

# Lecture Notes in Computer Science

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

2815

**Springer**

*Berlin*

*Heidelberg*

*New York*

*Hong Kong*

*London*

*Milan*

*Paris*

*Tokyo*

Yehuda Lindell

# Composition of Secure Multi-Party Protocols

A Comprehensive Study



Springer

## Series Editors

Gerhard Goos, Karlsruhe University, Germany  
Juris Hartmanis, Cornell University, NY, USA  
Jan van Leeuwen, Utrecht University, The Netherlands

## Author

Yehuda Lindell  
IBM T.J. Watson Research Center  
19 Skyline Drive, Hawthorne, NY 10532, USA  
E-mail: lindell@us.ibm.com

## Cataloging-in-Publication Data applied for

Bibliographic information published by Die Deutsche Bibliothek  
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie;  
detailed bibliographic data is available in the Internet at <<http://dnb.ddb.de>>.

CR Subject Classification (1998): E.3, C.2, D.4.6, K.4.1, K.4.4, K.6.5

ISSN 0302-9743

ISBN 3-540-20105-X Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag Berlin Heidelberg New York  
a member of BertelsmannSpringer Science+Business Media GmbH

<http://www.springer.de>

© Springer-Verlag Berlin Heidelberg 2003  
Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin GmbH  
Printed on acid-free paper SPIN: 10950029 06/3142 5 4 3 2 1 0

# Preface

In the setting of multi-party computation, sets of two or more parties with private inputs wish to jointly compute some (predetermined) function of their inputs. The computation should be such that the outputs received by the parties are correctly distributed, and furthermore, that the privacy of each party's input is preserved as much as possible. This encompasses any distributed computing task and includes computations as simple as coin-tossing and broadcast, and as complex as electronic voting, electronic auctions, electronic cash schemes, and anonymous transactions. The feasibility (and infeasibility) of multi-party computation has been extensively studied, resulting in a seemingly comprehensive understanding of what can and cannot be securely computed, and under what assumptions. However, most of this research relates only to the *stand-alone* setting, where a single set of parties execute a single protocol in isolation. In contrast, in modern network settings, it is usually the case that many different sets of parties run many different protocol executions at the same time. Furthermore, an active adversary can try to utilize these many different executions in order to somehow “break” the security of the individual protocols. A protocol that is secure in such a multi-execution setting is said to remain secure under “protocol composition”.

Unfortunately, protocols that are secure in the stand-alone setting do not necessarily remain secure under composition. In fact, as we show here, the situation is actually much worse. That is, in this work, we show that there exist multi-party tasks which have secure protocols for the stand-alone setting, but which cannot be solved by *any* protocol in the setting of composition. We demonstrate this fact on the important task of secure broadcast. Specifically, in the stand-alone model, it is known that a public-key infrastructure of digital signatures can be used in order to obtain broadcast that is secure *for any number of corrupted parties*. In contrast, we show that it is *impossible* to obtain a secure broadcast protocol that composes in parallel or concurrently (even assuming an infrastructure of digital signatures), unless one assumes that less than one third of the parties are corrupted. Thus, solving multi-party tasks when security under composition is required, is strictly harder than in the stand-alone model.

The above highlights the need for a renewed study, in the context of composition, of all the feasibility results that have been proven in the stand-alone

setting. Specifically, many fundamental questions regarding the feasibility and infeasibility of obtaining security under composition have yet to be even considered. The study of these questions therefore constitutes an important and pressing research project.

A recent important step forward in this direction was made with the introduction of a new security definition called “universal composability”. The importance of this definition is that protocols that are proven secure by it are guaranteed to remain secure in a very general setting of composition. In particular, it is guaranteed that such protocols will remain secure even when run concurrently with many other arbitrary protocols. This captures the security requirements of protocols in real settings, like that of the Internet. Previously, it has been shown that when a majority of the parties are assumed to be honest, universally composable protocols can be constructed for any multi-party task. (This result holds in the plain model, where no trusted setup phase is assumed.) It has also been shown that when no honest majority of parties is guaranteed (as is the case for the important two-party setting), there are important functions that *cannot* be securely realized in a universally composable way by any protocol in the plain model. Thus, an important open question is whether or not there exists a reasonable model in which universally composable multi-party computation of general functions can be achieved, when there is no honest majority. We show that in the common reference string model (where all parties have access to a common string that was generated in a trusted preprocessing phase), universally composable protocols exist for any two-party and multi-party function and for any number of corrupted parties. This result establishes the feasibility of obtaining secure two-party and multi-party computation (without an honest majority) under the most stringent security definition currently known. More importantly, it demonstrates that security in Internet-like settings can be achieved, at least in the common reference string model.

An additional result presented in this work is related to the fact that all known protocols for secure multi-party computation rely extensively on a broadcast channel, and implement this channel in a point-to-point network using a protocol for secure broadcast. Therefore, the impossibility of achieving broadcast under composition when a third or more parties are corrupted, implies that the above-mentioned multi-party protocols cannot be composed (again, when a third or more parties are corrupted). The use of broadcast therefore constitutes a serious obstacle to achieving protocol composition. In order to remove this obstacle, we first provide a new definition of secure multi-party computation that is a mild relaxation of previous definitions. Then, we show that under this definition, secure computation of any function can be achieved *without broadcast* by adapting known protocols. As a corollary we obtain secure multi-party protocols that compose (concurrently, when an honest majority is assumed, and in parallel when any number of parties may be corrupted). Another additional benefit of removing the need for broadcast

is that we also remove the need for a public-key infrastructure of digital signatures. Thus, we reduce the assumptions required for obtaining secure computation, even in the stand-alone model.

**Acknowledgements.** This work is a revised version of my Ph.D. thesis, written at the Weizmann Institute of Science in Israel. As such, first and foremost, I would like to thank my thesis advisors Oded Goldreich and Moni Naor. I am most grateful to Moni for helping me find my way in my first year. Although I spent most of this year studying (rather than researching), Moni continually threw research questions at me from many different fields of theoretical computer science. Eventually I found my way to cryptography, and by the time Moni went away on sabbatical leave I already had a large stack of problems to look at.

My work with Oded began in the second year of my studies and his contribution to me on both a personal and research level has been enormous. It was always a pleasure to work with Oded; his demand for excellence on the one hand and positive encouragement on the other served as great motivating factors throughout the ups and downs of the last few years. Oded has served as a role model for me, and I am sure that he will continue to be a source of inspiration. I have learnt many things from Oded; both technically and also regarding one's attitude and approach to research. I feel greatly indebted to him and incredibly lucky to have had him as my advisor. I hope and believe that I will have many opportunities to continue working with Oded in the future.

I had the great privilege of carrying out my Ph.D. research at the Weizmann Institute of Science. The atmosphere of the Computer Science department, due to both the faculty and students, significantly contributed to the success of my studies. Throughout my graduate studies, I learnt much from the faculty members of the department and from fellow students. I would like to especially mention Shafi Goldwasser, with whom I was very fortunate to work. I would also like to give special thanks to Alon Rosen and Boaz Barak with whom I had countless fruitful interactions.

I would also like to thank the many people with whom I worked and interacted. First, special thanks to my co-authors: Boaz Barak, Ran Canetti, Rosario Gennaro, Oded Goldreich, Shafi Goldwasser, Eyal Kushilevitz, Anna Lysyanskaya, Benny Pinkas, Tal Rabin, and Salil Vadhan. Thanks also to Marc Fischlin, Shai Halevi, Jonathan Katz, Hugo Krawczyk, Michael Langberg, Daniele Micciancio, Kobbi Nissim, and Omer Reingold. My work and discussions with you all were both enjoyable and productive.

In addition, I wish to acknowledge my co-authors on the results that make up this book. Chapter 2 is due to joint work with Anna Lysyanskaya and Tal Rabin [LLR02], Chapter 3 resulted from joint work with Shafi Goldwasser [GL02], and Chapter 4 is a result of joint work with Ran Canetti, Rafi Ostrovsky, and Amit Sahai [CLOS02].

Last but not least, I would like to thank my beautiful wife Yael for all her support. The pressures of a Ph.D. are sometimes taxing, and Yael's understanding throughout was always felt and appreciated.

June 2003

Yehuda Lindell



# Table of Contents

<b>Foreword</b> .....	VII
<b>Preface</b> .....	XI
<b>1. Introduction</b> .....	1
1.1 Secure Multi-party Computation .....	1
1.2 Protocol Composition .....	8
1.2.1 Types of Protocol Composition .....	8
1.2.2 Feasibility of Security under Composition .....	10
1.3 Our Results .....	15
1.3.1 The Composition of Authenticated Byzantine Agreement .....	16
1.3.2 Secure Computation without Agreement .....	18
1.3.3 Universally Composable Multi-party Computation ....	18
<b>2. The Composition of Authenticated Byzantine Agreement</b> .	21
2.1 Introduction .....	21
2.2 Definitions .....	25
2.2.1 Computational Model .....	25
2.2.2 Byzantine Generals/Agreement .....	26
2.2.3 Composition of Protocols .....	27
2.3 Impossibility for Parallel Composition .....	29
2.4 Sequential Composition of Deterministic Protocols .....	35
2.5 Authenticated Byzantine Agreement Using Unique Identifiers	38
<b>3. Secure Computation without Agreement</b> .....	45
3.1 Introduction .....	46
3.1.1 Byzantine Agreement and Secure Multi-party Computation .....	46
3.1.2 Our Results .....	47
3.1.3 Related Work .....	51
3.2 Definitions – Secure Computation .....	53
3.2.1 Execution in the Ideal Model .....	54
3.2.2 Execution in the Real Model .....	58

3.2.3	Security as Emulation of a Real Execution in the Ideal Model.....	60
3.3	Broadcast with Abort .....	61
3.3.1	Strengthening Broadcast with Abort .....	62
3.4	Secure Computation with Abort and No Fairness .....	62
3.5	Secure Computation with Abort and Partial Fairness .....	72
3.6	Obtaining Security under Self Composition .....	77
<b>4.</b>	<b>Universally Composable Multi-party Computation.....</b>	<b>81</b>
4.1	Introduction .....	81
4.2	Overview .....	84
4.2.1	The Model .....	84
4.2.2	An Outline of the Results and Techniques .....	86
4.3	Preliminaries.....	93
4.3.1	Universally Composable Security: The General Framework.....	93
4.3.2	Universal Composition with Joint State .....	102
4.3.3	Well-Formed Functionalities .....	106
4.4	Two-Party Secure Computation for Semi-honest Adversaries .	108
4.4.1	Universally Composable Oblivious Transfer .....	108
4.4.2	The General Construction .....	116
4.5	Universally Composable Commitments .....	128
4.6	Universally Composable Zero-Knowledge .....	141
4.7	The Commit-and-Prove Functionality $\mathcal{F}_{\text{CP}}$ .....	143
4.7.1	UC Realizing $\mathcal{F}_{\text{CP}}$ for Static Adversaries .....	144
4.7.2	UC Realizing $\mathcal{F}_{\text{CP}}$ for Adaptive Adversaries .....	149
4.8	Two-Party Secure Computation for Malicious Adversaries ...	155
4.8.1	The Protocol Compiler .....	155
4.8.2	Conclusions.....	161
4.9	Multi-party Secure Computation .....	162
4.9.1	Multi-party Secure Computation for Semi-honest Adversaries.....	163
4.9.2	Authenticated Broadcast .....	169
4.9.3	One-to-Many Extensions of $\mathcal{F}_{\text{MCOM}}$ , $\mathcal{F}_{\text{ZK}}$ , and $\mathcal{F}_{\text{CP}}$ .....	172
4.9.4	Multi-party Secure Computation for Malicious Adversaries .....	179
	<b>References .....</b>	<b>185</b>
	<b>Index .....</b>	<b>191</b>

# References

- [B01] B. Barak. How to go Beyond the Black-Box Simulation Barrier. In *42nd FOCS*, pages 106–115, 2001.
- [B02] B. Barak. Constant-Round Coin-Tossing With a Man in the Middle or Realizing the Shared Random String Model. In *43rd FOCS*, pages 345–355, 2002.
- [B91a] D. Beaver. Foundations of Secure Interactive Computing. In *CRYPTO'91*, Springer-Verlag (LNCS 576), pages 377–391, 1991.
- [B91b] D. Beaver. Secure Multi-party Protocols and Zero-Knowledge Proof Systems Tolerating a Faulty Minority. *Journal of Cryptology*, 4(2):75–122, 1991.
- [B97] D. Beaver. Plug and play encryption. In *CRYPTO'97*, Springer-Verlag (LNCS 1294), pages 75–89, 1997.
- [BBM00] M. Bellare, A. Boldyreva, and S. Micali. Public-Key Encryption in a Multi-user Setting: Security Proofs and Improvements. In *EUROCRYPT'00*, Springer-Verlag (LNCS 1807), pages 259–274, 2000.
- [BG89] D. Beaver and S. Goldwasser. Multiparty Computation with Fault Majority. In *CRYPTO'89*, Springer-Verlag (LNCS 435), 1989.
- [BMR90] D. Beaver, S. Micali and P. Rogaway. The Round Complexity of Secure Protocols. In *22nd STOC*, pages 503–513, 1990.
- [BGW88] M. Ben-Or, S. Goldwasser and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In *20th STOC*, pages 1–10, 1988.
- [B86] M. Blum. How to Prove a Theorem So No One Else Can Claim It. *Proceedings of the International Congress of Mathematicians*, pages 1444–1451, 1986.
- [BFM88] M. Blum, P. Feldman and S. Micali. Non-interactive zero-knowledge and its applications. In *20th STOC*, pages 103–112, 1988.
- [C00] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.
- [C01] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd FOCS*, pages 136–145, 2001. Full version available at <http://eprint.iacr.org/2000/067>.
- [CFGN96] R. Canetti, U. Feige, O. Goldreich and M. Naor. Adaptively Secure Multi-Party Computation. In *28th STOC*, pages 639–648, 1996.
- [CF01] R. Canetti and M. Fischlin. Universally Composable Commitments. In *CRYPTO'01*, Springer-Verlag (LNCS 2139), pages 19–40, 2001.
- [CH94] R. Canetti and A. Herzberg. Maintaining Security in the Presence of Transient Faults. In *CRYPTO'94*, Springer-Verlag (LNCS 839), pages 425–438, 1994.

- [CKPR01] R. Canetti, J. Kilian, E. Petrank, and A. Rosen. Black-Box Concurrent Zero-Knowledge Requires  $\Omega(\log n)$  Rounds. In *33rd STOC*, pages 570–579, 2001.
- [CK02] R. Canetti and H. Krawczyk. Universally Composable Notions of Key-Exchange and Secure Channels. In *EUROCRYPT'02*, Springer-Verlag (LNCS 2332), pages 337–351, 2002.
- [CKL03] R. Canetti, E. Kushilevitz and Y. Lindell. On the Limitations of Universal Composition Without Set-Up Assumptions. In *EUROCRYPT'03*, Springer-Verlag (LNCS 2656), pages 68–86, 2003.
- [CLOS02] R. Canetti, Y. Lindell, R. Ostrovsky and A. Sahai. Universally Composable Two-Party and Multi-Party Computation. In *34th STOC*, pages 494–503, 2002.
- [CR03] R. Canetti and T. Rabin. Universal Composition with Joint State. In *CRYPTO'03*, 2003.
- [CCD88] D. Chaum, C. Crepeau and I. Damgard. Multi-party Unconditionally Secure Protocols. In *20th STOC*, pages 11–19, 1988.
- [C86] R. Cleve. Limits on the Security of Coin Flips when Half the Processors are Faulty. In *18th STOC*, pages 364–369, 1986.
- [DG03] I. Damgard and J. Groth. Non-Interactive and Reusable Non-Malleable Commitment Schemes. In *35th STOC*, pages 426–435, 2003.
- [DN00] I. Damgard and J.B. Nielsen. Improved non-committing encryption schemes based on general complexity assumptions. In *CRYPTO'00*, Springer-Verlag (LNCS 1880), pages 432–450, 2000.
- [DN02] I. Damgard and J.B. Nielsen. Perfect Hiding or Perfect Binding Universally Composable Commitment Schemes with Constant Expansion Factor. In *CRYPTO'02*, Springer-Verlag (LNCS 2442), pages 581–596, 2002.
- [DDOPS01] A. De Santis, G. Di Crescenzo, R. Ostrovsky, G. Persiano, A. Sahai. Robust Non-interactive Zero-Knowledge. In *CRYPTO'01*, Springer-Verlag (LNCS 2139), pages 566–598, 2001.
- [DP92] A. De Santis and G. Persiano. Zero-Knowledge Proofs of Knowledge Without Interaction. In *33rd FOCS*, pages 427–436, 1992.
- [DIO98] G. Di Crescenzo, Y. Ishai, and R. Ostrovsky. Non-Interactive and Non-Malleable Commitment. In *30th STOC*, pages 141–150, 1998.
- [DKOS01] G. Di Crescenzo, J. Katz, R. Ostrovsky and A. Smith. Efficient and Non-interactive Non-malleable Commitment. In *EUROCRYPT'01*, Springer-Verlag (LNCS 2045), pages 40–59, 2001.
- [DM00] Y. Dodis and S. Micali. Parallel Reducibility for Information-Theoretically Secure Computation. In *CRYPTO'00*, Springer-Verlag (LNCS 1880), pages 74–92, 2000.
- [D82] D. Dolev. The Byzantine Generals Strike Again. *Journal of Algorithms*, 3(1):14–30, 1982.
- [DDN91] D. Dolev, C. Dwork and M. Naor. Non-malleable cryptography. *SIAM Journal on Computing*, 30(2):391–437, 2000. (Preliminary version in *23rd STOC*, 1991.)
- [DNS98] C. Dwork, M. Naor, and A. Sahai. Concurrent Zero-Knowledge. In *30th STOC*, pages 409–418, 1998.
- [DS83] D. Dolev and H.R. Strong. Authenticated Algorithms for Byzantine Agreement. *SIAM Journal on Computing*, 12(4):656–665, 1983.
- [EGL85] S. Even, O. Goldreich and A. Lempel. A Randomized Protocol for Signing Contracts. In *Communications of the ACM*, 28(6):637–647, 1985.

- [F90] U. Feige. *Alternative Models for Zero Knowledge Interactive Proofs*. Ph.D. Thesis, Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, 1990. Available from <http://www.wisdom.weizmann.ac.il/~feige>.
- [FS89] U. Feige and A. Shamir. Zero-Knowledge Proofs of Knowledge in Two Rounds. In *CRYPTO'89*, Springer-Verlag (LNCS 435), pages 526–544, 1989.
- [FM97] P. Feldman and S. Micali. An Optimal Algorithm for Synchronous Byzantine Agreement. *SIAM Journal on Computing*, 26(2):873–933, 1997.
- [FLM86] M. Fischer, N. Lynch, and M. Merritt. Easy Impossibility Proofs for Distributed Consensus Problems. *Distributed Computing*, 1(1):26–39, 1986.
- [FF00] M. Fischlin and R. Fischlin. Efficient Non-malleable Commitment Schemes. In *CRYPTO'00*, Springer-Verlag (LNCS 1880), pages 413–431, 2000.
- [FGMR02] M. Fitzi, N. Gisin, U. Maurer and O. Von Rotz. Unconditional Byzantine Agreement and Multi-Party Computation Secure Against Dishonest Minorities from Scratch. In *EUROCRYPT'02*, Springer-Verlag (LNCS 2332), pages 482–501, 2002.
- [FGHS02] M. Fitzi, D. Gottesman, M. Hirt, T. Holenstein and A. Smith. Byzantine Agreement Secure Against Faulty Majorities From Scratch. In *21st PODC*, pages 118–126, 2002.
- [FM00] M. Fitzi and U. Maurer. From Partial Consistency to Global Broadcast. In *32nd STOC*, pages 494–503, 2000.
- [GHY87] Z. Galil, S. Haber and M. Yung. Cryptographic Computation: Secure Fault Tolerant Protocols and the Public Key Model. In *CRYPTO'87*, Springer-Verlag (LNCS 293), pages 135–155, 1987.
- [GM00] J. Garay and P. Mackenzie. Concurrent Oblivious Transfer. In *41st FOCS*, pages 314–324, 2000.
- [G01] O. Goldreich. *Foundations of Cryptography: Volume 1 – Basic Tools*. Cambridge University Press, 2001.
- [G02a] O. Goldreich. *Secure Multi-Party Computation*. Manuscript, v. 1.4, 2002. Available from <http://www.wisdom.weizmann.ac.il/~oded/pp.html>.
- [G02b] O. Goldreich. Concurrent Zero-Knowledge With Timing Revisited. In *34th STOC*, pages 332–340, 2002.
- [GoKa96] O. Goldreich and A. Kahan. How To Construct Constant-Round Zero-Knowledge Proof Systems for NP. *Journal of Cryptology*, 9(3):167–190, 1996.
- [GoKr96] O. Goldreich and H. Krawczyk. On the composition of zero-knowledge proof systems. *SIAM Journal on Computing*, 25(1):169–192, 1996.
- [GL89] O. Goldreich and L. Levin. A Hard Predicate for All One-way Functions. In *21st STOC*, pages 25–32, 1989.
- [GO94] O. Goldreich and Y. Oren. Definitions and Properties of Zero-Knowledge Proof Systems. *Journal of Cryptology*, 7(1):1–32, 1994.
- [GMW86] O. Goldreich, S. Micali and A. Wigderson. Proofs that Yield Nothing but their Validity or All Languages in NP Have Zero-Knowledge Proof Systems. *Journal of the ACM*, 38(1):691–729, 1991. (Preliminary version in *27th FOCS*, 1986.)
- [GMW87] O. Goldreich, S. Micali and A. Wigderson. How to Play any Mental Game – A Completeness Theorem for Protocols with Honest Majority. In *19th STOC*, pages 218–229, 1987. For details see [G02a].

- [GL90] S. Goldwasser and L. Levin. Fair Computation of General Functions in Presence of Immoral Majority. In *CRYPTO'90*, Springer-Verlag (LNCS 537), pages 77–93, 1990.
- [GL02] S. Goldwasser and Y. Lindell. Secure Computation Without Agreement. In *16th DISC*, pages 17–32, 2002.
- [GMR85] S. Goldwasser, S. Micali and C. Rackoff The Knowledge Complexity of Interactive Proof Systems. *SIAM Journal on Computing*, 18(1):186–208, 1989. (Preliminary version in *17th STOC*, 1985.)
- [GMR88] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.
- [GLR95] L. Gong, P. Lincoln, and J. Rushby. Byzantine Agreement with Authentication: Observations and Applications in Tolerating Hybrid and Link Faults. In *Dependable Computing for Critical Applications*, pages 139–157, 1995.
- [G78] J.N. Gray. Notes on data base operating systems. In *Operating Systems: An Advanced Course*, Springer-Verlag (LNCS 60), chapter 3.F, page 465, 1978.
- [K89] J. Kilian. *Uses of Randomness in Algorithms and Protocols*. The ACM Distinguished Dissertation 1989, MIT press.
- [KP01] J. Kilian and E. Petrank. Concurrent and resettable zero-knowledge in poly-logarithmic rounds. In *33rd STOC*, pages 560–569, 2001.
- [KPR98] J. Kilian, E. Petrank and C. Rackoff. Lower Bounds for Zero Knowledge on the Internet. In *39th FOCS*, pages 484–492, 1998.
- [LSP82] L. Lamport, R. Shostack, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.
- [L83] L. Lamport. The weak byzantine generals problem. *Journal of the ACM*, 30(3):668–676, 1983.
- [L01] Y. Lindell. Parallel Coin-Tossing and Constant-Round Secure Two-Party Computation. In the *Journal of Cryptology*, 16(3):143–184, 2003. (Preliminary version in *CRYPTO'01*, 2001.)
- [L03a] Y. Lindell. Bounded-Concurrent Secure Two-Party Computation Without Setup Assumptions. In *35th STOC*, pages 683–692, 2003.
- [L03b] Y. Lindell. General Composition and Universal Composability in Secure Multi-Party Computation. In *44th FOCS*, 2003.
- [LLR02] Y. Lindell, A. Lysysanskaya and T. Rabin. On the Composition of Authenticated Byzantine Agreement. In *34th STOC*, pages 514–523, 2002.
- [MR91] S. Micali and P. Rogaway. Secure computation. Unpublished manuscript, 1992. Preliminary version in *CRYPTO'91*, Springer-Verlag (LNCS 576), pages 392–404, 1991.
- [N91] M. Naor. Bit Commitment using Pseudorandom Generators. *Journal of Cryptology*, 4(2):151–158, 1991.
- [OY91] R. Ostrovsky and M. Yung. How to Withstand Mobile Virus Attacks. In *10th PODC*, pages 51–59, 1991.
- [PR03] R. Pass and A. Rosen Bounded-Concurrent Secure Two-Party Computation in a Constant Number of Rounds. In *44th FOCS*, 2003.
- [PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2):228–234, 1980.
- [PW96] B. Pfitzmann and M. Waidner. Information-Theoretic Pseudosignatures and Byzantine Agreement for  $t \geq n/3$ . Technical Report RZ 2882 (#90830), IBM Research, 1996.

- [PW00] B. Pfitzmann and M. Waidner. Composition and integrity preservation of secure reactive systems. *7th ACM Conference on Computer and Communication Security*, pages 245–254, 2000.
- [PRS02] M. Prabhakaran, A. Rosen and A. Sahai. Concurrent Zero Knowledge With Logarithmic Round Complexity. In *43rd FOCS*, pages 366–375, 2002.
- [R81] M. Rabin. How to Exchange Secrets by Oblivious Transfer. Tech. Memo TR-81, Aiken Computation Laboratory, Harvard U., 1981.
- [RB89] T. Rabin and M. Ben-Or. Verifiable Secret Sharing and Multi-party Protocols with Honest Majority. In *21st STOC*, pages 73–85, 1989.
- [RK99] R. Richardson and J. Kilian. On the Concurrent Composition of Zero-Knowledge Proofs. In *EUROCRYPT'99*, Springer-Verlag (LNCS 1592), pages 415–413, 1999.
- [RSA78] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
- [R00] A. Rosen. A Note on the Round-Complexity of Concurrent Zero-Knowledge. In *CRYPTO'00*, Springer-Verlag (LNCS 1880), pages 451–468, 2000.
- [S99] A. Sahai. Non-Malleable Non-Interactive Zero-Knowledge and Adaptive Chosen-Ciphertext Security. In *40th FOCS*, pages 543–553, 1999.
- [Y86] A. Yao. How to Generate and Exchange Secrets. In *27th FOCS*, pages 162–167, 1986.

# Foreword

Modern society is quite preoccupied with statistics such as the average, median, and deviation of various attributes (e.g., salary) of its members. On the other hand, individuals often wish to keep their own attributes secret (although they are interested in the above statistics). Furthermore, on top of being suspicious of other people, individuals are growing to be suspicious of all (the society's) establishments and are unwilling to trust the latter with their secrets. Under these circumstances it is not clear whether there is a way for the members of the society to obtain various statistics (regarding all secrets) without revealing their individual secrets to other people.

The above question is a special case of a general problem. We are talking about computing some (predetermined) function of inputs that are scattered among different parties, without having these parties reveal their individual inputs. The mutually suspicious parties have to employ some distributed protocol in order to compute the function value, without leaking any other information regarding their inputs to one another. Furthermore, in some settings, some of the parties may deviate from the protocol, and it is desired that this should not be of any advantage to them. At best, we would like to “emulate” a trusted party (which collects the inputs from the parties, computes the corresponding outputs, and hands them to the corresponding parties), and do so in a distributed setting in which no trusted parties exist. This, in a nutshell, is what secure cryptographic protocols are all about.

General results concerning secure two-party and multi-party computations were first announced in the mid 1980s (cf. [GMW86, Y86, GMW87] and details in [G02a]). Put briefly, assuming the existence of trapdoor permutations, these results assert that one can construct protocols for securely computing any desired multi-party functionality. These results either require a majority of honest players or allow dishonest players to suspend the execution (while being detected as bad). Subsequent “unconditional” results in the “private channel model” require a  $2/3$ -majority of honest players [BGW88, CCD88].

The aforementioned results were discovered at a time in which *intensive* electronic multi-party interactions seemed a remote possibility. So it seems fair to say that, while generating considerable interest within the theory community, these results generated little interest in the applied cryptography



community. But times have changed: intensive electronic multi-party interactions seem almost a reality, and the entire cryptographic community seems very much interested in a variety of natural problems which arise from such a reality. This has triggered a serious reconsideration of the definitions and results established in the 1980s. The current work makes an important contribution to this fundamental project.

One important research direction initiated and developed in recent years is the study of the preservation of security under concurrent executions of protocols. This research is aimed at extending the treatment developed in the 1980s, which only refers to stand-alone executions. Needless to say, in most settings, one would like security to hold not only when a single execution takes place but also when multiple executions are taking place concurrently. Furthermore, whereas we do not want to require honest parties to coordinate their actions in the various executions (which may not even be possible), the adversary may coordinate its actions in the various executions. Thus, this research direction is of great theoretical and practical importance. Two of the three technical chapters of the current work present important contributions to this research direction:

1. Chapter 4 deals with the notion of environmental-security (a.k.a “Universal Composability” or UC-security) put forward by Canetti [C01]. In particular, environmental-security implies security under concurrent executions of protocols. It was known how to construct environmentally-secure multi-party computations *with an honest majority*, and that this is impossible (in the bare model) when the honest parties are not in the majority (and most importantly in the two-party case). The current work shows that augmenting the model with a “random reference string” yields a model in which environmentally-secure *two-party* computations are possible. More generally, environmentally-secure multi-party computations are possible in this model even without an honest majority. It should be noted that the availability of a “random reference string” is a reasonable assumption that has been used in several related contexts.
2. Chapter 2 deals with the composability of Byzantine Agreement protocols. It is well known that (in the plain model) a special majority (of two-thirds) of honest players is required for achieving Byzantine Agreement. To obtain higher resilience, one typically uses a public-key infrastructure, yielding the so-called *authenticated* Byzantine Agreement problem. The current work shows that the security of authenticated Byzantine Agreement protocols is not preserved under concurrent composition, unless distinct IDs can be externally assigned (in a secure manner) to individual executions. Thus, in contrast to previous beliefs, authenticated Byzantine Agreement protocols can be safely used *only* in contexts where each protocol invocation is assigned a distinct ID. Other related results are also presented in Chapter 2.

In addition, this work studies the phenomenon that Byzantine Agreement protocols (or a postulated broadcast channel) are used in all prior work regarding secure multi-party computation. This question of whether or not this phenomenon is inherent to secure multi-party computation arises even in the stand-alone context. In Chapter 3 it is shown that if the definition of secure multi-party computation is slightly relaxed (in a way that does not imply Byzantine Agreement as a special case) then one can obtain secure protocols without using Byzantine Agreement (or postulated broadcast channels). Avoiding the use of Byzantine Agreement has several advantages for the stand-alone model as well as for restricted types of protocol composition.

The focus of the current work is on the *general* study of secure multi-party computation (rather than on protocols for solving specific problems), which is natural in the context of the theoretical treatment of the subject matter. We wish to highlight the importance of this theoretical study to practice. Firstly, this study clarifies fundamental issues regarding security in a multi-party environment. Secondly, it draws the lines between what is possible in principle and what is not. Thirdly, it develops general techniques for designing secure protocols. And last, sometimes, it may even yield schemes (or modules) that may be incorporated in practical systems. Thus, we believe that the current work is both of theoretical and practical importance.

Oded Goldreich, Weizmann Institute of Science.

# 1. Introduction

## 1.1 Secure Multi-party Computation

*Distributed computing* considers the scenario where a number of distinct, yet connected, computing devices (or parties) wish to carry out a joint computation of some function. For example, these devices may be servers who hold a distributed database system, and the function to be computed may be a database update of some kind. The aim of *secure multi-party computation* is to enable parties to carry out such distributed computing tasks in a secure manner. Whereas distributed computing classically deals with questions of computing under the threat of machine crashes and other inadvertent faults, secure multi-party computation is concerned with the possibility of deliberately malicious behaviour by some adversarial entity. That is, it is assumed that a protocol execution may come under “attack” by an external entity, or even by a subset of the participating parties. The aim of this attack may be to learn private information or cause the result of the computation to be incorrect. Thus, two important requirements on any secure computation protocol are *privacy* and *correctness*. The privacy requirement states that nothing should be learned beyond what is absolutely necessary; more exactly, parties should learn their output and nothing else. The correctness requirement states that each party should receive its correct output. Therefore, the adversary must not be able to cause the result of the computation to deviate from the function that the parties had set out to compute.

The setting of secure multi-party computation encompasses tasks as simple as coin-tossing and broadcast, and as complex as electronic voting, electronic auctions, electronic cash schemes, contract signing, anonymous transactions, and private information retrieval schemes. Consider for a moment the tasks of voting and auctions. The privacy requirement for an election protocol ensures that no parties learn anything about the individual votes of other parties, and the correctness requirement ensures that no coalition of parties can influence the outcome of the election beyond just voting for their preferred candidate. Likewise, in an auction protocol, the privacy requirement ensures that only the winning bid is revealed (this may be desired), and the correctness requirement ensures that the highest bidder is indeed the party to win (and so the auctioneer, or any other party, cannot bias the outcome).

Due to its generality, the setting of secure multi-party computation can model almost every, if not every, cryptographic problem (including the classic tasks of encryption and authentication). Therefore, questions of feasibility and infeasibility for secure multi-party computation are fundamental to the theory and practice of cryptography.

**Security in multi-party computation.** As we have mentioned above, the model that we consider is one where an adversarial entity controls some subset of the parties and wishes to attack the protocol execution. The parties under the control of the adversary are called **corrupted**, and follow the adversary's instructions. Secure protocols should withstand any adversarial attack (where the exact power of the adversary will be discussed later). In order to formally claim and prove that a protocol is secure, a precise definition of security for multi-party computation is required. A number of different definitions have been proposed and these definitions aim to ensure a number of important security properties that are general enough to capture most (if not all) multi-party computation tasks. We now describe the most central of these properties:

- *Privacy:* No party should learn anything more than its prescribed output. In particular, the only information that should be learned about other parties' inputs is what can be derived from the output itself. For example, in an auction where the only bid revealed is that of the highest bidder, it is clearly possible to derive that all other bids were lower than the winning bid. However, this should be the only information revealed about the losing bids.
- *Correctness:* Each party is guaranteed that the output that it receives is correct. To continue with the example of an auction, this implies that the party with the highest bid is guaranteed to win, and no party including the auctioneer can influence this.
- *Independence of Inputs:* Corrupted parties must choose their inputs independently of the honest parties' inputs. This property is crucial in a sealed auction, where bids are kept secret and parties must fix their bids independently of others. We note that independence of inputs is *not* implied by privacy. For example, it may be possible to generate a higher bid, without knowing the value of the original one. Such an attack can actually be carried out on some encryption schemes (i.e., given an encryption of \$100, it is possible to generate a valid encryption of \$101, without knowing the original encrypted value).
- *Guaranteed Output Delivery:* Corrupted parties should not be able to prevent honest parties from receiving their output. In other words, the adversary should not be able to disrupt the computation by carrying out a "denial of service" attack.

- *Fairness*: Corrupted parties should receive their outputs if and only if the honest parties also receive their outputs. The scenario where a corrupted party obtains output and an honest party does not should not be allowed to occur. This property can be crucial, for example, in the case of contract signing. Specifically, it would be very problematic if the corrupted party received the signed contract and the honest party did not.

We stress that the above list does *not* constitute a definition of security, but rather a set of requirements that should hold for any secure protocol. Indeed, one possible approach to defining security is to just generate a list of separate requirements (as above) and then say that a protocol is secure if all of these requirements are fulfilled. However, this approach is not satisfactory for the following reasons. First, it may be possible that an important requirement was missed. This is especially true because different applications have different requirements, and we would like a definition that is general enough to capture all applications. Second, the definition should be simple enough so that it is trivial to see that *all* possible adversarial attacks are prevented by the proposed definition.

The standard definition today (cf. [C00] following [GL90, B91a, MR91]) therefore formalizes security in the following general way. As a mental experiment, consider an “ideal world” in which an external trusted (and incorruptible) party is willing to help the parties carry out their computation. In such a world, the parties can simply send their inputs to the trusted party, who then computes the desired function and passes each party its prescribed output. Since the only action carried out by a party is that of sending its input to the trusted party, the only freedom given to the adversary is in choosing the corrupted parties’ inputs. Notice that all of the above-described security properties (and more) hold in this ideal computation. For example, privacy holds because the only message ever received by a party is its output (and so it cannot learn any more than this). Likewise, correctness holds since the trusted party cannot be corrupted and so will always compute the function correctly.

Of course, in the “real world”, there is no external party that can be trusted by all parties. Rather, the parties run some protocol amongst themselves without any help. Despite this, a secure protocol should emulate the so-called “ideal world”. That is, a real protocol that is run by the parties (in a world where no trusted party exists) is said to be *secure*, if no adversary can do more harm in a real execution than in an execution that takes place in the ideal world. This can be formulated by saying that for any adversary carrying out a successful attack in the real world, there exists an adversary that successfully carries out the same attack in the ideal world. However, successful adversarial attacks *cannot* be carried out in the ideal world. We therefore conclude that all adversarial attacks on protocol executions in the real world must also fail.

More formally, the security of a protocol is established by comparing the outcome of a real protocol execution to the outcome of an ideal computation. That is, for any adversary attacking a real protocol execution, there exists an adversary attacking an ideal execution (with a trusted party) such that the input/output distributions of the adversary and the participating parties in the real and ideal executions are essentially the same. Thus a real protocol execution “emulates” the ideal world. This formulation of security is called the **ideal/real simulation paradigm**. In order to motivate the usefulness of this definition, we describe why all the properties described above are implied. Privacy follows from the fact that the adversary’s output is the same in the real and ideal executions. Since the adversary learns nothing beyond the corrupted party’s outputs in an ideal execution, the same must be true for a real execution. Correctness follows from the fact that the honest parties’ outputs are the same in the real and ideal executions, and from the fact that in an ideal execution, the honest parties all receive correct outputs as computed by the trusted party. Regarding independence of inputs, notice that in an ideal execution, all inputs are sent to the trusted party before any output is received. Therefore, the corrupted parties know nothing of the honest parties’ inputs at the time that they send their inputs. In other words, the corrupted parties’ inputs are chosen independently of the honest parties’ inputs, as required. Finally, guaranteed output delivery and fairness hold in the ideal world because the trusted party always returns all outputs. The fact that it also holds in the real world again follows from the fact that the honest parties’ outputs are the same in the real and ideal executions.

We remark that the above informal definition is actually “overly ideal” and needs to be relaxed in settings where the adversary controls a half or more of the participating parties (that is, in the case that there is no honest majority). When this number of parties is corrupted, it is known that it is *impossible* to obtain general protocols for secure multi-party computation that guarantee output delivery and fairness (e.g. [G78, C86]). Therefore, the definition is relaxed and the adversary is allowed to abort the computation (i.e., cause it to halt before termination), meaning that “guaranteed output delivery” is not fulfilled. Furthermore, the adversary can cause this abort to take place after it has already obtained its output, but before all the honest parties receive their outputs. Thus “fairness” is not achieved. Loosely speaking, the relaxed definition is obtained by modifying the ideal execution and giving the adversary the additional capability of instructing the trusted party to not send outputs to some of the honest parties. Otherwise, the definition remains identical and thus all the other properties are still preserved. It turns out that subtle differences in the way that this relaxation is defined can have significant ramifications on the feasibility of obtaining secure computation and on composition. This issue is dealt with in depth in Chapter 3; see Section 1.3.2 for more details.

We note that there are works that aim to provide intermediate notions of fairness [Y86, GHY87, BG89, GL90]. However, for now, we limit our reference to the cases that either (complete) fairness and output delivery are guaranteed, or neither fairness (of any type) nor output delivery are guaranteed.

**Adversarial power.** The above informal definition of security omits one very important issue: the power of the adversary that attacks a protocol execution. As we have mentioned, the adversary controls a subset of the participating parties in the protocol. However, we have not described the corruption strategy (i.e., when or how parties come under the “control” of the adversary), the allowed adversarial behaviour (i.e., does the adversary just passively gather information or can it instruct the corrupted parties to act maliciously), and what complexity the adversary is assumed to be (i.e., is it polynomial-time or computationally unbounded). We now describe the main types of adversaries that have been considered:

1. **Corruption strategy:** The corruption strategy deals with the question of when and how parties are corrupted. There are two main models:
  - a) **Static corruption model:** In this model, the adversary is given a fixed set of parties whom it controls. Honest parties remain honest throughout and corrupted parties remain corrupted.
  - b) **Adaptive corruption model:** Rather than having a fixed set of corrupted parties, adaptive adversaries are given the capability of corrupting parties during the computation. The choice of who to corrupt, and when, can be arbitrarily decided by the adversary and may depend on its view of the execution (for this reason it is called adaptive). This strategy models the threat of an external “hacker” breaking into a machine during an execution. We note that in this model, once a party is corrupted, it remains corrupted from that point on.

An additional model, called the **proactive model** [OY91, CH94], considers the possibility that parties are corrupted for a certain period of time only. Thus, honest parties may become corrupted throughout the computation (like in the adaptive adversarial model), but corrupted parties may also become honest.

2. **Allowed adversarial behaviour:** Another parameter that must be defined relates to the actions that corrupted parties are allowed to take. Once again, there are two main types of adversaries:
  - a) **Semi-honest adversaries:** In the semi-honest adversarial model, even corrupted parties correctly follow the protocol specification. However, the adversary obtains the internal state of all the corrupted parties (including the transcript of all the messages received), and attempts to use this to learn information that should remain private. This is a rather weak adversarial model. However, there are some settings where it can realistically model the threats to the system. Semi-honest adversaries are also called “honest-but-curious” and “passive”.

- b) **Malicious adversaries:** In this adversarial model, the corrupted parties can *arbitrarily* deviate from the protocol specification, according to the adversary’s instructions. In general, providing security in the presence of malicious adversaries is preferred, as it ensures that no adversarial attack can succeed. Malicious adversaries are also called “active”.
3. **Complexity:** Finally, we consider the assumed computational complexity of the adversary. As above, there are two categories here:
- a) **Polynomial-time:** The adversary is allowed to run in (probabilistic) polynomial-time (and sometimes, expected polynomial-time). The specific computational model used differs, depending on whether the adversary is uniform (in which case, it is a probabilistic polynomial-time Turing machine) or non-uniform (in which case, it is modelled by a polynomial-size family of circuits).
  - b) **Computationally unbounded:** In this model, the adversary has no computational limits whatsoever.

The above distinction regarding the complexity of the adversary yields two very different models for secure computation: the *information-theoretic* model [BGW88, CCD88] and the *computational* model [Y86, GMW87]. In the information-theoretic setting, the adversary is not bound to any complexity class (and in particular, is not assumed to run in polynomial-time). Therefore, results in this model hold unconditionally and do not rely on any complexity or cryptographic assumptions. The only assumption used is that parties are connected via ideally *private* channels (i.e., it is assumed that the adversary cannot eavesdrop or interfere with the communication between honest parties).

In contrast, in the computational setting the adversary is assumed to be polynomial-time. Results in this model typically assume cryptographic assumptions like the existence of trapdoor permutations. We note that it is not necessary here to assume that the parties have access to ideally private channels, because such channels can be implemented using public-key encryption. However, it is assumed that the communication channels between parties are authenticated; that is, if two honest parties communicate, then the adversary can eavesdrop but cannot modify any message that is sent. Such authentication can be achieved using digital signatures [GMR88] and a public-key infrastructure.

We remark that all possible combinations of the above types of adversaries have been considered in the literature.

**Feasibility of secure multi-party computation.** The above-described definition of security seems to be very restrictive in that no adversarial success is tolerated. Thus, one may wonder whether it is even possible to obtain secure protocols under this definition, and if yes, for which distributed computing tasks. Perhaps surprisingly, powerful feasibility results have been established, demonstrating that in fact, *any* distributed computing task can be securely



computed. We now briefly state the most central of these results; let  $n$  denote the number of participating parties and let  $t$  denote a bound on the number of parties that may be corrupted:

1. For  $t < n/3$  (i.e., when less than a third of the parties can be corrupted), secure multi-party protocols with fairness and guaranteed output delivery can be achieved for any function in a point-to-point network and without any setup assumptions. This can be achieved both in the computational setting [GMW87] (assuming the existence of trapdoor permutations), and in the information-theoretic (private channel) setting [BGW88, CCD88].
2. For  $t < n/2$  (i.e., in the case of a guaranteed honest majority), secure multi-party protocols with fairness and guaranteed output delivery can be achieved for any function assuming that the parties have access to a broadcast channel. This can be achieved in the computational setting [GMW87] (with the same assumptions as above), and in the information-theoretic setting [RB89].
3. For  $t \geq n/2$  (i.e., when the number of corrupted parties is not limited), secure multi-party protocols (without fairness or guaranteed output delivery) can be achieved assuming that the parties have access to a broadcast channel and in addition assuming the existence of trapdoor permutations [Y86, GMW87, G02a]. These feasibility results hold only in the computational setting; analogous results for the information-theoretic setting cannot be obtained when  $t \geq n/2$  [BGW88].

In summary, secure multi-party protocols exist for any distributed computing task. In the computational model, this holds for all possible numbers of corrupted parties, with the qualification that when no honest majority exists, then fairness and guaranteed output delivery are not obtained. We note that the above results all hold with respect to malicious adversaries. (The status regarding adaptive versus static adversaries is more involved and is therefore omitted here.)

**Stand-alone computation versus composition.** Given such broad feasibility results, it may seem that the only challenge remaining for the field of secure multi-party computation is to come up with more efficient protocols. However, this is far from the case. Specifically, all of the above-described results have been proven in the *stand-alone* model of computation only. In this model, *a single set of parties run a single protocol in isolation*. However, modern scenarios and networks are such that many sets of parties run many protocols simultaneously. Furthermore, the fact that a protocol is secure in the stand-alone model does *not* mean that it will remain secure when run in a more general setting (i.e., under composition). It is therefore of fundamental importance to re-study the feasibility of secure multi-party computation under composition. The study of this question is the focus of this work.

## 1.2 Protocol Composition

In a stand-alone setting of computation, a single set of parties carry out a single protocol execution in isolation. That is, it is assumed that these parties are the only ones running a protocol and that they run it only once. Generally speaking, the notion of “protocol composition” refers to a setting where many protocol executions take place. As we will see, this includes many possible scenarios, ranging from the case that a single set of parties runs the same protocol many times to the case that many sets of different parties run many different protocols many times.

The standard notion of composition, and the one considered in this work, is that of *stateless composition*. This means that the honest parties relate to each execution as if it is running in isolation, and therefore obliviously of the other executions taking place. In particular, this means that honest parties are not required to coordinate between different executions or keep track of the history of past executions. (This is called stateless composition because no “joint state” is kept between the executions.)

Stateless composition is clearly desirable: the complexity of simultaneously coordinating between many executions can be high, and keeping a lot of state information can become a burden on the system. However, even if the required amount of joint state is manageably small, stateful composition may still be difficult, if not impossible, to carry out. For example, ensuring successful coordination between protocols that are designed independently of each other may be very problematic.

We note that in contrast to the honest parties, the adversary may keep joint state and coordinate its actions between the protocol executions. This asymmetry between the adversary (who is “stateful”) and the honest parties (who are “stateless”), is due to the fact that some level of coordination is clearly possible. Thus, although it is undesirable to rely on such coordination in the construction of protocols, it would be careless to assume that the adversary cannot utilize it to some extent. Furthermore, the adversary selects its strategy after all protocols are designed and can therefore implement a (joint) malicious strategy that takes all protocol executions into account.

### 1.2.1 Types of Protocol Composition

As we have described, the notion of protocol composition relates to settings in which many protocol executions take place. However, there are actually many ways to interpret this notion. We single out three different important parameters: the *context* in which the protocol runs, the *participating parties* and the *scheduling*.

1. **The context:** This refers to the question of *which protocols* are being run together in the network, or in other words, with which protocols should the protocol in question compose. There are two contexts that have been considered, defining two classes of composition:

- a) **Self composition:** A protocol is said to *self compose* if it remains secure when it alone is executed many times in a network. We stress that in this setting, there is only one protocol that is being run. As we have mentioned, the honest parties run each protocol execution obliviously of the other executions (even though the same protocol is run each time).
- b) **General composition:** In this type of composition, many different protocols are run together in the network. Furthermore, these protocols may have been designed independently of one another. A protocol is said to maintain security under *general composition* if its security is maintained even when it is run along with other arbitrary protocols.

We stress a crucial difference between self and general composition. In self composition, the protocol designer has control over everything that is being run in the network. However, in general composition, the other protocols being run may even have been designed maliciously after the secure protocol is fixed.

2. **The participating parties:** Another parameter which must be considered is whether or not the same set of parties is involved in all executions:
  - a) **A single set of parties:** In this setting, the same set of parties participates in all executions. Typically, when self composition is considered, this also means that the same party assumes the same role in each of the executions (e.g., in the case of zero-knowledge it is assumed that the same party plays the “prover” in all executions).
  - b) **Arbitrary sets of parties:** In this setting, arbitrary (and possibly intersecting) sets of parties run each protocol execution.
3. **The scheduling:** There are three main types of scheduling that appear in the literature:
  - a) **Sequential:** each new execution begins strictly after the previous one terminates. In this case, at any give time, only one protocol is running.
  - b) **Parallel:** all executions begin at the same time and proceed at the same rate (i.e., in a synchronous fashion).
  - c) **Concurrent:** the scheduling of the protocol executions, including when they start and the rate at which they proceed, is maliciously determined by the adversary. That is, the adversary has full control over when messages sent by the parties are delivered (as is typical in an asynchronous network).

Another type of scheduling, called **concurrent with timing**, lies somewhere between parallel and concurrent scheduling. Here, the adversary can determine when protocols begin but is limited in its ability to vary the rate at which the protocols proceed. (In particular, the adversary cannot delay the delivery of messages for an arbitrary long period of time.)

Another restriction on concurrent scheduling that has been considered is that of **bounded concurrency**. In the standard definition of composition,

the protocol should remain secure for any polynomial number of invocations. In contrast, bounded concurrency considers a scenario where the number of concurrent executions is fixed ahead of time. Furthermore, the protocol specification is allowed to depend on this predetermined bound.

### 1.2.2 Feasibility of Security under Composition

One of the first questions that one may ask regarding the feasibility of obtaining security under composition is whether or not it actually makes any difference. That is, is it really harder to achieve security under composition than in the stand-alone setting? In Chapter 2, based on [LLR02], we show that this is indeed the case. Specifically, we demonstrate that the (seemingly simple) problem of secure broadcast is strictly harder to achieve under composition. In fact, we show that there are certain settings (related to the setup assumption and number of corrupted parties) in which stand-alone broadcast can be achieved, and yet it is *impossible* to achieve broadcast under parallel self composition (even for a single set of parties). Furthermore, this holds even if only two parallel executions take place. Since this impossibility result is proven for a relatively weak setting of composition (specifically, parallel self composition with a single set of parties), it also holds for all stronger notions of composition (including all levels of the taxonomy up to concurrent general composition for arbitrary sets of parties). Thus security under composition can be much more problematic than in the stand-alone model. See Section 1.3.1 for more details.

We now proceed to briefly survey what is known regarding the feasibility and infeasibility of obtaining security under composition. We remark that unless specifically stated, all results mentioned refer to a setting where no trusted setup phase is assumed. We call this the *plain model*.

**Self composition for a single set of parties.** Most of the research on the security of protocols under composition has been in the context of *self composition* for a *single set of parties*. Although this seems to be a rather weak model of concurrency, many difficulties already arise in this setting. It therefore serves as a good first step in understanding this difficult problem.

The main focus of research in this setting has been for the specific problem of zero-knowledge [GMR85]. Beginning with the case of *sequential composition*, it has been shown that sequential composition does not necessarily hold for the original formulation of zero-knowledge in which uniform adversaries were considered [GoK96]. That is, there exist protocols that are zero-knowledge for a single execution, but are *not* zero-knowledge if the protocol is executed even twice sequentially. In contrast, it has been shown that any protocol that is secure under *auxiliary input* zero-knowledge (where the adversary may be a non-uniform algorithm) remains secure under sequential composition [Go94]. Thus, allowing the adversary to have auxiliary input is essential for guaranteeing that any secure zero-knowledge protocol

remains secure under sequential composition. The situation for *parallel composition* is somewhat worse. That is, security under parallel composition does not necessarily hold, even for auxiliary input zero-knowledge [Gokr96]. Nevertheless, zero-knowledge protocols that do compose in parallel have been demonstrated [Goka96, G02b]. Furthermore, obtaining security under parallel composition does not seem to require greater round complexity than for the stand-alone case (this is in contrast to concurrent composition, see below). The question related to composition that has received the most attention is that of *concurrent zero-knowledge*. This was first formally considered by [DNS98] who demonstrated the existence of protocols that remain secure under concurrency with timing. However, [DNS98] did not answer the question of whether or not there exist protocols that are concurrent zero-knowledge with respect to arbitrary, adversarial scheduling (i.e., “full” concurrency). In a breakthrough work, such protocols were shown to exist in [RK99]. Interestingly, the protocol of [RK99] required polynomially many rounds, in contrast to the stand-alone setting where zero-knowledge can be achieved in a constant number of rounds only. This sparked off a number of works attempting to characterize the round complexity of black-box<sup>1</sup> concurrent zero-knowledge [KPR98, R00, KP01, CKPR01, PRS02]. The final result is an almost matching upper and lower bound of  $\tilde{\Omega}(\log k)$  rounds (where  $k$  denotes the security parameter) for black-box concurrent zero-knowledge [CKPR01, PRS02]. Thus, at least for black-box zero-knowledge, security in the concurrent setting is *strictly harder to achieve* than in the stand-alone setting. A further breakthrough regarding concurrent zero-knowledge was demonstrated by [B01] who showed that non-black-box simulation can be used in order to obtain constant-round protocols that remain secure under *bounded* concurrency. Although this model of concurrency is weaker than the standard one, it still “breaks” the  $\tilde{\Omega}(\log k)$  lower-bound of [CKPR01], which holds also for bounded concurrency.

Another specific problem that has been studied in the context of concurrent self composition is that of oblivious transfer. In [GM00], it was shown that there exist protocols for 2-out-of-1 oblivious transfer [R81, EGL85], that remain secure under concurrent self composition for a single set of parties. However, the model of concurrency considered by [GM00] in proving the security of their protocol is such that all the inputs in all the executions are independent of each other. This model is rather limited and does not reflect the reality of concurrent computation where related inputs are often inherent. Nevertheless, we remark that even in this setting, it is unknown what can and cannot be achieved beyond what was shown in [GM00].

<sup>1</sup> In a black-box zero-knowledge protocol, the simulator is allowed only oracle access to the adversarial verifier. This is a restriction of the standard notion of zero-knowledge that allows the simulator to view the actual description of the verifier. We note that until recently, all zero-knowledge protocols were black-box.

In addition to the above work on specific protocol problems, research has also been carried out regarding the composition of protocols for *general secure multi-party computation*. (A general protocol is one that can be used for obtaining the secure computation of any function.) The security of multi-party protocols under *sequential* self composition (and arbitrary sets of parties) was considered by [B91b]. Specifically, a general definition of security was presented and it was shown that any protocol that is secure by the proposed definition remains secure under sequential self composition [B91b]. The question of *parallel* self composition for general multi-party computation has actually received no explicit attention. Nevertheless, it is not difficult to obtain general protocols that achieve parallel self composition for a single set of parties, by combining the construction of [GMW87] with zero-knowledge protocols that compose in parallel. This will be discussed in more detail in Chapter 3 (Section 3.6). Regarding *concurrent* self composition and a single set of parties, the feasibility of obtaining secure protocols in this setting was considered in [L03a]. The specific case considered by [L03a] is where no honest majority exists, and in particular, the important special case of two parties. In this setting, [L03a] showed that a general two-party protocol that remains secure for  $m$  concurrent executions and can be proven via black-box simulation, must have more than  $m$  rounds of communication. This implies that security for (unbounded) concurrency *cannot* be achieved using black-box techniques. This is in stark contrast to the case of black-box zero-knowledge where a logarithmic number of rounds suffice for any polynomial number of concurrent executions. Thus, concurrent composition for general two-party computation is strictly harder to achieve than concurrent zero-knowledge, and is also strictly harder to achieve than stand-alone two-party computation for which constant-round protocols exist [L01] (building on the work of [Y86, GMW87]). We conclude that even in this relatively weak model of concurrent composition, it is “hard” to achieve security. On the positive side, [L03a] present general two-party protocols that have  $O(m)$  rounds and remain secure for  $m$  concurrent executions. This was then improved by [PR03], who present constant round protocols that achieve security under bounded concurrency. (The protocol of [PR03] does not use black-box simulation and so this does not contradict the black-box lower bound of [L03a].) An important open question remaining in this context is the feasibility of obtaining general two-party computation that remains secure under *unbounded* concurrent self composition for a single set of parties.

**Self composition for arbitrary sets of parties.** The model studied in the work on non-malleability [DDN91] is actually a restricted version of self composition for arbitrary sets of parties. Specifically, the case that has been considered is one where only two pairs of parties co-exist, and where each pair carries out only a single execution. That is, the setting is one where *two* pairs of parties each run a single execution of a *single* protocol, *concurrently*. Despite this limitation on the concurrency, many difficulties arise when attempt-

ing to construct non-malleable protocols. Indeed, the only protocols known for the plain model are for commitment and zero-knowledge [DDN91, B02], and these constructions are themselves very involved. We note that simpler and more efficient constructions are known in the common reference string model<sup>2</sup> [DIO98, S99, FF00, DDOPS01, DKOS01].

**General composition.** A protocol that is secure under general composition can be run in a network where many different and arbitrary protocols are being executed. This scenario is typical of modern networks like the Internet. In fact, the type of composition required for obtaining security in the Internet is that of concurrent general composition for arbitrary sets of parties.

The notion of general composition was first introduced by [MR91], who considered it for the sequential scheduling case. It was then formally shown that proving security under the stand-alone definitions for secure multi-party computation suffices for obtaining security under sequential general composition [C00].<sup>3</sup> We therefore have that any multi-party functionality can be securely computed under sequential general composition, and for any number of corrupted parties [GMW87, C00].

A recent and important breakthrough in the study of the security of protocols under general composition was made with the introduction of **universal composability**<sup>4</sup> (UC) [C01]. Universal composability is a security definition for which a strong composition theorem has been proven, stating that security is preserved under concurrent general composition with arbitrary sets of parties. The UC definition follows the standard ideal/real simulation paradigm that compares a real protocol execution to an ideal execution involving an external trusted party. However, it also differs from previous definitions in a very important way. The traditional model considered for secure computation includes the parties running the protocol, plus an adversary  $\mathcal{A}$  that controls a set of corrupted parties. However, in the universally composable framework, an additional adversarial entity called the **environment**  $\mathcal{Z}$  is introduced. This environment generates the inputs to all parties, reads all outputs, and in addition interacts with the adversary in an arbitrary way throughout the computation. (As is hinted by its name,  $\mathcal{Z}$  represents the external environment that consists of arbitrary protocol executions that may be running concurrently with the given protocol.) A protocol is said to **UC realize** a given functionality  $f$  if for any real-model adversary  $\mathcal{A}$  that interacts with the protocol, there exists an ideal-model adversary  $\mathcal{A}'$ , such that *no environment*  $\mathcal{Z}$  can tell whether it is interacting with  $\mathcal{A}$  and parties running the protocol, or with  $\mathcal{A}'$  and parties that are running in the ideal model. Such a

<sup>2</sup> In the common reference string model, all parties are given access to a common reference string that is ideally chosen from some efficiently samplable distribution. Essentially, this means that a trusted setup phase is assumed.

<sup>3</sup> For sequential general composition, security for a single set of parties is equivalent to security for arbitrary sets of parties. Therefore, no distinction is made here.

<sup>4</sup> This notion has also been called **environmental security**.

protocol is called **universally composable**. (In a sense,  $\mathcal{Z}$  serves as an “interactive distinguisher” between a run of the protocol and an ideal execution involving a trusted party.) The importance of this new definition is due to a composition theorem that states that any universally composable protocol remains secure under *concurrent general composition with arbitrary sets of parties* [C01]. Therefore, universal composability provides the security guarantees needed for running protocols in modern network settings. As such, it is a very important contribution to the theory of secure computation and protocol composition.

It has been shown that in the case of an *honest majority*, universally composable protocols exist for any multi-party functionality [C01] (based on the protocols of [BGW88, RB89]). On the negative side, it has also been shown that in the two-party case (where there is no honest majority) there exist functionalities that cannot be UC realized unless some trusted setup phase is assumed.<sup>5</sup> In particular, the commitment and zero-knowledge functionalities *cannot* be UC realized by any protocol in the plain model [C01, CF01]. On the other hand, in the common reference string model, both the commitment and zero-knowledge functionalities *can* be UC realized [CF01, DN02]. The immediate question that follows from the above is what functionalities can and cannot be UC computed in the common reference string model and in the plain model.

On the positive side, in Chapter 4 (based on [CLOS02]) we show that essentially *any* multi-party functionality can be UC computed in the common reference string model, for any number of corrupted parties; see Section 1.3.3 for more details. Thus, when trusted setup can be assumed, it is possible to obtain concurrent general composition for arbitrary sets of parties, as desired. On the negative side, wide-reaching impossibility results have been shown for the case where no honest majority is present and when no trusted setup phase is assumed [CKL03]. In particular, for deterministic two-party functions where both parties receive the same output, UC secure protocols can only be achieved if the function depends on only one party’s input and is efficiently invertible (in some sense, this is just bit transmission). Thus, Yao’s famous millionaires problem (where two millionaires wish to compare who has more money) cannot be UC realized. Impossibility results for other general classes of both deterministic and probabilistic functions have also been proven; see [CKL03].

The severe impossibility results of [CKL03] were proven for the *definition* of universal composability. As we have mentioned, UC security implies security under concurrent general composition, but the reverse is not known to hold. Therefore, [CKL03] still leaves open the possibility that concurrent general composition can be achieved in the plain model, even when no honest

---

<sup>5</sup> Note that two-party computation is a special case of multi-party computation when there is no honest majority. Therefore, these results also prove that there exist *multi-party* functionalities that cannot be UC realized without an honest majority.



majority is present. Nevertheless, it was recently shown that security under concurrent general composition implies a weaker variant of universal composability [L03b]. Furthermore, [CKL03] actually show that many of their impossibility results also hold for this weaker variant. We therefore conclude that for large classes of two-party functions, *security under concurrent general composition cannot be achieved in the plain model*. To a certain extent, this means that one must either settle for weaker notions of composition (which is problematic given the reality of today’s network setting) or accept the common reference string model, in which case the construction of Chapter 4 can be used.

We remark that additional work on parallel and concurrent general composition in more limited settings appears in [DM00] which deals specifically with the information-theoretic setting and assumes ideally private channels, and in [PW00] which relates to a setting where a single protocol execution takes place concurrently to one other arbitrary protocol (that is, the “amount” of concurrency is strictly limited).

## 1.3 Our Results

This book studies the feasibility of protocols under different types of composition. The main results can be summarized as follows: First, in Chapter 2, we demonstrate that obtaining security under (even rather weak notions of) composition can be strictly harder than obtaining stand-alone security. Specifically, we show that achieving broadcast under parallel self composition is “hard”. That is, in the stand-alone setting, it is possible to achieve broadcast that tolerates any number of corrupted parties (assuming a public-key infrastructure for digital signatures). In contrast, under parallel self composition, it is impossible to achieve broadcast if a third or more of the parties are corrupted (even assuming a public-key infrastructure for digital signatures). This result has a significant impact on the composability of secure multi-party protocols because such protocols typically use a broadcast protocol extensively. However, when (at least) parallel self composition is considered and a third or more of the parties are corrupted, the required broadcast protocol cannot be obtained. Nevertheless, in Chapter 3, we show that it is possible to replace the broadcast protocol used in secure multi-party protocols with a weaker primitive that can be achieved under composition. This involves a very mild relaxation of the definition, yet removes the obstacle to obtaining protocols that remain secure under composition. Finally, in Chapter 4, we present protocols for UC realizing any multi-party functionality in the common reference string model, in the case of no honest majority. This is an important result, demonstrating that it is possible to obtain concurrent general composition without an honest majority, and using a reasonable setup assumption (namely, that of a common reference string).

Each chapter of this book is preceded by a detailed introduction. Here, we briefly state the results and explain how they relate to each other.

### 1.3.1 The Composition of Authenticated Byzantine Agreement

We begin by studying the specific and basic problem of achieving secure broadcast in a point-to-point network, where parties are connected via pairwise communication channel. This problem is known as Byzantine Agreement (or Generals) and has been the focus of much research. In a broadcast protocol, a designated party, called the broadcaster, wishes to send a message to the rest of the parties, called the recipients. Loosely speaking, a *secure* broadcast protocol ensures that all the honest recipients will receive the message that an honest broadcaster wishes to send. (I.e., the adversary cannot disrupt the communication and can also not interfere with the message being broadcast.) Furthermore, if the broadcaster itself is corrupted, then all the recipients are guaranteed to receive the same message. (I.e., a corrupted broadcaster cannot cause some recipients to receive a different message from others; something which cannot occur with a physical broadcast channel.) We note that Byzantine Agreement is a closely related problem to that of broadcast, but is actually defined differently. Nevertheless, the same results hold for both and so we abuse terminology and refer to broadcast as Byzantine Agreement for now.

It has been shown that Byzantine Agreement can be achieved in the plain model *if and only if* strictly less than  $1/3$  of the parties may be corrupted [PSL80, LSP82]. Furthermore, by augmenting the network with a public-key infrastructure for digital signatures, it is possible to obtain secure protocols for Byzantine Agreement for *any* number of corrupted parties [PSL80, LSP82]. This augmented problem is called “authenticated Byzantine Agreement” (since the digital signatures are used for “authentication”).

Despite the extensive research on this problem, the setting that has been considered for Byzantine Agreement was that of the stand-alone model only. In particular, the above-mentioned feasibility results have not been proven in the setting of composition. Due to the fact that Byzantine Agreement is a basic primitive that is used extensively, establishing its feasibility under composition is of great importance. We study this question and ask whether or not it is possible to obtain protocols for Byzantine Agreement and authenticated Byzantine Agreement that self compose in parallel or concurrently. Our results are as follows:

1. *Byzantine Agreement (in the plain model)*: It is easily shown that any protocol for Byzantine Agreement in the plain model remains secure under concurrent self composition. Thus, when less than a third of the participating parties are corrupted, Byzantine Agreement can be achieved in the setting of concurrent self composition.

2. *Authenticated Byzantine Agreement*: Somewhat surprisingly, we show that when a third or more of the parties are corrupted, authenticated Byzantine Agreement that remains secure under parallel (or concurrent) self composition *cannot* be achieved. This holds even if only two parallel executions take place. This impossibility result is unconditional and does not depend on any computational or other assumptions. It also holds for both randomized and deterministic protocols.<sup>6</sup>

Thus, the augmentation of the plain model with a public-key infrastructure for digital signatures is of *no benefit* when parallel or concurrent composition is needed. In particular, if less than  $1/3$  of the parties are corrupted, then Byzantine Agreement for the plain model can be used and security holds for concurrent self composition. On the other hand, if  $1/3$  or more of the parties are corrupted and composition is needed, then it is impossible to achieve Byzantine Agreement, even given a public-key infrastructure for digital signatures.

Conceptually, this result demonstrates that achieving security under composition (even a relatively weak type of composition such as two-execution parallel self composition) can be strictly harder than achieving security in the stand-alone model. We note that such a “separation” between stand-alone security and security under composition can be made on many levels. One type of separation would state that protocols that are stand-alone secure are not necessarily secure under composition (this is the type of separation demonstrated regarding the parallel composition of zero-knowledge protocols [Gok96]). A stronger separation may state that more computational resources are needed for achieving security under composition than in the stand-alone setting (the black-box lower bounds for concurrent zero-knowledge are actually separations of this kind; e.g., see [CKPR01]). However, the separation proven here is far stronger. It states that in the setting of composition, a certain problem (i.e., secure broadcast with  $1/3$  or more corruptions) cannot be solved by *any* protocol, whereas solutions do exist for the stand-alone case. Thus, the additional difficulty of obtaining security under composition is not only with respect to protocol design, but also with respect to what problems can and cannot be solved.

Another ramification of this result is due to the fact that authenticated Byzantine Agreement protocols are widely used. Specifically, all known protocols for secure multi-party computation, in the case that a third or more of the parties may be corrupted (e.g., [GMW87, RB89, BMR90]), rely extensively on broadcast and implement it using authenticated Byzantine Agreement. Therefore, none of these multi-party protocols compose in the point-to-point model.

---

<sup>6</sup> We also show a severe lower bound regarding the round complexity of deterministic protocols that remain secure under sequential self composition; see Section 2.4.

### 1.3.2 Secure Computation without Agreement

As we have mentioned, an immediate concern that arises out of the previous result relates to the feasibility of obtaining composition of secure multi-party protocols in a point-to-point network. This concern motivates the following question: Is it possible to obtain secure multi-party computation *without relying on a broadcast channel* (and thus without using authenticated Byzantine Agreement) when a third or more of the parties are corrupted?

We answer this question in the affirmative. Specifically, we mildly relax the definition of secure computation in the case that output delivery is not guaranteed, and show how this definition can be achieved without the use of a broadcast channel, and for any number of corrupted parties. The novelty of our definition is in decoupling the issue of *agreement* from the central security issues of privacy, correctness and independence of inputs in secure computation. The only difference between our definition and previous ones is as follows. Recall that when output delivery is not guaranteed, the adversary is allowed to disrupt the computation and have the honest parties “abort”, rather than receive their prescribed output. Previously, it was required that if one honest party aborted then all other honest parties also abort. Thus, the parties *agree* on whether or not the protocol execution terminated successfully or not. In our new definition, it is possible that some honest parties abort while others receive their correct output. Thus, there is no agreement regarding the success of the protocol execution. We stress that in all other aspects, our definition remains the same (in particular, the properties of privacy, correctness and independence of inputs are all preserved).

Having removed the need for a broadcast channel (and thus Byzantine Agreement), we show how it is possible to obtain multi-party protocols that remain secure under self composition. Specifically, we show that there exist protocols that remain secure under parallel self composition (for a single set of parties), and tolerate any number of corrupted parties. Furthermore, in the case of an honest majority (but with a third or more corruptions), we show that security under concurrent self composition can be achieved. This constitutes some progress towards obtaining security under composition. However, the case of concurrent composition when no honest majority exists is still left unresolved. This question is dealt with next.

### 1.3.3 Universally Composable Multi-party Computation

The broadest type of composition described above is that of concurrent general composition with arbitrary sets of parties. Furthermore, this type of composition really reflects the setting of today’s networks like the Internet. Therefore, the feasibility of obtaining universally composable secure multi-party computation is of great interest (recall that universal composability implies security under concurrent general composition with arbitrary sets of parties).

As we have described, broad impossibility results have been proven with respect to the feasibility of obtaining universally composable protocols in the plain model when there is no honest majority. Therefore, a natural and important question to ask is whether or not there exists a reasonable model (based on some trusted setup phase) in which it is possible to obtain general universally composable secure multi-party computation when there is *no honest majority*? An important special case of this setting is that of universally composable secure *two-party* computation (notice that in the two party case, as soon as one party is corrupted, there is no honest majority).

Recall that universally composable protocols for the *specific* two-party functionalities of commitment and zero-knowledge have been constructed in the common reference string model [CF01, DN02]. In this model, all parties are given access to some string that is ideally chosen from a given distribution. The string must be chosen in a trusted manner, and therefore this setup assumption is definitely undesirable. However, due to the impossibility results of the plain model mentioned above, some setup assumption is required, and that of a common reference string seems rather minimal. We note that there are some scenarios where the use of a common reference string is very reasonable. Take for example the case of a large organization that wishes to have its employees use secure protocols for a number of different tasks. Then, in this context, the organization itself is trusted to properly choose a common reference string. We prove the following fundamental feasibility result:

*There exist universally composable protocols in the common reference string model for securely computing any multi-party functionality and that tolerate any number of corrupted parties.*

An important corollary of this result is the feasibility of obtaining universally composable *two-party* computation for any functionality. We stress that all previous feasibility results for secure computation (in the case of no honest majority) related to the stand-alone model only. Thus, this result is the first general construction to guarantee security under *concurrent general composition*, without assuming an honest majority.

## An Apparent Contradiction

We conclude with a remark regarding the relation between the results described in Sections 1.3.1 and 1.3.3. On the one hand, in Section 1.3.1 we claim that it is impossible to construct a protocol for the broadcast functionality (i.e., Byzantine Agreement) that self-composes in parallel, when  $1/3$  or more of the parties are corrupted. (This impossibility result holds also in the common reference string model, see Section 2.2.2.) On the other hand, in Section 1.3.3 we claim that for any functionality (and, in particular, for the broadcast functionality), there exists a protocol in the common reference string model that remains secure under concurrent general composition and

for *any* number of corrupted parties. Thus, it would seem that these claims contradict each other.

However, there is actually no contradiction because the results are in different models. Specifically, the impossibility result of Section 1.3.1 is demonstrated in a model where guaranteed output delivery is required (i.e., all honest parties must receive their output and the adversary cannot carry out a denial of service attack). Indeed, secure broadcast is almost meaningless without guaranteed output delivery.

In contrast, the definition of universal composability, similarly to the relaxed definitions described in Section 1.3.2, does not guarantee output delivery and thus honest parties may abort (the outline of the definition of universal composability in Section 1.2.2 is very brief and therefore does not include this point). We stress that the proof of the impossibility result for composing authenticated Byzantine Agreement relies heavily on the requirement that parties *must* receive output; whereas the fact that output delivery is *not* required is essential for achieving universal composability in the case of no honest majority. Thus, this difference in modelling is actually central to both of the results. See the formal definitions of Byzantine Agreement and universal composability in Sections 2.2 and 4.3, respectively, for a full description of the model used in each result.<sup>7</sup>

We note that the fact that output delivery is not guaranteed for universal composability is actually due to *additional* power that is given to the adversary. That is, in the definition of universal composability, the adversary is given the power to block messages sent over the network. Such an adversarial attack could be carried out, for example, by a network router in the Internet. In the face of such adversarial power, output delivery can never be guaranteed, because the adversary can always just cut off all the communication.

## Organization

Although the results presented in this book are all on the subject of the composition of secure protocols, each result can be viewed in its own right. Therefore, despite the risk of a small amount of repetition, each chapter is presented in a self-contained manner, enabling a reader to focus on one chapter without having to read the others.

---

<sup>7</sup> There is another technical difference between the models in that the current formalization of universal composability assumes that each execution is assigned a unique session identifier. In such a case, it is actually possible to achieve authenticated Byzantine Agreement that self-composes concurrently, for any number of corrupted parties; see Section 2.5. Nevertheless, we believe that the feasibility result for universal composability holds even when unique session identifiers are not assumed.

## 2. The Composition of Authenticated Byzantine Agreement

In this chapter we show that the basic problem of achieving secure broadcast in a distributed network is strictly harder when composition is required. In particular, assuming a public-key infrastructure for digital signatures, it is known that secure broadcast can be achieved for *any* number of corrupted parties in the stand-alone model. In this chapter, we show that an analogous result that composes cannot be achieved. That is, when a third or more of the parties may be corrupt, it is *impossible* to obtain secure broadcast that remains secure under parallel self composition, (even given a public-key infrastructure). Notice that the type of composition considered here is rather weak. That is, impossibility occurs even in the case of parallel self composition. We also present lower bounds for the sequential self composition of deterministic protocols (the impossibility result for the parallel case holds even for randomized protocols).

### 2.1 Introduction

The Byzantine Generals (Byzantine Agreement<sup>1</sup>) problem is one of the most researched areas in distributed computing. Numerous variations of the problem have been considered under different communication models, and both positive results (i.e., protocols) and negative results (i.e., lower bounds on efficiency and fault tolerance) have been established. The reason for this vast interest is the fact that the Byzantine Generals problem is the algorithmic implementation of a broadcast channel within a point-to-point network. In addition to its importance as a primitive in its own right, broadcast is a key tool in the design of secure protocols for multi-party computation.

The problem of Byzantine Generals is (informally) defined as follows. The setting is that of  $n$  parties connected via a point-to-point network, where one party is designated as the General (or dealer) who holds an input message  $x$ . In addition, there is an adversary who controls up to  $t$  of the parties and can arbitrarily deviate from the designated protocol specification. The aim of the protocol is to securely simulate a broadcast channel. Thus, first and foremost, all (honest) parties must receive the same message. Furthermore,

---

<sup>1</sup> These two problems are essentially equivalent.

if the General is honest, then the message received by the honest parties must be  $x$  (i.e., the adversary is unable to prevent an honest General from successfully broadcasting its given input message).

Pease et al. [PSL80, LSP82] provided a solution to the Byzantine Generals problem in the plain model, i.e. the information-theoretic model with point-to-point communication lines (and no setup assumptions). For their solution, the number of corrupted parties,  $t$ , must be less than  $n/3$ . Furthermore, they complemented this result by showing that the requirement of  $t < n/3$  is in fact inherent. That is, no protocol which solves the Byzantine Generals problem in the plain model can tolerate a third or more corrupted parties.

The above bound on the number of corrupted parties in the plain model is a severe limitation. It is therefore of great importance to find a different (and realistic) model in which it is possible to achieve higher fault tolerance. One possibility involves augmenting the plain model so that messages that are sent can be authenticated. By authentication, we mean the ability to verify that a message originated from a certain party, even when shown the message by a different party. This can be achieved using a trusted preprocessing phase in which a public-key infrastructure for digital signatures (e.g. [RSA78, GMR88]) is set up. (We note that this requires that the adversary be computationally bounded. However, there exist preprocessing phases which do not require any computational assumptions; see [PW96].) Indeed, Pease et al. [PSL80, LSP82] use such an augmentation and obtain a protocol for the Byzantine Generals problem which can tolerate *any* number of corrupted parties (this is very dramatic considering the limitation to  $1/3$  corrupt in the plain model). The Byzantine Generals problem in this model is called *authenticated* Byzantine Generals. We often informally refer to this model as the “authenticated model”.

A common use of Byzantine Generals is to substitute a broadcast channel in multi-party protocols. As such, it is likely to be executed many times. The question of whether these protocols remain secure when executed concurrently, in parallel or sequentially is thus an important one. However, existing work on this problem (in both the standard and authenticated models) focused on the security and correctness of protocols in the stand-alone model only.

It is not difficult to show that the “unauthenticated” protocol of Pease et al. [PSL80], and in fact all protocols in the plain model, do compose concurrently (and hence in parallel and sequentially). However, this is not the case with respect to *authenticated* Byzantine Generals. The first to notice that composition in this model is problematic were Gong et al. [GLR95], who also suggest methods for overcoming the problem. Our work shows that these suggestions and any others are in fact futile because composition in this model is impossible (as long as  $1/3$  or more of the parties may be corrupted). (We note that by composition, we actually refer to stateless composition; see Sections 1.2 and 2.2.3 for a formal discussion.)



**The main impossibility result – parallel composition.** The main result of this chapter is a proof that authenticated Byzantine Generals protocols, both deterministic and randomized, cannot be self composed in parallel (and thus concurrently). This is a powerful statement with respect to the value of enhancing the plain model by the addition of authentication. Indeed, it shows that despite popular belief, this enhancement *does not* provide the ability to improve fault tolerance when composition is required. That is, if there is a need for parallel composition, then the number of corrupted players cannot be  $n/3$  or more, and hence the authenticated model provides no advantage over the plain model. This result is summarized in the following theorem:

**Theorem 2.1.1** *No protocol for authenticated Byzantine Generals or Agreement that remains secure under parallel self composition (for even just two executions) can tolerate  $n/3$  or more corrupted parties.*

**Sequential composition – deterministic protocols.** Theorem 2.1.1 relates to the feasibility of parallel (and therefore concurrent) composition. However, sequential composition is also an important concern. As we have mentioned in Section 1.2, it has been proven that the standard definitions for secure multi-party computation imply security under sequential composition [C00]. However, this theorem only holds when no joint state between protocols is used. The theorem therefore does not necessarily hold for authenticated Byzantine Agreement, where the parties use the same public-key infrastructure in all executions.<sup>2</sup> Indeed, it turns out that the known protocols for authenticated Byzantine Agreement [PSL80, LSP82] do *not* remain secure under sequential composition. (It is easy to construct real attacks for these protocols.)

In Section 2.4, we show that this is not a particular problem with the protocols of [PSL80, LSP82]. Rather, lower bounds actually hold for all *deterministic* protocols. In particular, any deterministic protocol for authenticated Byzantine Agreement that runs for  $r$  rounds and tolerates  $t \geq n/3$  corrupted parties, can be composed sequentially at most  $2r - 1$  times. That is,

**Theorem 2.1.2** *Let  $\Pi$  be a deterministic protocol for authenticated Byzantine Agreement that terminates within  $r$  rounds of communication and remains secure under sequential self composition for  $2r$  or more executions. Then  $\Pi$  can tolerate at most  $t < n/3$  corrupted parties.*

We note that this lower bound can be overcome by using randomization. That is, randomized protocols that can be composed sequentially any polynomial number of times are known for the case that  $t < n/2$  (notice that Theorem 2.1.2 rules out such protocols for the range  $n/3 \leq t < n/2$ ). Such

<sup>2</sup> Interestingly, the existence of the public-key infrastructure is what enables higher fault tolerance in the stand-alone setting. However, it is also the source of difficulty under composition.

protocols are also known for the case that  $t \geq n/2$ ; however, these protocols have message complexity that is exponential in the number of parties. Therefore, they are only efficient when the number of parties is  $O(\log n)$ . We refer the interested reader to [LLR02] for further details.

**On the use of unique session identifiers.** As will be apparent from the proof of Theorem 2.1.1, the obstacle to achieving agreement in this setting is the fact that honest parties cannot tell in which execution of the protocol a given message was authenticated. This allows the adversary to “borrow” messages from one execution to another, and by that attack the system. In Section 2.5, we show that if we further augment the authenticated model so that unique and common indices are assigned to each execution, then security under many concurrent executions can be achieved for any number of corrupted parties.

Thus, on the one hand, our results strengthen the common belief that session identifiers are necessary for achieving authenticated Byzantine Generals. On the other hand, we show that such identifiers *cannot be generated within the system*. Typical suggestions for generating session identifiers in practice include having the General choose one, or having the parties exchange random strings and then set the identifier to be the concatenation of all these strings. However, Theorem 2.1.1 rules out all such solutions. Rather, one must assume the existence of some *trusted external means* for coming up with unique and common indices. This seems to be a very difficult, if not impossible, assumption to realize.

**Implications for secure multi-party computation.** As we have stated above, one important use for Byzantine Generals protocols is to substitute the broadcast channel in a multi-party protocol. In fact, all known solutions for general multi-party computation assume a broadcast channel. The implicit claim in all these works is that this broadcast channel can be substituted by a Byzantine Generals protocol without any complications. However, Theorem 2.1.1 shows that the use of authenticated Byzantine Generals in such a way prevents the composition of the larger protocol (even if this protocol does compose when using a physical broadcast channel).

Another important implication of Theorem 2.1.1 is due to the fact that the definition of general secure multi-party computation in the case of an honest majority, implies a solution to the Byzantine Generals problem. Therefore, any secure protocol for solving general multi-party tasks can be used to solve Byzantine Generals. This means that none of these protocols can be composed in parallel or concurrently, unless more than  $2/3$  of the parties are honest or a physical broadcast channel is available. This issue is dealt with in depth in Chapter 3.

**A comparison to zero-knowledge.** It is instructive to compare this result to the work of Goldreich and Krawczyk [Gokr96] on zero-knowledge. They show that there exist protocols that are zero-knowledge when executed stand-

alone, and yet do not remain zero-knowledge when composed in parallel (even twice). Thus, they show that zero-knowledge does not *necessarily* compose in parallel. Completing this picture, we note that there exist zero-knowledge protocols that do compose in parallel (for example, see [Goka96, G02b]). In contrast, we show that it is impossible to obtain *any protocol* for Byzantine Agreement that will compose in parallel (even twice).

## 2.2 Definitions

### 2.2.1 Computational Model

We consider a setting involving  $n$  parties,  $P_1, \dots, P_n$ , that interact in a *synchronous* point-to-point network. In such a network, each pair of parties is directly connected, and it is assumed that the adversary cannot modify messages sent between honest parties. Each party is formally modelled by an interactive Turing machine with  $n - 1$  pairs of communication tapes. The communication of the network proceeds in synchronized rounds, where each round consists of a *send* phase followed by a *receive* phase. In the *send* phase of each round, the parties write messages onto their output tapes, and in the *receive* phase, the parties read the contents of their input tapes. We note that since we prove an *impossibility* result in this chapter, the result is only strengthened by considering synchronous networks. (Specifically, since synchronous networks are a special case of asynchronous ones, impossibility is automatically derived in the asynchronous setting as well.)

This chapter refers to the *authenticated* model, where some type of *trusted preprocessing phase* is assumed. This is modelled by all parties also having an additional **setup-tape** that is generated during the preprocessing phase. Typically, in such a preprocessing phase, a public-key infrastructure of signature keys is generated. That is, each party receives its own secret signing key, and in addition, public verification keys associated with all other parties. (This enables parties to use the signature scheme to authenticate messages that they receive, and is thus the source of the name “authenticated”.) However, we stress that our lower bound holds for all preprocessing phases, even those that cannot be efficiently generated; see Definition 2.2.2.

In this model, a  $t$ -adversary is a party that controls up to  $t < n$  of the parties  $P_1, \dots, P_n$  (these parties are said to be corrupted). The adversary receives the corrupted parties’ views and determines the messages that they send. These messages need not be according to the protocol execution, but rather can be computed by the adversary as an arbitrary function of its view. We consider static adversaries for whom the set of corrupted parties is fixed before the execution begins. (Again, by taking a weaker adversary, we strengthen our impossibility result.) Finally, our impossibility results hold for adversaries (and honest parties) whose running time is of any complexity. In

fact, our lower bound holds as long as the adversary is allowed to be of the same complexity as the honest parties.

### 2.2.2 Byzantine Generals/Agreement

The existing literature defines two related problems: Byzantine Generals and Byzantine Agreement. In the first problem, there is one designated party, the General or dealer, who wishes to broadcast its value to all the other parties. In the second problem, each party has an input and the parties wish to agree on a value, with a validity condition that if a majority of honest parties begin with the same value, then they must terminate with that value. The Byzantine Generals problem can be used to solve the Byzantine Agreement problem. Therefore, it suffices to prove impossibility for Byzantine Agreement, and impossibility for Byzantine Generals is automatically derived.<sup>3</sup> We relax the standard requirements on protocols for Byzantine Agreement in that we allow a protocol to fail with probability that is negligible in some security parameter. This relaxation is needed for the case of authenticated Byzantine protocols where signature schemes are used (and can always be forged with some negligible probability). We present the formal definition for the Byzantine Agreement problem only, and all references from here on will be with respect to this problem. Below,  $[n]$  denotes the set of indices  $\{1, \dots, n\}$ .

**Definition 2.2.1** (Byzantine Agreement): *Let  $P_1, \dots, P_n$  be  $n$  parties with associated inputs  $x_1, \dots, x_n$  and let  $\mathcal{A}$  be an adversary. Furthermore, let  $I \subseteq [n]$  ( $|I| \leq t$ ) be a fixed index set such that for every  $i \in I$ , the adversary  $\mathcal{A}$  controls the party  $P_i$ . Then, a protocol solves the Byzantine Agreement problem, tolerating  $t$  corruptions, if for any adversary  $\mathcal{A}$  and any  $I$  such that  $|I| \leq t$ , the following two properties hold (except with negligible probability):*

1. Agreement: *All honest parties output the same value.*
2. Validity: *If more than  $n/2$  honest parties have the same input value  $x$ , then all honest parties output  $x$ .*

We note that for cases of  $t < n/2$  (e.g., for  $t < n/3$ ), the validity requirement is sometimes stated so that it must hold only when all honest parties have the same input. Our impossibility result holds in this case as well.

**Authenticated Byzantine Agreement.** In the model for authenticated Byzantine Agreement, some trusted preprocessing phase is run before any executions begin. In this phase, a trusted party distributes keys to every participating party. Formally,

<sup>3</sup> In the case of an honest majority, Byzantine Agreement can also be used to solve Byzantine Generals. However, this is not known to be the case when there is no honest majority.

**Definition 2.2.2** (authenticated Byzantine Agreement): *A protocol for authenticated Byzantine Agreement is a Byzantine Agreement protocol with the following augmentation:*

- *Each party has an additional setup-tape.*
- *Prior to any protocol execution, a (trusted) party chooses a series of strings  $s_1, \dots, s_n$  according to some distribution, and sets party  $P_i$ 's setup-tape to equal  $s_i$  (for every  $i = 1, \dots, n$ ).*

*Following the above preprocessing stage, the protocol is run in the standard communication model for Byzantine Agreement protocols.*

As we have mentioned, a natural example of such a preprocessing phase is one where the strings  $s_1, \dots, s_n$  constitute a public-key infrastructure for digital signatures. That is, the trusted party chooses verification and signing key-pairs  $(vk_1, sk_1), \dots, (vk_n, sk_n)$  from a secure signature scheme, and sets the contents of party  $P_i$ 's tape to equal  $s_i = (vk_1, \dots, vk_{i-1}, sk_i, vk_{i+1}, \dots, vk_n)$ . In other words, all parties are given their own signing key and the verification keys of all the other parties. We note that this preprocessing phase can also be used to setup a *common reference string* to be accessed by all parties (in this case, all the  $s_i$ 's are set to the same reference string).

We remark that the above-defined preprocessing phase is very strong. First, it is assumed that it is run completely by a trusted party. Furthermore, there is no computational bound on the power of the trusted party generating the keys. Nevertheless, our impossibility results hold even for such a preprocessing phase.

### 2.2.3 Composition of Protocols

This chapter deals with the security of authenticated Byzantine Agreement protocols under composition. As we have described in Section 1.2, the type of composition considered here is *stateless composition*. In this type of composition, the parties run identical copies of the protocol in each execution. This implies, amongst other things, that there is no information that is externally provided to the parties and is unique to every execution (like a common session identifier). (See Section 2.1 for a discussion on session identifiers and their role.) Formally, composition is captured by the following process:

**Definition 2.2.3** (sequential and parallel composition): *Let  $P_1, \dots, P_n$  be parties for an authenticated Byzantine Agreement protocol  $\Pi$ . Let  $I \subseteq [n]$  ( $|I| \leq t$ ) be a fixed index set such that for every  $i \in I$ , the adversary  $\mathcal{A}$  controls the party  $P_i$ . Then, the sequential (resp., parallel) composition of  $\Pi$  involves the following process:*

- *Run the preprocessing phase associated with  $\Pi$  and obtain the strings  $s_1, \dots, s_n$ . Then, for every  $j$ , set the setup-tape of  $P_j$  to equal  $s_j$ .*

- Repeat the following process a polynomial number of times sequentially (resp., in parallel):
  1. The adversary  $\mathcal{A}$  chooses an input vector  $x_1, \dots, x_n$ .
  2. Fix the input tape of every honest  $P_j$  to be  $x_j$  and the random-tape to be a uniformly (and independently) chosen random string.
  3. Invoke all parties for an execution of  $\Pi$  (using the strings generated in the preprocessing phase above). The execution is such that for  $i \in I$ , the messages sent by party  $P_i$  are determined by  $\mathcal{A}$  (who also sees  $P_i$ 's view). On the other hand, all other parties follow the instructions as defined in  $\Pi$ .

We stress that the preprocessing phase is executed only once, and all executions use the strings distributed in this phase. Furthermore, we note that Definition 2.2.3 implies that all honest parties are oblivious of the other executions that have taken place (or that are taking place in parallel). This is implicit in the fact that in each execution the parties are invoked with no additional state information, beyond the contents of their input, random and setup tapes (i.e., the composition is *stateless*). In contrast, the adversary  $\mathcal{A}$  can coordinate between the executions, and its view at any given time includes all the messages received in all the executions.<sup>4</sup>

Concurrent composition is defined analogously, with the only difference being that the adversary can determine the scheduling of the executions (i.e., when they begin and at what rate they proceed). This is standard for a completely asynchronous network. Before continuing, we show that any Byzantine Agreement protocol in the plain model composes concurrently.

**Proposition 2.2.4** *Let  $\Pi$  be a protocol that solves the Byzantine Agreement problem in the plain model and tolerates  $t$  corruptions. Then,  $\Pi$  solves the Byzantine Agreement problem under concurrent self composition, and tolerates  $t$  corruptions.*<sup>5</sup>

**Proof (sketch).** We reduce the security of  $\Pi$  under concurrent composition to its security for a single execution. Assume by contradiction that there exists an adversary  $\mathcal{A}$  who runs  $N$  concurrent executions of  $\Pi$ , such that with non-negligible probability, in one of the executions the outputs of the parties do not meet the requirement on a Byzantine Agreement protocol. Then, we construct an adversary  $\mathcal{A}'$  who internally incorporates  $\mathcal{A}$  and attacks a single execution of  $\Pi$ . Intuitively,  $\mathcal{A}'$  simulates all executions apart from the one in which  $\mathcal{A}$  succeeds in its attack. Formally,  $\mathcal{A}'$  begins by choosing an index  $i \in_R \{1, \dots, N\}$ . Then, for all but the  $i^{\text{th}}$  execution of the protocol,  $\mathcal{A}'$

<sup>4</sup> The analogous definition for the composition of *unauthenticated* Byzantine Agreement is derived from Definition 2.2.3 by removing the reference to the preprocessing stage and setup-tapes.

<sup>5</sup> It can actually be shown that  $\Pi$  remains secure even under concurrent *general* composition.

plays the roles of the honest parties in an interaction with  $\mathcal{A}$  (this simulation is internal to  $\mathcal{A}'$ ). However, for the  $i^{\text{th}}$  execution,  $\mathcal{A}'$  externally interacts with the honest parties and passes messages between them and  $\mathcal{A}$ . The key point to notice is that the honest parties hold no secret information and run each execution independently of the others. Therefore, the simulation of the concurrent setting by  $\mathcal{A}'$  for  $\mathcal{A}$  is *perfect*. Thus, with probability  $1/N$ , the  $i^{\text{th}}$  execution is the one in which  $\mathcal{A}$  succeeds. However, this means that  $\mathcal{A}'$  succeeds in breaking the protocol for a single execution with success probability that equals  $1/N$  times the success probability of  $\mathcal{A}$ . This contradicts the stand-alone security of  $\Pi$ . ■

## 2.3 Impossibility for Parallel Composition

In this section we show that it is impossible to construct an authenticated Byzantine Agreement protocol that self composes in parallel (or concurrently), and is secure when  $n/3$  or more parties are corrupt. This result is analogous to the [PSL80, LSP82] lower bounds for Byzantine Agreement in the plain model (i.e., without authentication). We stress that our result does not merely show that authenticated Byzantine Agreement protocols do not necessarily compose; rather, we show that one *cannot* construct protocols that will compose.

**Intuition.** Let us first provide some intuition into why the added power of the preprocessing step in authenticated Byzantine Agreement does not help when composition is required. (Recall that on the one hand in the stand-alone setting, there exist authenticated Byzantine Agreement protocols that tolerate any number of corrupted parties. On the other hand, under parallel composition, only  $t < n/3$  corruptions can be tolerated.) An instructive step is to first see how authenticated Byzantine Agreement protocols typically utilize authentication (i.e., digital signatures) in order to increase fault tolerance. Consider three parties  $A$ ,  $B$  and  $C$  participating in a standard (unauthenticated) Byzantine Agreement protocol. Furthermore, assume that during the execution  $A$  claims to  $B$  that  $C$  sent it some message  $x$ . Then,  $B$  cannot differentiate between the case that  $C$  actually sent  $x$  to  $A$ , and the case that  $C$  did not send this value and  $A$  is lying. Thus,  $B$  cannot be sure that  $A$  really received  $x$  from  $C$ . Indeed, such a model has been called the “oral message” model, in contrast to the “signed message” model of authenticated Byzantine Agreement [LSP82]. The use of signature schemes helps to overcome this exact problem: If  $C$  had *signed* on the message  $x$  and sent this signature to  $A$ , then  $A$  could forward the signature to  $B$ . Since  $A$  cannot forge  $C$ ’s signature, this would then constitute a proof that  $C$  had indeed sent  $x$  to  $A$ . Utilizing the unforgeability property of signatures, it is thus possible to achieve Byzantine Agreement for any number of corrupted parties.

However, the above intuition holds only in a setting where a single execution of the agreement protocol takes place. Specifically, if a number of executions were to take place, then  $A$  may send  $B$  a value  $x$  along with  $C$ 's signature on  $x$ , yet  $B$  would still not know whether  $C$  signed on  $x$  in this execution, or in a different (concurrent or parallel) execution. Thus, the mere fact that  $A$  produces  $C$ 's signature on a value does not provide proof that  $C$  signed *this* value in *this* execution. As we will see in the proof, this is enough to render the public-key infrastructure useless under parallel composition.

**Theorem 2.3.1** (Theorem 2.1.1 – restated): *No protocol for authenticated Byzantine Agreement that remains secure under parallel self composition (for even just two executions) can tolerate  $n/3$  or more corrupted parties.*

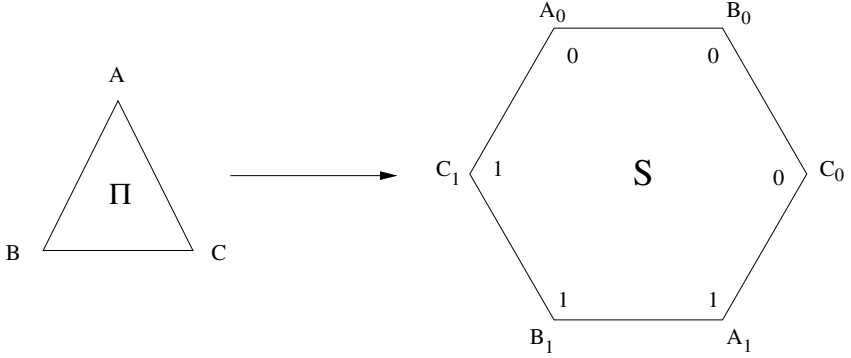
**Proof.** Our proof of Theorem 2.1.1 uses ideas from the proof by Fischer et al. [FLM86] that no unauthenticated Byzantine Agreement protocol can tolerate  $n/3$  or more corrupted parties. We begin by proving the following lemma:

**Lemma 2.3.2** *There exists no protocol for authenticated Byzantine Agreement for three parties, that remains secure under parallel self composition (for even just two executions) and can tolerate one corrupted party.*

**Proof.** Assume, by contradiction, that there exists a protocol  $\Pi$  that solves the Byzantine Agreement problem for three parties  $A$ ,  $B$  and  $C$ , where one may be corrupt. Furthermore,  $\Pi$  remains secure even when composed in parallel twice. Exactly as in the proof of Fischer et al. [FLM86], we define a hexagonal system  $S$  that intertwines two independent copies of  $\Pi$ . That is, let  $A_0, B_0, C_0$  and  $A_1, B_1$  and  $C_1$  be independent copies of the three parties participating in  $\Pi$ . By independent copies, we mean that  $A_0$  and  $A_1$  are the same party  $A$  with the same key tape, that runs in two different parallel executions of  $\Pi$ , as defined in Definition 2.2.3. The system  $S$  is defined by connecting party  $A_0$  to  $C_1$  and  $B_0$  (rather than to  $C_0$  and  $B_0$ ); party  $B_0$  to  $A_0$  and  $C_0$ ; party  $C_0$  to  $B_0$  and  $A_1$ ; and so on, as in Figure 2.1.

In the system  $S$ , parties  $A_0, B_0$ , and  $C_0$  have input 0; while parties  $A_1, B_1$  and  $C_1$  have input 1. Note that within  $S$ , all parties follow the instructions of  $\Pi$  exactly. We stress that  $S$  is *not* a Byzantine Agreement setting (where the parties are joined in a complete graph on three nodes), and therefore the definitions of Byzantine Agreement tell us nothing directly of what the parties' outputs should be. However,  $S$  is a well-defined system and this implies that the parties have well-defined output distributions. The proof proceeds by showing that if  $\Pi$  is a correct Byzantine Agreement protocol, then we arrive at a contradiction regarding the output distribution in  $S$ . We begin by showing that  $B_0$  and  $C_0$  always output 0 in  $S$ . We denote by  $\text{rounds}(\Pi)$  the upper bound on the number of rounds of  $\Pi$  (when run in a Byzantine Agreement setting).





**Fig. 2.1.** Combining two copies of  $\Pi$  in a hexagonal system  $S$ .

**Claim 2.3.3** *Except with negligible probability, parties  $B_0$  and  $C_0$  halt within  $\text{rounds}(\Pi)$  steps and output 0 in the system  $S$ .*

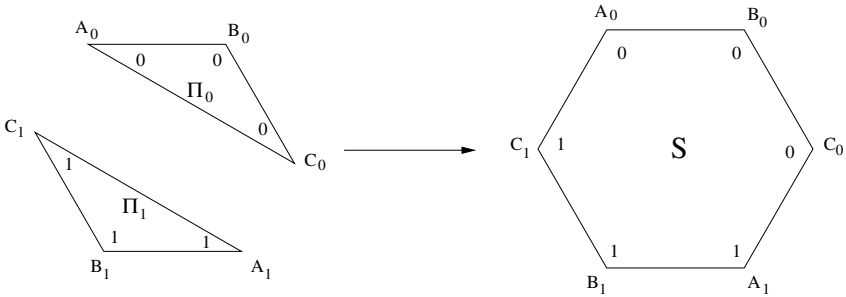
**Proof.** We prove this claim by showing that there exists an adversary  $\mathcal{A}$  controlling  $A_0$  and  $A_1$  who participates in two parallel copies of  $\Pi$  and simulates the system  $S$ , with respect to  $B_0$  and  $C_0$ 's view. The adversary  $\mathcal{A}$  (and the other honest parties participating in the parallel executions) work within a Byzantine Agreement setting where there are well-defined requirements on their output distribution. Therefore, by analyzing their output in this parallel execution setting, we are able to make claims regarding their output in the system  $S$ .

Before formally proving the above, we introduce some terminology and notation. A **system**  $X$  is defined by a set of parties along with protocol instructions for these parties, an adversary along with a set of corrupted parties, and a set of inputs for all parties. In addition, part of the system definition includes the network structure by which the parties are connected. Let  $X$  be a system and let  $P$  be a party in  $X$ . Then,  $\text{view}_X(P)$  denotes the **view** of party  $P$  in  $X$ ; this view contains the contents of  $P$ 's input and random tapes along with the series of messages received during the execution. If the parties within the system are deterministic (including the adversary), then  $\text{view}_X(P)$  is a single value (since all messages are predetermined by the parties' strategies and their inputs). If the parties are probabilistic, then  $\text{view}_X(P)$  is a random variable assuming values over  $P$ 's view, when all parties' random tapes are chosen uniformly at random.

We define two different systems and prove some properties of these systems. First, let  $S$  be the above-defined hexagonal system involving parties  $A_0, A_1, B_0, B_1, C_0$  and  $C_1$ . As described  $A_0, B_0, C_0$  all have input 0 and  $A_1, B_1, C_1$  all have input 1. The parties instructions are to all honestly follow the protocol instructions of  $\Pi$  and the network structure is as shown in Figure 2.1. (In this system, there is no adversary; formally, the set of corrupted inputs is empty.)

Next, we define a system 2BA which is made up of two parallel executions of  $\Pi$  (thus defining the network structure); denote these two executions by  $\Pi_0$  and  $\Pi_1$ . In the first execution (i.e., in  $\Pi_0$ ), parties  $A_0, B_0$  and  $C_0$  participate, each with input 0; in the second execution (i.e., in  $\Pi_1$ ), parties  $A_1, B_1$  and  $C_1$  participate, each with input 1. Recall that  $A_0$  and  $A_1$  are independent copies of the party  $A$  with the same key tape (as in Definition 2.2.3); likewise for  $B_0, B_1$  and  $C_0, C_1$ . Uncorrupted parties in the system all honestly follow the instructions of  $\Pi$ .

In order to complete the description of system 2BA, it remains to describe the adversary and the set of corrupted parties. Let  $\mathcal{A}$  be an adversary and let the set of corrupted parties be  $A_0$  and  $A_1$ . Intuitively, party  $\mathcal{A}$ 's strategy is to maliciously generate an execution in which  $B_0$ 's and  $C_0$ 's view in 2BA is *identical* to their view in  $S$ . That is, we will construct  $\mathcal{A}$  such that  $\text{view}_{2\text{BA}}(B_0)$  is distributed exactly according to  $\text{view}_S(B_0)$ ; likewise,  $\text{view}_{2\text{BA}}(C_0)$  will have the same distribution as  $\text{view}_S(C_0)$ . (We remark that if  $\Pi$  is deterministic, then we will have that  $\text{view}_{2\text{BA}}(B_0) = \text{view}_S(B_0)$  and  $\text{view}_{2\text{BA}}(C_0) = \text{view}_S(C_0)$ .) We now describe how  $\mathcal{A}$  achieves this:  $\mathcal{A}$  works by redirecting edges in the two parallel triangles (representing the parallel execution), so that the overall system has the same behavior as  $S$ ; see Figure 2.2.



**Fig. 2.2.** Redirecting edges of  $\Pi_0$  and  $\Pi_1$  to make a hexagon.

Specifically, in 2BA the  $(A_0, C_0)$  and  $(A_1, C_1)$  edges of  $\Pi_0$  and  $\Pi_1$  respectively are removed, and the  $(A_0, C_1)$  and  $(A_1, C_0)$  edges of  $S$  are added in their place.  $\mathcal{A}$  is able to make such a modification because it only involves redirecting messages to and from parties that it controls (i.e.,  $A_0$  and  $A_1$ ). (Recall that the corrupted party can coordinate between the different executions.) We now formally describe how  $\mathcal{A}$  works (in the description below,  $\text{msg}_i(A_0, B_0)$  denotes the message sent from  $A_0$  to  $B_0$  in the  $i^{\text{th}}$  round of the 2BA execution):

$\mathcal{A}$  invokes parties  $A_0$  and  $A_1$ , upon inputs 0 and 1 respectively. We stress that  $A_0$  and  $A_1$  follow the instructions of protocol  $\Pi$  exactly. However,  $\mathcal{A}$  provides them with their incoming messages and sends their outgoing mes-

sages for them. The only malicious behavior of  $\mathcal{A}$  is in the redirection of messages to and from  $A_0$  and  $A_1$ . A full description of  $\mathcal{A}$ 's code is as follows (we recommend the reader to refer to Figure 2.2 in order to clarify the following):

1. *Send outgoing messages of round  $i$ :*  $\mathcal{A}$  obtains messages  $\text{msg}_i(A_0, B_0)$  and  $\text{msg}_i(A_0, C_0)$  from  $A_0$  in execution  $\Pi_0$ , and messages  $\text{msg}_i(A_1, B_1)$  and  $\text{msg}_i(A_1, C_1)$  from  $A_1$  in  $\Pi_1$  (these are the round  $i$  messages sent by  $A_0$  and  $A_1$  to the other parties in system 2BA; as we have mentioned,  $A_0$  and  $A_1$  compute these messages according to the protocol definition and based on their view).
  - In  $\Pi_0$ ,  $\mathcal{A}$  sends  $B_0$  the message  $\text{msg}_i(A_0, B_0)$  and sends  $C_0$  the message  $\text{msg}_i(A_1, C_1)$  (and thus the  $(A_1, C_1)$  directed edge is replaced by the directed edge  $(A_1, C_0)$ ).
  - In  $\Pi_1$ ,  $\mathcal{A}$  sends  $B_1$  the message  $\text{msg}_i(A_1, B_1)$  and sends  $C_1$  the message  $\text{msg}_i(A_0, C_0)$  (and thus the  $(A_0, C_0)$  directed edge is replaced by the directed edge  $(A_0, C_1)$ ).
2. *Obtain incoming messages from round  $i$ :*  $\mathcal{A}$  receives messages  $\text{msg}_i(B_0, A_0)$  and  $\text{msg}_i(C_0, A_0)$  from  $B_0$  and  $C_0$  in round  $i$  of  $\Pi_0$ , and messages  $\text{msg}_i(B_1, A_1)$  and  $\text{msg}_i(C_1, A_1)$  from  $B_1$  and  $C_1$  in round  $i$  of  $\Pi_1$ .
  - $\mathcal{A}$  passes  $A_0$  in  $\Pi_0$  the messages  $\text{msg}_i(B_0, A_0)$  and  $\text{msg}_i(C_1, A_1)$  (and thus the  $(C_1, A_1)$  directed edge is replaced by the directed edge  $(C_1, A_0)$ ).
  - $\mathcal{A}$  passes  $A_1$  in  $\Pi_1$  the messages  $\text{msg}_i(B_1, A_1)$  and  $\text{msg}_i(C_0, A_0)$  (and thus the  $(C_0, A_0)$  directed edge is replaced by the directed edge  $(C_0, A_1)$ ).

This completes the description of  $\mathcal{A}$ , and thus the definition of the system 2BA.

We now claim that  $\text{view}_{2\text{BA}}(B_0)$  and  $\text{view}_{2\text{BA}}(C_0)$  are distributed exactly like  $\text{view}_S(B_0)$  and  $\text{view}_S(C_0)$ .<sup>6</sup> This holds because in 2BA, all parties follow the protocol definition (including  $A_0$  and  $A_1$ ). The same is true in the system  $S$ , except that party  $A_0$  is connected to  $B_0$  and  $C_1$  instead of to  $B_0$  and  $C_0$ . Likewise,  $A_1$  is connected to  $B_1$  and  $C_0$  instead of to  $B_1$  and  $C_1$ . However, by the definition of  $\mathcal{A}$ , the messages received by all the parties in 2BA are exactly the same as the messages received by the parties in  $S$  (e.g., the messages seen by  $C_0$  in 2BA are those sent by  $B_0$  and  $A_1$ , exactly as in  $S$ ). Therefore, the views of  $B_0$  and  $C_0$  in the parallel execution maliciously controlled by  $\mathcal{A}$ , are *identically distributed* to their views in  $S$ .<sup>7</sup>

<sup>6</sup> In fact, the views of *all* the honest parties in 2BA with  $\mathcal{A}$  are identical to their views in the system  $S$ . However, in order to obtain Claim 2.3.3, we need only analyze the views of  $B_0$  and  $C_0$ .

<sup>7</sup> We note a crucial difference between this proof and that of Fischer et al. [FLM86]. In [FLM86], the corrupted party  $\mathcal{A}$  is able to simulate the entire  $A_0$ – $C_1$ – $B_1$ – $A_1$

By the assumption that  $\Pi$  is a correct Byzantine Agreement protocol that composes twice in parallel, we have that in execution  $\Pi_0$  of 2BA, both  $B_0$  and  $C_0$  halt within  $\text{rounds}(\Pi)$  steps and output 0 (except with negligible probability). The fact that they both output 0 is derived from the fact that  $B_0$  and  $C_0$  are an honest majority with the same input value 0. Therefore, they must output 0 in the face of *any* adversary  $\mathcal{A}$  controlling  $A_0$ ; in particular this holds with respect to the specific adversary  $\mathcal{A}$  described above. Since the views of  $B_0$  and  $C_0$  in  $S$  are identically distributed to their views in  $\Pi_0$ , we conclude that in the system  $S$  they also halt within  $\text{rounds}(\Pi)$  steps and output 0 (except with negligible probability). This completes the proof of the claim. ■

Using analogous arguments, we obtain the following two claims:

**Claim 2.3.4** *Except with negligible probability, parties  $A_1$  and  $B_1$  halt within  $\text{rounds}(\Pi)$  steps and output 1 in the system  $S$ .*

In order to prove this claim, the adversary is  $\mathcal{C}$  who controls  $C_1$  and  $C_2$  and works in a similar way to  $\mathcal{A}$  in the proof of Claim 2.3.3 above. (The only difference is regarding the specific edges that are redirected.)

**Claim 2.3.5** *Except with negligible probability, parties  $A_1$  and  $C_0$  halt within  $\text{rounds}(\Pi)$  steps and output the same value in the system  $S$ .*

Similarly, this claim is proven by defining an adversary  $\mathcal{B}$  who controls  $B_1$  and  $B_2$  and follows a similar strategy to  $\mathcal{A}$  in the proof of Claim 2.3.3 above.

Combining Claims 2.3.3, 2.3.4 and 2.3.5 we obtain a contradiction. This is because, on the one hand,  $C_0$  must output 0 in  $S$  (Claim 2.3.3), and  $A_1$  must output 1 in  $S$  (Claim 2.3.4). On the other hand, by Claim 2.3.5, parties  $A_1$  and  $C_0$  must output the same value. We conclude that there does not exist a 3-party protocol for Byzantine Agreement that tolerates one corruption and composes twice in parallel. This concludes the proof of the lemma. ■

Theorem 2.1.1 is derived from Lemma 2.3.2 in the standard way [PSL80, LSP82] by showing that if there exists a protocol that is correct for any  $n \geq 3$  and  $n/3$  corrupted parties, then one can construct a protocol for 3 parties that can tolerate one corrupted party. This is in contradiction to Lemma 2.3.2, and thus Theorem 2.1.1 is implied. ■

The following corollary, referring to concurrent composition, is immediately derived from the fact that parallel composition (where the scheduling of the messages is fixed and synchronized) is merely a special case of concurrent composition (where the adversary controls the scheduling).

---

segment of the hexagon system  $S$  by itself. Thus, in a *single* execution of  $\Pi$  with  $B_0$  and  $C_0$ , party  $\mathcal{A}$  can simulate the hexagon. Here, due to the fact that the parties  $B_1$  and  $C_1$  may have secret information that  $\mathcal{A}$  does not have access to,  $\mathcal{A}$  is unable to simulate their behavior itself. Rather,  $\mathcal{A}$  needs to redirect messages from the parallel execution of  $\Pi_1$  in order to complete the hexagon.

**Corollary 2.3.6** *No protocol for authenticated Byzantine Agreement that remains secure under concurrent self composition can tolerate  $n/3$  or more corrupted parties.*

## 2.4 Sequential Composition of Deterministic Protocols

Theorem 2.1.1 states that it is impossible to obtain authenticated Byzantine Agreement that remains secure under parallel (self) composition. The impossibility result holds even if only two parallel executions take place and even for randomized protocols. In this section, we consider a much more limited type of composition and a limited class of protocols. That is, we study the feasibility of obtaining *deterministic* protocols for authenticated Byzantine Agreement that remain secure under *sequential* (self) composition. We show that any protocol that terminates within  $r$  rounds can only be composed sequentially at most  $2r-1$  times. Thus, deterministic protocols for unbounded concurrency (i.e., protocols that remain secure for any polynomial number of executions) *do not exist*. We remark that the known protocols for (stand-alone) authenticated Byzantine Agreement are deterministic [PSL80, LSP82].

The lower bound here is derived by showing that for any deterministic protocol  $\Pi$ ,  $r$  rounds of the hexagonal system  $S$  (see Figure 2.1) can be simulated in  $2r$  sequential executions of  $\Pi$ . As we have seen in the proof of Theorem 2.1.1, the ability to simulate  $S$  results in a contradiction to the correctness of the Byzantine Agreement protocol  $\Pi$ . However, a contradiction is only derived if the system  $S$  halts. Since  $\Pi$  terminates within  $r$  rounds, the system  $S$  also halts within  $r$  rounds. Therefore, a contradiction is reached if  $\Pi$  is run  $2r$  or more times. We conclude that the protocol  $\Pi$  can be sequentially composed at most  $2r-1$  times.

We remark that in actuality, one can prove a more general statement that says that for *any* deterministic protocol,  $r$  rounds of 2 parallel executions of the protocol can be perfectly simulated in  $2r$  sequential executions of the same protocol. (More generally,  $r$  rounds of  $k$  parallel executions of a protocol can be simulated in  $k \cdot r$  sequential executions.) Thus, essentially, the deterministic sequential lower bound is derived by reducing it to the parallel composition case of Theorem 2.1.1. That is,

**Theorem 2.4.1** (Theorem 2.1.2 – restated): *Let  $\Pi$  be a deterministic protocol for authenticated Byzantine Agreement that terminates within  $r$  rounds of communication and remains secure under sequential self composition for  $2r$  or more executions. Then  $\Pi$  can tolerate at most  $t < n/3$  corrupted parties.*

**Proof.** As we have mentioned, we prove this theorem by showing that a faulty party in a deterministic authenticated Byzantine Agreement protocol  $\Pi$  for three parties, can perfectly simulate  $r/2$  rounds of the hexagonal system

$S$  using  $r$  sequential executions. Thus the proof here is very similar to the proof of Theorem 2.1.1.

Assume by contradiction that there exists a deterministic protocol for authenticated Byzantine Agreement  $\Pi$  for three parties with the following properties:  $\Pi$  tolerates one faulty party, always halts after at most  $r/2$  rounds, and is secure for  $r$  sequential executions. We show that the existence of such a  $\Pi$  results in a contradiction. Exactly as in the proof of Theorem 2.1.1, we combine two copies of  $\Pi$  into a hexagonal system  $S$  with parties  $A_0$ ,  $B_0$  and  $C_0$  (who all have input 0), and  $A_1$ ,  $B_1$  and  $C_1$  (who all have input 1). We begin by proving the following claim (which implies an analog to Claim 2.3.3).

**Claim 2.4.2** *There exists a faulty party  $A$  such that in the  $r^{\text{th}}$  sequential execution of  $\Pi$ , the views of  $B$  and  $C$  (upon input 0 into that execution) are identical to the respective views of  $B_0$  and  $C_0$  after  $r/2$  rounds of the system  $S$ .*

**Proof.** Party  $A$  works by simulating the hexagonal system  $S$ . Essentially, this consists of simulating two parallel executions of  $\Pi$  (with the edges “redirected” to make up  $S$ ); recall that this involves intertwining *two* different copies of  $\Pi$  (one in which  $A$ ,  $B$  and  $C$  input 0, and one in which they input 1). However, in our setting,  $\Pi$  can only be run sequentially and thus only *one* copy of  $\Pi$  can be running at any given time. This problem is solved by simulating one round of the hexagon over two sequential executions. That is, in the 1<sup>st</sup> and 2<sup>nd</sup> execution of  $\Pi$ , the first round of  $S$  is simulated; in the 3<sup>rd</sup> and 4<sup>th</sup> execution of  $\Pi$ , the first two rounds of  $S$  are simulated, and so on. We now show how this simulation is achieved (we prove the claim for the case that  $r$  is even; the other case is easily derived).

Let  $A_0$  and  $A_1$  be identical copies of  $A$ , except that  $A_0$  has input 0 and  $A_1$  has input 1; likewise define  $B_0$ ,  $B_1$ ,  $C_0$  and  $C_1$ . The sequential executions are such that  $A_0$ ,  $B_0$  and  $C_0$  execute in the  $i^{\text{th}}$  execution for *even* values of  $i$ ; and  $A_1$ ,  $B_1$  and  $C_1$  execute in the  $i^{\text{th}}$  execution for *odd* values of  $i$ . The adversarial party  $\mathcal{A}$  controls  $A_0$  and  $A_1$ , but as in the proof of Theorem 2.1.1,  $\mathcal{A}$ ’s faulty behavior consists merely of redirecting messages. We denote by  $\text{msg}_j(A, B)$  the  $j^{\text{th}}$  message sent by  $A$  to  $B$  in an execution of  $\Pi$ .  $\mathcal{A}$  works as follows ( $i$  denotes the index of the execution taking place in the series of sequential executions):

1.  $i = 1$ :  $\mathcal{A}$  invokes  $A_1$  who runs with  $B_1$  and  $C_1$ . Party  $\mathcal{A}$  records the messages output by  $A_1$ :  $\text{msg}_1(A_1, B_1)$  and  $\text{msg}_1(A_1, C_1)$  (we stress that  $A_1$  is run internally by  $\mathcal{A}$  and therefore these messages are obtained internally). Furthermore,  $\mathcal{A}$  receives and records the messages:  $\text{msg}_1(B_1, A_1)$  and  $\text{msg}_1(C_1, A_1)$  (these messages are received by  $\mathcal{A}$  through external interaction with  $B_1$  and  $C_1$ ). Finally,  $\mathcal{A}$  runs the execution until it concludes, ignoring the continuation.
2.  $i = 2$ :  $\mathcal{A}$  invokes  $A_0$  who runs with  $B_0$  and  $C_0$ . Party  $\mathcal{A}$  records the messages output by  $A_0$ :  $\text{msg}_1(A_0, B_0)$  and  $\text{msg}_1(A_0, C_0)$ . Then,  $\mathcal{A}$  sends  $B_0$

and  $C_0$  the messages  $\text{msg}_1(A_1, B_1)$  and  $\text{msg}_1(A_0, C_0)$ , respectively. Next,  $\mathcal{A}$  receives and records the messages:  $\text{msg}_1(B_0, A_0)$  and  $\text{msg}_1(C_0, A_0)$ . Finally,  $\mathcal{A}$  runs the execution until it concludes, ignoring the continuation.

3. *Odd  $i$* :  $\mathcal{A}$  invokes  $A_1$  who runs with  $B_1$  and  $C_1$ . For every  $j < i/2$  (starting at  $j = 1$  and increasing in sequential order), party  $\mathcal{A}$  works as follows (we note that all the messages sent in these rounds were obtained in previous executions):

- $\mathcal{A}$  passes  $A_1$  the messages  $\text{msg}_j(B_1, A_1)$  and  $\text{msg}_j(C_0, A_0)$  (informally speaking, this second message is redirected).
- $\mathcal{A}$  sends  $B_1$  the message  $\text{msg}_j(A_1, B_1)$ .
- $\mathcal{A}$  sends  $C_1$  the message  $\text{msg}_j(A_0, C_0)$  (this message is redirected).

Notice that the  $\text{msg}_j(\cdot, \cdot)$  messages were previously obtained in executions  $2j-1$  and  $2j$  of  $\Pi$ .

In round  $\lceil i/2 \rceil$ ,  $\mathcal{A}$  works as follows:

- $\mathcal{A}$  records the messages that are output by  $A_1$ :  $\text{msg}_{\lceil \frac{i}{2} \rceil}(A_1, B_1)$  and  $\text{msg}_{\lceil \frac{i}{2} \rceil}(A_1, C_1)$ .
- $\mathcal{A}$  obtains the messages  $\text{msg}_{\lceil \frac{i}{2} \rceil}(B_1, A_1)$  and  $\text{msg}_{\lceil \frac{i}{2} \rceil}(C_1, A_1)$  from  $B_1$  and  $C_1$  and records them.

Finally,  $\mathcal{A}$  runs the execution until it concludes, ignoring the continuation.

4. *Even  $i$* :  $\mathcal{A}$  invokes  $A_0$  who runs with  $B_0$  and  $C_0$ . For every  $j < i/2$  (starting at  $j = 1$  and increasing in sequential order), party  $\mathcal{A}$  works as follows:

- $\mathcal{A}$  passes  $A_0$  the messages  $\text{msg}_j(B_0, A_0)$  and  $\text{msg}_j(C_1, A_1)$ .
- $\mathcal{A}$  sends  $B_0$  the message  $\text{msg}_j(A_0, B_0)$ .
- $\mathcal{A}$  sends  $C_0$  the message  $\text{msg}_j(A_1, C_1)$

In round  $i/2$ ,  $\mathcal{A}$  works as follows:

- $\mathcal{A}$  records the messages output by  $A_0$ :  $\text{msg}_{\frac{i}{2}}(A_0, B_0)$  and  $\text{msg}_{\frac{i}{2}}(A_0, C_0)$ .
- $\mathcal{A}$  sends  $B_0$  and  $C_0$  the messages:  $\text{msg}_{\frac{i}{2}}(A_0, B_0)$  and  $\text{msg}_{\frac{i}{2}}(A_1, C_1)$ , respectively (this second message was obtained by  $\mathcal{A}$  in the previous execution).
- $\mathcal{A}$  obtains the messages  $\text{msg}_{\frac{i}{2}}(B_0, A_0)$  and  $\text{msg}_{\frac{i}{2}}(C_0, A_0)$  from  $B_0$  and  $C_0$  and records them.

Finally,  $\mathcal{A}$  runs the execution until it concludes, ignoring the continuation.

First, note that at the conclusion of round 1 of the second execution ( $i = 2$ ), the views of parties  $B_0$  and  $C_0$  are identical to their views at the conclusion

of round 1 of  $S$  (in particular,  $B_0$  sees messages from  $A_0$  and  $C_0$ , and  $C_0$  sees messages from  $B_0$  and  $A_1$ ). Then, in the fourth sequential execution,  $A$  begins by sending  $B_0$  and  $C_0$  the same round 1 messages. Since  $\Pi$  is a deterministic protocol,  $B_0$  and  $C_0$  reply with the *same* messages as in the second execution (likewise, the messages they send to each other are the same as in the second execution). Thus, the round 2 messages that they receive (that are computed based on the messages sent in previous executions) are consistent with their views in  $S$ . Using the same argument, we have that for every even  $i$ , after  $i/2$  rounds of the  $i^{\text{th}}$  sequential execution, the views of  $B_0$  and  $C_0$  are identical to their views after  $i/2$  rounds of  $S$ . This concludes the proof of the claim. ■

Recall that by the contradicting assumption,  $\Pi$  is a protocol for Byzantine Agreement that always halts within  $r/2$  rounds and is secure for  $r$  sequential executions. Thus, in the  $r^{\text{th}}$  sequential execution of  $\Pi$ , we have that  $B$  and  $C$  both halt within  $r/2$  rounds and output 0 (their output must equal 0 as they both have 0 for input). By Claim 2.4.2, it follows that in  $S$ , parties  $B_0$  and  $C_0$  also halt within  $r/2$  rounds and both output 0. Thus, we obtain the following analog to Claim 2.3.3:

**Claim 2.4.3** *Except with negligible probability, parties  $B_0$  and  $C_0$  halt within  $r/2$  rounds and output 0 in the system  $S$ .*

The following two claims (analogous to Claims 2.3.4 and 2.3.5) can be shown in a similar fashion:

**Claim 2.4.4** *Except with negligible probability, parties  $A_1$  and  $B_1$  halt within  $r/2$  rounds and output 1 in the system  $S$ .*

**Claim 2.4.5** *Except with negligible probability, parties  $A_1$  and  $C_0$  halt within  $r/2$  rounds and output the same value in the system  $S$ .*

Combining Claims 2.4.3, 2.4.4 and 2.4.5, we reach a contradiction. We thus conclude that there does not exist a deterministic protocol for authenticated Byzantine Agreement for three parties that tolerates one faulty party, runs for at most  $r/2$  rounds and is secure for  $r$  sequential executions. As in the proof of Theorem 2.1.1, the general case of  $n$  parties (for any  $n$ ) is obtained in a standard way. This completes the proof of the theorem. ■

## 2.5 Authenticated Byzantine Agreement Using Unique Identifiers

In this section we consider an augmentation to the authenticated model in which each execution is assigned a unique and common identifier. We show that in such a model, it is possible to achieve Byzantine Agreement that



composes concurrently, for *any* number of corrupted parties. We stress that in the authenticated model itself, it is not possible for the parties to agree on unique and common identifiers, without some external help. This is because by the results of this section, agreeing on a common identifier amounts to solving the Byzantine Agreement problem, and we have proven that this cannot be achieved for  $t \geq n/3$  when composition is required. Therefore, these identifiers must come from outside the system (and as such, assuming their existence is an augmentation to the authenticated model).

Intuitively, the existence of unique identifiers helps in the authenticated model for the following reason. Recall that our impossibility result is based on the ability of the adversary to *borrow* signed messages from one execution to another. Now, if each signature also includes the session identifier, then the honest parties can easily distinguish between messages signed in this execution and messages signed in a different execution. It turns out that this is enough. That is, we give a transformation of protocols for authenticated Byzantine Agreement to protocols that compose concurrently in a setting where unique identifiers exist. Loosely speaking, our transformation holds for protocols that utilize the signature scheme for signing and verifying only (as is natural).

**Secure signature schemes.** Before proceeding, we present an informal definition of secure signature schemes. A signature scheme is a triplet of algorithms  $(G, S, V)$ , where  $G$  is a probabilistic generator that outputs a pair of signing and verification keys  $(sk, vk)$ ,  $S$  is a signing algorithm and  $V$  is a verification algorithm. The **validity** requirement for signature scheme states that except with negligible probability, for every message  $m$ ,  $V(vk, m, S(sk, m)) = 1$ , where  $(vk, sk) \leftarrow G(1^n)$ ; i.e., honestly generated signatures are almost always accepted.

The security requirement of a signature scheme states that the probability that an efficient forging algorithm  $\mathcal{A}$  can succeed in generating a valid forgery is negligible. This should hold even when  $\mathcal{A}$  is given oracle access to a signing oracle (this oracle represents valid signatures that  $\mathcal{A}$  may obtain in a real attack). In order for  $\mathcal{A}$  to succeed, it must generate a valid signature on a message that was not queried to the signing oracle. More formally, the following experiment is defined: The generator  $G$  is run, outputting a key-pair  $(vk, sk)$ . Then,  $\mathcal{A}$  is given  $vk$  and oracle access to the signing oracle  $S(sk, \cdot)$ . At the conclusion of the experiment,  $\mathcal{A}$  outputs a pair  $(m^*, \sigma^*)$ . Let  $Q_m$  be the set of oracle queries by  $\mathcal{A}$ . Then, we say that  $\mathcal{A}$  has succeeded if  $V(vk, m^*, \sigma^*) = 1$  and  $m^* \notin Q_m$ . (That is,  $\mathcal{A}$  output a message along with a valid signature, and  $\mathcal{A}$  did not query its oracle with this message.) A signature scheme is **existentially secure against chosen-message attacks** if for every probabilistic polynomial-time  $\mathcal{A}$ , the probability that  $\mathcal{A}$  succeeds is negligible.

**Generalizing the security of signature schemes.** Clearly, the focus of the above definition of security is on the fact that  $\mathcal{A}$  cannot succeed in gen-

erating any forgery (except with negligible probability). However, according to the specific formulation,  $\mathcal{A}$  receives oracle access to  $S(sk, \cdot)$  only. For our purposes below, we wish to consider what happens when  $\mathcal{A}$  is given oracle access to some oracle  $\text{Aux}(sk, \cdot)$  that does *not* generate valid signatures, but rather computes some other function of  $sk$  and the query. That is,  $\mathcal{A}$  can receive information connected to  $sk$  that is not necessarily limited to valid signatures. Of course, if this additional information consists of fully revealing  $sk$ , then  $\mathcal{A}$  could easily forge signatures. However, other information may be revealed that does not enable  $\mathcal{A}$  to forge signatures. More formally, consider a setting where the adversary is given access to two oracles:  $S(sk, \cdot)$  and  $\text{Aux}(sk, \cdot)$ , where  $\text{Aux}$  is an auxiliary information oracle. Security is defined as previously; however, we stress that the only limitation on the message  $m^*$  output by  $\mathcal{A}$  at the conclusion of the experiment is that it was not queried to the  $S(sk, \cdot)$  oracle. In particular,  $\mathcal{A}$  may have queried the  $\text{Aux}$  oracle with  $m^*$  and this does not affect the validity of the forgery.<sup>8</sup> Formally, we define an identical experiment as above, except that  $\mathcal{A}$  is given oracle access to both  $S$  and  $\text{Aux}$ . We stress that the set of queries  $Q_m$  consists *only* of  $\mathcal{A}$ 's queries to  $S(sk, \cdot)$ . We say that  $\langle (S, V), \text{Aux} \rangle$  is **existentially secure against generalized chosen-message attacks** if for every probabilistic polynomial-time  $\mathcal{A}$ , the probability that  $\mathcal{A}$  succeeds in outputting a forgery not in  $Q_m$  is negligible. Notice that  $\text{Aux}$  must be specified along with  $S$  and  $V$  in order to determine whether or not the scheme is secure.

We will now define a specific signature scheme and show that it is existentially secure against generalized chosen-message attacks. Let  $(S, V)$  be any signature scheme that is existentially secure against chosen-message attacks and let  $id$  be a string (of any length). Define  $(S_{id}, V_{id})$  as follows:  $S_{id}(sk, m) = S(sk, id \circ m)$  and  $V_{id}(vk, m, \sigma) = V(vk, id \circ m, \sigma)$ , where  $\circ$  denotes concatenation. That is,  $(S_{id}, V_{id})$  are the same as  $(S, V)$  except that the message  $m$  is always prefixed by the string  $id$ . Next, define the oracle  $\text{Aux}(sk, \cdot) = S_{-id}(sk, \cdot)$  as follows:  $S_{-id}(sk, m) = S(sk, m)$  if the prefix of  $m$  does *not* equal  $id$ . Otherwise,  $S_{-id}(sk, m) = \perp$ . In other words, the oracle  $\text{Aux}(sk, \cdot) = S_{-id}(sk, \cdot)$  signs on any message that does not have  $id$  as a prefix. We now claim that  $\langle (S_{id}, V_{id}), S_{-id} \rangle$  is existentially secure against generalized chosen-message attacks. This can be seen as follows. Intuitively, the oracle queries to  $S_{-id}$  cannot be of any help to an adversary  $\mathcal{A}$  because a successful forgery must be prefixed by  $id$  and all oracle queries to  $S_{-id}$  must be prefixed by some  $id' \neq id$ . More formally, assume that there exists an adversary  $\mathcal{A}$  that successfully generates a forgery in the setting of a generalized chosen-message attack against  $\langle (S_{id}, V_{id}), S_{-id} \rangle$ . Then, we construct an adversary  $\mathcal{A}'$  who successfully generates a forgery in a standard chosen-message attack against  $(S, V)$ .  $\mathcal{A}'$  invokes  $\mathcal{A}$  and answers all its oracle queries for it.  $\mathcal{A}'$

<sup>8</sup> Clearly, it must be infeasible to derive a valid signature  $\sigma = S(sk, m)$  from the oracle-response  $\text{Aux}(sk, m)$ . Otherwise, the adversary  $\mathcal{A}$  can always succeed in generating a forgery: It simply obtains  $\text{Aux}(sk, m)$  and derives the signature  $\sigma$ .

can do this because it has access to the  $S(sk, \cdot)$  oracle. By the contradicting assumption, with non-negligible probability  $\mathcal{A}$  outputs a pair  $(m^*, \sigma^*)$  such that  $V(vk, id \circ m^*, \sigma^*) = 1$  and  $m^*$  was not queried to the  $S_{id}(sk, \cdot)$  oracle. However, this implies that in the simulation,  $\mathcal{A}'$  did not query its oracle with  $id \circ m^*$ . (This holds because all queries of  $\mathcal{A}$  to the  $S_{-id}$  oracle have a different prefix.) Therefore,  $\mathcal{A}'$  halts, outputting the successfully forged pair  $(id \circ m^*, \sigma^*)$ . This contradicts the security of  $(S, V)$ .

We are now ready to present the transformation itself:

**The transformation.** Let  $\Pi$  be a protocol for authenticated Byzantine Agreement that uses a secure signature scheme  $(S, V)$ . We define a modified protocol  $\Pi(id)$  that is exactly the same as  $\Pi$  except that the parties use the signature scheme  $(S_{id}, V_{id})$  as defined above. We note that the common value  $id$  is given to each party as auxiliary input.

In the following theorem we show that the above simple transformation suffices for achieving security in a setting where many concurrent executions take place.

**Theorem 2.5.1** *Let  $\Pi$  be a protocol for authenticated Byzantine Agreement that is secure when using a signature scheme that is existentially secure against generalized chosen-message attacks. Furthermore, the (honest) parties use their secret keys for signing only.<sup>9</sup> Let  $\Pi(id)$  be obtained from  $\Pi$  as in the above transformation, and let  $id_1, \dots, id_\ell$  be a series of  $\ell$  equal-length strings.<sup>10</sup> Then the protocols  $\Pi(id_1), \dots, \Pi(id_\ell)$  all solve the Byzantine Agreement problem, even when run concurrently.*

**Proof.** Intuitively, the security of the protocols  $\Pi(id_1), \dots, \Pi(id_\ell)$  is due to the fact that signatures from  $\Pi(id_i)$  cannot be of any help to the adversary in  $\Pi(id_j)$ . This is because in  $\Pi(id_j)$ , the honest parties reject any signature on a message that begins with an identifier that is not  $id_j$ . Since  $id_i \neq id_j$ , we have that signatures sent in  $\Pi(id_i)$  are of no help in  $\Pi(id_j)$ . Our formal proof of this intuition proceeds by showing how an adversary for a single execution of  $\Pi(id)$  can internally simulate the concurrent executions of  $\Pi(id_1), \dots, \Pi(id_\ell)$ , thereby reducing the security of the concurrent setting to the stand-alone setting. However, in order to enable the adversary to simulate other executions, it must be able to simulate the signatures generated by the honest parties in these executions. By supplying the adversary with the oracle  $S_{-id}$  as described above, this becomes possible.

First, we remark that  $\Pi(id)$  constitutes a secure Byzantine Agreement protocol even when an adversary  $\mathcal{A}_{id}$  is given access to all of the oracles  $S_{-id}(sk_1, \cdot), \dots, S_{-id}(sk_n, \cdot)$ . This is the case because  $\Pi$  is secure when using

<sup>9</sup> Formally, this means that party  $P_i$ 's instructions can be formulated with access to an oracle  $S(sk, \cdot)$  instead of the key  $sk$  itself.

<sup>10</sup> More generally, any set of  $\ell$  prefix-free strings suffice.

a signature scheme that is existentially secure against *generalized* chosen-message attacks, and  $\langle (S_{id}, V_{id}), S_{-id} \rangle$  is such a scheme. (Recall that  $\Pi(id)$  is identical to  $\Pi$  except that the signature scheme used is  $(S_{id}, V_{id})$ .) We also note that the security of  $\Pi(id)$  trivially holds even when  $\mathcal{A}_{id}$  is able to choose  $id$ .

Next, we show that an adversary  $\mathcal{A}$  who successfully attacks the concurrent executions of  $\Pi(id_1), \dots, \Pi(id_\ell)$  can be used by  $\mathcal{A}_{id}$  to successfully attack a single execution of  $\Pi(id)$  for some string  $id$ . This then contradicts the security of the underlying protocol  $\Pi$ . Now, assume by contradiction that  $\mathcal{A}$  succeeds in “breaking” one of the  $\Pi(id_i)$  executions with non-negligible probability. The adversary  $\mathcal{A}_{id}$  internally incorporates  $\mathcal{A}$  and attacks a single execution of  $\Pi(id)$ . Intuitively,  $\mathcal{A}_{id}$  internally simulates all executions for  $\mathcal{A}$ , except for the one in which  $\mathcal{A}$  succeeds in its attack. Formally,  $\mathcal{A}_{id}$  first randomly selects an execution  $i \in_R \{1, \dots, \ell\}$  and sets  $id = id_i$ . Next,  $\mathcal{A}_{id}$  invokes  $\mathcal{A}$  and emulates the concurrent executions of  $\Pi(id_1), \dots, \Pi(id_\ell)$  for  $\mathcal{A}$ . Adversary  $\mathcal{A}_{id}$  does this by playing the roles of the honest parties in all but the  $i^{\text{th}}$  execution  $\Pi(id_i)$ . In contrast, in  $\Pi(id_i)$  adversary  $\mathcal{A}_{id}$  externally interacts with the honest parties and passes messages between them and  $\mathcal{A}$ . Since  $\mathcal{A}_{id}$  is given access to the signing oracles  $S_{-id}(sk_1, \cdot), \dots, S_{-id}(sk_n, \cdot)$  and the honest parties use their signing keys to generate signatures only,  $\mathcal{A}_{id}$  is able to generate the honest parties’ messages in all the executions  $\Pi(id_j)$  for  $j \neq i$ . (Recall that in these executions, the prefix of every signed message is  $id_j \neq id_i$  and thus these oracles suffice.) Therefore, the emulation by  $\mathcal{A}_{id}$  of the concurrent executions for  $\mathcal{A}$  is perfect. This implies that  $\mathcal{A}_{id}$  succeeds in “breaking”  $\Pi(id)$  with success probability that equals  $1/\ell$  times  $\mathcal{A}$ ’s success probability in the concurrent setting. Thus,  $\mathcal{A}_{id}$  succeeds with non-negligible probability and this contradicts the stand-alone security of  $\Pi(id)$ . ■

It is easy to verify that the protocols of [PSL80, LSP82, DS83] for authenticated Byzantine Agreement all fulfill the requirements in the assumption of Theorem 2.5.1. We therefore obtain the following corollary:

**Corollary 2.5.2** *In a model where global unique identifiers are allocated to each execution, there exist protocols for authenticated Byzantine Agreement that tolerate any  $t < n$  corruptions and remain secure under concurrent self composition.*

We conclude by noting that it is not at all clear how it is possible to augment the authenticated model with unique identifiers. In particular, requiring the on-line participation of a trusted party who assigns identifiers to every execution is clearly impractical. (Furthermore, such a party could just be used to directly implement broadcast.) However, we do note one important scenario where Theorem 2.5.1 *can* be applied. As we have mentioned, secure protocols often use many invocations of a broadcast primitive. Furthermore, in order to improve round efficiency, in any given round many broadcasts may be simultaneously executed. The key point here is that *within* the secure protocol,

unique identifiers can be allocated to each broadcast by the protocol designer. Therefore, authenticated Byzantine Agreement can be used. Of course, this does not change the fact that the secure protocol itself will not compose in parallel or concurrently. However, it does mean that its security is guaranteed in the stand-alone setting, and a physical broadcast channel is not necessary.

### 3. Secure Computation without Agreement

In the previous chapter, we have shown that authenticated Byzantine Agreement protocols cannot be composed concurrently (or even in parallel) when a third or more of the parties are corrupted. An immediate and important ramification of this result relates to the composition of secure multi-party computation. All known protocols for general secure multi-party computation strongly rely on the extensive use of a broadcast primitive. When a third or more of the parties are corrupted, this broadcast is implemented using authenticated Byzantine Agreement. Essentially, this use of Byzantine Agreement cannot be eliminated since the standard definition of secure computation (for the case of an honest majority) actually implies Byzantine Agreement. Moreover, it is accepted folklore that the use of a broadcast channel is essential for achieving secure multiparty computation, even when there is no honest majority. Due to the above state of affairs, *all known protocols for secure multi-party computation in the point-to-point network model do not remain secure under concurrent or even parallel composition.*

In this chapter we show that secure computation *can* be achieved without a broadcast channel, and thus without authenticated Byzantine Agreement. Specifically, we present a new definition of secure computation that only mildly relaxes previous definitions and that can be achieved without using a broadcast channel. The new definition separates the issue of agreement from the central security issues of privacy, correctness and independence of inputs in secure computation. As a result, security for general multi-party computation does not imply Byzantine Agreement. Therefore, the lower bounds of Byzantine Agreement no longer apply to secure multi-party computation. Indeed, we prove that secure multi-party computation can be achieved for any number of corrupted parties and without a broadcast channel (or trusted preprocessing phase as required for running authenticated Byzantine Agreement). An important corollary of the results in this chapter is the ability to obtain multi-party protocols in a point-to-point network that remain secure under self composition (with concurrent scheduling in the case of an honest majority, and with parallel scheduling when there is no such honest majority).

### 3.1 Introduction

#### 3.1.1 Byzantine Agreement and Secure Multi-party Computation

There is a close connection between Byzantine Agreement and secure multi-party computation. First, Byzantine Agreement (or broadcast) is used as a basic and central tool in the construction of secure protocols. In particular, all known protocols for general multi-party computation use a broadcast channel (and implement it using Byzantine Agreement or authenticated Byzantine Agreement). Second, Byzantine Agreement is actually a special case of secure computation (this holds by the standard definition taken for the case of an honest majority, where output delivery is guaranteed). Therefore, all the lower bounds relating to Byzantine Agreement immediately apply to secure multi-party computation. In particular, the Byzantine Agreement problem cannot be solved in the plain model when a third or more of the parties are corrupted [PSL80]. Thus, in the plain model and when a third or more of the parties are corrupted, it is also impossible to achieve general secure computation with guaranteed output delivery. On the other hand, when there is an honest majority (but possibly more than  $1/3$  corrupted) it *is* possible to obtain secure computation with guaranteed output delivery, assuming a broadcast channel. This means that in order to achieve such secure computation for the range of  $n/3 \leq t < n/2$ , either a physical broadcast channel or a trusted pre-processing phase for running authenticated Byzantine Agreement *must* be assumed. (Recall that  $n$  denotes the overall number of parties and  $t$  the number of corrupted parties.)

In the previous chapter, it was shown that authenticated Byzantine Agreement cannot be composed (concurrently or even in parallel), unless  $t < n/3$ . This has the following ramifications. On the one hand, in the range of  $n/3 \leq t < n/2$ , it is *impossible* to obtain secure computation that composes without using a physical broadcast channel.<sup>1</sup> This is because such a protocol in the point-to-point network model and with trusted pre-processing would imply authenticated Byzantine Agreement that composes. On the other hand, as we have mentioned, in the range of  $t \geq n/2$  the definitions of secure computation are relaxed and thus do not imply Byzantine Agreement. Nevertheless, all protocols for secure computation in this range make extensive use of a broadcast primitive. The impossibility of composing authenticated Byzantine Agreement puts this whole body of work into question when composition is required. Specifically without using a physical broadcast channel, none of these protocols compose (even in parallel). In summary, there are no known protocols for secure computation in a point-to-point network that compose

---

<sup>1</sup> Of course, what we mean is that it is impossible to obtain protocols that would provide a solution to the Byzantine Agreement problem. In particular, *general* protocols that can be used for securely computing *any* function (and thus also Byzantine Agreement), cannot be obtained.

in parallel or concurrently, for any  $t \geq n/3$ . Needless to say, the requirement of a physical broadcast channel is very undesirable (and often unrealistic).

### 3.1.2 Our Results

As we described in Section 1.2, in the case that no honest majority exists (i.e., when  $t \geq n/2$ ), guaranteed output delivery and complete fairness cannot be achieved. Therefore, the ideal model is modified so that these properties are not implied. Specifically, under certain conditions, parties are allowed to abort (in which case they output a special symbol  $\perp$  rather than their specified output). In addition, different notions of fairness are required for different values of  $t$ . We now describe three types of fairness. On the one extreme, there is “complete fairness” that guarantees that if a corrupt party gets its output then all honest parties also get their output. (As we have mentioned, this is only possible when  $t < n/2$ .) On the other extreme, there is “no fairness” in which case the adversary always gets its output and has the power to decide whether or not the honest parties also get output. An intermediate notion, called “partial fairness”, specifies out a single party such that if the specified party is honest then complete fairness is achieved. On the other hand, if the specified party is corrupt, then no fairness is achieved. Thus, fairness is partial. Regarding the abort policy regarding the honest parties, previous definitions required that either *all* honest parties receive their outputs or *all* honest parties output  $\perp$  [G02a].<sup>2</sup> Thus the parties all agree on whether or not output was received.

In this chapter, we present a further relaxation of the definition of secure multi-party computation for the case that output delivery is not guaranteed. The additional relaxation is very mild, yet has the effect of decoupling the issue of *agreement* from the issue of *security* in multi-party computation. Specifically, in our new definition some honest parties may receive output while some receive  $\perp$ , and the requirement of agreement of abort is removed. We stress that this is the only difference between our definition and previous ones. All the other security properties (e.g., privacy, correctness and independence of inputs) are preserved.

Our main result is the construction of a protocol for secure computation according to the new definition for *any*  $t < n$  and *without* a broadcast channel or setup assumption. (Our protocol assumes the same computational assumptions made, if any, by corresponding protocols that assume a broadcast channel.) We note that our results hold in both the information-theoretic and computational models.

**A hierarchy of definitions.** In order to describe our results in more detail, we present a hierarchy of definitions for secure computation. The hierarchy

---

<sup>2</sup> We note that in private communication, Goldreich stated that the requirement in [G02a] of having all parties abort or all parties receive output was only made in order to simplify the definition.



that we present here relates to the issues of abort (or failure to receive output) and fairness.

1. *Secure computation without abort*: According to this definition, all parties are guaranteed to receive their output. (This is what we previously called “guaranteed output delivery”.) This is the standard definition for the case of honest majority (i.e.,  $t < n/2$ ). Since all honest parties receive output, complete fairness is always obtained here.
2. *Secure computation with unanimous abort*: In this definition, it is ensured that either all honest parties receive their outputs or all honest parties abort. This definition can be considered with different levels of fairness:
  - a) *Complete fairness*: Recall that when complete fairness is achieved, the honest parties are guaranteed to receive output if the adversary does. Therefore, here one of two cases can occur: Either all parties receive output or all parties abort. This means that the adversary can conduct a denial of service attack, but nothing else. (This definition can only be achieved in the case of  $t < n/2$ .)
  - b) *Partial fairness*: As in the case of complete fairness, the adversary may disrupt the computation and cause the honest parties to abort without receiving their prescribed output. However, unlike above, the adversary may receive the corrupted parties’ outputs, even if the honest parties abort (and thus the abort is not always fair). In particular, the protocol *specifies* a single party such that the following holds. If this party is honest, then complete fairness is essentially achieved (i.e., either all parties abort or all parties receive correct output). If the specified party is corrupt, then fairness may be violated. That is, the adversary receives the corrupted parties’ outputs first, and then decides whether or not the honest parties all receive their correct output or all receive abort (and thus the adversary may receive output while the honest parties do not).  
 Although fairness is only guaranteed in the case that the specified party is not corrupted, there are applications where this feature may be of importance. For example, in a scenario where one of the parties may be “more trusted” than others (yet not too trusted), it may be of advantage to make this party the specified party. Another setting where this can be of advantage is one where all the participating parties are trusted. However, the security problem that we are trying to protect against is that of an external party “hacking” into the machines of one of more of the parties. In such a case, it may be possible to provide additional protection to the specified party.
  - c) *No fairness*: This is the same as in the case of partial fairness except that the adversary always receives the corrupted parties’ outputs first (i.e., there is no specified party).

We stress that in all the above three definitions, if one honest party aborts then so do all honest parties, and so the abort is *unanimous*. This

means that in the case that an abort occurs, all of the parties are aware of the fact that the protocol did not successfully terminate. This feature of having all parties succeed or fail together may be an important one in some applications.

3. *Secure computation with abort*: The only difference between this definition and the previous one is that some honest parties may receive output while others abort. That is, the requirement of unanimity with respect to abort is removed. This yields two different definitions, depending on whether partial fairness or no fairness is taken. (Complete fairness is not considered here because it only makes sense in a setting where all the parties, including the corrupted parties, either all receive output or all abort. Therefore, it is not relevant in the setting of secure computation with non-unanimous abort.)

Using the above terminology, the definition proposed by [G02a] for the case of  $t \geq n/2$  is that of secure computation with unanimous abort and partial fairness. Our new definition is that of secure computation with abort (and partial or no fairness), and as we have mentioned, its key feature is a decoupling of the issues of secure computation and agreement (or unanimity) of abort.

**Achieving secure computation with abort.** In this chapter, we show that secure computation with abort and partial fairness can be achieved for any  $t < n$ , and without a broadcast channel or a trusted pre-processing phase. We achieve this result in the following way. We begin by defining a weak variant of the Byzantine Generals problem, called *broadcast with abort*, with the following properties. First, there exists a *single* value  $x$  such that every party either outputs  $x$  or aborts. Second, when the broadcasting party is honest, the value  $x$  equals its input, similarly to the validity condition of Byzantine Generals. We call this “broadcast with abort” because as with secure computation with abort, some parties may output the correct value while other honest parties abort. We show how to achieve this type of broadcast with a simple deterministic protocol that runs in 2 rounds. Secure multi-party computation is then achieved by replacing the broadcast channel in known protocols with a broadcast with abort protocol. Despite the weak nature of agreement in this broadcast protocol, it is nevertheless enough for achieving secure multi-party computation with abort. Since our broadcast with abort protocol runs in only 2 rounds, we also obtain a very efficient transformation of protocols that work with a broadcast channel into protocols that require only a point-to-point network. In summary, we obtain the following theorem:

**Theorem 3.1.1** (efficient transformation): *There exists an efficient protocol compiler that receives any protocol  $\Pi$  for the broadcast model and outputs a protocol  $\Pi'$  for the point-to-point model such that the following holds: If  $\Pi$  securely computes a functionality  $f$  with unanimous abort and with any level of fairness, then  $\Pi'$  securely computes  $f$  with abort and with no fairness.*

Furthermore, if  $\Pi$  tolerates up to  $t$  corruptions and runs for  $R$  rounds, then  $\Pi'$  tolerates up to  $t$  corruptions and runs for  $O(R)$  rounds.

Notice that in the transformation of Theorem 3.1.1, protocol  $\Pi'$  does not achieve complete fairness or partial fairness, even if  $\Pi$  did. Thus, fairness may be lost in the transformation. Nevertheless, meaningful secure computation is still obtained, and at virtually no additional cost.

When obtaining some level of fairness is important, Theorem 3.1.1 does not provide a solution. We therefore also show that partial fairness *can* be obtained without a broadcast channel for the range of  $t \geq n/2$ . Recall that even with a broadcast channel, complete fairness cannot be obtained in this range; therefore, we do not lose any fairness even though we work in the point-to-point model only. This result is stated in the following theorem:

**Theorem 3.1.2** (partial fairness): *For any probabilistic polynomial-time  $n$ -party functionality  $f$ , there exists a protocol in the point-to-point model for computing  $f$  that is secure with abort, partially fair and tolerates any  $t < n$  corruptions.*

The theorem is proved by first showing that fairness can be boosted in the point-to-point model. That is, given a generic protocol for secure multi-party computation that achieves no fairness, one can construct a generic protocol for secure multi-party computation that achieves partial fairness. (Loosely speaking, a generic protocol is one that can be used to securely compute any efficient functionality.) Applying Theorem 3.1.1 to known protocols for the broadcast model, we obtain secure multi-party computation that achieves no fairness. Then, using the above “fairness boosting”, we obtain Theorem 3.1.2. We note that the round complexity of the resulting protocol is of the same order of the “best” generic protocol that works in the broadcast model. In particular, based on the protocol of [BMR90], we obtain the first constant-round protocol in the point-to-point network for the range of  $n/3 \leq t < n/2$ .<sup>3</sup> That is:

**Corollary 3.1.3** (constant round protocols without broadcast for  $t < n/2$ ): *Assume that there exist public-key encryption schemes (or, alternatively, assume the existence of one-way functions and a model with private channels). Then, for every probabilistic polynomial-time functionality  $f$ , there exists a constant round protocol in the point-to-point network for computing  $f$  that is secure with abort, partially fair and tolerates  $t < n/2$  corruptions.*

<sup>3</sup> For the range of  $t < n/3$ , the broadcast channel in the protocol of [BMR90] can be replaced by the expected constant-round Byzantine Agreement protocol of [FM97]. However, when  $n/3 \leq t < n/2$ , authenticated Byzantine Agreement must be used. Since there are no known (expected) constant-round protocols for authenticated Byzantine Agreement, we have that the resulting protocol for secure computation is not constant-round.

**Composition of secure multi-party protocols.** An important corollary of our new definition is the ability to obtain secure multi-party protocols for  $t > n/3$  that self-compose in parallel (for any  $t$ ) or concurrently (for  $t < n/2$ ), without a broadcast channel. Since our protocols do not use a broadcast channel (or authenticated Byzantine Agreement), the lower bound of Chapter 2 does not apply. Specifically, if a protocol composes when using a broadcast channel, then the transformed protocol in the point-to-point network also composes. This is in contrast to all previous protocols that used authenticated Byzantine Agreement in order to replace the broadcast channel.<sup>4</sup> Jumping ahead, we compare the composition achieved here to that obtained in Chapter 4. In Chapter 4, the strong notion of concurrent general composition is obtained. However, the protocol there relies on a common reference string. In contrast, in this chapter, we obtain only the weaker notion of self composition. Nevertheless, our protocols are in the plain model and do not require any setup assumptions.

**Discussion.** We propose that the basic definition of secure computation should focus on the issues of privacy, correctness and independence of inputs. In contrast, the property of agreement should be treated as an additional, and not central, feature. The benefit of taking such a position (irrespective of whether one is convinced conceptually) is that the feasibility of secure computation is completely decoupled from the feasibility of Byzantine Agreement. Thus, the lower bounds relating to Byzantine Agreement (and authenticated Byzantine Agreement) do not imply anything regarding secure computation. Indeed, as we show, “broadcast with abort” is sufficient for secure computation. However, it lacks any flavour of agreement in the classical sense. This brings us to an important observation. Usually, proving a lower bound for a special case casts light on the difficulties in solving the general problem. However, in the case of secure computation this is not the case. Rather, *the fact that the lower bounds of Byzantine Agreement apply to secure computation is due to marginal issues relating to unanimity regarding the delivery of outputs, and not due to the main issues of security.*

### 3.1.3 Related Work

There are two independent and concurrent results [FGMR02, FGHHS02] studying the necessity of broadcast in secure computation, although apparently for

<sup>4</sup> Unfortunately, full proofs regarding the composition of protocols using a broadcast channel do not appear in the literature. In this footnote we momentarily allow ourselves to rely on the unproven claim that the protocol of [RB89] that uses a broadcast channel and achieves security for  $t < n/2$ , remains secure under concurrent self composition. Regarding parallel self composition, it is not difficult to show that a variant of the (broadcast-model) protocol of [GMW87] that achieves security for any  $t < n$ , remains secure under parallel self composition; see Section 3.6 for more details. We can therefore conclude that in the point-to-point network, concurrent self composition of secure multi-party computation is possible for  $t < n/2$ , and parallel self composition is possible for any  $t$ .

different motivation. In [FGMR02], the question of multi-party computation in the case that the number of corruptions is  $t < n/2$  is studied. They show that in this case, it is possible to achieve weak Byzantine Agreement (where loosely speaking, either all honest parties abort or all honest parties agree on the broadcasted value). (We note that their protocol is randomized and therefore bypasses the  $t < n/3$  lower-bound on deterministic weak Byzantine Agreement protocols of [L83].) They further show how this can be used in order to obtain secure computation with unanimous abort and complete fairness for the case of  $t < n/2$ . Thus for the range of  $n/2 \leq t < n/3$ , their solution achieves complete fairness whereas ours achieves only partial fairness.

In subsequent work, [FGHHS02] studied the question of Byzantine Agreement for any  $t < n$  and whether its relaxation to weak Byzantine Agreement can be achieved without preprocessing. They show that it is indeed possible to achieve (randomized) weak Byzantine Agreement for *any*  $t < n$ , in  $O(t)$  rounds. They also show how their weak Byzantine Agreement protocol can be used to obtain secure computation with unanimous abort and partial fairness for any  $t < n$ .

In comparison, we achieve secure computation with (non-unanimous) abort and partial fairness for any  $t < n$ . However, our focus is different. In particular, our results emphasize the fact that the issue of agreement is not central to the task of secure computation. Furthermore, removing this requirement enables us to remove the broadcast channel with almost no cost. Thus, we obtain a round-preserving transformation of secure protocols in the broadcast model to those in the point-to-point model. This is in contrast to [FGMR02, FGHHS02] who use their weak Byzantine Agreement protocol in order to setup a public-key infrastructure for authenticated Byzantine Agreement. They therefore incur the cost of setting up this infrastructure along with a cost of  $t+1$  rounds for simulating every broadcast in the original protocol. Our protocols are therefore significantly more round efficient.<sup>5</sup> Finally we note that we can use the weak Byzantine Agreement protocol of [FGHHS02] to transform any generic  $r$ -round protocol for secure computation with abort into an  $(r+t)$ -round protocol with unanimous abort (and the same level of fairness). This is achieved by having the parties broadcast whether they received outputs or not after the protocol with abort concludes. It is enough to

<sup>5</sup> We note one subtle, yet important caveat. Given a *general* protocol for secure computation (i.e., a protocol that can be used for securely computing any functionality) that uses a broadcast channel and runs for  $r$  rounds, we obtain an  $O(r)$ -round protocol that is secure with abort and partially fair (this is in contrast to the  $O(tr)$  round complexity of [FGMR02, FGHHS02]). However, given a protocol that solves a specific secure computation problem, our transformation provides no fairness (and in particular does not achieve partial fairness). In order to achieve partial fairness, we must revert to a general protocol. In contrast, the transformation of [FGMR02, FGHHS02] works for any protocol. Thus, given a very efficient protocol for a specific problem that achieves partial fairness, it may actually be “cheaper” to use the transformation of [FGMR02, FGHHS02].

use weak Byzantine Agreement for this broadcast. We therefore reduce the  $O(tr)$  round complexity of [FGHS02] to  $O(r+t)$ , while achieving the same level of security.

In [C01], a communication model where the adversary has control over the delivery of messages was introduced. Essentially, this definition also decouples secure computation from agreement because parties are never guaranteed to get output. In particular, the adversary is allowed to deliver output to whomever it wishes, and only these parties will ever receive output. However, the motivation of [C01] is different; it aims to decouple the issue of guaranteed output delivery from the main issues of secure computation. In contrast, we focus on the question of agreement by the parties on whether or not output was delivered.

### 3.2 Definitions – Secure Computation

In this section we present definitions for secure multi-party computation. The basic description and definitions are based on [G02a], which in turn follows [GL90, B91a, MR91, C00]. We actually consider a number of definitions here. In particular, we present formal definitions for secure computation with *unanimous abort* and with *abort*, with *complete fairness*, *partial fairness*, and *no fairness*. In addition, we refer to *secure computation without abort*, which is the standard definition used when more than half the parties are honest. According to this definition, all parties receive the output and the adversary cannot disrupt the computation. However, we will not formally present this definition here.

**Notation:** We denote by  $U_k$  the uniform distribution over  $\{0, 1\}^k$ ; for a set  $S$  we denote  $s \in_R S$  when  $s$  is chosen uniformly from  $S$ ; we let  $[n]$  denote the set of integers  $\{1, \dots, n\}$ ; finally, computational indistinguishability is denoted by  $\stackrel{c}{\equiv}$  and statistical closeness by  $\stackrel{s}{\equiv}$ . The security parameter is denoted by  $k$ .

**Multi-party computation.** A multi-party protocol problem (for  $n$  parties  $P_1, \dots, P_n$ ) is cast by specifying a random process that maps vectors of inputs to vectors of outputs (one input and one output for each party). We refer to such a process as an  $n$ -ary functionality and denote it  $f : (\{0, 1\}^*)^n \rightarrow (\{0, 1\}^*)^n$ , where  $f = (f_1, \dots, f_n)$ . That is, for a vector of inputs  $\bar{x} = (x_1, \dots, x_n)$ , the output-vector is a random variable  $(f_1(\bar{x}), \dots, f_n(\bar{x}))$  ranging over vectors of strings. The output for the  $i^{\text{th}}$  party (with input  $x_i$ ) is defined to be  $f_i(\bar{x})$ .

**Adversarial behaviour.** Loosely speaking, the aim of a secure multi-party protocol is to protect the honest parties against dishonest behaviour from the corrupted parties. This “dishonest behaviour” can manifest itself in a number of ways; in this chapter we focus on *malicious* adversaries who may arbitrarily deviate from the protocol specification. When considering malicious

adversaries, there are certain undesirable actions that cannot be prevented. Specifically, parties may refuse to participate in the protocol, may substitute their local input (and enter with a different input) and may cease participating in the protocol before it terminates. Essentially, secure protocols limit the adversary to such behaviour only.

Formally, the adversary is modelled by a non-uniform interactive Turing machine: in the computational model this machine is polynomial-time whereas in the information-theoretic model it is unbounded. (We note that by standard arguments, we can assume that the adversary is deterministic.) For simplicity, in this work we consider a *static corruption* model. Therefore, at the beginning of the execution, the adversary is given a set  $I$  of corrupted parties which it controls. That is, the adversary obtains the views of the corrupted parties, and provides them with the messages that they are to send in the execution.

**Security of protocols (informal).** The security of a protocol is analyzed by comparing what an adversary can do in the protocol to what it can do in an ideal scenario that is secure by definition. This is formalized by considering an *ideal* computation involving an incorruptible *trusted party* to whom the parties send their inputs. The trusted party computes the functionality on the inputs and returns to each party its respective output. Loosely speaking, a protocol is secure if any adversary interacting in the real protocol (where no trusted party exists) can do no more harm than if it was involved in the above-described ideal computation. We begin by formally defining this ideal computation.

### 3.2.1 Execution in the Ideal Model

The ideal model differs for each of the definitions. We therefore present each one separately (see Section 3.1.2 for an outline of the different definitions).

**1. Secure computation with unanimous abort and complete fairness:** According to this definition, there are two possible termination cases. In the first case, all parties (including the corrupted parties) abort without receiving output. In the second case, the protocol terminates and all parties receive their prescribed output. As we have mentioned, even assuming a broadcast channel, this definition is only achievable when the number of corrupted parties is less than  $n/2$  (i.e.,  $|I| < n/2$ ). Recall that a malicious party can always substitute its input or refuse to participate. Therefore, the ideal model takes these inherent adversarial behaviours into account; i.e., by giving the adversary the ability to do this also in the ideal model. An ideal execution proceeds as follows:

*Inputs:* Each party obtains its respective input from the input vector  $\bar{x} = (x_1, \dots, x_n)$ .

*Send inputs to trusted party:* An honest party  $P_j$  always sends its input  $x_j$  to the trusted party. The corrupted parties may, depending on their inputs  $\{x_i\}_{i \in I}$ , either abort or send modified values  $x'_i \in \{0, 1\}^{|x_i|}$  to the trusted party. Denote the sequence of inputs obtained by the trusted party by  $\bar{x}' = (x'_1, \dots, x'_n)$  (for an honest party  $P_j$ , it always holds that  $x'_j = x_j$ ).

*Trusted party answers the parties:* In case  $\bar{x}'$  is a valid input sequence, the trusted party computes  $f(\bar{x}')$  and sends  $f_i(\bar{x}')$  to party  $P_i$  for every  $i$ . Otherwise (i.e., in case a corrupted party aborted or sent a non-valid input), the trusted party replies to all parties with a special abort symbol  $\perp$ .

*Outputs:* An honest party always outputs the message that it received from the trusted party and the corrupted parties output nothing. The adversary outputs an arbitrary function of the initial inputs  $\{x_i\}_{i \in I}$  and the messages that the corrupted parties received from the trusted party.

**Definition 3.2.1** (ideal-model computation with unanimous abort and complete fairness): *Let  $f : (\{0, 1\}^*)^n \rightarrow (\{0, 1\}^*)^n$  be an  $n$ -ary functionality, where  $f = (f_1, \dots, f_n)$ , and let  $I \subset [n]$  be such that for every  $i \in I$ , the adversary  $\mathcal{A}$  controls  $P_i$  (this is the set of corrupted parties). Then, the joint execution of  $f$  under  $(\mathcal{A}, I)$  in the ideal model on input vector  $\bar{x} = (x_1, \dots, x_n)$  and auxiliary input  $z$  to  $\mathcal{A}$ , denoted  $\text{IDEAL}_{f, I, \mathcal{A}(z)}^{(1)}(\bar{x})$ , is defined as the output vector of  $P_1, \dots, P_n$  and  $\mathcal{A}$  resulting from the above described ideal process.*

**2. Secure computation with unanimous abort and partial fairness:** As before, a malicious party can always substitute its input or refuse to participate. However, when there are a half or less honest parties, it is not possible to continue computing in the case that the adversary ceases prematurely. Thus, we cannot prevent the “early abort” phenomenon in which the adversary receives its output, whereas the honest parties do not receive theirs (i.e., complete fairness cannot be achieved). Nevertheless, a partial notion of fairness can be achieved. That is, a party  $P_1$  is specified so that if it is honest, then complete fairness is achieved. In contrast, if it is corrupted, then the adversary receives the corrupted parties’ outputs first and then can decide whether or not the honest parties receive output or abort. We note that the abort is unanimous and thus if one honest party aborts, then so do all honest parties. The only difference from the previous definition is in the “trusted party answers remaining parties” stage. An ideal execution proceeds as follows:

*Inputs:* Each party obtains its respective input from the input vector  $\bar{x} = (x_1, \dots, x_n)$ .

*Send inputs to trusted party:* An honest party  $P_j$  always sends its input  $x_j$  to the trusted party. The corrupted parties may, depending on their inputs  $\{x_i\}_{i \in I}$ , either abort or send modified values  $x'_i \in \{0, 1\}^{|x_i|}$  to the trusted party. Denote the sequence of inputs obtained by the trusted party by  $\bar{x}' = (x'_1, \dots, x'_n)$  (for an honest party  $P_j$ , it always holds that  $x'_j = x_j$ ).



*Trusted party answers first party:* In case  $\bar{x}'$  is a valid input sequence, the trusted party computes  $f(\bar{x}')$  and sends  $f_1(\bar{x}')$  to party  $P_1$ . Otherwise (i.e., in case a corrupted party aborted or sent a non-valid input), the trusted party replies to all parties with a special symbol,  $\perp$ .

*Trusted party answers remaining parties:* If the first party is not corrupted (i.e.,  $1 \notin I$ ), then the trusted party sends  $f_j(\bar{x}')$  to party  $P_j$ , for every  $j$ . In case the first party is corrupted, then for every  $i \in I$ , the trusted party sends  $f_i(\bar{x})$  to party  $P_i$  (i.e., the corrupted parties receive their outputs first). Then the corrupted  $P_1$ , depending on the views of all the corrupted parties, instructs the trusted party to either send  $f_j(\bar{x}')$  to  $P_j$  for every  $j \notin I$ , or to send  $\perp$  to  $P_j$  for every  $j \notin I$ .

*Outputs:* An honest party always outputs the message that it received from the trusted party and the corrupted parties output nothing. The adversary outputs an arbitrary function of the initial inputs  $\{x_i\}_{i \in I}$  and the messages that the corrupted parties received from the trusted party.

**Definition 3.2.2** (ideal-model computation with unanimous abort and partial fairness): *Let  $f : (\{0, 1\}^*)^n \rightarrow (\{0, 1\}^*)^n$  be an  $n$ -ary functionality, where  $f = (f_1, \dots, f_n)$ , and let  $I \subset [n]$  be such that for every  $i \in I$ , the adversary  $\mathcal{A}$  controls  $P_i$  (this is the set of corrupted parties). Then, the joint execution of  $f$  under  $(\mathcal{A}, I)$  in the ideal model on input vector  $\bar{x} = (x_1, \dots, x_n)$  and auxiliary input  $z$  to  $\mathcal{A}$ , denoted  $\text{IDEAL}_{f, I, \mathcal{A}(z)}^{(2)}(\bar{x})$ , is defined as the output vector of  $P_1, \dots, P_n$  and  $\mathcal{A}$  resulting from the above described ideal process.*

We note that this is the definition of [G02a] for the case of  $t \geq n/2$ .

**3. Secure computation with unanimous abort and no fairness:** This definition is very similar to the previous one, except that there is no specified party. Rather, the adversary first receives the output of the corrupted parties. Then, it decides whether all the honest parties receive output or they all abort. Formally,

*Inputs:* Each party obtains its respective input from the input vector  $\bar{x} = (x_1, \dots, x_n)$ .

*Send inputs to trusted party:* An honest party  $P_j$  always sends its input  $x_j$  to the trusted party. The corrupted parties may, depending on their inputs  $\{x_i\}_{i \in I}$ , either abort or send modified values  $x'_i \in \{0, 1\}^{|x_i|}$  to the trusted party. Denote the sequence of inputs obtained by the trusted party by  $\bar{x}' = (x'_1, \dots, x'_n)$  (for an honest party  $P_j$ , it always holds that  $x'_j = x_j$ ).

*Trusted party answers adversary:* In case  $\bar{x}'$  is a valid input sequence, the trusted party computes  $f(\bar{x}')$  and sends  $f_i(\bar{x}')$  to party  $P_i$  for every  $i \in I$ . Otherwise (i.e., in case a corrupted party aborted or sent a non-valid input), the trusted party replies to all parties with a special symbol,  $\perp$ .

*Trusted party answers remaining parties:* The adversary, depending on the views of all the corrupted parties, instructs the trusted party to either send  $f_j(\bar{x}')$  to  $P_j$  for every  $j \notin I$ , or to send  $\perp$  to  $P_j$  for every  $j \notin I$ .

*Outputs:* An honest party always outputs the message that it received from the trusted party and the corrupted parties output nothing. The adversary outputs an arbitrary function of the initial inputs  $\{x_i\}_{i \in I}$  and the messages that the corrupted parties received from the trusted party.

**Definition 3.2.3** (ideal-model computation with unanimous abort and no fairness): *Let  $f : (\{0, 1\}^*)^n \rightarrow (\{0, 1\}^*)^n$  be an  $n$ -ary functionality, where  $f = (f_1, \dots, f_n)$ , and let  $I \subset [n]$  be such that for every  $i \in I$ , the adversary  $\mathcal{A}$  controls  $P_i$  (this is the set of corrupted parties). Then, the joint execution of  $f$  under  $(\mathcal{A}, I)$  in the ideal model on input vector  $\bar{x} = (x_1, \dots, x_n)$  and auxiliary input  $z$  to  $\mathcal{A}$ , denoted  $\text{IDEAL}_{f, I, \mathcal{A}(z)}^{(3)}(\bar{x})$ , is defined as the output vector of  $P_1, \dots, P_n$  and  $\mathcal{A}$  resulting from the above described ideal process.*

The above three definitions all relate to the case of secure computation with unanimous abort. We now present the analogous definitions for the case of secure computation with abort. The only difference between the definitions is regarding the “trusted party answers remaining parties” item. In the above definitions all honest parties either receive their output or they receive  $\perp$ . However, here some of these parties may receive their (correct) output and some may receive  $\perp$ . This is where our new definition differs from past ones. We only present definitions for partial and no fairness (complete fairness only makes sense if all parties, including the adversary, either receive their outputs or  $\perp$ ).

**4. Secure computation with abort and partial fairness:** As we have mentioned, the only difference between this definition and the analogous definition with unanimous abort is that if party  $P_1$  is corrupted, then it may designate who does and does not receive output. We repeat only the relevant item:

*Trusted party answers remaining parties:* If the first party is not corrupted (i.e.,  $1 \notin I$ ), then the trusted party sends  $f_j(\bar{x}')$  to party  $P_j$ , for every  $j$ . In case the first party is corrupted, then for every  $i \in I$ , the trusted party sends  $f_i(\bar{x}')$  to  $P_i$  (i.e., the corrupted parties receive their output first). Then the corrupted  $P_1$ , depending on the views of all the corrupted parties, chooses a subset of the honest parties  $J \subseteq [n] \setminus I$  and sends  $J$  to the trusted party. The trusted party then sends  $f_j(\bar{x}')$  to  $P_j$  for every  $j \in J$ , and  $\perp$  to all other honest parties.

**Definition 3.2.4** (ideal-model computation with abort and partial fairness): *Let  $f : (\{0, 1\}^*)^n \rightarrow (\{0, 1\}^*)^n$  be an  $n$ -ary functionality, where  $f = (f_1, \dots, f_n)$ , and let  $I \subset [n]$  be such that for every  $i \in I$ , the adversary  $\mathcal{A}$  controls  $P_i$  (this is the set of corrupted parties). Then, the joint execution of  $f$  under  $(\mathcal{A}, I)$  in the ideal model on input vector  $\bar{x} = (x_1, \dots, x_n)$  and auxiliary input  $z$  to  $\mathcal{A}$ , denoted  $\text{IDEAL}_{f, I, \mathcal{A}(z)}^{(4)}(\bar{x})$ , is defined as the output vector of  $P_1, \dots, P_n$  and  $\mathcal{A}$  resulting from the above described ideal process.*

**5. Secure computation with abort and no fairness:** This definition is the same as Definition 3.2.3 with respect to the fact that the adversary always receives the output of the corrupted parties first. However, as with Definition 3.2.4, the honest parties' abort is not necessarily unanimous. That is, the adversary designates which honest parties receive their output and which receive  $\perp$ . We repeat the only item in which this definition differs from Definition 3.2.3:

*Trusted party answers remaining parties:* The adversary, depending on the views of all the corrupted parties, chooses a subset of the honest parties  $J \subseteq [n] \setminus I$  and sends  $J$  to the trusted party. The trusted party then sends  $f_j(\bar{x}')$  to  $P_j$  for every  $j \in J$ , and  $\perp$  to all other honest parties.

**Definition 3.2.5** (ideal-model computation with abort and no fairness): *Let  $f : (\{0, 1\}^*)^n \rightarrow (\{0, 1\}^*)^n$  be an  $n$ -ary functionality, where  $f = (f_1, \dots, f_n)$ , and let  $I \subset [n]$  be such that for every  $i \in I$ , the adversary  $\mathcal{A}$  controls  $P_i$  (this is the set of corrupted parties). Then, the joint execution of  $f$  under  $(\mathcal{A}, I)$  in the ideal model on input vector  $\bar{x} = (x_1, \dots, x_n)$  and auxiliary input  $z$  to  $\mathcal{A}$ , denoted  $\text{IDEAL}_{f, I, \mathcal{A}(z)}^{(5)}(\bar{x})$ , is defined as the output vector of  $P_1, \dots, P_n$  and  $\mathcal{A}$  resulting from the above described ideal process.*

### 3.2.2 Execution in the Real Model

We now define a real model execution. In the real model, the parties execute the protocol in a *synchronous* network with *rushing*. That is, the execution proceeds in rounds: each round consists of a send phase (where parties send their message from this round) followed by a receive phase (where they receive messages from other parties). This means that the messages sent by an honest party in a given round depend on the messages that it received in previous rounds only. However, the adversary can compute its messages in a given round based on the messages that it receives from the honest parties in the same round. The term “rushing” refers to this additional adversarial capability.

In this work, we consider a scenario where the parties are connected via a fully connected point-to-point network (and there is no broadcast channel). We refer to this model as the *point-to-point model* (in contrast to the *broadcast model* where the parties are given access to a physical broadcast channel in addition to the point-to-point network). The communication lines between parties are assumed to be ideally authenticated and private (and thus the adversary cannot modify or read messages sent between two honest parties).<sup>6</sup> Furthermore, the delivery of messages between honest parties is guaranteed.

<sup>6</sup> We note that when the parties are assumed to be computationally bounded, privacy can be achieved over authenticated channels by using public-key encryption. Therefore, in such a setting, the requirement that the channels be private is not essential. However, we include it for simplicity.

Finally, we note that we do not assume any trusted preprocessing phase (that can be used to setup a public-key infrastructure, for example).<sup>7</sup>

Throughout a real execution, the honest parties all follow the instructions of the prescribed protocol, whereas the corrupted parties receive their (arbitrary) instructions from the adversary. Then, at the conclusion of the execution, the honest parties output their prescribed output from the protocol, the corrupted parties output nothing and the adversary outputs an arbitrary function of its view of the computation (which contains the views of all the corrupted parties). Without loss of generality, we assume that the adversary always outputs its view (and not some function of it). Formally,

**Definition 3.2.6** (real-model execution): *Let  $f$  be an  $n$ -ary functionality and let  $\Pi$  be a multi-party protocol for computing  $f$ . Furthermore, let  $I \subset [n]$  be such that for every  $i \in I$ , the adversary  $\mathcal{A}$  controls  $P_i$  (this is the set of corrupted parties). Then, the joint execution of  $\Pi$  under  $(\mathcal{A}, I)$  in the real model on input vector  $\bar{x} = (x_1, \dots, x_n)$  and auxiliary input  $z$  to  $\mathcal{A}$ , denoted  $\text{REAL}_{\Pi, I, \mathcal{A}(z)}(\bar{x})$ , is defined as the output vector of  $P_1, \dots, P_n$  and  $\mathcal{A}$  resulting from the protocol interaction, where for every  $i \in I$ , party  $P_i$  computes its messages according to  $\mathcal{A}$ , and for every  $j \notin I$ , party  $P_j$  computes its messages according to  $\Pi$ .*

**On synchronous versus asynchronous networks.** As we have mentioned above, the results in this chapter are presented in the synchronous network model. However, this is for the sake of simplicity only. In fact, when output delivery is *not* guaranteed, asynchronous communication does not present any additional hardship. Specifically, the honest parties should merely wait until receiving all round  $i$  messages before sending their message for round  $i+1$ . This ensures that the only difference between a synchronous and an asynchronous execution is the ability of the adversary to prevent the parties from receiving output. When output delivery is not guaranteed, this is anyway allowed.

---

<sup>7</sup> One can argue that achieving authenticated and private channels in practice essentially requires a trusted pre-processing phase for setting up a public-key infrastructure. Therefore, there is no reason not to utilize this preprocessing phase in the secure multi-party computation as well. In such a case, the preprocessing phase could be used in order to implement authenticated Byzantine Agreement (and thereby achieve secure broadcast for any number of corrupted parties). However, we claim that the issue of achieving “secure communication channels” should be separated from the issue of achieving “secure broadcast”. An example of why this is important was demonstrated in Chapter 2, where we showed that authenticated Byzantine Agreement does not compose (in parallel or concurrently) when  $2/3$  or less of the parties are honest. In contrast, secure channels can be achieved without any limitation on the protocol using them [CK02]; in particular, without restrictions on composability and the number of corrupted parties.

### 3.2.3 Security as Emulation of a Real Execution in the Ideal Model

Having defined the ideal and real models, we can now define security of protocols. Loosely speaking, the definition asserts that the adversary can do no more harm in a real protocol execution than in the ideal model (where security trivially holds). This is formulated by saying that adversaries in the ideal model are able to simulate adversaries in an execution of a secure real-model protocol. The definition of security comes in two flavours. In the first, we consider polynomial-time bounded adversaries, and require that the simulation be such that the real-model and ideal-model output distributions are computationally indistinguishable. In the second, we consider unbounded adversaries and require that the simulation be such that the output distributions of the two executions are statistically close.

**Definition 3.2.7** (computational security): *Let  $f$  and  $\Pi$  be as above. We say that protocol  $\Pi$  is a protocol for computational  $t$ -secure computation with unanimous abort (resp., with abort) and with complete fairness (resp., with partial fairness or with no fairness), if for every non-uniform polynomial-time adversary  $\mathcal{A}$  for the real model, there exists a non-uniform probabilistic (expected) polynomial-time adversary  $\mathcal{S}$  for the ideal model, such that for every  $I \subset [n]$  with  $|I| < t$ ,*

$$\left\{ \text{IDEAL}_{f,I,\mathcal{S}(z)}^{(\alpha)}(\bar{x}) \right\}_{k \in \mathbb{N}, \bar{x} \in (\{0,1\}^k)^n, z \in \{0,1\}^{\text{poly}(k)}} \\ \stackrel{c}{=} \left\{ \text{REAL}_{\Pi,I,\mathcal{A}(z)}(\bar{x}) \right\}_{k \in \mathbb{N}, \bar{x} \in (\{0,1\}^k)^n, z \in \{0,1\}^{\text{poly}(k)}}$$

where the value of  $\alpha \in \{1, 2, 3, 4, 5\}$  depends on whether secure computation with unanimous abort or with abort is being considered, and whether complete fairness, partial fairness or no fairness is required.

**Definition 3.2.8** (information-theoretic security): *Let  $f$  and  $\Pi$  be as above. We say that protocol  $\Pi$  is a protocol for information-theoretic  $t$ -secure computation with unanimous abort (resp., with abort) and with complete fairness (resp., with partial fairness or with no fairness), if for every non-uniform adversary  $\mathcal{A}$  for the real model, there exists a non-uniform probabilistic adversary  $\mathcal{S}$  for the ideal model such that for every  $I \subset [n]$  with  $|I| < t$ ,*

$$\left\{ \text{IDEAL}_{f,I,\mathcal{S}(z)}^{(\alpha)}(\bar{x}) \right\}_{k \in \mathbb{N}, \bar{x} \in (\{0,1\}^k)^n, z \in \{0,1\}^*} \\ \stackrel{s}{=} \left\{ \text{REAL}_{\Pi,I,\mathcal{A}(z)}(\bar{x}) \right\}_{k \in \mathbb{N}, \bar{x} \in (\{0,1\}^k)^n, z \in \{0,1\}^*}$$

where the value of  $\alpha \in \{1, 2, 3, 4, 5\}$  depends on whether secure computation with unanimous abort or with abort is being considered, and whether complete fairness, partial fairness or no fairness is required.

### 3.3 Broadcast with Abort

In this section, we present a weak variant of the Byzantine Generals problem, that we call “*broadcast with abort*”. The main idea is to weaken both the agreement and validity requirements so that some parties may output the broadcast value  $x$  while others output  $\perp$ . Formally,

**Definition 3.3.1** (broadcast with abort): *Let  $P_1, \dots, P_n$ , be  $n$  parties and let  $P_1$  be the dealer with input  $x$ . In addition, let  $\mathcal{A}$  be an adversary who controls up to  $t$  of the parties (which may include  $P_1$ ). A protocol solves the broadcast with abort problem, tolerating  $t$  corruptions, if for any adversary  $\mathcal{A}$  the following three properties hold:*

1. Agreement: *If an honest party outputs  $x'$ , then all honest parties output either  $x'$  or  $\perp$ .*
2. Validity: *If  $P_1$  is honest, then all honest parties output either  $x$  or  $\perp$ .*
3. Non-triviality: *If all parties are honest, then all parties output  $x$ .*

(The non-triviality requirement is needed to rule out a protocol in which all parties simply output  $\perp$  and halt.) We now present a simple protocol that solves the broadcast with abort problem for *any*  $t$ . As we will see later, despite its simplicity, this protocol suffices for obtaining secure computation with abort.

**Protocol 3.3.2** (broadcast with abort):

- **Input:**  $P_1$  has a value  $x$  to broadcast.

- **The Protocol:**

1.  $P_1$  sends  $x$  to all parties.
2. Denote by  $x^i$  the value received by  $P_i$  from  $P_1$  in the previous round. Then, every party  $P_i$  (for  $i > 1$ ) sends its value  $x^i$  to all other parties.
3. Denote the value received by  $P_i$  from  $P_j$  in the previous round by  $x_j^i$  (recall that  $x^i$  denotes the value  $P_i$  received from  $P_1$  in the first round). Then,  $P_i$  outputs  $x^i$  if this is the only value that it saw (i.e., if for every  $x_j^i$  that  $P_i$  receives, it holds that  $x_j^i = x^i$ ). Otherwise, it outputs  $\perp$ .  
We note that if  $P_i$  did not receive any value in the first round, then it always outputs  $\perp$ .

We now prove that Protocol 3.3.2 is secure, for any number of corrupted parties. That is,

**Proposition 3.3.3** *Protocol 3.3.2 solves the broadcast with abort problem, and tolerates any  $t < n$  corruptions.*

**Proof.** The fact that the non-triviality condition is fulfilled is immediate. We now prove the other two conditions:

1. *Agreement*: Let  $P_i$  be an honest party such that  $P_i$  outputs a value  $x'$ . Then, it must be that  $P_i$  received  $x'$  from  $P_1$  in the first round (i.e.,  $x^i = x'$ ). Therefore,  $P_i$  sent this value to all other parties in the second round. Now, a party  $P_j$  will output  $x^j$  if this is the only value that it saw during the execution. However, as we have just seen,  $P_j$  definitely saw  $x'$  in the second round. Thus,  $P_j$  will only output  $x^j$  if  $x^j = x'$ . Furthermore, if  $P_j$  does not output  $x^j$ , then it outputs  $\perp$ .
2. *Validity*: If  $P_1$  is honest, then all parties receive  $x$  in the first round. Therefore, they will only output  $x$  or  $\perp$ .



### 3.3.1 Strengthening Broadcast with Abort

A natural question to ask is whether or not we can strengthen Definition 3.3.1 in one of the following two ways (and still obtain a protocol that tolerates  $t \geq n/3$  corruptions):

1. *Strengthen the agreement requirement*: If an honest party outputs a value  $x'$ , then all honest parties output  $x'$ . (Note that the validity requirement remains unchanged.)
2. *Strengthen the validity requirement*: If  $P_1$  is honest, then all honest parties output  $x$ . (Note that the agreement requirement remains unchanged.)

It is easy to see that the above strengthening of the agreement requirement results in the definition of weak Byzantine Generals. (The validity and non-triviality requirements combined together are equivalent to the validity requirement of weak Byzantine Generals.) Therefore, there do not exist *deterministic* protocols for the case of  $t \geq n/3$  [L83]. For what can be done if one utilizes randomized protocols, see the section on recent *related work* (Section 3.1.3). Regarding the strengthening of the validity requirement, the resulting definition implies a problem known as “Crusader Agreement” (outputting  $\perp$  can be interpreted as “explicitly” knowing that the dealer is corrupted). This was shown to be unachievable for any  $t \geq n/3$  in [D82] (we note that this lower bound holds for randomized protocols as well). We therefore conclude that the “broadcast with abort” requirements cannot be strengthened in either of the above two ways (for deterministic protocols), without incurring a  $t < n/3$  lower bound.

## 3.4 Secure Computation with Abort and No Fairness

In this section, we show that any protocol for secure computation (with unanimous abort and any level of fairness) that uses a broadcast channel can be “compiled” into a protocol for the point-to-point network that achieves secure computation with abort and no fairness. Furthermore, the fault tolerance of

the compiled protocol is the same as the original one. Actually, we assume that the protocol is such that all parties terminate in the same round. We say that such a protocol has *simultaneous termination*. Without loss of generality, we also assume that all parties generate their output in the last round. The result of this section is formally stated in the following theorem:

**Theorem 3.4.1** *There exists a (polynomial-time) protocol compiler that takes any protocol  $\Pi$  (with simultaneous termination) for the broadcast model, and outputs a protocol  $\Pi'$  for the point-to-point model such that the following holds: If  $\Pi$  is a protocol for information-theoretic (resp., computational)  $t$ -secure computation with unanimous abort and any level of fairness, then  $\Pi'$  is a protocol for information-theoretic (resp., computational)  $t$ -secure computation with abort and no fairness.*

Combining Theorem 3.4.1 with known protocols (specifically, the protocols of [RB89] and [GMW87]<sup>8</sup>), we obtain the following corollaries:

**Corollary 3.4.2** (information-theoretic security – compilation of [RB89]): *For any probabilistic polynomial-time  $n$ -ary functionality  $f$ , there exists a protocol in the point-to-point model, for the information-theoretic  $n/2$ -secure computation of  $f$  with abort and no fairness.*

**Corollary 3.4.3** (computational security – compilation of [GMW87]): *For any probabilistic polynomial-time  $n$ -ary functionality  $f$ , there exists a protocol in the point-to-point model, for the computational  $t$ -secure computation of  $f$  with abort and no fairness, for any  $t$ .*

We now proceed to prove Theorem 3.4.1.

**Proof of Theorem 3.4.1.** Intuitively, we construct a protocol for the point-to-point model from a protocol for the broadcast model, by having the parties in the point-to-point network simulate the broadcast channel. This simulation is not carried out using Byzantine Generals because this is not possible for  $t \geq n/3$ . Rather, it is simulated using a protocol for “broadcast with abort”, which as we have seen can be achieved for any  $t$ . Recall that in such a protocol, either the correct value is delivered to all parties, or some parties output  $\perp$ . The idea is to halt the computation in the case that any honest party receives  $\perp$  from a broadcast execution. The point at which the computation halts dictates which parties (if any) receive output. The key point is that if no honest party receives  $\perp$ , then the broadcast with abort protocol perfectly simulates a broadcast channel. Therefore, the result is that the original protocol (for the broadcast channel) is simulated perfectly until the point that it may prematurely halt.

<sup>8</sup> Both the [RB89] and [GMW87] protocols have simultaneous termination



### Components of the compiler:

1. **Broadcast with abort executions:** Each broadcast of the original protocol (using the assumed broadcast channel) is replaced with an execution of the broadcast with abort protocol.
2. **Blank rounds:** Following each broadcast with abort execution, a blank round is added in which no protocol messages are sent. Rather, these blank rounds are used to enable the parties to notify each other in the case that they abort. Specifically, if a party receives  $\perp$  in a broadcast with abort execution, then it sends  $\perp$  to all parties in the blank round that immediately follows and halts. Likewise, if a party receives  $\perp$  in a blank round, then it sends  $\perp$  to all parties in the next blank round and halts.

Thus each round of the original protocol is transformed into 3 rounds in the compiled protocol (2 rounds for broadcast with abort and an additional blank round). We now formally define the protocol compiler:

**Construction 3.4.4** (protocol compiler): *Given a protocol  $\Pi$ , the compiler produces a protocol  $\Pi'$ . The specification of protocol  $\Pi'$  is as follows:*

- *The parties use broadcast with abort in order to emulate each broadcast message of protocol  $\Pi$ . Each round of  $\Pi$  is expanded into 3 rounds in  $\Pi'$ : broadcast with abort is run in the first 2 rounds, and the third round is a blank round. Point-to-point messages of  $\Pi$  are sent unmodified in  $\Pi'$ . More exactly, the parties emulate  $\Pi$  according to the following instructions (for simplicity, we count the first round of  $\Pi$  to be round 0):*
  1. **Broadcasting messages:** *Let  $P_i$  be a party who is supposed to broadcast a message  $m$  in the  $j^{\text{th}}$  round of  $\Pi$ . Then, in the  $j^{\text{th}}$  broadcast simulation of  $\Pi'$  (i.e., in rounds  $3j$  and  $3j + 1$  of  $\Pi'$ ), all parties run an execution of broadcast with abort in which  $P_i$  plays the dealer role and sends  $m$ .*
  2. **Sending point-to-point messages:** *Any message that  $P_i$  is supposed to send to  $P_j$  over the point-to-point network in the  $j^{\text{th}}$  round of  $\Pi$  is sent by  $P_i$  to  $P_j$  over the point-to-point network in round  $3j$  of  $\Pi'$ .*
  3. **Receiving messages:** *For each message that party  $P_i$  is supposed to receive from a broadcast in  $\Pi$ , party  $P_i$  participates in an execution of broadcast with abort as a receiver. If its output from this execution is a message  $m$ , then it appends  $m$  to its view (to be used for determining its later steps according to  $\Pi$ ).  
If it receives  $\perp$  from this execution, then it sends  $\perp$  to all parties in the next round (i.e., in the blank round following the execution of broadcast with abort), and halts immediately thereafter.*
  4. **Blank rounds:** *If a party  $P_i$  receives  $\perp$  in a blank round, then it sends  $\perp$  to all parties in the next blank round and halts. In the 2 rounds preceding the next blank round, Party  $P_i$  does not send any point-to-point messages or messages belonging to a broadcast execution. (We*

*note that if this blank round is the last round of the execution, then  $P_i$  simply halts.)*

5. **Output:** *If a party  $P_i$  received  $\perp$  at any point in the execution (in an execution of broadcast with abort or in a blank round), then it outputs  $\perp$ . Otherwise, it outputs the value specified by  $\Pi$ .*

By the assumption in the theorem,  $\Pi$  is  $t$ -secure with unanimous abort (and any level of fairness). In order to prove that the compiled protocol  $\Pi'$  is  $t$ -secure with abort and no fairness, we first define a different transformation of  $\Pi$  to  $\tilde{\Pi}$  which is a hybrid protocol between  $\Pi$  and  $\Pi'$ . In particular,  $\tilde{\Pi}$  is still run in the broadcast model. However, it provides the adversary with the additional ability of prematurely halting honest parties. We now define the hybrid protocol  $\tilde{\Pi}$  and show that it is  $t$ -secure with abort and no fairness:

**Lemma 3.4.5** *Let  $\Pi$  be a protocol in the broadcast model that is computational (resp., information-theoretic)  $t$ -secure with unanimous abort and any level of fairness. Then, define protocol  $\tilde{\Pi}$  (also for the broadcast model) as follows:*

1. *Following each round of  $\Pi$ , add a blank round.*
2. *If in a blank round,  $P_i$  receives a  $\perp$  message, then  $P_i$  sends  $\perp$  to  $P_j$  for all  $j \neq i$  in the next blank round and halts.  $P_i$  also does not broadcast any message or send any point-to-point messages in the next round of  $\Pi$  (before the blank round where it sends all the  $\perp$  messages).*
3. *Apart from the above, the parties follow the instructions of  $\Pi$ .*
4. **Output:** *If a party  $P_i$  received  $\perp$  in any blank round, then it outputs  $\perp$ . Otherwise, it outputs the value specified by  $\Pi$ .*

*Then,  $\tilde{\Pi}$  is computational (resp., information-theoretic)  $t$ -secure with abort and no fairness.*

**Proof.** We prove this theorem for the case that  $\Pi$  is computationally  $t$ -secure with unanimous abort and partial fairness. The other cases (information-theoretic security and security with complete fairness or no fairness) are proved in a similar way. Let  $\tilde{\mathcal{A}}$  be a real-model adversary attacking  $\tilde{\Pi}$ . Our aim is to construct an ideal-model simulator  $\tilde{\mathcal{S}}$  for  $\tilde{\mathcal{A}}$ . In order to do this, we must use the fact that for any adversary  $\mathcal{A}$  attacking protocol  $\Pi$ , there exists an ideal-model simulator  $\mathcal{S}$ . Unfortunately we cannot apply  $\mathcal{S}$  to  $\tilde{\mathcal{A}}$  because  $\mathcal{S}$  is a simulator for protocol  $\Pi$  and  $\tilde{\mathcal{A}}$  participates in protocol  $\tilde{\Pi}$ . We therefore first construct an adversary  $\mathcal{A}$  that attacks  $\Pi$  from the adversary  $\tilde{\mathcal{A}}$  that attacks  $\tilde{\Pi}$ . The construction of  $\mathcal{A}$  is such that the output distribution of an execution of  $\Pi$  with  $\mathcal{A}$  is very similar to the output distribution of an execution of  $\tilde{\Pi}$  with  $\tilde{\mathcal{A}}$ . Having constructed  $\mathcal{A}$ , it is then possible to apply the simulator  $\mathcal{S}$  that we know is guaranteed to exist. We therefore obtain a simulator  $\tilde{\mathcal{S}}$  for  $\tilde{\mathcal{A}}$  by first “transforming”  $\tilde{\mathcal{A}}$  into  $\mathcal{A}$  and then applying  $\mathcal{S}$ . As we will show, the resulting simulator  $\tilde{\mathcal{S}}$  is as required for  $\tilde{\Pi}$ . Details follow.

We begin by defining the adversary  $\mathcal{A}$  who attacks  $\Pi$ . Adversary  $\mathcal{A}$  internally incorporates  $\tilde{\mathcal{A}}$  and has internal communication with  $\tilde{\mathcal{A}}$  and external communication with the honest parties executing  $\Pi$ .

#### Adversary $\mathcal{A}$ for $\Pi$ :

- **Input:**  $\mathcal{A}$  receives an auxiliary input  $z$ , an input sequence  $\{x_i\}_{i \in I}$  and a series of random-tapes  $\{r_i\}_{i \in I}$ . (Recall that  $\mathcal{A}$  controls all the parties in the set  $I$ . Thus, corrupted party  $P_i$ 's input and random-tape equal  $x_i$  and  $r_i$ , respectively.)
- **Execution:**
  1. *Invoke  $\tilde{\mathcal{A}}$ :*  $\mathcal{A}$  begins by invoking  $\tilde{\mathcal{A}}$  upon auxiliary input  $z$ , input sequence  $\{x_i\}_{i \in I}$  and random-tapes  $\{r_i\}_{i \in I}$ .
  2. *Emulation before  $\tilde{\mathcal{A}}$  sends any  $\perp$  messages:*  $\mathcal{A}$  internally passes to  $\tilde{\mathcal{A}}$  all the messages that it externally receives from the honest parties (through broadcast or point-to-point communication). Likewise,  $\mathcal{A}$  externally broadcasts in  $\Pi$  any message that  $\tilde{\mathcal{A}}$  broadcasts in  $\tilde{\Pi}$ , and  $\mathcal{A}$  externally sends  $P_j$  in  $\Pi$  any message that  $\tilde{\mathcal{A}}$  sends  $P_j$  in  $\tilde{\Pi}$ .
  3. *Emulation after  $\tilde{\mathcal{A}}$  sends a  $\perp$  message:* Once  $\tilde{\mathcal{A}}$  sends a  $\perp$  message in an execution of  $\tilde{\Pi}$ , the honest parties in  $\Pi$  and  $\tilde{\Pi}$  may behave differently (in particular, a party receiving  $\perp$  may continue to send messages in  $\Pi$ , whereas it would halt in  $\tilde{\Pi}$ ). Therefore,  $\mathcal{A}$  filters messages sent by honest parties in  $\Pi$  so that  $\tilde{\mathcal{A}}$ 's view equals what it would in an execution of  $\tilde{\Pi}$ . That is:  
 In the round of  $\Pi$  following the first  $\perp$  message sent by  $\tilde{\mathcal{A}}$ , adversary  $\mathcal{A}$  forwards to  $\tilde{\mathcal{A}}$  only the point-to-point and broadcast messages sent by a party  $P_j$  who did not receive  $\perp$  from  $\tilde{\mathcal{A}}$  in the previous (simulated) blank round of  $\tilde{\Pi}$ . Furthermore,  $\mathcal{A}$  simulates for  $\tilde{\mathcal{A}}$  the sending of all the  $\perp$  messages that would be sent in the next blank round of  $\tilde{\Pi}$  (if one exists), and halts.
  4. *Output:*  $\mathcal{A}$  outputs whatever  $\tilde{\mathcal{A}}$  does.

Before proceeding, we show that the only difference between an execution of  $\Pi$  with  $\mathcal{A}$ , and  $\tilde{\Pi}$  with  $\tilde{\mathcal{A}}$ , is that some additional honest parties may output  $\perp$  in the execution of  $\tilde{\Pi}$ . That is, we claim that the joint distribution of the outputs of all parties not outputting  $\perp$  and the adversary is identical in  $\Pi$  and  $\tilde{\Pi}$ . We begin with some notation:

- Let  $\text{REAL}_{\Pi, I, \mathcal{A}(z)}(\bar{x}, \bar{r})$  be the global output of an execution of  $\Pi$  with adversary  $\mathcal{A}$  with auxiliary input  $z$ , inputs  $\bar{x}$ , and random-tapes  $\bar{r}$  (i.e.,  $\bar{r} = (r_1, \dots, r_n)$  where  $P_i$  receives random-tape  $r_i$ ). (Thus,  $\text{REAL}_{\Pi, I, \mathcal{A}(z)}(\bar{x}) = \{\text{REAL}_{\Pi, I, \mathcal{A}(z)}(\bar{x}, U_{|\bar{r}|})\}$ .)
- For any subset  $J \subseteq [n]$ , denote by  $\text{REAL}_{\Pi, I, \mathcal{A}(z)}(\bar{x}, \bar{r})|_J$ , the *restriction* of  $\text{REAL}_{\Pi, I, \mathcal{A}(z)}(\bar{x}, \bar{r})$  to the outputs of  $\mathcal{A}$  and all parties  $P_j$  for  $j \in J$ . We stress that  $\mathcal{A}$ 's output is included in this restriction.

- In any execution of  $\tilde{\Pi}$ , it is possible to divide the parties into those who output  $\perp$  and those who do not output  $\perp$ . We note that the set of parties outputting  $\perp$  is chosen by the adversary  $\tilde{\mathcal{A}}$  and is dependent solely on the auxiliary input  $z$  and the parties' inputs  $\bar{x}$  and random-tapes  $\bar{r}$ . We denote by  $J = J_{\tilde{\mathcal{A}}(z)}(\bar{x}, \bar{r})$  the set of parties who do not output  $\perp$  in an execution of  $\tilde{\Pi}$  with adversary  $\tilde{\mathcal{A}}$  (and where the auxiliary input is  $z$  and the parties' inputs and random-tapes are  $\bar{x}$  and  $\bar{r}$ ). (Notice that  $J_{\tilde{\mathcal{A}}}$  is a deterministic function depending only on  $z$ ,  $\bar{x}$  and  $\bar{r}$ . This is because, without loss of generality, the adversary  $\tilde{\mathcal{A}}$  is deterministic.) We also denote by  $J_{\tilde{\mathcal{A}}(z)}(\bar{x})$  a random variable taking values over  $J_{\tilde{\mathcal{A}}(z)}(\bar{x}, \bar{r})$  for uniformly distributed  $\bar{r}$ .

We now consider the joint distribution of the outputs of the adversary and the parties in  $J_{\tilde{\mathcal{A}}(z)}(\bar{x}, \bar{r})$  (i.e., those not outputting  $\perp$ ). We claim that these distributions are identical in  $\Pi$  with  $\mathcal{A}$  and in  $\tilde{\Pi}$  with  $\tilde{\mathcal{A}}$ . Using the above notation, we claim that for every adversary  $\tilde{\mathcal{A}}$  and set of corrupted parties  $I$ , for all auxiliary inputs  $z$ , and for all input and random-tape sequences  $\bar{x}$  and  $\bar{r}$ ,

$$\text{REAL}_{\Pi, I, \mathcal{A}(z)}(\bar{x}, \bar{r})|_{J_{\tilde{\mathcal{A}}(z)}(\bar{x}, \bar{r})} = \text{REAL}_{\tilde{\Pi}, I, \tilde{\mathcal{A}}(z)}(\bar{x}, \bar{r})|_{J_{\tilde{\mathcal{A}}(z)}(\bar{x}, \bar{r})} \quad (3.1)$$

where  $\mathcal{A}$  is as defined above. We now prove Eq. (3.1). First, it is clear that  $\tilde{\mathcal{A}}$ 's view in a real execution of  $\tilde{\Pi}$  is identical to its view in the simulation by  $\mathcal{A}$ . Therefore,  $\mathcal{A}$ 's output in  $\Pi$  equals  $\tilde{\mathcal{A}}$ 's output in  $\tilde{\Pi}$ . Next, notice that if there exists an honest party that does not output  $\perp$  in  $\tilde{\Pi}$ , then it must be that  $\perp$  messages were sent in the last blank round only. However, this means that any honest party not receiving such a message has an identical view in  $\Pi$  and  $\tilde{\Pi}$ . Therefore, the outputs of all such parties are identical in  $\Pi$  and  $\tilde{\Pi}$ . Eq. (3.1) follows. We stress that the parties who output  $\perp$  in  $\tilde{\Pi}$  may have very different outputs in  $\Pi$  (and in particular may output their prescribed outputs). Nevertheless, at this stage we are only interested in those parties not outputting  $\perp$ .

We now proceed to construct a simulator  $\tilde{\mathcal{S}}$  for  $\tilde{\mathcal{A}}$ . Intuitively,  $\tilde{\mathcal{S}}$  works by using the simulator  $\mathcal{S}$  for  $\mathcal{A}$ , where  $\mathcal{A}$  is derived from  $\tilde{\mathcal{A}}$  as above. Recall that  $\mathcal{A}$  is an adversary for the secure protocol  $\Pi$ , and thus a simulator  $\mathcal{S}$  is guaranteed to exist for  $\mathcal{A}$ .

**Simulator  $\tilde{\mathcal{S}}$ :** First consider an adversary  $\mathcal{A}$  for  $\Pi$  constructed from the adversary  $\tilde{\mathcal{A}}$  as described above. By the security requirements of  $\Pi$ , for every adversary  $\mathcal{A}$  there exists an ideal-model simulator  $\mathcal{S}$  for  $\mathcal{A}$ . Simulator  $\tilde{\mathcal{S}}$  for  $\tilde{\mathcal{A}}$  works by emulating an ideal execution for this  $\mathcal{S}$ . This involves emulating the interaction that  $\mathcal{S}$  has with its trusted third party. Recall that  $\tilde{\mathcal{S}}$  works in an ideal model for secure computation with abort and no fairness (i.e., according to Definition 3.2.5), whereas  $\mathcal{S}$  works in an ideal model for secure computation with unanimous abort and partial fairness (i.e., according to Definition 3.2.2). Thus, essentially  $\tilde{\mathcal{S}}$  has “more power” than  $\mathcal{S}$ . Also, recall

that where partial fairness holds, there are two cases depending on whether or not  $P_1$  is corrupted. If  $P_1$  is not corrupted, then essentially all parties receive output at the same time. If  $P_1$  is corrupted, then the adversary receives the corrupted parties' outputs first and then decides whether the honest parties all receive output or all abort.

We now describe the simulator. Let  $\tilde{\mathcal{S}}$ 's auxiliary input be  $z$  and its input be a set of values  $\{x_i\}_{i \in I}$ . Furthermore, let  $r$  be the contents of  $\tilde{\mathcal{S}}$ 's random tape. Then,  $\tilde{\mathcal{S}}$  sets  $\mathcal{S}$ 's random tape to  $r$  and internally invokes  $\mathcal{S}$  upon auxiliary input  $z$  and input series  $\{x_i\}_{i \in I}$ . Next,  $\tilde{\mathcal{S}}$  works as an intermediary between  $\mathcal{S}$  and the trusted party. That is,  $\tilde{\mathcal{S}}$  obtains the input values  $\{x'_i\}_{i \in I}$  sent by  $\mathcal{S}$  and externally sends these same values to the trusted party. Once  $\tilde{\mathcal{S}}$  forwards these inputs to the trusted party, it receives back all the corrupted parties' outputs (recall that  $\tilde{\mathcal{S}}$  interacts in an ideal model with no fairness).  $\tilde{\mathcal{S}}$  then forwards these outputs to  $\mathcal{S}$ . We distinguish two cases:

1.  $P_1$  is not corrupted: in this case,  $\mathcal{S}$  concludes its execution at this point, outputting some value.
2.  $P_1$  is corrupted: in this case,  $\mathcal{S}$  first instructs the trusted party to either send all the honest parties their outputs or send them all  $\perp$ .  $\mathcal{S}$  then concludes, outputting some value.

$\tilde{\mathcal{S}}$  ignores the instruction sent to the trusted party in the second case, and sets its output to be whatever  $\mathcal{S}$  output. It remains to define the set  $J$  that  $\tilde{\mathcal{S}}$  sends to the trusted party in the “trusted party answers remaining parties” stage of its ideal execution (recall that all honest parties in  $J$  receive their output and all others receive  $\perp$ ). First notice that the string output by  $\mathcal{S}$  is computationally indistinguishable to  $\mathcal{A}$ 's output from a real execution. However, by the definition of  $\mathcal{A}$ , this output contains  $\tilde{\mathcal{A}}$ 's view of an execution of  $\tilde{\Pi}$ . Furthermore,  $\tilde{\mathcal{A}}$ 's view fully defines which honest parties in an execution of  $\tilde{\Pi}$  output  $\perp$  and which receive their output. In particular, if  $\tilde{\mathcal{A}}$  sent a  $\perp$  message before the last blank round, then all honest parties abort (and  $J = \phi$ ). Otherwise, all parties receive output except for those receiving  $\perp$  messages in the last blank round. Therefore,  $\tilde{\mathcal{S}}$  examines this view and defines the set  $J$  accordingly.<sup>9</sup> Once  $J$  is defined,  $\tilde{\mathcal{S}}$  sends it to the trusted party and halts. This completes the description of  $\tilde{\mathcal{S}}$ .

We now wish to show that the output of an ideal execution with abort and no fairness with adversary  $\tilde{\mathcal{S}}$  is computationally indistinguishable to the output of a real execution of  $\tilde{\Pi}$  with  $\tilde{\mathcal{A}}$ . We begin by showing an analog to Eq. (3.1) in the ideal model. That is, we show that the outputs of parties not outputting  $\perp$  in an execution of  $\tilde{\Pi}$  are the same in an ideal execution (by Def. 2) with  $\mathcal{S}$  and in an ideal execution (by Def. 5) with  $\tilde{\mathcal{S}}$ . Before claiming this formally,

<sup>9</sup> This is the point in the proof where we need the original protocol to be such that all parties terminate simultaneously. Actually, it suffices that each party defines its output at a fixed and predetermined point. However, in such a case, simultaneous termination can anyway be achieved.

we introduce the following notation. Let  $\text{IDEAL}_{f,I,\tilde{S}(z,r)}^{(\alpha)}(\bar{x})$  denote the output of an ideal execution (by Definition  $\alpha$ ) with input series  $\bar{x}$  and where the ideal model adversary  $\tilde{S}$ 's auxiliary input and random tape equal  $z$  and  $r$ , respectively. Furthermore, for any subset  $J \subseteq [n]$ , let  $\text{IDEAL}_{f,I,\tilde{S}(z,r)}^{(\alpha)}(\bar{x})|_J$  be the restriction of  $\text{IDEAL}_{f,I,\tilde{S}(z,r)}^{(\alpha)}(\bar{x})$  to the output of  $\tilde{S}$  and the parties in  $J$ . Finally, let  $J = J_{\tilde{S}(z,r)}(\bar{x})$  be the set of parties in the ideal execution with  $\tilde{S}$  who do not output  $\perp$  (this set is a deterministic function of  $z$ ,  $r$  and  $\bar{x}$ ), and let  $J_{\tilde{S}(z)}(\bar{x})$  denote the associated random variable when  $r$  is chosen uniformly. Now, we claim that for every set  $I$ , every auxiliary input  $z$  and random tape  $r$  (for  $S$  or  $\tilde{S}$ ), and every set of inputs  $\bar{x}$ ,

$$\text{IDEAL}_{f,I,\tilde{S}(z,r)}^{(5)}(\bar{x})|_{J_{\tilde{S}(z,r)}(\bar{x})} = \text{IDEAL}_{f,I,S(z,r)}^{(2)}(\bar{x})|_{J_{\tilde{S}(z,r)}(\bar{x})} \quad (3.2)$$

where  $S$  and  $\tilde{S}$  are invoked with auxiliary input  $z$  and random-tape  $r$ . In order to see that Eq. (3.2) holds, notice the following. First,  $\tilde{S}$  (upon auxiliary input  $z$ , random tape  $r$  and inputs  $\{x_i\}_{i \in I}$ ) sends exactly the same inputs to the trusted party as  $S$  does (upon auxiliary input  $z$ , random tape  $r$  and inputs  $\{x_i\}_{i \in I}$ ). Next, the outputs of all honest parties not outputting  $\perp$  are fixed by  $\bar{x}$  and the inputs sent to the trusted party by the simulators  $S$  or  $\tilde{S}$ . Therefore, if  $S$  and  $\tilde{S}$  send the same inputs, it follows that all parties not outputting  $\perp$  have exactly the same output. In addition,  $\tilde{S}$  outputs exactly the same string that  $S$  outputs. Eq. (3.2) therefore follows.

By assumption,  $\Pi$  is computationally  $t$ -secure with unanimous abort and partial fairness. It therefore holds that for every set  $I \subset [n]$  such that  $|I| < t$ , and for *every* set  $J \subseteq [n]$

$$\left\{ \text{IDEAL}_{f,I,S(z)}^{(2)}(\bar{x})|_J \right\}_{\bar{x},z} \stackrel{c}{=} \left\{ \text{REAL}_{\Pi,I,\mathcal{A}(z)}(\bar{x})|_J \right\}_{\bar{x},z}$$

Next, notice that the sets  $J_{\tilde{S}}$  and  $J_{\tilde{\mathcal{A}}}$  are fully defined given  $\tilde{\mathcal{A}}$  and  $\tilde{S}$ 's outputs respectively. (Recall that  $J_{\tilde{S}}$  equals the set of parties not outputting  $\perp$  in an ideal execution with  $\tilde{S}$ , and  $J_{\tilde{\mathcal{A}}}$  equals the set of parties not outputting  $\perp$  in a real execution of  $\tilde{\Pi}$  with  $\tilde{\mathcal{A}}$ .) Furthermore, by the definitions of  $\mathcal{A}$  and  $S$ , it follows that their outputs also fully define  $J_{\tilde{S}}$  and  $J_{\tilde{\mathcal{A}}}$ . Therefore,  $J_{\tilde{S}}$  (resp.,  $J_{\tilde{\mathcal{A}}}$ ) is part of the global output  $\text{IDEAL}$  (resp.,  $\text{REAL}$ ). This implies that,

$$\left\{ \text{IDEAL}_{f,I,S(z)}^{(2)}(\bar{x})|_{J_{\tilde{S}(z)}(\bar{x})} \right\} \stackrel{c}{=} \left\{ \text{REAL}_{\Pi,I,\mathcal{A}(z)}(\bar{x})|_{J_{\tilde{\mathcal{A}}(z)}(\bar{x})} \right\} \quad (3.3)$$

(Otherwise, we could distinguish  $\text{IDEAL}_{f,I,S(z)}^{(2)}(\bar{x})$  from  $\text{REAL}_{\Pi,I,\mathcal{A}(z)}(\bar{x})$  by considering the restriction to  $J_{\tilde{S}}$  or to  $J_{\tilde{\mathcal{A}}}$ , respectively.) Combining Eq. (3.3) with Equations (3.1) and (3.2), we have that

$$\left\{ \text{IDEAL}_{f,I,\tilde{S}(z)}^{(5)}(\bar{x})|_{J_{\tilde{S}(z)}(\bar{x})} \right\} \stackrel{c}{=} \left\{ \text{REAL}_{\tilde{\Pi},I,\tilde{\mathcal{A}}(z)}(\bar{x})|_{J_{\tilde{\mathcal{A}}(z)}(\bar{x})} \right\}$$

It remains to show that the entire output distributions (including the honest parties *not* in  $J$ ) are computationally indistinguishable. However, this is immediate, because for every party  $P_i$  for which  $i \notin J$ , it holds that  $P_i$  outputs  $\perp$  (this is true for both the real and ideal executions). Therefore,

$$\left\{ \text{IDEAL}_{f,I,\tilde{S}(z)}^{(5)}(\bar{x}) \right\} \stackrel{c}{=} \left\{ \text{REAL}_{\tilde{\Pi},I,\tilde{\mathcal{A}}(z)}(\bar{x}) \right\}$$

completing the proof of Lemma 3.4.5. ■

Recall that our aim is to show the security of the compiled protocol  $\Pi'$  (and not  $\tilde{\Pi}$ ). However, intuitively, there is no difference between  $\tilde{\Pi}$  and  $\Pi'$ . The reason is as follows: the only difference between  $\Pi'$  and  $\tilde{\Pi}$  is that in  $\Pi'$  the adversary can cause a party  $P_i$  to receive  $\perp$  in a broadcast with abort execution. Now, when this happens,  $P_i$  sends  $\perp$  to all honest parties in the next blank round of  $\Pi'$ . However, the adversary for  $\tilde{\Pi}$  can just send  $\perp$  itself to all the honest parties in the next blank round. The same effect is therefore achieved. Formally:

**Lemma 3.4.6** *Let  $\Pi$  be any protocol in the broadcast model and let  $\tilde{\Pi}$  be the transformation of  $\Pi$  as described in Lemma 3.4.5. Then, for every real-model adversary  $\mathcal{A}'$  for  $\Pi'$  of Construction 3.4.4, there exists a real-model adversary  $\tilde{\mathcal{A}}$  for  $\tilde{\Pi}$ , such that for every  $I \subset [n]$  with  $|I| < t$ ,*

$$\left\{ \text{REAL}_{\tilde{\Pi},I,\tilde{\mathcal{A}}(z)}(\bar{x}) \right\} \equiv \left\{ \text{REAL}_{\Pi',I,\mathcal{A}'(z)}(\bar{x}) \right\}$$

**Proof.** We begin by describing the adversary  $\tilde{\mathcal{A}}$ . Intuitively,  $\tilde{\mathcal{A}}$  works by simulating the executions of *broadcast with abort* for  $\mathcal{A}'$ . If a party receives  $\perp$  in  $\Pi'$  (in a broadcast with abort execution or in a blank round), then  $\tilde{\mathcal{A}}$  sends the appropriate  $\perp$  messages in a blank round of  $\tilde{\Pi}$ . This strategy works because in  $\Pi'$ , a party's output is the same if it receives  $\perp$  in a broadcast with abort execution or in the blank round that immediately follows it. Formally, adversary  $\tilde{\mathcal{A}}$  invokes  $\mathcal{A}'$  and in every round of the execution of  $\tilde{\Pi}$  works as follows:

1. *Receiving messages in rounds  $r, r+1$ :*  $\tilde{\mathcal{A}}$  receives the broadcast and point-to-point messages from the honest parties in  $\tilde{\Pi}$ . For every message broadcast by an honest party,  $\tilde{\mathcal{A}}$  simulates a “broadcast with abort” execution, playing the honest parties' roles (where  $\mathcal{A}'$  plays the corrupted parties' roles). In addition,  $\tilde{\mathcal{A}}$  forwards any point-to-point messages unchanged to  $\mathcal{A}'$ .
2. *Sending messages in round  $r, r+1$ :*  $\tilde{\mathcal{A}}$  plays the honest parties' roles in “broadcast with abort” executions, in which  $\mathcal{A}'$  broadcasts messages to the honest parties. Consider a particular execution in which a corrupted party  $P$  plays the dealer. If all the honest parties receive  $\perp$  in this execution, then  $\tilde{\mathcal{A}}$  broadcasts nothing in  $\tilde{\Pi}$ . However, if at least one honest

party outputs a message  $m$ , then  $\tilde{\mathcal{A}}$  broadcasts  $m$ . As before, point-to-point messages are forwarded unchanged.

3. *Blank round following round  $r + 1$ :* If an honest party  $P_i$  receives  $\perp$  in the simulated “broadcast with abort” execution of rounds  $r$  and  $r + 1$ , then  $\tilde{\mathcal{A}}$  sends  $\perp$  to *all* the honest parties in this blank round.

At the conclusion of the execution,  $\tilde{\mathcal{A}}$  outputs whatever  $\mathcal{A}'$  does. This completes the description of  $\tilde{\mathcal{A}}$ . The fact that  $\tilde{\mathcal{A}}$  perfectly simulates an execution of  $\Pi'$  with  $\mathcal{A}'$  follows directly from the definition of  $\Pi'$ . That is, the only difference between  $\tilde{\Pi}$  and  $\Pi'$  is that in  $\Pi'$  the broadcast channel is replaced by broadcast with abort. This means that some parties may receive  $\perp$  instead of the broadcasted message. However, in this case,  $\tilde{\mathcal{A}}$  knows that this event occurred and can send  $\perp$  to all the honest parties in the following blank round, as would occur in an execution of  $\Pi'$ . The key point is that in  $\Pi'$  it *makes no difference* to an honest parties if it received  $\perp$  in a broadcast with abort execution or in the following blank round. We conclude that the outputs of all the honest parties and  $\tilde{\mathcal{A}}$  in  $\tilde{\Pi}$ , are identically distributed to the outputs of the honest parties and  $\mathcal{A}'$  in  $\Pi'$ . ■

**Concluding the proof of Theorem 3.4.1:** Let  $\mathcal{A}'$  be an adversary attacking  $\Pi'$ . By Lemma 3.4.6, we have that there exists an adversary  $\tilde{\mathcal{A}}$  attacking  $\tilde{\Pi}$  such that the output distributions of  $\Pi'$  with  $\mathcal{A}'$ , and  $\tilde{\Pi}$  with  $\tilde{\mathcal{A}}$  are identical. Then, by Lemma 3.4.5, we have that for every real-model adversary  $\tilde{\mathcal{A}}$  for  $\tilde{\Pi}$ , there exists an ideal-model simulator  $\tilde{\mathcal{S}}$  such that the output distribution of a real execution with  $\tilde{\mathcal{A}}$  is computationally indistinguishable (or statistically close) to an ideal execution with abort and no fairness with  $\tilde{\mathcal{S}}$ . We conclude that the output distribution of a real execution of  $\Pi'$  with adversary  $\mathcal{A}'$  is computationally indistinguishable (or statistically close) to an ideal execution with abort and no fairness with  $\tilde{\mathcal{S}}$ . That is,  $\Pi'$  is  $t$ -secure with abort and no fairness, as required. ■

**The complexity of protocol  $\Pi'$ :** We remark that the transformation of  $\Pi$  to  $\Pi'$  preserves the round complexity of  $\Pi$ . In particular, the number of rounds in  $\Pi'$  equals exactly 3 times the number of rounds in  $\Pi$ . Regarding the communication complexity (i.e., bandwidth), this is the same in  $\Pi$  and  $\Pi'$  except for the cost incurred in simulating the broadcast channel. Notice that in the “broadcast with abort” protocol, if a dealer sends a  $k$ -bit message, then the total bandwidth equals  $n \cdot k$ . (If the dealer cheats and sends “long” messages, then the bandwidth is upper-bound by the length of the longest acceptable message times  $n$ .) Therefore, the number of bits sent in an execution of  $\Pi'$  is at most  $n$  times that sent in an execution of  $\Pi$ .



### 3.5 Secure Computation with Abort and Partial Fairness

In this section we show that for any functionality  $f$ , there exists a protocol for the *computational*  $t$ -secure computation of  $f$  with abort and partial fairness, for any  $t$  (assuming the existence of trapdoor permutations). Furthermore, for any functionality  $f$ , there exists a protocol for the *information-theoretic*  $n/2$ -secure computation of  $f$  with abort and partial fairness (and without any complexity assumptions).

**Outline:** We begin by motivating why the strategy used in Section 3.4 to obtain secure computation with abort does not provide partial fairness. The problem lies in the fact that due to the use of a “broadcast with abort” protocol (and not a real broadcast channel), the adversary can disrupt messages that are broadcast by honest parties. Now, in the definition of security with abort and partial fairness, once an honest  $P_1$  receives its output, all honest parties must receive their output. Therefore, the adversary must not be allowed to disrupt the communication following the time that an honest  $P_1$  receives its output. If broadcast is utilized in the remainder of the protocol (as is indeed the case for known protocols), then the adversary will be able to disrupt the communication and prevent all other honest parties from receiving output.

We solve this problem here by having the parties compute a different functionality. This functionality is such that once  $P_1$  gets its output, it can supply all the other parties with their output directly and without broadcast. Of course, this functionality must not reveal anything about the other parties’ outputs to  $P_1$ . As a first attempt, consider what happens if instead of computing the original functionality  $f$ , the parties first compute the following:

**First attempt:**

*Inputs:*  $\bar{x} = (x_1, \dots, x_n)$

*Outputs:*

- Party  $P_1$  receives its own output  $f_1(\bar{x})$ . In addition, for every  $i > 1$ , it receives  $c_i = f_i(\bar{x}) \oplus r_i$  for a uniformly distributed  $r_i$ .
- For every  $i > 1$ , party  $P_i$  receives the string  $r_i$ .

That is, for each  $i > 1$ , party  $P_i$  receives a random pad  $r_i$  and  $P_1$  receives an “encryption” of  $f_i(\bar{x})$  with that random pad. Now, assume that the parties use a protocol that is secure with abort and *no* fairness in order to compute this new functionality. Then, there are two possible outcomes to such a protocol execution: either all parties receive their prescribed output, or at least one honest party receives  $\perp$ . In the case that at least one honest party receives  $\perp$ , this party can notify  $P_1$  who can then immediately halt. The result is that no

parties, including the corrupted ones, receive output (if  $P_1$  does not send the  $c_i$  values, then the parties only obtain  $r_i$  which contains no information on  $f_i(\bar{x})$ ). In contrast, if all parties received their prescribed output, then party  $P_1$  can send each party  $P_i$  its encryption  $c_i$ , allowing it to reconstruct the output by computing  $f_i(\bar{x}) = c_i \oplus r_i$ . The key point is that the adversary is unable to prevent  $P_1$  from sending these  $c_i$  values and no broadcast is needed in this last step. Of course, if  $P_1$  is corrupted, then it will learn all the corrupted parties' outputs first. However, under the definition of partial fairness, this is allowed.

The flaw in the above strategy arises in the case that  $P_1$  is corrupted. Specifically, a corrupted  $P_1$  can send the honest parties modified values, causing them to conclude with incorrect outputs. This is in contradiction to what is required of secure computation. Therefore, we modify the functionality that is computed so that a corrupted  $P_1$  is unable to cheat. In particular, the aim is to prevent the adversary from modifying  $c_i = f_i(\bar{x}) \oplus r_i$  without  $P_i$  detecting this modification. If the adversary can be restrained in this way, then it can choose not to deliver an output; however, any output delivered is guaranteed to be correct. The above-described aim can be achieved using standard (information-theoretic) authentication techniques, based on pairwise independent hash functions. That is, let  $\mathcal{H}$  be a family of pairwise independent hash functions  $h : \{0, 1\}^k \rightarrow \{0, 1\}^k$ . Then, the functionality that the parties compute is as follows:

**Functionality  $F$ :**

*Inputs:*  $\bar{x} = (x_1, \dots, x_n)$

*Outputs:*

- Party  $P_1$  receives its own output  $f_1(\bar{x})$ . In addition, for every  $i > 1$ , it receives  $c_i = f_i(\bar{x}) \oplus r_i$  for a uniformly distributed  $r_i$ , and  $a_i = h_i(c_i)$  for  $h_i \in_R \mathcal{H}$ . (Note,  $P_1$  receives  $a_i$  but not the function description  $h_i$ .)
- For every  $i > 1$ , party  $P_i$  receives the string  $r_i$  and the description of the hash function  $h_i$ .

Notice that as in the first attempt,  $P_1$  learns nothing of the output of any honest  $P_i$  (since  $f_i(\bar{x})$  is encrypted with a random pad). Furthermore, if  $P_1$  attempts to modify  $c_i$  to  $c'_i$  in any way, then the probability that it will generate the correct authentication value  $a'_i = h_i(c'_i)$  is at most  $2^{-k}$  (by the pairwise independent properties of  $h_i$ ). Thus, the only thing a corrupt  $P_1$  can do is refuse to deliver the output. We now formally prove the above:

**Theorem 3.5.1** *For any probabilistic polynomial-time  $n$ -ary functionality  $f$ , there exists a protocol in the point-to-point model for the computational  $t$ -secure computation of  $f$  with abort and partial fairness, for any  $t$ . Furthermore, there exists a protocol in the point-to-point model for the information-theoretic  $n/2$ -secure computation of  $f$  with abort and partial fairness.*

**Proof.** We begin by describing the protocol for computing  $f$ , as motivated above.

**Protocol 3.5.2** (protocol for secure computation with abort and partial fairness for any  $f$ ):

1. **Stage 1 – computation:** *The parties use any protocol for secure (computational or information-theoretic) computation with abort and no fairness in order to compute the functionality  $F$  defined above.<sup>10</sup> Thus,  $P_1$  receives  $f_1(\bar{x})$  and a pair  $(c_i, a_i)$  for every  $i > 1$ , and each  $P_i$  ( $i > 1$ ) receives  $(r_i, h_i)$  such that  $c_i = f_i(\bar{x}) \oplus r_i$  and  $a_i = h_i(c_i)$ .*
2. **Stage 2 – blank round:** *After the above protocol concludes, a blank-round is added so that if any party receives  $\perp$  for its output from Stage 1, then it sends  $\perp$  to  $P_1$  in this blank round.*
3. **Stage 3 – outputs:** *If  $P_1$  received any  $\perp$ -messages in the blank round, then it sends  $\perp$  to all parties and halts outputting  $\perp$ . Otherwise, for every  $i$ , it sends  $(c_i, a_i)$  to  $P_i$  and halts, outputting  $f_1(\bar{x})$ . Party  $P_i$  outputs  $\perp$  if it received  $\perp$  from  $P_1$  (it ignores any  $\perp$  it may receive from other parties). If it received  $(c_i, a_i)$  from  $P_1$  (and not  $\perp$ ), then it checks that  $a_i = h_i(c_i)$ . If yes, it outputs  $f_i(\bar{x}) = c_i \oplus r_i$ . Otherwise, it outputs  $\perp$ .*

Intuitively, the security of Protocol 3.5.2 with abort and partial fairness is derived from the fact that Stage 1 is run using a protocol that is secure with abort (even though it has no fairness property). Consider the two cases regarding whether or not  $P_1$  is corrupted:

1.  $P_1$  is corrupt: in this case,  $\mathcal{A}$  receives all the outputs of the corrupted parties first. Furthermore,  $\mathcal{A}$  can decide exactly which parties to give output to and which not. However, this is allowed in the setting of secure computation with abort and partial fairness, and so is fine. We stress that  $\mathcal{A}$  cannot cause an honest party to output any value apart from  $\perp$  or its correct output. This is because the authentication properties of pairwise independent hash functions guarantee that  $\mathcal{A}$  does not modify  $c_i$ , and the correctness of the protocol of Stage 1 guarantees that  $c_i \oplus r_i$  equals the correct output  $f_i(\bar{x})$ .
2.  $P_1$  is honest: there are two possible cases here; either some honest party received  $\perp$  in the computation of Stage 1 or all honest parties received their correct outputs. If some honest party received  $\perp$ , then this party sends  $\perp$  to  $P_1$  in Stage 2 and thus no parties (including the corrupted parties) receive output. (Similarly, if  $\mathcal{A}$  sends  $\perp$  to  $P_1$  in Stage 2 then no parties receive output.) In contrast, if all honest parties received their

<sup>10</sup> By Corollaries 3.4.2 and 3.4.3 in Section 3.4, appropriate protocols exist. That is, information-theoretic secure protocols exist for  $t < n/2$ . Furthermore, assuming the existence of trapdoor permutations, computational secure protocols exist for any  $t$ .

outputs and  $\mathcal{A}$  does not send  $\perp$  to  $P_1$  in Stage 2, then all parties receive outputs and the adversary cannot cause any honest party to abort. We therefore have that in this case complete fairness is achieved, as required.

In order to formally prove the security of the protocol, we use the sequential composition theorem of [C00]. This theorem states that we can consider a model which is a hybrid of the ideal and real models. In this hybrid model, the parties all interact with a trusted party for the computation of Stage 1 (where the ideal model for this computation is that of secure computation with abort and no fairness). Then, Stages 2 and 3 take place as in a real execution. The result is a protocol that is a hybrid of real and ideal executions. In order to prove the security of the (real) protocol, it suffices to construct an ideal-model simulator for the hybrid protocol. Thus, the description of the parties and adversary below relate to this hybrid model (the parties send messages to each other, as in a real execution, and to a trusted party, as in an ideal execution). The proof of [C00] is stated for secure computation without abort (and complete fairness); however it holds also for secure computation with abort and no fairness.

Let  $\mathcal{A}$  be an adversary attacking Protocol 3.5.2 in the above-described hybrid model. We construct an adversary  $\mathcal{S}$  who works in the ideal model for  $f$  with abort and partial fairness.  $\mathcal{S}$  internally incorporates  $\mathcal{A}$  and simulates the hybrid execution for  $\mathcal{A}$ . Therefore,  $\mathcal{S}$  has *external* communication with the trusted party of its ideal model and *internal*, simulated communication with the adversary  $\mathcal{A}$ . In our description of  $\mathcal{S}$ , we differentiate between the cases that  $P_1$  is corrupt and  $P_1$  is honest:

1.  $P_1$  is corrupt:  $\mathcal{S}$  invokes  $\mathcal{A}$  and receives the inputs that  $\mathcal{A}$  intends to send to the trusted party of the hybrid model. Then,  $\mathcal{S}$  externally sends these inputs unmodified to the trusted party of its ideal model for computing  $f$ . If the inputs are not valid, then in the hybrid model all parties receive  $\perp$  as output. Therefore,  $\mathcal{S}$  internally hands  $\perp$  to  $\mathcal{A}$  as its output from Stage 1 and simulates all honest parties sending  $\perp$  in Stage 2 (as would occur in a hybrid execution).  $\mathcal{S}$  then halts, outputting whatever  $\mathcal{A}$  does. Otherwise, if the inputs are valid,  $\mathcal{S}$  receives all the corrupted parties outputs  $\{f_i(\bar{x})\}_{i \in I}$  (this is the case because  $\mathcal{S}$  controls  $P_1$  and by partial fairness, when  $P_1$  is corrupt the adversary receives the corrupted parties' outputs first).  $\mathcal{S}$  then constructs the corrupted parties' outputs from Stage 1 that  $\mathcal{A}$  expects to see in the hybrid execution.  $\mathcal{S}$  defines  $P_1$ 's output as follows: First,  $P_1$ 's personal output is  $f_1(\bar{x})$ . Next, for every corrupted party  $P_i$ , party  $P_1$ 's output contains the pair  $(c_i = f_i(\bar{x}) \oplus r_i, h_i(c_i))$  for  $r_i \in_R \{0, 1\}^k$  and  $h_i \in_R \mathcal{H}$ . Finally, for every honest party  $P_j$ , party  $P_1$ 's output contains a pair  $(c_j, a_j)$  where  $c_j, a_j \in_R \{0, 1\}^k$ . This defines  $P_1$ 's output. We now define how  $\mathcal{S}$  constructs the other corrupted parties' outputs: for every corrupted  $P_i$ , simulator  $\mathcal{S}$  defines  $P_i$ 's output to equal  $(r_i, h_i)$  where these are the values used in preparing the corresponding pair  $(c_i, h_i(c_i))$  in  $P_1$ 's output. (We note that  $\mathcal{S}$  can

prepare these values because it knows  $f_i(\bar{x})$  for every corrupted party  $P_i$ .)  $\mathcal{S}$  then internally passes  $\mathcal{A}$  all of these outputs. In the hybrid model, after receiving the outputs from Stage 1,  $\mathcal{A}$  sends a set  $J'$  to the trusted party instructing it to give outputs to the parties specified in this set (all other parties receive  $\perp$ ).  $\mathcal{S}$  obtains this set  $J'$  from  $\mathcal{A}$  and records it.

$\mathcal{S}$  continues by simulating  $P_l$  sending  $\perp$  to  $P_1$  in the blank round, for every honest party  $P_l$  for which  $l \notin J'$  (as would occur in a hybrid execution). Then, in the last stage,  $\mathcal{A}$  (controlling  $P_1$ ) sends to each honest party  $P_j$  a pair  $(c'_j, a'_j)$  or  $\perp$ .  $\mathcal{S}$  receives these strings and defines the set of parties  $J$  to receive outputs to equal those parties in  $J'$  to whom  $\mathcal{A}$  sends the same  $(c_j, a_j)$  that  $\mathcal{S}$  gave  $\mathcal{A}$  in Stage 1. (These are the parties who do not see  $\perp$  in the execution and whose checks of Stage 3 succeed; they therefore do not abort.)  $\mathcal{S}$  concludes by externally sending  $J$  to the ideal-model trusted party and outputting whatever  $\mathcal{A}$  does.

2.  $P_1$  is honest: In this case,  $\mathcal{S}$  begins in the same way. That is,  $\mathcal{S}$  invokes  $\mathcal{A}$  and receives the inputs that  $\mathcal{A}$  intends to send the trusted party of the hybrid model. However, unlike in the previous case,  $\mathcal{S}$  does not forward these inputs to its trusted party; rather it just records them.<sup>11</sup> (If any of these inputs are invalid, then  $\mathcal{S}$  internally sends  $\perp$  to all corrupted parties, externally sends invalid inputs to the trusted party and halts. In the sequel, we assume that all inputs sent by  $\mathcal{A}$  are valid.) Now,  $\mathcal{A}$  expects to receive outputs from Stage 1 before it sends the trusted party the set  $J'$  of honest parties receive output from Stage 1. However,  $\mathcal{S}$  does *not* have the corrupted parties' outputs yet. Fortunately, when  $P_1$  is not corrupted,  $\mathcal{S}$  can perfectly simulate the corrupted parties outputs from Stage 1 by merely providing them with  $(r_i, h_i)$  where  $r_i \in_R \{0, 1\}^k$  and  $h_i \in_R \mathcal{H}$ . After internally passing  $\mathcal{A}$  the simulated corrupted parties' outputs,  $\mathcal{S}$  obtains a set  $J'$  from  $\mathcal{A}$ , instructing the trusted party of the hybrid model which parties should receive output.

$\mathcal{S}$  continues by simulating Stages 2 and 3 of the protocol. As above,  $\mathcal{S}$  simulates every honest party  $P_l$  for which  $l \notin J'$  sending  $\perp$  to  $P_1$  in Stage 2. Furthermore,  $\mathcal{S}$  obtains any messages sent by  $\mathcal{A}$  in this stage. If  $\mathcal{A}$  sends  $\perp$  to  $P_1$  in Stage 2, then  $\mathcal{S}$  simulates  $P_1$  sending  $\perp$  to all parties, sends invalid inputs to the trusted party and halts. Likewise, if  $J'$  does not contain *all* the honest parties, then  $\mathcal{S}$  internally simulates  $P_1$  sending  $\perp$  to all the corrupted parties, and externally sends invalid inputs to the trusted party. (These cases correspond to the case that no parties receive their prescribed output.)

In contrast, if  $J'$  contains all the honest parties (i.e., no honest party received  $\perp$  from Stage 1) and  $\mathcal{A}$  did not send  $\perp$  to  $P_1$  in Stage 2 of the simulation, then  $\mathcal{S}$  externally sends the trusted party the inputs that it

<sup>11</sup>  $\mathcal{S}$  cannot forward the inputs to the trusted party yet, because in the model of partial fairness as soon as it does this all parties receive output. However, in the execution of Protocol 3.5.2,  $\mathcal{A}$  can cause the execution to abort at a later stage.

recorded from  $\mathcal{A}$  above, receiving back all of the corrupted parties outputs  $\{f_i(\bar{x})\}_{i \in I}$ . Then, for each corrupted party's output  $f_i(\bar{x})$ , simulator  $\mathcal{S}$  generates the pair that corrupted  $P_i$  would see in a hybrid execution. In particular, previously in the simulation  $\mathcal{S}$  provided  $P_i$  with a pair  $(r_i, h_i)$  where  $r_i \in_R \{0, 1\}^k$  and  $h_i \in_R \mathcal{H}$ . Now,  $\mathcal{S}$  simulates  $P_1$  sending corrupted party  $P_i$  the pair  $(c_i, a_i)$  where  $c_i = f_i(\bar{x}) \oplus r_i$  and  $a_i = h_i(c_i)$ . ( $\mathcal{S}$  can do this because it knows the random-pad  $r_i$  and the hash function  $h_i$ .) Finally,  $\mathcal{S}$  outputs whatever  $\mathcal{A}$  does and halts.

The fact that the global output in the ideal execution with  $\mathcal{S}$  is identically distributed to the global output in a hybrid execution with  $\mathcal{A}$  is derived from the following observations. First,  $\mathcal{A}$ 's outputs from Stage 1 can be perfectly simulated, both when  $P_1$  is corrupt and when  $P_1$  is honest. Second, the honest parties' messages in Stage 2 can be perfectly simulated given only the set  $J'$  sent by  $\mathcal{A}$  to the hybrid-model trusted party in the ideal execution of Stage 1. Therefore,  $\mathcal{A}$ 's view in the hybrid-model execution is identical to its view in a real execution. It remains to show that the honest parties' outputs are also correctly distributed.

First, consider the case that  $P_1$  is corrupt. In this case, with overwhelming probability, the set of honest parties receiving output in the real model are exactly those parties  $P_j$  for whom  $P_1$  (controlled by  $\mathcal{A}$ ) sends the exact pair  $(c_j, a_j)$  that it received as output from Stage 1 (and who did not see  $\perp$  at any time in the execution). This is due to the authentication properties of pairwise independent hash functions. Likewise, in the ideal-model simulation,  $\mathcal{S}$  designates these same parties to be the ones receiving output. Therefore, except with negligible probability, the set  $J$  sent by  $\mathcal{S}$  to the trusted party contains exactly those parties who would receive output in a real execution.

Next, consider the case that  $P_1$  is honest. In this case, all parties receive output unless  $P_1$  sees  $\perp$  in Stage 2. This can happen if  $\mathcal{A}$  sends  $P_1$  such a value, or if any honest party received  $\perp$  from Stage 1. Both of these cases are detected by  $\mathcal{S}$  in the hybrid-model simulation, and therefore the case that all parties abort in the hybrid model corresponds to this case in the real model (and likewise for the case that all parties receive output). This completes the proof of Theorem 3.5.1 ■

### 3.6 Obtaining Security under Self Composition

As we have discussed, one of our main aims in removing the reliance on a broadcast channel in secure protocols was to remove the obstacle that such a channel poses when security under composition is considered. (Of course, achieving secure computation without a trusted setup phase, as needed for authenticated Byzantine Agreement, is also of importance. However, the possibility of obtaining secure protocols that compose was unknown, even assuming a trusted setup phase.) In this section we *informally discuss* the ramifica-

tions of the results of this chapter on protocol composition. We consider both parallel self composition (for the case of no honest majority) and concurrent self composition (when there is an honest majority).

As we have shown, the protocol compiler of Construction 3.4.4 is such that the only difference between the original protocol that uses a broadcast channel and the resulting protocol that uses only point-to-point channels is with respect to the unanimity of abort. The formal proof of this fact was demonstrated in the stand-alone model only. However, it is not hard to see that all the claims also go through in the setting of composition. Therefore, it suffices to demonstrate the existence of protocols in the broadcast model that remain secure under composition in order to derive the existence of analogous protocols in the point-to-point network model. The presentation here is very informal; in particular, we do not formally define what is meant by security under self composition; see [L03a] for formal definitions. We now present the results:

**Concurrent composition for  $t < n/2$ .** The broadcast-model protocol of [RB89] for the information-theoretic  $n/2$ -secure computation of any functionality has been shown to remain secure under concurrent (general) composition [c01].<sup>12</sup> Therefore, by applying the protocol compiler of Construction 3.4.4 to the protocol of [RB89], we obtain a protocol that provides information-theoretic  $n/2$ -secure computation (with abort and no fairness) in the point-to-point model, and remains secure under concurrent (general) composition. Next, using the transformation in the proof of Theorem 3.5.1, we obtain an analogous protocol with abort and partial fairness.<sup>13</sup>

**Parallel composition for  $t \geq n/2$ .** A variant of the protocol of [GMW87] for the broadcast model can be shown to compose in parallel. Specifically, consider a variant of [GMW87] where the only rewinding that is carried out by the ideal-model simulator is in simulating zero-knowledge proofs and extracting witnesses from proofs of knowledge (we note that the protocol as described in [G02a] has this property). Then, if the zero-knowledge proofs and proofs of knowledge used compose in parallel, it follows that the entire secure protocol composes in parallel. It remains to demonstrate the existence of zero-knowledge proofs and proofs of knowledge that compose in parallel. The fact that there exist zero-knowledge proofs that compose in parallel was proven in [G02b]. In contrast, we do not know of any published theorem stating the existence of a zero-knowledge proof of knowledge that composes in parallel (i.e., for which knowledge extraction succeeds even in the set-

<sup>12</sup> Actually, this claim does not yet have an accompanying proof.

<sup>13</sup> We remark that Theorem 3.5.1 actually provides a transformation of *any* general secure protocol. That is, any general protocol for the broadcast model that is secure with abort and no fairness can be transformed into a general protocol for the point-to-point model, where the resulting protocol is secure with abort and partially fair. The theorem was not stated in this way so that it would not be overly cumbersome.

ting of parallel composition). Nevertheless, it is not difficult to show that the knowledge extractors for standard zero-knowledge proofs of knowledge of  $\mathcal{NP}$  (e.g., [B86, GMW86]) succeed under parallel composition. Given this observation, it is possible to obtain zero-knowledge proofs of knowledge for which both the simulation and extraction succeed under parallel composition. This can be achieved using ideas from the protocol of [FS89, F90] (where the only rewinding carried out by the simulator is in running a knowledge extractor). We thus conclude that in the broadcast model, there exist secure protocols for  $t \geq n/2$  that compose in parallel. By applying Construction 3.4.4 and the transformation in the proof of Theorem 3.5.1 to such protocols, we obtain secure protocols with abort and partial fairness in the point-to-point model that *compose in parallel*.

We remark that little is known regarding the feasibility of obtaining security under *concurrent* composition for the case of  $t \geq n/2$ , even assuming a broadcast channel. In the next chapter, we show that universally composable protocols exist in the common reference string model. However, in the plain model, the only construction known is for the case of two parties, and security is only achieved for *bounded concurrency* [L03a].

**Comparison to Chapter 4.** We conclude by noting that the above results on the composition of secure protocols are incomparable to those on universal composability achieved in Chapter 4. On the one hand, the notion of composition achieved by universal composability is that of concurrent general composition with arbitrary sets of parties. However, this result requires a common reference string to achieve. On the other hand, the type of composition demonstrated here is that of concurrent or parallel self composition for a single set of parties. Nevertheless, these results are in the plain model, and require no trusted setup phase (like that needed for obtaining a common reference string).



## 4. Universally Composable Multi-party Computation

In this chapter we prove a fundamental result stating that secure multi-party computation that remains secure under *concurrent general composition* can be achieved, for any number of corrupted parties. That is, we consider an asynchronous multi-party network and an adversary that can adaptively corrupt as many parties as it wishes. We present protocols that allow any subset of parties in this setting to securely realize any desired functionality of their inputs, and be guaranteed that security is preserved *regardless of the activity in the rest of the network*. Our protocols are in the common reference string model and rely on standard intractability assumptions.

### 4.1 Introduction

As we have seen, the model of “stand-alone computation” does not really capture the security requirements from cryptographic protocols in modern computer networks. In such networks, a protocol execution may run concurrently with an unknown number of other protocols. These arbitrary protocols may be executed by the same parties or other parties, they may have potentially related inputs and the scheduling of message delivery may be adversarially coordinated. Furthermore, the local outputs of a protocol execution may be used by other protocols in an unpredictable way. These concerns, or “attacks” on a protocol are not captured by the stand-alone model.

One way to guarantee that protocols withstand some specific security threats in multi-execution environments is to explicitly incorporate these threats into the security model and analysis. Such an approach was taken, for instance, in the case of non-malleability of protocols [DDN91]. However, this approach is inherently limited since it needs to explicitly address each new concern, whereas in a realistic network setting, the threats may be unpredictable. Furthermore, it inevitably results in definitions with ever-growing complexity.

An alternative, and arguably preferable, approach is to design and analyze protocols as “stand alone”, and then derive security in a multi-execution environment via a *secure composition theorem*. This is the approach taken by the recently proposed framework of *universally composable security* [C01]. Here a generic definition is given for what it means for a protocol to “securely

realize a given ideal functionality”, where an “ideal functionality” is a natural algorithmic way of defining the protocol problem at hand. In addition, it has been shown that protocols that are secure under this definition remain secure under concurrent general composition with arbitrary sets of parties. Thus, any protocol that securely realizes an ideal functionality when considered as stand-alone (according to this definition), continues to securely realize the same functionality even when composed with any other set of protocols that may be running concurrently in the same system. A protocol that is secure within the [C01] framework is called **universally composable (UC)**, and we say that the protocol UC **realizes** the given functionality.

Briefly recapping known feasibility results for universal composability, it has been shown that *any* ideal functionality can be UC realized using known constructions, as long as a majority of the participants are honest [C01] (building upon [BGW88, RB89, CFGN96]). However, this result does not hold when half or more of the parties may be corrupted. In particular, it does not hold for the important case of *two-party* protocols, where each party wishes to maintain its security even if the other party is corrupted. In fact, it has been shown that in the plain model, many two-party functionalities cannot be securely realized in this framework by two-party protocols [C01, CF01, CKL03]. Nonetheless, protocols that UC realize the commitment and zero-knowledge functionalities in the **common reference string (CRS)** model were shown in [CF01, DDOPS01]. (In the CRS model all parties are given a common, public reference string that is ideally chosen from a given distribution. This model was originally proposed in the context of non-interactive zero-knowledge proofs [BFM88] and since then has proved useful in other cases as well.)

**Our results.** Loosely speaking, we show that any functionality can be UC realized in the CRS model, regardless of the number of corrupted parties. More specifically, consider an *asynchronous* multi-party network where the communication is open and delivery of messages is not guaranteed. (For simplicity, we assume that delivered messages are authenticated. This can be achieved using standard methods.) The network contains an unspecified number of parties, and any number of these parties can be adaptively corrupted throughout the computation. In this setting, we show how arbitrary subsets of parties can UC realize any functionality of their inputs. The functionality may be reactive, namely it may receive inputs and generate outputs multiple times throughout the computation.

In addition to our general constructions for two-party and multi-party computation, we also present a new adaptively secure UC commitment scheme in the CRS model, assuming only the existence of trapdoor permutations. (UC commitment schemes are protocols that UC realize the ideal commitment functionality [CF01]. Existing constructions of UC commitments

[CF01, DN02] rely on specific cryptographic assumptions.<sup>1</sup>) Since UC zero-knowledge can be obtained given a UC commitment scheme [CF01], we can plug our new scheme into the UC zero-knowledge protocol of [CF01] and thereby obtain an adaptively secure UC zero-knowledge protocol in the CRS model, for any NP relation, and based on any trapdoor permutation. Beyond being interesting in its own right, we use this commitment scheme in order to base our constructions on general cryptographic assumptions.

**Adaptive security.** Our protocol is the first general construction that guarantees security against adaptive adversaries in the two-party case and in the case of multi-party protocols with honest minority. (We note that no adaptively secure general construction was known in these cases even in the traditional stand-alone model; all previous adaptively secure protocols for general multi-party computation assumed an honest majority.) We remark that, in contrast to the case of stand-alone protocols, in the setting of composition with arbitrary sets of parties, adaptive security is a relevant concern even for protocols with only two participants. Furthermore, it is important to protect even against adversaries that eventually break into *all* the participants in an interaction. This is because when arbitrary sets of parties are considered, multiple interactions take place between different sets of parties in the system. Therefore, all the participants in one interaction may constitute a proper subset of the participants in another interaction. Our results hold even in a model where no data can ever be erased.

**Cryptographic assumptions.** Our protocols are based on the following cryptographic assumptions. For the non-adaptive case (both semi-honest and malicious) we assume the existence of trapdoor permutations only. For the adaptive case we additionally assume the existence of *augmented* non-committing encryption protocols [CFG96]. The augmentation includes oblivious key generation and invertible samplability [DN00]. Loosely speaking, oblivious key generation states that public keys can be generated without knowing the corresponding private keys, and invertible samplability states that given a public/private key-pair it is possible to obtain the random coin tosses of the key generator when outputting this key-pair (the oblivious key generator should also be invertible). Such encryption schemes are known to exist under the RSA and DDH assumptions.

As we have mentioned, our protocols are in the CRS model. The above assumptions suffice if we use a common reference string that is not uniformly distributed (but is rather taken from some different distribution). If a uniformly distributed common reference string is to be used, then we additionally assume the existence of dense cryptosystems [DP92].

**Organization.** In Section 4.2 we provide an overview of the model of [C01] and an outline of our construction of UC two-party and multi-party pro-

<sup>1</sup> Subsequent to this work, other constructions based on general assumptions have also been presented [DG03].

protocols. Section 4.3 contains a number of important preliminaries: First, in Section 4.3.1, a more detailed description of the [C01] framework and of the composition theorem itself is presented. Then, in Section 4.3.2, the issue of universal composition and joint state is discussed (this is important where a common reference string is used, as is the case in our constructions). Protocols for UC commitments and UC zero-knowledge, both important tools in our general constructions, are presented in Sections 4.5 and 4.6.

We then begin our presentation of constructions for UC two-party and multi-party computation. That is, in Section 4.4, we show how to obtain UC two-party secure computation in the presence of semi-honest adversaries. Next we proceed to the case of malicious adversaries. That is, in Section 4.7 we define and realize the two-party commit-and-prove functionality. This is then used in Section 4.8 to construct a two-party protocol compiler that is used to transform the protocol of Section 4.4 into a protocol that is secure against malicious adversaries. Finally, in Section 4.9, we extend our two-party constructions to the multi-party case. We present the two-party case separately because most of the cryptographic ideas already arise in this setting.

## 4.2 Overview

This section provides a high-level overview of the model and our constructions. Section 4.2.1 contains an overview of the general framework of universal composability, the definition of security and the composition theorem. Then, in Section 4.2.2 we provide a brief outline of our constructions for two-party and multi-party computation. The aim of this outline is to provide the reader with the “big picture”, before delving into details.

### 4.2.1 The Model

We begin by outlining the framework for universal composability; for more details see Section 4.3.1 and [C01]. The framework provides a rigorous method for defining the security of cryptographic tasks, while ensuring that security is maintained under concurrent general composition. Protocols that fulfill the definitions of security in this framework are called **universally composable (UC)**.

As in other general definitions (e.g., [GL90, MR91, B91b, PW00, C00]), the security requirements of a given task (i.e., the functionality expected from a protocol that carries out the task) are captured via a set of instructions for a “trusted party” that obtains the inputs of the participants and provides them with the desired outputs (in one or more iterations). We call the algorithm run by the trusted party an **ideal functionality**. Since the trusted party just runs the ideal functionality, we do not distinguish between them. Rather, we refer to interaction between the parties and the *functionality*. Informally, a

protocol securely carries out a given task if no adversary can gain more from an attack on a real execution of the protocol, than from an attack on an ideal process where the parties merely hand their inputs to a trusted party with the appropriate functionality and obtain their outputs from it, without any other interaction. In other words, it is required that a real execution can be *emulated* in the above ideal process (where the meaning of *emulation* is described below). We stress that in a real execution of the protocol, no trusted party exists and the parties interact amongst themselves only.

In order to prove the universal composition theorem, the notion of emulation in this framework is considerably stronger than in previous ones. Traditionally, the model of computation includes the parties running the protocol, plus an adversary  $\mathcal{A}$  that controls the communication channels and potentially corrupts parties. Emulation means that for any adversary  $\mathcal{A}$  attacking a real protocol execution, there should exist an “ideal process adversary” or simulator  $\mathcal{S}$ , that causes the outputs of the parties in the ideal process to be essentially the same as the outputs of the parties in a real execution. In the universally composable framework, an additional adversarial entity called the **environment**  $\mathcal{Z}$  is introduced. This environment generates the inputs to all parties, reads all outputs, and in addition interacts with the adversary in an arbitrary way throughout the computation. (As is hinted by its name,  $\mathcal{Z}$  represents the external environment that consists of arbitrary protocol executions that may be running concurrently with the given protocol.) A protocol is said to UC realize a given ideal functionality  $\mathcal{F}$  if for any “real-life” adversary  $\mathcal{A}$  that interacts with the protocol there exists an “ideal-process adversary”  $\mathcal{S}$ , such that *no environment*  $\mathcal{Z}$  can tell whether it is interacting with  $\mathcal{A}$  and parties running the protocol, or with  $\mathcal{S}$  and parties that interact with  $\mathcal{F}$  in the ideal process. (In a sense, here  $\mathcal{Z}$  serves as an “interactive distinguisher” between a run of the protocol and the ideal process with access to  $\mathcal{F}$ . See [C01] for more motivating discussion on the role of the environment.) Note that the definition requires the “ideal-process adversary” (or simulator)  $\mathcal{S}$  to interact with  $\mathcal{Z}$  throughout the computation. Furthermore,  $\mathcal{Z}$  cannot be “rewound”.

The following *universal composition theorem* is proven in [C01]: Consider a protocol  $\pi$  that operates in a *hybrid* model of computation where parties can communicate as usual, and in addition have ideal access to an unbounded number of copies of some ideal functionality  $\mathcal{F}$ . (This model is called the  $\mathcal{F}$ -hybrid model.) Furthermore, let  $\rho$  be a protocol that UC realizes  $\mathcal{F}$  as sketched above, and let  $\pi^\rho$  be the “composed protocol”. That is,  $\pi^\rho$  is identical to  $\pi$  with the exception that each interaction with the ideal functionality  $\mathcal{F}$  is replaced with a call to (or an activation of) an appropriate instance of the protocol  $\rho$ . Similarly,  $\rho$ -outputs are treated as values provided by the functionality  $\mathcal{F}$ . The theorem states that in such a case,  $\pi$  and  $\pi^\rho$  have essentially the same input/output behaviour. Thus,  $\rho$  behaves just like the ideal functionality  $\mathcal{F}$ , even when composed concurrently with an arbitrary

protocol  $\pi$ . This implies the notion of concurrent general composition as described in Section 1.2. A special case of the composition theorem states that if  $\pi$  UC realizes some ideal functionality  $\mathcal{G}$  in the  $\mathcal{F}$ -hybrid model, then  $\pi^\rho$  UC realizes  $\mathcal{G}$  from scratch.

We consider a network where the adversary sees all the messages sent, and delivers or blocks these messages at will. (The fact that message delivery is not guaranteed frees us from the need to explicitly deal with the “early stopping” problem of protocols run between two parties or amongst many parties where only a minority may be honest. This is because even the ideal process allows the adversary to abort the execution at any time.) We note that although the adversary may block messages, it cannot modify messages sent by honest parties (i.e., the communication lines are ideally authenticated). Our protocols are cast in a completely asynchronous point-to-point network (and thus the adversary has full control over when messages are delivered, if at all). Also, as usual, the adversary is allowed to **corrupt** parties. In the case of **static** adversaries the set of corrupted parties is fixed at the onset of the computation. In the **adaptive** case the adversary corrupts parties at will throughout the computation. We also distinguish between malicious and semi-honest adversaries: If the adversary is **malicious** then corrupted parties follow the arbitrary instructions of the adversary. In the **semi-honest** case, even corrupted parties follow the prescribed protocol and the adversary essentially only gets read access to the states of corrupted parties.

#### 4.2.2 An Outline of the Results and Techniques

In this section we provide a high-level description of our protocols for two-party and multi-party computation, and the techniques used in obtaining them. Our construction is conceptually very similar to the construction of Goldreich, Micali and Wigderson [GMW87, G02a]. This construction (which we call the GMW construction) is comprised of two stages. First, they present a protocol for securely realizing any functionality in the semi-honest adversarial model. Next, they construct a *protocol compiler* that takes any semi-honest protocol and transforms it into a protocol that has the same functionality in the malicious adversarial model. (However, as discussed above, they consider a model where only a single protocol execution takes place in the system. In contrast, we construct protocols for universally composable secure computation.) We begin by considering the two-party case.

**Two-party computation in the case of semi-honest adversaries.** Recall that in the case of semi-honest adversaries, even the corrupted parties follow the protocol specification. However, the adversary may attempt to learn more information than intended by examining the transcript of messages that it received during the protocol execution. Despite the seemingly weak nature of the adversarial model, obtaining protocols secure against semi-honest adversaries is a non-trivial task.

We begin by briefly recalling the [GMW87, G02a] construction for secure two-party computation in the semi-honest adversarial model. Let  $f$  be the two-party functionality that is to be securely computed. Then, the parties are given an arithmetic circuit over  $GF(2)$  that computes the function  $f$ . The protocol starts with the parties sharing their inputs with each other using simple bitwise-xor secret sharing, and thus following this stage, they both hold shares of the input lines of the circuit. That is, for each input line  $l$ , party  $A$  holds a value  $a_l$  and party  $B$  holds a value  $b_l$ , such that both  $a_l$  and  $b_l$  are random under the constraint that  $a_l + b_l$  equals the value of the input into this line. Next, the parties evaluate the circuit gate-by-gate, computing random shares of the output line of the gate from the random shares of the input lines to the gate. There are two types of gates in the circuit: addition gates and multiplication gates. Addition gates are evaluated by each party locally adding its shares of the input values. Multiplication gates are evaluated using 1-out-of-4 oblivious transfer (the oblivious transfer protocol used is basically that of [EGL85]). In the above way, the parties jointly compute the circuit and obtain shares of the output gates. The protocol concludes with each party revealing the prescribed shares of the output gates to the other party (i.e, if a certain output gate provides a bit of  $A$ 's input, then  $B$  will reveal its share of this output line to  $A$ ).

Our general construction is exactly that of GMW, except that the oblivious transfer protocol used is universally composable. That is, we first define an ideal oblivious transfer functionality,  $\mathcal{F}_{\text{OT}}$ , and show that in the  $\mathcal{F}_{\text{OT}}$ -hybrid model, the GMW protocol UC realizes any two-party functionality in the presence of semi-honest, *adaptive* adversaries. This holds unconditionally and even if the adversary and environment are computationally unbounded. Of course, computational assumptions are used for UC realizing  $\mathcal{F}_{\text{OT}}$  itself. (Our overall construction is actually somewhat more general than that of GMW in that it deals with reactive functionalities that have multiple stages which are separately activated. This is achieved by having the parties hold shares of the state of the ideal functionality between activations.)

Next we present protocols that UC realize  $\mathcal{F}_{\text{OT}}$  in the semi-honest case. In the non-adaptive case, the protocol of [EGL85, G02a] suffices. In the adaptive case, our protocol uses an augmented version of non-committing encryption [CFGN96]. The augmentation consists of two additional properties. First, the encryption scheme should have an alternative key generation algorithm that generates only public encryption keys without the corresponding decryption key. Second, the standard and additional key generation algorithms should be invertible in the sense that given the output key or keys, it is possible to find the random coin tosses used in generating these keys. (Following [DN00], we call these properties *oblivious key generation* and *invertible samplability*.) All known non-committing encryption schemes have this properties. In particular, such schemes exist under either the RSA assumption or the DDH assumption.) In all, we show:

**Proposition 4.2.1** (semi-honest computation – informal): *Assume that trap-door permutations exist. Then, for any two-party ideal functionality  $\mathcal{F}$ , there exists a protocol  $\Pi$  that UC realizes  $\mathcal{F}$  in the presence of semi-honest, static adversaries. Furthermore, if augmented two-party non-committing encryption protocols exist, then there exists a protocol  $\Pi$  that UC realizes  $\mathcal{F}$  in the presence of semi-honest, adaptive adversaries.*

Proposition 4.2.1 as stated above is not precise. This is due to two technicalities regarding the model of computation as defined in [C01]. We therefore define a class of functionalities for which these technical problems do not arise and then construct secure protocols for any functionality in this class. See Section 4.3.3 for more discussion and an exact definition.

Another point where our results formally differ from Proposition 4.2.1 is due to the fact that, according to the definitions used here, protocols which do not generate any output are technically secure (for any functionality). Thus, Proposition 4.2.1 as stated, can be easily (but un-interestingly) achieved. In contrast, we prove the existence of protocols which *do* generate output and UC realize any functionality (we call such a protocol non-trivial; for more details, see the discussion after Definition 4.3.2 in Section 4.3.1). Proposition 4.2.1 is formally restated in Section 4.4.2.

**Obtaining two-party computation secure against malicious adversaries.** Having constructed a protocol that is universally composable when the adversary is limited to semi-honest behaviour, we construct a protocol compiler that transforms this protocol into one that is secure even against malicious adversaries. From here on, we refer to the protocol that is secure against semi-honest adversaries as the “basic protocol”. Recall that the basic protocol is only secure in the case that even the corrupted parties follow the protocol specification exactly, using a uniformly chosen random tape. Thus, in order to obtain a protocol secure against malicious adversaries, we need to enforce potentially malicious corrupted parties to behave in a semi-honest manner. First and foremost, this involves forcing the parties to follow the prescribed protocol. However, this only makes sense relative to a *given* input and random tape. Furthermore, a malicious party must be forced into using a *uniformly chosen* random tape. This is because the security of the basic protocol may depend on the fact that the party has no freedom in setting its own randomness. We begin with a high-level description of the GMW compiler.

**An informal description of the GMW compiler.** The GMW compiler begins by having each party commit to its input. Next, the parties run a coin-tossing protocol in order to fix their random tapes. A simple coin-tossing protocol in which both parties receive the same uniformly distributed string is not sufficient here. This is because the parties’ random tapes must remain secret. Instead, an augmented coin-tossing protocol is used, where one party



receives a uniformly distributed string (to be used as its random tape) and the other party receives a commitment to that string. Now, following these two steps, each party holds its own input and uniformly distributed random tape, and a commitment to the other party's input and random tape.

Next, the commitments to the random tape and to the inputs are used to “enforce” semi-honest behaviour. Observe that a protocol specification is a deterministic function of a party's view consisting of its input, random tape and messages received so far. Further observe that each party holds a commitment to the input and random tape of the other party and that the messages sent so far are public. Therefore, the assertion that a new message is computed according to the protocol is an NP statement (and the party sending the message knows an adequate NP-witness to it). This means that the parties can use zero-knowledge proofs to show that their steps are indeed according to the protocol specification. Therefore, in the protocol emulation phase, the parties send messages according to the instructions of the basic protocol, while proving at each step that the messages sent are correct. The key point is that, due to the soundness of the proofs, even a malicious adversary cannot deviate from the protocol specification without being detected. Therefore, the adversary is limited to semi-honest behaviour. Furthermore, since the proofs are zero-knowledge, nothing “more” is revealed in the compiled protocol than in the basic protocol. We conclude that the security of the compiled protocol (against malicious adversaries) is directly derived from the security of the basic protocol (against semi-honest adversaries).

In summary, the GMW compiler has three components: input commitment, coin-tossing and protocol emulation (where the parties prove that their steps are according to the protocol specification).

**Universally composable protocol compilation.** A natural way of adapting the GMW compiler to the setting of universally composable secure computation would be to take the same compiler, but rather use *universally composable* commitments, coin-tossing and zero-knowledge as sub-protocols. However, such a strategy fails because the receiver of a universally composable commitment receives *no information* about the value that was committed to. (Instead, the recipient receives only a formal “receipt” assuring it that a value was committed to. See Section 4.5 for more details.) Thus, there is no NP-statement that a party can prove relative to its input commitment. This is in contrast to the GMW protocol where standard (perfectly binding) commitments are used and thus each party holds a string that uniquely determines the other party's input and random tape.

A different strategy is therefore required for constructing a universally composable compiler. Before describing our strategy, observe that in GMW the use of the commitment scheme is not standard. Specifically, although both parties commit to their inputs etc., they never decommit. Rather, they *prove* NP-statements relative to their committed values. Thus, a natural primitive to use would be a “commit-and-prove” functionality, which is comprised of

two phases. In the first phase, a party “commits” (or is bound) to a specific value. In the second phase, this party proves NP-statements (in zero-knowledge) relative to the committed value. We formulate this notion in a universally composable commit-and-prove functionality,<sup>2</sup> denoted  $\mathcal{F}_{\text{CP}}$ , and then use this functionality to implement all three phases of the compiler. More specifically, our protocol compiler uses the “commit” phase of the  $\mathcal{F}_{\text{CP}}$  functionality in order to execute the input and coin-tossing phases of the compiler. The “prove” phase of the  $\mathcal{F}_{\text{CP}}$  functionality is then used to force the adversary to send messages according to the protocol specification and consistent with the committed input and the random tape resulting from the coin-tossing. The result is a universally composable analog to the GMW compiler. We remark that in the  $\mathcal{F}_{\text{CP}}$ -hybrid model the compiler is unconditionally secure against *adaptive* adversaries, even if the adversary and the environment are computationally unbounded.

We show how to UC realize  $\mathcal{F}_{\text{CP}}$  in the  $\mathcal{F}_{\text{ZK}}$ -hybrid model, i.e. in a hybrid model with ideal access to an ideal zero-knowledge functionality,  $\mathcal{F}_{\text{ZK}}$ . (Functionality  $\mathcal{F}_{\text{ZK}}$  expects to receive a statement  $x$  and a witness  $w$  from the prover. It then forwards  $x$  to the verifier, together with an assertion whether  $R(x, w)$  holds, where  $R$  is a predetermined relation.) Essentially, in the commit phase of the commit-and-prove protocol, the committer commits to its input value  $w$  using some commitment scheme  $C$ , and in addition it proves to the receiver, using  $\mathcal{F}_{\text{ZK}}$  with an appropriate relation, that it “knows” the committed value. In the prove phase, where the committer wishes to assert that the committed value  $w$  stands in relation  $R$  with some public value  $x$ , the committer presents  $x$  and  $w$  to  $\mathcal{F}_{\text{ZK}}$  again — but this time the relation used by  $\mathcal{F}_{\text{ZK}}$  asserts two properties: first that  $R(x, w)$  holds, and second that  $w$  is the same value that was previously committed to.

To guarantee security against static adversaries, the commitment scheme of Naor [N91] is sufficient as an instantiation of the scheme  $C$ . We thus obtain a protocol for UC realizing  $\mathcal{F}_{\text{CP}}$  in the  $\mathcal{F}_{\text{ZK}}$ -hybrid model, based on any one-way function. To guarantee security against *adaptive* adversaries we need “adaptively secure” commitment schemes, namely commitment schemes where a simulator can generate “dummy commitments” which can be later opened in multiple ways. (In fact, a slightly stronger property is needed here, see details within.) Such commitments exist assuming the existence of trap-door permutations, as is demonstrated by our construction of universally composable commitments in Section 4.5. In all we obtain:

<sup>2</sup> In a concurrent and independent work [DN02], Damgård and Nielsen consider a functionality that has great resemblance to our commit-and-prove functionality, and construct universally composable protocols that realize this functionality under specific number-theoretic assumptions. Our commit-and-prove protocol is based on more general assumptions, whereas their protocol is considerably more efficient.

**Theorem 4.2.2** (two-party computation in the malicious model – informal): *Assume that trapdoor permutations exist. Then, for any two-party ideal functionality  $\mathcal{F}$ , there exists a protocol  $\Pi$  that UC realizes  $\mathcal{F}$  in the  $\mathcal{F}_{\text{ZK}}$ -hybrid model in the presence of malicious, static adversaries. Furthermore, if augmented two-party non-committing encryption protocols exist, then there exists a protocol  $\Pi$  that UC realizes  $\mathcal{F}$  in the  $\mathcal{F}_{\text{ZK}}$ -hybrid model in the presence of malicious, adaptive adversaries.*

As with Proposition 4.2.1, Theorem 4.2.2 is not stated exactly. It is formally restated in Section 4.8.2.

Let  $\mathcal{F}_{\text{CRS}}$  denote the common random string functionality (that is,  $\mathcal{F}_{\text{CRS}}$  provides all parties with a common, public string drawn from a predefined distribution). Then, as we show in Section 4.5, universally composable commitments can be UC realized in the  $\mathcal{F}_{\text{CRS}}$ -hybrid model, assuming the existence of trapdoor permutations. Furthermore, [CF01] showed that the  $\mathcal{F}_{\text{ZK}}$  functionality can be UC realized given universally composable commitments. Combining these results together, we have that  $\mathcal{F}_{\text{ZK}}$  can be UC realized in the  $\mathcal{F}_{\text{CRS}}$ -hybrid model, assuming the existence of trapdoor permutations. Using the composition theorem we obtain a similar result to Theorem 4.2.2, with the exception that  $\mathcal{F}$  is realized in the  $\mathcal{F}_{\text{CRS}}$ -hybrid model (rather than in the  $\mathcal{F}_{\text{ZK}}$ -hybrid model).

**On the distribution of the reference string.** In obtaining the above corollary, the common reference string is used only in the construction of the universally composable commitment scheme (which is used for obtaining  $\mathcal{F}_{\text{ZK}}$ ). As we have mentioned, in the  $\mathcal{F}_{\text{CRS}}$ -hybrid model, universally composable commitments can be obtained assuming the existence of trapdoor permutations only. However, in this case, the common reference string is *not* uniformly distributed. Nevertheless, a uniformly distributed string can be used, under the additional assumption of the existence of *dense cryptosystems* [DP92]. We therefore conclude that universally composable two-party computation can be obtained with a *uniformly distributed* reference string, under the assumption that the following primitives exist: trapdoor permutations, dense cryptosystems and, in the adaptive case, augmented two-party non-committing encryption protocols.

**Extensions to multi-party computation.** We now describe how the two-party construction of Theorem 4.2.2 is extended to the setting of multi-party computation, where any number of parties may be corrupt. For our construction, we assume that each set of interacting parties is given access to an *authenticated* broadcast channel. In the UC setting, this assumption can be securely realized for any number of corrupted parties.

The outline of our construction is as follows. Similarly to the two-party case, we first construct a multi-party protocol that is secure against semi-honest adversaries (as above, this protocol is essentially that of GMW). Then, we construct a protocol compiler (again, like that of GMW), that transforms

semi-honest protocols into ones that are secure even against malicious adversaries. This protocol compiler is constructed using a one-to-many extension of the commit-and-prove functionality, denoted  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ . (In the one-to-many extension, a *single* party commits and proves to *many* receivers/verifiers.) The extension of the protocol that UC realizes two-party  $\mathcal{F}_{\text{CP}}$  to a protocol that UC realizes one-to-many  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$  constitutes the main difference between the two-party and multi-party constructions. Therefore, in this outline, we focus exclusively on how this extension is achieved.

The first step in realizing  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ , is to construct one-to-many extensions of universal commitments and zero-knowledge. In a one-to-many commitment scheme, all parties receive the commitment (and the committer is bound to the same value for all parties). Likewise, in one-to-many zero-knowledge, all parties verify the proof (and they either all accept or all reject the proof). Now, any *non-interactive* commitment scheme can be transformed into a one-to-many equivalent by simply having the committer broadcast its message to all parties. Thus, this functionality is immediately obtained from our commitment scheme in Section 4.5 or from the scheme of [CF01] (both of these constructions are non-interactive). However, obtaining one-to-many zero-knowledge is more involved, since we do not know how to construct non-interactive adaptively-secure universally composable zero-knowledge.<sup>3</sup> Nevertheless, a one-to-many zero-knowledge protocol can be constructed based on the universally-composable zero-knowledge protocol of [CF01] and the methodology of [G02a] for obtaining a multi-party extension of zero-knowledge. Specifically, [CF01] show that parallel executions of the 3-round zero-knowledge protocol of Hamiltonicity is universally composable, when a universally composable commitment scheme is used for the prover's commitments. Thus, as in [G02a], the prover runs a copy of the above zero-knowledge protocol with each receiver over the broadcast channel, using the one-to-many commitment scheme for its commitments. Furthermore, each verifying party checks that the proofs of all the other parties are accepting (this is possible because the proof of Hamiltonicity is publicly verifiable and because all parties view all the communication). Thus, at the end of the protocol, all honest parties agree (without any additional communication) on whether the proof was successful or not. (Note also that the adversary cannot cause an honest prover's proof to be rejected.)

It remains to describe how to realize  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$  in the  $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$ -hybrid model. The basic idea is to generalize the  $\mathcal{F}_{\text{CP}}$  protocol. As with zero-knowledge, this is not straightforward because in the protocol for adaptive adversaries, the  $\mathcal{F}_{\text{CP}}$  commit-phase is interactive. Nevertheless, this problem is solved by having the committer commit to its input value  $w$  by separately running the protocol for the commit-phase of (two-party)  $\mathcal{F}_{\text{CP}}$  with every party over the broadcast

<sup>3</sup> In the case of static adversaries, the non-interactive zero-knowledge protocol of [DDOPS01] suffices. Thus, here too, the prover message can simply be broadcast and one-to-many zero-knowledge is obtained.

channel. Following this, the committer uses one-to-many zero-knowledge to prove that it committed to the same value in all of these commitments. (Since each party views the communication from all the commitments, every party can verify this zero-knowledge proof.) The prove phase is similar to the two-party case, except that the one-to-many extension of zero-knowledge is used (instead of two-party zero-knowledge).

Finally, we note that, as in the two-party case, a multi-party protocol compiler can be constructed in the  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ -hybrid model, with no further assumptions. Combining all of the above and recalling that  $\mathcal{F}_{\text{ZK}}$  can be obtained in the  $\mathcal{F}_{\text{CRS}}$ -hybrid model, we have the following theorem:

**Theorem 4.2.3** (multi-party computation in the malicious model – informal): *Assume that trapdoor permutations exist. Then, for any multi-party ideal functionality  $\mathcal{F}$ , there exists a protocol  $\Pi$  that UC realizes  $\mathcal{F}$  in the  $\mathcal{F}_{\text{CRS}}$ -hybrid model in the presence of malicious, static adversaries, and for any number of corruptions. Furthermore, if augmented two-party non-committing encryption protocols exist, then there exists a protocol  $\Pi$  that UC realizes  $\mathcal{F}$  in the  $\mathcal{F}_{\text{CRS}}$ -hybrid model in the presence of malicious, adaptive adversaries, and for any number of corruptions.*

As with Proposition 4.2.1, Theorem 4.2.3 is not stated exactly. It is formally restated in Section 4.9.4.

## 4.3 Preliminaries

Section 4.3.1 reviews the framework of [C01] and the universal composition theorem. In Section 4.3.2 we discuss issues that arise regarding universal composition when some amount of joint state between protocols is desired. Finally, Section 4.3.3 presents the class of functionalities which we will show how to UC realize. Before proceeding, we recall the definition of computational indistinguishability. A **distribution ensemble**  $X = \{X(k, a)\}_{k \in \mathbf{N}, a \in \{0,1\}^*}$  is an infinite set of probability distributions, where a distribution  $X(k, a)$  is associated with each  $k \in \mathbf{N}$  and  $a \in \{0,1\}^*$ . The ensembles considered in this work describe outputs where the parameter  $a$  represents input, and the parameter  $k$  is taken to be the **security parameter**. A distribution ensemble is called **binary** if it consists only of distributions over  $\{0,1\}$ . Then,

**Definition 4.3.1** *Two binary distribution ensembles  $X$  and  $Y$  are indistinguishable (written  $X \stackrel{c}{\equiv} Y$ ) if for any  $c \in \mathbf{N}$  there exists  $k_0 \in \mathbf{N}$  such that for all  $k > k_0$  and for all  $a$  we have*

$$|\Pr(X(k, a) = 1) - \Pr(Y(k, a) = 1)| < k^{-c}.$$

### 4.3.1 Universally Composable Security: The General Framework

We start by reviewing the syntax of message-driven protocols in asynchronous networks. We then present the real-life model of computation, the ideal process, and the general definition of UC realizing an ideal functionality. Next, we present the hybrid model and the composition theorem. The text is somewhat informal for clarity and brevity, and is mostly taken from the Overview section of [C01]. For full details see there.

**Protocol syntax.** Following [GMR85, G01], a protocol is represented as a system of probabilistic interactive Turing machines (ITMs), where each ITM represents the program to be run within a different party. Specifically, the input and output tapes model inputs and outputs that are received from and given to other programs running on the same machine, and the communication tapes model messages sent to and received from the network. Adversarial entities are also modelled as ITMs. We concentrate on a model where the adversaries have an arbitrary additional input, or an “advice” string. From a complexity-theoretic point of view, this essentially implies that adversaries are non-uniform ITMs.

In order to simplify the exposition, we introduce the following convention. We assume that all protocols are such that the parties read their input tapes only at the onset of a protocol execution. This can easily be achieved by having the parties copy their input tape onto an internal work tape. This convention prevents problems that may occur when parties’ input tapes are modified in the middle of a protocol execution (as is allowed in the model).

**The basic framework.** As sketched in Section 4.2, protocols that UC carry out a given task (or, protocol problem) are defined in three steps, as follows. First, the process of executing a protocol in the presence of an adversary and in a given computational environment is formalized. This is called the *real-life model*. Next, an *ideal process* for carrying out the task at hand is formalized. In the ideal process the parties do not communicate with each other. Instead they have access to an “ideal functionality”, which is essentially an incorruptible “trusted party” that is programmed to capture the desired functionality of the given task. A protocol is said to UC realize an ideal functionality if the process of running the protocol amounts to “emulating” the ideal process for that ideal functionality. We overview the model for protocol execution (called the *real-life model*), the ideal process, and the notion of protocol emulation.

We concentrate on the following model of computation, aimed at representing current realistic communication networks (such as the Internet). The communication takes place in an asynchronous, public network, without guaranteed delivery of messages. We assume that the communication is *authenticated* and thus the adversary cannot modify messages sent by honest

parties.<sup>4</sup> Furthermore, the adversary may only deliver messages that were previously sent by parties, and may deliver each message sent only once. The fact that the network is asynchronous means that the messages are not necessarily delivered in the order which they are sent. Parties may be broken into (i.e., become corrupted) throughout the computation, and once corrupted their behaviour is arbitrary (or, *malicious*). (Thus, our main consideration is that of malicious, adaptive adversaries. However, below we present the modifications necessary for modelling static and semi-honest adversaries.) We do not trust data erasures; rather, we postulate that all past states are available to the adversary upon corruption. Finally, all the involved entities are restricted to probabilistic polynomial time (or “feasible”) computation.

**Protocol execution in the real-life model.** We sketch the process of executing a given protocol  $\pi$  (run by parties  $P_1, \dots, P_n$ ) with some adversary  $\mathcal{A}$  and an environment machine  $\mathcal{Z}$  with input  $z$ . All parties have a security parameter  $k \in \mathbb{N}$  and are polynomial in  $k$ . The execution consists of a sequence of *activations*, where in each activation a single participant (either  $\mathcal{Z}$ ,  $\mathcal{A}$ , or some  $P_i$ ) is activated. The environment is activated first. In each activation it may read the contents of the output tapes of all the *uncorrupted* parties<sup>5</sup> and the adversary, and may write information on the input tape of one of the parties or of the adversary. Once the activation of the environment is complete (i.e, once the environment enters a special waiting state), the entity whose input tape was written on is activated next.

Once the adversary is activated, it may read its own tapes and the outgoing communication tapes of all parties. It may either deliver a message to some party by writing this message on the party’s incoming communication tape or *corrupt* a party. Only messages that were sent in the past by some party can be delivered, and each message can be delivered at most once. Upon corrupting a party, the adversary gains access to all the tapes of that party and controls all the party’s future actions. (We assume that the adversary also learns all the past internal states of the corrupted party. This means that the model does not assume effective cryptographic erasure of data.) In addition, whenever a party is corrupted the environment is notified (say, via a message that is added to the output tape of the adversary). If the adversary delivered a message to some uncorrupted party in its activation then this party is activated once the activation of the adversary is complete. Otherwise the environment is activated next.

<sup>4</sup> We remark that the basic model in [C01] postulates *unauthenticated* communication, i.e. the adversary may delete, modify, and generate messages at wish. Here we concentrate on authenticated networks for sake of simplicity. Authentication can be added in standard ways. Formally, the model here corresponds to the  $\mathcal{F}_{\text{AUTH}}$ -hybrid model in [C01].

<sup>5</sup> The environment is not given read access to the corrupted parties’ output tapes because once a party is corrupted, it is no longer activated. Rather, the adversary sends messages in its name. Therefore, the output tapes of corrupted parties are not relevant.

Once a party is activated (either due to an input given by the environment or due to a message delivered by the adversary), it follows its code and possibly writes local outputs on its output tape and outgoing messages on its outgoing communication tape. Once the activation of the party is complete the environment is activated. The protocol execution ends when the environment completes an activation without writing on the input tape of any entity. The output of the protocol execution is the output of the environment. We assume that this output consists of only a single bit.

In summary, the order of activations is as follows. The environment  $\mathcal{Z}$  is always activated first. The environment then either activates the adversary  $\mathcal{A}$  or some party  $P_i$  by writing on an input tape. If the adversary is activated, it may return control to the environment, or it may activate some party  $P_i$  by delivery a message to  $P_i$ . After  $P_i$  is activated, control is always returned to  $\mathcal{Z}$ . We stress that at any point, only a single party is activated. Furthermore,  $\mathcal{Z}$  and  $\mathcal{A}$  can only activate one other entity (thus only a single input is written by  $\mathcal{Z}$  per activation and likewise  $\mathcal{A}$  can deliver only message per activation).

Let  $\text{REAL}_{\pi, \mathcal{A}, \mathcal{Z}}(k, z, \bar{r})$  denote the output of environment  $\mathcal{Z}$  when interacting with adversary  $\mathcal{A}$  and parties running protocol  $\pi$  on security parameter  $k$ , input  $z$  and random tapes  $\bar{r} = r_{\mathcal{Z}}, r_{\mathcal{A}}, r_1, \dots, r_n$  as described above ( $z$  and  $r_{\mathcal{Z}}$  for  $\mathcal{Z}$ ,  $r_{\mathcal{A}}$  for  $\mathcal{A}$ ;  $r_i$  for party  $P_i$ ). Let  $\text{REAL}_{\pi, \mathcal{A}, \mathcal{Z}}(k, z)$  denote the random variable describing  $\text{REAL}_{\pi, \mathcal{A}, \mathcal{Z}}(k, z, \bar{r})$  when  $\bar{r}$  is uniformly chosen.

**The ideal process.** Security of protocols is defined via comparing the protocol execution in the real-life model to an *ideal process* for carrying out (a single instance of) the task at hand. A key ingredient in the ideal process is the **ideal functionality** that captures the desired functionality, or the specification, of the task. The ideal functionality is modelled as another ITM that interacts with the environment and the adversary via a process described below. More specifically, the ideal process involves an ideal functionality  $\mathcal{F}$ , an **ideal process adversary** (or **simulator**)  $\mathcal{S}$ , an environment  $\mathcal{Z}$  with input  $z$ , and a set of **dummy parties**  $\tilde{P}_1, \dots, \tilde{P}_n$ .

As in the process of protocol execution in the real-life model, the environment is activated first. As there, in each activation it may read the contents of the output tapes of all (dummy) parties and the adversary, and may write information on the input tape of either one of the (dummy) parties or of the adversary. Once the activation of the environment is completed the entity whose input tape was written on is activated next.

The dummy parties are fixed and simple ITMs: Whenever a dummy party is activated with an input, it writes it on its outgoing communication tape for the ideal functionality  $\mathcal{F}$ . Furthermore, whenever a dummy party is activated due to delivery of some message (from  $\mathcal{F}$ ), it copies this message to its output. At the conclusion of a dummy party's activation, the environment  $\mathcal{Z}$  is activated. The communication by the dummy parties is with the ideal functionality  $\mathcal{F}$  only. In principle, these messages sent between the dummy parties and  $\mathcal{F}$  are secret and cannot be read by the adversary  $\mathcal{S}$ . However,



these messages are actually comprised of two parts: a *header* and *contents*. The header is public and can be read by  $\mathcal{S}$ , whereas the contents is private and cannot be read by  $\mathcal{S}$ . The definition of the functionality states which information is in the “header” and which is in the “contents”. Some information must clearly be in the public header; for example, the identity of the party to whom the functionality wishes to send output must be public so that it can be delivered. Beyond that, the functionality definition should specify what is in the “header” and what is in the “contents”. However, for all the functionalities considered in this paper, there is a fixed format for the header. Specifically, the header contains the type of action being taken, the session identifier and the participating parties. For example, a commit message is of the following format:  $(\text{commit}, \text{sid}, P_i, P_j, b)$ , where “commit” states that the party is committing to a new value, *sid* is the session identifier,  $P_i$  is the committing party,  $P_j$  is the receiving party and  $b$  is the value being committed to. In this case, the header consists of  $(\text{commit}, \text{sid}, P_i, P_j)$  and the contents consists of  $(b)$  only. We adopt this format by convention for all (but one of) the functionalities defined in this paper. We therefore omit specific reference to the header and contents in our definitions of functionalities (and explicitly describe the difference for the one functionality which does not have this format).

When the ideal functionality  $\mathcal{F}$  is activated, it reads the contents of its incoming communication tape, and potentially sends messages to the parties and to the adversary by writing these messages on its outgoing communication tape. Once the activation of  $\mathcal{F}$  is complete, the environment  $\mathcal{Z}$  is activated next.

When the adversary  $\mathcal{S}$  is activated, it may read its own input tape and in addition it can read the *public headers* of the messages on the outgoing communication tape of  $\mathcal{F}$ . In contrast,  $\mathcal{S}$  cannot read the *private contents* of these messages (unless the recipient of the message is  $\mathcal{S}$  or a corrupted party<sup>6</sup>). Likewise,  $\mathcal{S}$  can read the public headers of the messages intended for  $\mathcal{F}$  that appear on the outgoing communication tapes of the dummy parties. Then,  $\mathcal{S}$  can execute one of the following four actions: It may either **write** a message from itself on  $\mathcal{F}$ ’s incoming communication tape<sup>7</sup>, **deliver** a message to  $\mathcal{F}$  from some party  $P_i$  by copying it from  $P_i$ ’s outgoing communication

<sup>6</sup> Note that the ideal process allows  $\mathcal{S}$  to obtain the output values sent by  $\mathcal{F}$  to the corrupted parties as soon as they are generated. Furthermore, if at the time that  $\mathcal{S}$  corrupts some party  $P_i$  there are messages sent from  $\mathcal{F}$  to  $P_i$ , then  $\mathcal{S}$  immediately obtains the contents of these messages.

<sup>7</sup> Many natural ideal functionalities indeed send messages to the adversary  $\mathcal{S}$  (see the commitments and zero-knowledge functionalities of Sections 4.5 and 4.6 for examples). On the other hand, having the adversary send messages to  $\mathcal{F}$  is less common. Nevertheless, this option can be useful in order to relax the requirements on protocols that realize the functionality. For example, it may be easier to obtain coin-tossing if the adversary is allowed to bias some of the bits of the result. If this is acceptable for the application in mind, we can allow the adversary this capability by having it send its desired bias to  $\mathcal{F}$ .

tape to  $\mathcal{Z}$ 's incoming communication tape, deliver a message from  $\mathcal{F}$  to  $P_i$  by copying the appropriate message from  $\mathcal{Z}$ 's outgoing communication tape to  $P_i$ 's incoming communication tape, or corrupt a party. Upon corrupting a party, both  $\mathcal{Z}$  and  $\mathcal{F}$  learn the identity of the corrupted party (say, a special message is written on their respective incoming communication tapes).<sup>8</sup> In addition, the adversary learns all the past inputs and outputs of the party. Finally, the adversary controls the party's actions from the time that the corruption takes place.

If the adversary delivered a message to some uncorrupted (dummy) party  $P_i$  or to the functionality  $\mathcal{F}$  in an activation, then this entity (i.e.,  $P_i$  or  $\mathcal{F}$ ) is activated once the activation of the adversary  $\mathcal{S}$  is complete. Otherwise the environment  $\mathcal{Z}$  is activated next.

As in the real-life model, the protocol execution ends when the environment completes an activation without writing on the input tape of any entity. The output of the protocol execution is the (one bit) output of  $\mathcal{Z}$ .

In summary, the order of activations in the ideal model is as follows. As in the real model, the environment  $\mathcal{Z}$  is always activated first, and then activates either the adversary  $\mathcal{S}$  or some dummy party  $P_i$  by writing an input. If the adversary  $\mathcal{S}$  is activated, then it either activates a dummy party  $P_i$  or the ideal functionality  $\mathcal{F}$  by delivering the entity a message, or it returns control to the environment. After the activation of a dummy party or the functionality, the environment is always activated next.

Let  $\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k, z, \bar{r})$  denote the output of environment  $\mathcal{Z}$  after interacting in the ideal process with adversary  $\mathcal{S}$  and ideal functionality  $\mathcal{F}$ , on security parameter  $k$ , input  $z$ , and random input  $\bar{r} = r_{\mathcal{Z}}, r_{\mathcal{S}}, r_{\mathcal{F}}$  as described above ( $z$  and  $r_{\mathcal{Z}}$  for  $\mathcal{Z}$ ,  $r_{\mathcal{S}}$  for  $\mathcal{S}$ ;  $r_{\mathcal{F}}$  for  $\mathcal{F}$ ). Let  $\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k, z)$  denote the random variable describing  $\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k, z, \bar{r})$  when  $\bar{r}$  is uniformly chosen.

*Remark.* The above definition of the ideal model slightly differs from that of [C01]. Specifically, in [C01] messages between the dummy parties and ideal functionality are delivered *immediately*. In contrast, in our presentation, this message delivery is carried out by the adversary. Thus, in both the real and ideal models, *all* message delivery is the responsibility of the adversary alone. We note that our results can also be stated in the model of “immediate delivery” as defined in [C01]. However, in such a case, the functionality should always ask the adversary whether or not to accept an input and send an output. This has the same effect as when the adversary is responsible for the delivery of these messages. Therefore, the final result is the same.

**UC realizing an ideal functionality.** We say that a protocol  $\pi$  UC realizes an ideal functionality  $\mathcal{F}$  if for any real-life adversary  $\mathcal{A}$  there exists an ideal-process adversary  $\mathcal{S}$  such that no environment  $\mathcal{Z}$ , on any input, can tell

<sup>8</sup> Allowing  $\mathcal{F}$  to know which parties are corrupted gives it considerable power. This power provides greater freedom in formulating ideal functionalities for capturing the requirements of given tasks. On the other hand, it also inherently limits the scope of general realizability theorems. See more discussion in Section 4.3.3.

with non-negligible probability whether it is interacting with  $\mathcal{A}$  and parties running  $\pi$  in the real-life process, or with  $\mathcal{S}$  and  $\mathcal{F}$  in the ideal process. This means that, from the point of view of the environment, running protocol  $\pi$  is ‘just as good’ as interacting with an ideal process for  $\mathcal{F}$ . (In a way,  $\mathcal{Z}$  serves as an “interactive distinguisher” between the two processes. Here it is important that  $\mathcal{Z}$  can provide the process in question with *adaptively chosen* inputs throughout the computation.) We have:

**Definition 4.3.2** *Let  $n \in \mathbf{N}$ . Let  $\mathcal{F}$  be an ideal functionality and let  $\pi$  be an  $n$ -party protocol. We say that  $\pi$  UC realizes  $\mathcal{F}$  if for any adversary  $\mathcal{A}$  there exists an ideal-process adversary  $\mathcal{S}$  such that for any environment  $\mathcal{Z}$ ,*

$$\{\text{IDEAL}_{\mathcal{F}, \mathcal{S}, \mathcal{Z}}(k, z)\}_{k \in \mathbf{N}, z \in \{0,1\}^*} \stackrel{c}{=} \{\text{REAL}_{\pi, \mathcal{A}, \mathcal{Z}}(k, z)\}_{k \in \mathbf{N}, z \in \{0,1\}^*} \quad (4.1)$$

**Non-trivial protocols and the requirement to generate output.** Recall that the ideal process does not require the ideal-process adversary to deliver messages that are sent by the ideal functionality to the dummy parties. Consequently, the definition provides no guarantee that a protocol will ever generate output or “return” to the calling protocol. Indeed, in our setting where message delivery is not guaranteed, it is impossible to ensure that a protocol “terminates” or generates output. Rather, the definition concentrates on the security requirements *in the case that the protocol generates output*.

A corollary of the above fact is that a protocol that “hangs”, never sends any messages and never generates output, UC realizes any ideal functionality. Thus, in order to obtain a meaningful feasibility result, we introduce the notion of a **non-trivial protocol**. Such a protocol has the property that if the real-life adversary delivers all messages and does not corrupt any parties, then the ideal-process adversary also delivers all messages and does not corrupt any parties. Note that in a non-trivial protocol, a party may not necessarily receive output. However, this only happens if either the functionality does not specify output for this party, or if the real-life adversary actively interferes in the execution (by either corrupting parties or refusing to deliver some messages). Our main result is to show the existence of *non-trivial* protocols for UC realizing any ideal functionality. All our protocols are in fact clearly non-trivial; therefore, we ignore this issue from here on.

**Relaxations of Definition 4.3.2.** We recall two standard relaxations of the definition:

- *Static (non-adaptive) adversaries.* Definition 4.3.2 allows the adversary to corrupt parties throughout the computation. A simpler (and somewhat weaker) variant forces the real-life adversary to corrupt parties only at the onset of the computation, before any uncorrupted party is activated. We call such adversaries **static**.

- *Passive (semi-honest) adversaries.* Definition 4.3.2 gives the adversary complete control over corrupted parties (such an adversary is called **malicious**). Specifically, the model states that from the time of corruption the corrupted party is no longer activated, and instead the adversary sends messages in the name of that party. In contrast, when a **semi-honest** adversary corrupts a party, the party continues to follow the prescribed protocol. Nevertheless, the adversary is given read access to the internal state of the party at all times, and is also able to modify the values that the environment writes on the corrupted parties' input tapes.<sup>9</sup> Formally, if in a given activation, the environment wishes to write information on the input tape of a *corrupted* party, then the environment first passes the adversary the value  $x$  that it wishes to write (along with the identity of the party whose input tape it wishes to write to). The adversary then passes a (possibly different) value  $x'$  back to the environment. Finally, the environment writes  $x'$  on the input tape of the corrupted party, following which the corrupted party is activated. We stress that when the environment writes on the input tape of an honest party, the adversary learns nothing of the value and cannot modify it. Everything else remains the same as in the above-described malicious model. We say that protocol  $\pi$  UC realizes functionality  $\mathcal{F}$  for *semi-honest adversaries*, if for any semi-honest real-life adversary  $\mathcal{A}$  there exists an ideal-process semi-honest adversary  $\mathcal{S}$  such that Eq. (4.1) holds for any environment  $\mathcal{Z}$ .

**The composition theorem.** In order to state the composition theorem, and in particular in order to formalize the notion of a real-life protocol with access to multiple copies of an ideal functionality, the **hybrid model of computation** with access to an ideal functionality  $\mathcal{F}$  (or, in short, the  $\mathcal{F}$ -hybrid model) is formulated. This model is identical to the real-life model, with the following additions. On top of sending messages to each other, the parties may send messages to and receive messages from an unbounded number of copies of  $\mathcal{F}$ . Each copy of  $\mathcal{F}$  is identified via a unique **session identifier** (SID); all messages addressed to this copy and all message sent by this copy carry the corresponding SID. (Sometimes a copy of  $\mathcal{F}$  will interact only with a subset of the parties. The identities of these parties is determined by the protocol in the  $\mathcal{F}$ -hybrid model.)

The communication between the parties and each one of the copies of  $\mathcal{F}$  mimics the ideal process. That is, when the adversary delivers a message from a party to a copy of  $\mathcal{F}$  with a particular SID, that copy of  $\mathcal{F}$  is the next entity to be activated. (If no such copy of  $\mathcal{F}$  exists then a new copy of  $\mathcal{F}$  is created and activated to receive the message.) Furthermore, although the adversary

<sup>9</sup> Allowing a semi-honest adversary to modify a corrupted party's input is somewhat non-standard. However, this simplifies the presentation of this work (and in particular the protocol compiler). All the protocols presented for the semi-honest model in this paper are secure both when the adversary can modify a corrupted party's input tape and when it cannot.

in the hybrid model is responsible for delivering the messages between the copies of  $\mathcal{F}$  and the parties, it does not have access to the contents of these messages.

The hybrid model does not specify how the SIDs are generated, nor does it specify how parties “agree” on the SID of a certain protocol copy that is to be run by them. These tasks are left to the protocol in the hybrid model. This convention simplifies formulating ideal functionalities, and designing protocols that UC realize them, by freeing the functionality from the need to choose the SIDs and guarantee their uniqueness. In addition, it seems to reflect common practice of protocol design in existing networks. See more discussion following Theorem 4.3.3 below.

Let  $\text{HYBRID}_{\pi, \mathcal{A}, \mathcal{Z}}^{\mathcal{F}}(k, z)$  denote the random variable describing the output of environment machine  $\mathcal{Z}$  on input  $z$ , after interacting in the  $\mathcal{F}$ -hybrid model with protocol  $\pi$  and adversary  $\mathcal{A}$ , analogously to the definition of  $\text{REAL}_{\pi, \mathcal{A}, \mathcal{Z}}(k, z)$ . (We stress that here  $\pi$  is a hybrid of a real-life protocol with ideal evaluation calls to  $\mathcal{F}$ .)

**Replacing a call to  $\mathcal{F}$  with a protocol invocation.** Let  $\pi$  be a protocol in the  $\mathcal{F}$ -hybrid model, and let  $\rho$  be a protocol that UC realizes  $\mathcal{F}$  (with respect to some class of adversaries). The composed protocol  $\pi^\rho$  is constructed by modifying the code of each ITM in  $\pi$  so that the first message sent to each copy of  $\mathcal{F}$  is replaced with an invocation of a new copy of  $\rho$  with fresh random input, with the same SID, and with the contents of that message as input. Each subsequent message to that copy of  $\mathcal{F}$  is replaced with an activation of the corresponding copy of  $\rho$ , with the contents of that message given to  $\rho$  as new input. Each output value generated by a copy of  $\rho$  is treated as a message received from the corresponding copy of  $\mathcal{F}$ . (See [c01] for more details on the operation of “composed protocols”, where a party, i.e. an ITM, runs multiple protocol-instances concurrently.)

If protocol  $\rho$  is a protocol in the real-life model then so is  $\pi^\rho$ . If  $\rho$  is a protocol in some  $\mathcal{G}$ -hybrid model (i.e.,  $\rho$  uses ideal evaluation calls to some functionality  $\mathcal{G}$ ) then so is  $\pi^\rho$ .

**Theorem statement.** In its general form, the composition theorem basically says that if  $\rho$  UC realizes  $\mathcal{F}$  in the  $\mathcal{G}$ -hybrid model for some functionality  $\mathcal{G}$ , then an execution of the composed protocol  $\pi^\rho$ , running in the  $\mathcal{G}$ -hybrid model, “emulates” an execution of protocol  $\pi$  in the  $\mathcal{F}$ -hybrid model. That is, for any adversary  $\mathcal{A}$  in the  $\mathcal{G}$ -hybrid model there exists an adversary  $\mathcal{S}$  in the  $\mathcal{F}$ -hybrid model such that no environment machine  $\mathcal{Z}$  can tell with non-negligible probability whether it is interacting with  $\mathcal{A}$  and  $\pi^\rho$  in the  $\mathcal{G}$ -hybrid model or it is interacting with  $\mathcal{S}$  and  $\pi$  in the  $\mathcal{F}$ -hybrid model.

A corollary of the general theorem states that if  $\pi$  UC realizes some functionality  $\mathcal{I}$  in the  $\mathcal{F}$ -hybrid model, and  $\rho$  UC realizes  $\mathcal{F}$  in the  $\mathcal{G}$ -hybrid model, then  $\pi^\rho$  UC realizes  $\mathcal{I}$  in the  $\mathcal{G}$ -hybrid model. (Here one has to define what it means to UC realize functionality  $\mathcal{I}$  in the  $\mathcal{F}$ -hybrid model. This is done in the natural way.) That is:

**Theorem 4.3.3** ([C01]) *Let  $\mathcal{F}, \mathcal{G}, \mathcal{I}$  be ideal functionalities. Let  $\pi$  be an  $n$ -party protocol in the  $\mathcal{F}$ -hybrid model, and let  $\rho$  be an  $n$ -party protocol that UC realizes  $\mathcal{F}$  in the  $\mathcal{G}$ -hybrid model. Then for any adversary  $\mathcal{A}$  in the  $\mathcal{G}$ -hybrid model there exists an adversary  $\mathcal{S}$  in the  $\mathcal{F}$ -hybrid model such that for any environment machine  $\mathcal{Z}$  we have:*

$$\left\{ \text{HYBRID}_{\pi^\rho, \mathcal{A}, \mathcal{Z}}^{\mathcal{G}}(k, z) \right\}_{k \in \mathbb{N}, z \in \{0,1\}^*} \stackrel{c}{=} \left\{ \text{HYBRID}_{\pi, \mathcal{S}, \mathcal{Z}}^{\mathcal{F}}(k, z) \right\}_{k \in \mathbb{N}, z \in \{0,1\}^*}$$

*In particular, if  $\pi$  UC realizes functionality  $\mathcal{I}$  in the  $\mathcal{F}$ -hybrid model then  $\pi^\rho$  UC realizes  $\mathcal{I}$  in the  $\mathcal{G}$ -hybrid model.*

Consider the case that  $\mathcal{G}$  is the empty functionality, and so the  $\mathcal{G}$ -hybrid model is actually the *real model*. Then, Theorem 4.3.3 states that  $\rho$  remains secure when run concurrently with *any* protocol  $\pi$ . In other words,  $\rho$  remains secure under concurrent general composition (with arbitrary sets of parties). We note that although  $\pi$  technically seems to be a “calling protocol”, it can also represent arbitrary network activity. Thus, we obtain that  $\rho$  remains secure when run concurrently in an arbitrary network.

**On the uniqueness of the session IDs.** The session IDs play a central role in the hybrid model and the composition operation, in that they enable the parties to distinguish different instances of a protocol. Indeed, differentiating protocol instances via session IDs is a natural and common mechanism in protocol design.

Yet, the current formulation of the hybrid model provides a somewhat over-idealized treatment of session IDs. Specifically, it is assumed that the session IDs are *globally unique* and *common to all parties*. That is, it is assumed that no two copies of an ideal functionality with the same session ID exist, even if the two copies have different (and even disjoint) sets of participants. Furthermore, all parties are assumed to hold the same SID (and they must somehow have agreed upon it). This treatment greatly simplifies the exposition of the model and the definition of ideal functionalities and protocols that realize them. Nonetheless, it is somewhat restrictive in that it requires the protocol in the hybrid model to guarantee global uniqueness of common session IDs. This may be hard (or even impossible) to achieve in the case that the protocol in the hybrid model is truly distributed and does not involve global coordination. See [LLR02] for more discussion on this point.

### 4.3.2 Universal Composition with Joint State

Traditionally, composition operations among protocols assume that the composed protocol instances have disjoint states, and in particular independent local randomness. The universal composition operation is no exception: if protocol  $\rho$  UC realizes some ideal functionality  $\mathcal{F}$ , and protocol  $\pi$  in the  $\mathcal{F}$ -hybrid model uses  $m$  copies of  $\mathcal{F}$ , then the composed protocol  $\pi^\rho$  uses  $m$  independent copies of  $\rho$ , and no two copies of  $\rho$  share any amount of state.

This property of universal composition (and of protocol composition in general) is bothersome in our context, where we wish to construct and analyze protocols in the common reference string (CRS) model. Let us elaborate. Assume that we follow the natural formalization of the CRS model as the  $\mathcal{F}_{\text{CRS}}$ -hybrid model, where  $\mathcal{F}_{\text{CRS}}$  is the functionality that chooses a string from the specified distribution and hands it to all parties. Now, assume that we construct a protocol  $\rho$  that UC realizes some ideal functionality  $\mathcal{F}$  in the  $\mathcal{F}_{\text{CRS}}$ -hybrid model (say, let  $\mathcal{F}$  be the commitment functionality,  $\mathcal{F}_{\text{COM}}$ ). Assume further that some higher level protocol  $\pi$  (in the  $\mathcal{F}$ -hybrid model) uses multiple copies of  $\mathcal{F}$ , and that we use the universal composition operation to replace each copy of  $\mathcal{F}$  with an instance of  $\rho$ . We now obtain a protocol  $\pi^\rho$  that runs in the  $\mathcal{F}_{\text{CRS}}$ -hybrid model and emulates  $\pi$ . However, this protocol is highly wasteful of the reference string. Specifically, each instance of  $\rho$  in  $\pi^\rho$  has its own separate copy of  $\mathcal{F}_{\text{CRS}}$ , or in other words each instance of  $\rho$  requires its own independent copy of the reference string. This stands in sharp contrast with our common view of the CRS model, where an unbounded number of protocol instances should be able to use the *same copy* of the reference string.

One way to get around this limitation of universal composition (and composition theorems in general) is to treat the entire, multi-session interaction as a single instance of a more complex protocol, and then to explicitly require that all sessions use the same copy of the reference string. More specifically, proceed as follows. First, given a functionality  $\mathcal{F}$  as described above, define a functionality,  $\hat{\mathcal{F}}$ , called the “multi-session extension of  $\mathcal{F}$ ”. Functionality  $\hat{\mathcal{F}}$  will run multiple copies of  $\mathcal{F}$ , where each copy will be identified by a special “sub-session identifier”, *ssid*. Upon receiving a message for the copy associated with *ssid*,  $\hat{\mathcal{F}}$  activates the appropriate copy of  $\mathcal{F}$  (running within  $\hat{\mathcal{F}}$ ), and forwards the incoming message to that copy. If no such copy of  $\mathcal{F}$  exists then a new copy is invoked and is given that *ssid*. Outputs generated by the copies of  $\mathcal{F}$  are copied to  $\hat{\mathcal{F}}$ ’s output. The next step after having defined  $\hat{\mathcal{F}}$  is to construct protocols that directly realize  $\hat{\mathcal{F}}$  in the  $\mathcal{F}_{\text{CRS}}$ -hybrid model, while making sure that the constructed protocols use only a single copy of  $\mathcal{F}_{\text{CRS}}$ .

This approach works, in the sense that it allows constructing and analyzing universally composable protocols that are efficient in their use of the reference string. However, it results in a cumbersome and non-modular formalization of ideal functionalities and protocols in the CRS model. Specifically, if we want to make sure that multiple sessions of some protocol (or set of protocols) use the same copy of the reference string, then we must treat all of these sessions (that may take place among different sets of parties) as a single instance of some more complex protocol. For example, assume that we want to construct commitments in the CRS model, then use these commitments to construct UC zero-knowledge protocols, and then use these protocols in yet higher-level protocols. Then, in any level of protocol design, we must design functionalities and protocols that explicitly deal with multiple

sessions. Furthermore, we must prove the security of these protocols within this multi-session setting. This complexity obviates much of the advantages of universal composition (and protocol composition in general).

In contrast, we would like to be able to formulate a functionality that captures only a single instance of some interaction, realize this functionality by some protocol  $\pi$  in the CRS model, and then UC compose multiple copies of  $\pi$  in spite of the fact that all copies use the same copy of the reference string. This approach is, in general, dangerous, since it can lead to insecure protocols. However, there are conditions under which such “composition with joint state” maintains security. This section describes a general tool that enables the composition of protocols even when they have some amount of joint state, under some conditions. Using this tool, suggested in [CR03] and called **universal composition with joint state (JUC)**, we are able to state and realize most of the functionalities in this work as functionalities for a single session, while still ending up with protocols where an unbounded number of instances use the same copy of the common reference string. This greatly simplifies the presentation while not detracting from the composability and efficiency of the presented protocols.

In a nutshell, universal composition with joint state is a new composition operation that can be sketched as follows. Let  $\mathcal{F}$  be an ideal functionality, and let  $\pi$  be a protocol in the  $\mathcal{F}$ -hybrid model. Let  $\hat{\mathcal{F}}$  denote the “multi-session extension of  $\mathcal{F}$ ” sketched above, and let  $\hat{\rho}$  be a protocol that UC realizes  $\hat{\mathcal{F}}$ . Then construct the composed protocol  $\pi^{[\hat{\rho}]}$  by replacing *all copies* of  $\mathcal{F}$  in  $\pi$  by a *single copy* of  $\hat{\rho}$ . (We stress that  $\pi$  assumes that it has access to multiple *independent* copies of  $\mathcal{F}$ . Still, we replace all copies of  $\mathcal{F}$  with a single copy of some protocol.) The JUC theorem states that protocol  $\pi^{[\hat{\rho}]}$ , running in the real-life model, “emulates”  $\pi$  in the usual sense. A more detailed presentation follows.

**The multi-session extension of an ideal functionality.** We formalize the notion of a multi-session extension of an ideal functionality, sketched above. Let  $\mathcal{F}$  be an ideal functionality. Recall that  $\mathcal{F}$  expects each incoming message to contain a special field consisting of its **session ID (SID)**. All messages received by  $\mathcal{F}$  are expected to have the same SID. (Messages that have different SIDs than that of the first message are ignored.) Similarly, all outgoing messages generated by  $\mathcal{F}$  carry the same SID.

The multi-session extension of  $\mathcal{F}$ , denoted  $\hat{\mathcal{F}}$ , is defined as follows.  $\hat{\mathcal{F}}$  expects each incoming message to contain *two* special fields. The first is the usual SID field as in any ideal functionality. The second field is called the **sub-session ID (SSID)** field. Upon receiving a message  $(sid, ssid, v)$  (where  $sid$  is the SID,  $ssid$  is the SSID, and  $v$  is an arbitrary value or list of values),  $\hat{\mathcal{F}}$  first verifies that  $sid$  is the same as that of the first message, otherwise the message is ignored. Next,  $\hat{\mathcal{F}}$  checks if there is a running copy of  $\mathcal{F}$  whose session ID is  $ssid$ . If so, then  $\hat{\mathcal{F}}$  activates that copy of  $\mathcal{F}$  with incoming message  $(ssid, v)$ , and follows the instructions of this copy. Otherwise, a new copy of  $\mathcal{F}$  is



invoked (within  $\hat{\mathcal{F}}$ ) and immediately activated with input  $(ssid, v)$ . From now on, this copy is associated with sub-session ID  $ssid$ . Whenever a copy of  $\mathcal{F}$  sends a message  $(ssid, v')$  to some party  $P_i$ ,  $\hat{\mathcal{F}}$  sends  $(sid, ssid, v')$  to  $P_i$ , and sends  $ssid$  to the adversary. (Sending  $ssid$  to the adversary implies that  $\hat{\mathcal{F}}$  does not hide which copy of  $\mathcal{F}$  is being activated within  $\hat{\mathcal{F}}$ .)

**The composition operation.** Let  $\mathcal{F}$  be an ideal functionality. The composition operation, called **universal composition with joint state (JUC)**, takes two protocols as arguments: a protocol  $\pi$  in the  $\mathcal{F}$ -hybrid model and a protocol  $\hat{\rho}$  that UC realizes  $\hat{\mathcal{F}}$ . Notice that  $\pi$  utilizes calls to  $\mathcal{F}$ , and not to  $\hat{\mathcal{F}}$ . Nevertheless, the JUC operation shows how to compose these together. The result is a composed protocol denoted  $\pi^{[\hat{\rho}]}$  and described as follows.

Recall that the  $\mathcal{F}$ -hybrid model is identical to the real-life model of computation, with the exception that the parties have access to multiple copies of  $\mathcal{F}$ . The different copies of  $\mathcal{F}$  are identified via their SIDs as described above. Let  $\mathcal{F}_{(sid)}$  denote the copy of functionality  $\mathcal{F}$  with SID  $sid$ . Protocol  $\pi^{[\hat{\rho}]}$  behaves like  $\pi$  with the following exceptions:<sup>10</sup>

1. When activated for the first time within party  $P_i$ ,  $\pi^{[\hat{\rho}]}$  invokes a copy of protocol  $\hat{\rho}$  with SID  $sid_0$ . That is, a copy of the  $i^{\text{th}}$  Interactive Turing Machine in  $\hat{\rho}$  is invoked as a subroutine within  $P_i$ , and is (locally) given identifier  $sid_0$ . No activation of  $\hat{\rho}$  occurs yet. ( $sid_0$  is some fixed, predefined value. For instance, set  $sid_0 = 0$ .)
2. Whenever  $\pi$  instructs party  $P_i$  to send a message  $(sid, v)$  to  $\mathcal{F}_{(sid)}$ , protocol  $\pi^{[\hat{\rho}]}$  instructs  $P_i$  to activate  $\hat{\rho}$  with input value  $(sid_0, sid, v)$ .
3. Whenever protocol  $\hat{\rho}$  instructs  $P_i$  to send a message  $m$  to some party  $P_j$ ,  $P_i$  writes the message  $(\hat{\rho}, sid, P_j, m)$  on its outgoing communication tape.
4. Whenever activated due to delivery of a message  $(\hat{\rho}, sid, P_i, m)$  from  $P_j$ ,  $P_i$  activates  $\hat{\rho}$  with incoming message  $(sid, P_j, m)$ .
5. Whenever (the single copy of)  $\hat{\rho}$  generates an output value  $(sid_0, sid, v)$ , proceed just as  $\pi$  proceeds with incoming message  $(sid, v)$  from  $\mathcal{F}_{(sid)}$ .

In short, whenever  $\pi$  makes an ideal call to  $\mathcal{F}$  with a given identifier  $sid$ , protocol  $\hat{\rho}$  is called with identifier  $sid_0$  and *sub-session* identifier  $sid$ . This is consistent with the fact that  $\hat{\rho}$  is a multi-session protocol. Conversely, whenever  $\hat{\rho}$  returns a value associated with identifiers  $(sid_0, sid)$ , this value is returned to  $\pi$  with (the single) identifier  $sid$ . This is consistent with the fact that  $\pi$  “thinks” that it is interacting with a single-session functionality  $\mathcal{F}$ .

**Theorem statement.** The JUC theorem asserts that if  $\hat{\rho}$  UC realizes  $\hat{\mathcal{F}}$ , then protocol  $\pi^{[\hat{\rho}]}$  behaves essentially like  $\pi$  with ideal access to multiple independent copies of  $\mathcal{F}$ . More precisely,

<sup>10</sup> For simplicity, we assume that  $\hat{\rho}$  UC realizes  $\hat{\mathcal{F}}$  in the real-life model of computation. The composition operation and theorem can be extended in a natural way to account for protocols  $\hat{\rho}$  that UC realize  $\hat{\mathcal{F}}$  in the  $\mathcal{G}$ -hybrid model for some ideal functionality  $\mathcal{G}$ .

**Theorem 4.3.4** (universal composition with joint state [CR03]): *Let  $\mathcal{F}, \mathcal{G}$  be ideal functionalities. Let  $\pi$  be a protocol in the  $\mathcal{F}$ -hybrid model, and let  $\hat{\rho}$  be a protocol that UC realizes  $\hat{\mathcal{F}}$ , the multi-session extension of  $\mathcal{F}$ , in the  $\mathcal{G}$ -hybrid model. Then the composed protocol  $\pi^{[\hat{\rho}]}$  in the  $\mathcal{G}$ -hybrid model emulates protocol  $\pi$  in the  $\mathcal{F}$ -hybrid model. That is, for any adversary  $\mathcal{A}$  there exists an adversary  $\mathcal{S}$  such that for any environment  $\mathcal{Z}$  we have*

$$\left\{ \text{HYBRID}_{\pi, \mathcal{S}, \mathcal{Z}}^{\mathcal{F}}(k, z) \right\}_{k \in \mathbf{N}, z \in \{0,1\}^*} \stackrel{c}{=} \left\{ \text{HYBRID}_{\pi^{[\hat{\rho}]}, \mathcal{A}, \mathcal{Z}}^{\mathcal{G}}(k, z) \right\}_{k \in \mathbf{N}, z \in \{0,1\}^*}$$

*In particular, if  $\pi$  UC realizes some functionality  $\mathcal{I}$  in the  $\mathcal{F}$ -hybrid model then  $\pi^{[\hat{\rho}]}$  UC realizes  $\mathcal{I}$  in the  $\mathcal{G}$ -hybrid model.*

**Discussion.** Jumping ahead, we sketch our use of the JUC theorem. Recall the commitment functionality,  $\mathcal{F}_{\text{COM}}$ , formalized in [CF01]. This functionality captures the process of commitment and decommitment to a single value, performed by two parties. In addition, [CF01] show how to realize  $\hat{\mathcal{F}}_{\text{COM}}$  in the CRS model, using a single copy of the CRS for all commitments. (In [CF01] the multi-session extension  $\hat{\mathcal{F}}_{\text{COM}}$  is called  $\mathcal{F}_{\text{MCOM}}$ .) An alternative protocol that UC realizes  $\hat{\mathcal{F}}_{\text{COM}}$  is also presented here.

In this work we construct protocols that use these commitment protocols. However, to preserve modularity of exposition, we present our protocols in the  $\mathcal{F}_{\text{COM}}$ -hybrid model, while allowing the protocols to use multiple copies of  $\mathcal{F}_{\text{COM}}$  and thus enjoy full modularity. We then use universal composition with joint state to compose any protocol  $\pi$  in the  $\mathcal{F}_{\text{COM}}$ -hybrid model with any protocol  $\hat{\rho}$  that UC realizes  $\hat{\mathcal{F}}_{\text{COM}}$  using a single copy of the reference string, to obtain a protocol  $\pi^{[\hat{\rho}]}$  that emulates  $\pi$  and uses only a single copy of the reference string for all the commitments. (We remark that the same technique is applied also to protocols that use the ideal zero-knowledge functionality,  $\mathcal{F}_{\text{ZK}}$ . See more details in Section 4.6.)

### 4.3.3 Well-Formed Functionalities

In this section, we define the set of functionalities for which our feasibility results apply. Clearly, we would like to be able to state a theorem saying that *any* ideal functionality can be UC realized. However, for technical reasons, such a claim cannot be made in our model. The first problem that arises is as follows. Since the ideal functionality is informed of the identities of the corrupted parties, it can do things that cannot be realized by any protocol. For example, consider the ideal functionality that lets all parties know which parties are corrupted. Then this functionality cannot be realized in the face of an adversary that corrupts a single random party but instructs that party to continue following the prescribed protocol.

In order to bypass this problem, we define a special class of functionalities that do not utilize their direct knowledge of the identities of the corrupted parties. For the lack of a better name, we call these functionalities

**well-formed.** A well-formed functionality consists of a main procedure (called the **shell**) and a subroutine (called the **core**.) The core is an arbitrary probabilistic polynomial-time algorithm, while the shell is a simple procedure described as follows. The shell forwards any incoming message to the core, with the exception that notifications of corruptions of parties are *not* forwarded. Outgoing messages generated by the core are copied by the shell to the outgoing communication tape. The above definition guarantees that the code of a well-formed ideal functionality “does not depend” on its direct knowledge regarding who is corrupted.

In subsequent sections, we show how to realize any well-formed functionality in the face of *static* adversaries. However, another technicality arises when considering *adaptive* adversaries. Consider for instance a two-party ideal functionality  $\mathcal{F}$  that works as follows: Upon activation, it chooses two large random primes  $p$  and  $q$  and sends  $n = pq$  to both parties. The value  $n$  is the only message output by the functionality; in particular, it never reveals the values  $p$  and  $q$ . The important property of this functionality that we wish to focus on is the fact that it has *private randomness* that is never revealed. Such a functionality can be UC realized in the static corruption model. However, consider what happens in a real execution if an adaptive adversary corrupts *both* parties after they output  $n$ . In this case, all prior randomness is revealed (recall that we assume no erasures). Therefore, if this randomness explicitly defines the primes  $p$  and  $q$  (as is the case in all known protocols for such a problem), these values will necessarily be revealed to the adversary. On the other hand, in the ideal process, even if both parties are corrupted,  $p$  and  $q$  are never revealed.<sup>11</sup> (We stress that the fact that  $p$  and  $q$  are revealed in the real model does not cause any real security concern. This is because when all the participating parties are corrupted, there are no security requirements on a protocol. In particular, there are no honest parties to “protect”). In light of the above discussion, we define **adaptively well-formed functionalities** that do not keep private randomness when all parties are corrupted. Formally, such functionalities have a shell and a core, as described above. However, in addition to forwarding messages to and from the core, the shell keeps track of the parties with whom the functionality interacts. If at some activation all these parties are corrupted, then the shell sends the random tape of the core to the adversary. Thus, when all the parties participating in some activation are corrupted, all the randomness is revealed (even in the ideal process). We show how any adaptively well-formed functionality can be UC realized in the face of adaptive adversaries.

In order to make sure that the multi-session extension of an adaptively well-formed functionality remains adaptively well-formed, we slightly modify the definition of the multi-session extension of an ideal functionality (see

<sup>11</sup> We do not claim that it is impossible to realize this specific functionality. Indeed, it may be possible to sample the domain  $\{n \mid n = pq\}$  (or a domain that is computationally indistinguishable from it) without knowing  $p$  or  $q$ . Nevertheless, the example clearly demonstrates the problem that arises.

Section 4.3.2) as follows. If the given ideal functionality  $\mathcal{F}$  is adaptively well-formed, then  $\hat{\mathcal{F}}$ , the multi-session extension of  $\mathcal{F}$ , is an adaptively well-formed functionality defined as follows. The core of  $\hat{\mathcal{F}}$  consists of the multi-session extension (in the usual sense) of the core of  $\mathcal{F}$ . The shell of  $\hat{\mathcal{F}}$  is as defined above except that it separately keeps track of the participating parties of each session. Then, if all the participating parties of some session are corrupted in some activation, the shell sends the random tape of the core for that session to the adversary. (Recall that each session of the multi-session functionality uses an independent random tape.) We note that the JUC theorem (Theorem 4.3.4) holds even with respect to the modified definition of multi-session extensions.

## 4.4 Two-Party Secure Computation for Semi-honest Adversaries

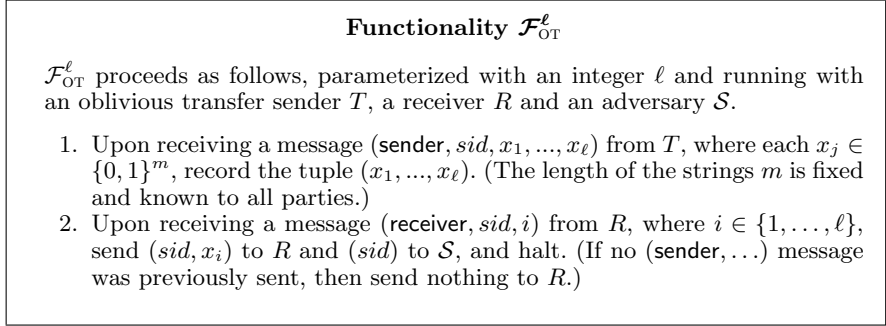
This section presents general constructions for UC realizing any two-party ideal functionality in the presence of semi-honest adversaries. The high-level construction is basically that of Goldreich et al. (GMW) [GMW87, G02a]. However, there are two differences. First, [GMW87] consider static adversaries whereas we consider adaptive adversaries. This actually only makes a difference in the oblivious transfer protocol; the rest of the protocol for circuit evaluation remains unchanged. We note that although the protocol constructions are very similar, our proof of security is significantly different. This is due to the fact that we deal with adaptive adversaries and in addition show universal composability (in contrast with static adversaries and the standard, stand-alone definitions of security). The second difference between the GMW construction and ours is that while GMW considered function evaluation, we consider more general **reactive** functionalities. In this case, parties may receive new inputs during the protocol execution and each new input may depend on the current adversarial view of the system. In particular, it may depend on previous outputs of this execution and on the activity in other executions. We note that despite the additional generality, this makes only a small difference to the construction.

We begin by presenting the oblivious-transfer ideal functionality  $\mathcal{F}_{\text{OT}}$ , and demonstrate how to UC realize this functionality in the presence of semi-honest adversaries (both static and adaptive). Following this we present our protocol for UC realizing any two-party functionality, in the  $\mathcal{F}_{\text{OT}}$ -hybrid model.

### 4.4.1 Universally Composable Oblivious Transfer

Oblivious transfer [R81, EGL85] is a two-party functionality, involving a sender with input  $x_1, \dots, x_\ell$ , and a receiver with input  $i \in \{1, \dots, \ell\}$ . The

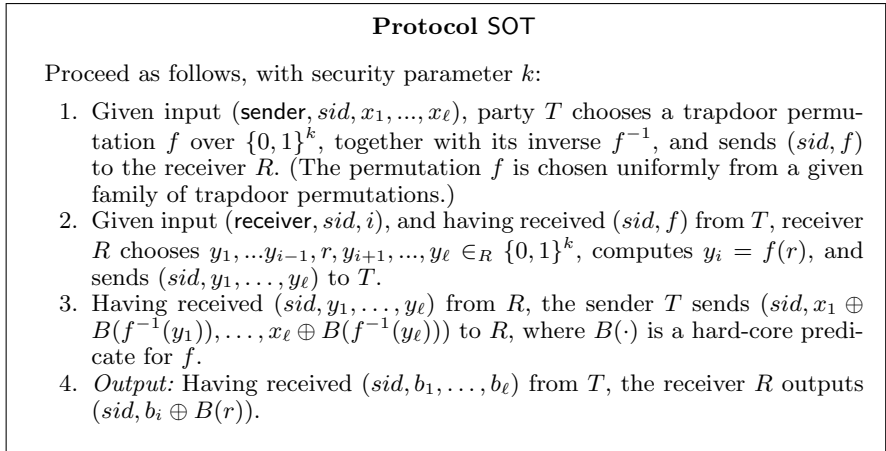
receiver should learn  $x_i$  (and nothing else) and the sender should learn nothing. An exact definition of the ideal oblivious transfer functionality, denoted  $\mathcal{F}_{\text{OT}}^\ell$ , appears in Figure 4.1. (Using standard terminology,  $\mathcal{F}_{\text{OT}}^\ell$  captures 1-out-of- $\ell$  OT.)



**Fig. 4.1.** The oblivious transfer functionality

We present protocols that UC realize  $\mathcal{F}_{\text{OT}}$  for both static and adaptive adversaries.

**Static UC oblivious transfer.** The oblivious transfer protocol of [GMW87, G02a], denoted SOT (for Static Oblivious Transfer) is presented in Figure 4.2. For simplicity we present the protocol for the case where each of the  $\ell$  input values is a single bit. (In the semi-honest case, oblivious transfer for strings can be constructed from this one via the composition theorem.)



**Fig. 4.2.** The static, semi-honest oblivious transfer protocol

**Claim 4.4.1** *Assuming that  $f$  is a trapdoor permutation, Protocol SOT UC realizes  $\mathcal{F}_{\text{OT}}^\ell$  in the presence of semi-honest, static adversaries.*

**Proof.** Let  $\mathcal{A}$  be a semi-honest, static adversary that interacts with parties running the above protocol. We construct an adversary  $\mathcal{S}$  for the ideal process for  $\mathcal{F}_{\text{OT}}^\ell$  such that no environment  $\mathcal{Z}$  can tell with non-negligible probability whether it is interacting with  $\mathcal{A}$  and the above protocol or with  $\mathcal{S}$  in the ideal process for  $\mathcal{F}_{\text{OT}}^\ell$ . Recall that  $\mathcal{S}$  interacts with the ideal functionality  $\mathcal{F}_{\text{OT}}^\ell$  and with the environment  $\mathcal{Z}$ . Simulator  $\mathcal{S}$  starts by invoking a copy of  $\mathcal{A}$  and running a simulated interaction of  $\mathcal{A}$  with  $\mathcal{Z}$  and parties running the protocol. (We refer to the interaction of  $\mathcal{S}$  in the ideal process as *external interaction*. The interaction of  $\mathcal{A}$  with the simulated  $\mathcal{A}$  is called *internal interaction*.)  $\mathcal{S}$  proceeds as follows:

**Simulating the communication with  $\mathcal{Z}$ :** Every input value that  $\mathcal{S}$  receives from  $\mathcal{Z}$  is written on  $\mathcal{A}$ 's input tape (as if coming from  $\mathcal{A}$ 's environment). Likewise, every output value written by  $\mathcal{A}$  on its output tape is copied to  $\mathcal{S}$ 's own output tape (to be read by  $\mathcal{S}$ 's environment  $\mathcal{Z}$ ).

**Simulating the case where the sender  $T$  only is corrupted:**  $\mathcal{S}$  simulates a real execution in which  $T$  is corrupted.  $\mathcal{S}$  begins by activating  $\mathcal{A}$  and receiving the message  $(sid, f)$  that  $\mathcal{A}$  (controlling  $T$ ) would send  $R$  in a real execution. Then,  $\mathcal{S}$  chooses  $y_1, \dots, y_\ell \in_R \{0, 1\}^k$  and simulates  $R$  sending  $T$  the message  $(sid, y_1, \dots, y_\ell)$  in the internal interaction. Finally, when  $\mathcal{A}$  sends the message  $(sid, b_1, \dots, b_\ell)$  from  $T$  to  $R$  in the internal interaction,  $\mathcal{S}$  externally sends  $T$ 's input  $x_1, \dots, x_\ell$  to  $\mathcal{F}_{\text{OT}}^\ell$  and delivers the output from  $\mathcal{F}_{\text{OT}}^\ell$  to  $R$ . (Recall that in the semi-honest model as defined here,  $\mathcal{A}$  is able to modify the input tape of  $T$ . Therefore, the value  $x_1, \dots, x_\ell$  sent by  $\mathcal{S}$  to  $\mathcal{F}_{\text{OT}}^\ell$  is the (possibly) modified value. Formally this causes no problem because actually it is the environment who writes the modified value, after “consultation” with  $\mathcal{A}$ . Since all communication is forwarded unmodified between  $\mathcal{A}$  and  $\mathcal{Z}$ , the value that  $\mathcal{Z}$  writes on  $T$ 's input tape is the already-modified value. We ignore this formality in the subsequent proofs in this section.)

**Simulating the case where the receiver  $R$  only is corrupted:** Simulator  $\mathcal{S}$  begins by activating  $\mathcal{A}$  and internally sending it the message that  $\mathcal{A}$  (controlling  $R$ ) expects to receive from  $T$  in a real execution. That is,  $\mathcal{S}$  chooses a random trapdoor permutation  $f$  and its inverse  $f^{-1}$ , and sends  $(sid, f)$  to  $\mathcal{A}$ . Next, it internally receives a message of the form  $(sid, y_1, \dots, y_\ell)$  from  $\mathcal{A}$ . Simulator  $\mathcal{S}$  then externally sends  $R$ 's input  $i$  to  $\mathcal{F}_{\text{OT}}^\ell$  and receives back the output  $x_i$ .  $\mathcal{S}$  concludes the simulation by choosing  $b_1, \dots, b_{i-1}, b_{i+1}, \dots, b_\ell$  uniformly, setting  $b_i = x_i \oplus B(f^{-1}(y_i))$ , and internally sending  $\mathcal{A}$  the message  $(sid, b_1, \dots, b_\ell)$ . (Recall that  $x_i$  is the output as obtained by  $\mathcal{S}$  from the ideal functionality  $\mathcal{F}_{\text{OT}}^\ell$ .)

**Simulating the case that neither party is corrupted:** In this case,  $\mathcal{S}$  receives a message  $(sid)$  signalling it that  $T$  and  $R$  concluded an ideal

execution with  $\mathcal{F}_{\text{OT}}$ .  $\mathcal{S}$  then generates a simulated transcript of messages between the real model parties. That is,  $\mathcal{S}$  generates  $T$ 's first message  $(sid, f)$  as the real  $T$  would, sets  $R$ 's reply to be  $(sid, y_1, \dots, y_\ell)$  where  $y_j \in_R \{0, 1\}^k$  for each  $j$ , and finally sets  $T$ 's second message to  $(sid, b_1, \dots, b_\ell)$  where  $b_j \in_R \{0, 1\}$  for every  $j$ .

**Simulating the case that both parties are corrupted:** In this case,  $\mathcal{S}$  knows both parties' inputs and can therefore simulate a protocol execution by generating the actual messages that the parties would send in a real execution.

We demonstrate that  $\{\text{IDEAL}_{\mathcal{F}_{\text{OT}}, \mathcal{S}, \mathcal{Z}}(k, z)\} \stackrel{c}{=} \{\text{REAL}_{\text{SOT}, \mathcal{A}, \mathcal{Z}}(k, z)\}$ . This is done by showing that the joint view of  $\mathcal{Z}$  and  $\mathcal{A}$  in the execution of  $\text{SOT}$  is indistinguishable from the joint view of  $\mathcal{Z}$  and the simulated  $\mathcal{A}$  within  $\mathcal{S}$  in the ideal process. First, notice that the simulation for the case where  $T$  is corrupted is perfect. This is because in both the ideal simulation and a real execution, the message received by  $T$  consists of  $\ell$  uniformly distributed  $k$ -bit strings, and the output of  $R$  is the same in both executions. (Notice that since  $f$  is a permutation, choosing  $r$  uniformly and computing  $y_i = f(r)$ , as occurs in a real execution, results in a uniformly distributed  $y_i$ . Furthermore, the output of  $R$  is  $b_i \oplus B(f^{-1}(y_i))$  where  $b_i$  is the  $i^{\text{th}}$  value sent by  $T$ .) Second, we claim that the simulation for the case where  $R$  is corrupted is indistinguishable from in a real execution. The only difference between the two is in the message  $b_1, \dots, b_\ell$  received by  $R$ . The bit  $b_i$  is identically distributed in both cases (in particular, in both the simulation and a real execution it equals  $x_i \oplus B(f^{-1}(y_i))$ ). However, in the ideal simulation, all the bits  $b_j$  for  $j \neq i$  are uniformly chosen and are not distributed according to  $x_j \oplus B(f^{-1}(y_j))$ . Nevertheless, due to the hard-core properties of  $B$ , the bit  $B(f^{-1}(y_j))$  for a random  $y_j$  is indistinguishable from a random-bit  $b_j \in_R \{0, 1\}$ . The same is also true for  $x_j \oplus B(f^{-1}(y_j))$  when  $x_j$  is fixed before  $y_j$  is chosen. (More precisely, given an environment that distinguishes with non-negligible probability between the real-life and the ideal interactions, we can construct an adversary that contradicts the hard-core property of  $B$ .) Thus the views are indistinguishable. By the same argument, we also have that the simulation for the case that neither party is corrupted results in a view that is indistinguishable from a real execution. This completes the proof.  $\blacksquare$

Our proof of security of the above protocol fails in the case of adaptive adversaries. Intuitively the reason is that when a party gets corrupted,  $\mathcal{S}$  cannot present the simulated  $\mathcal{A}$  with a valid internal state of the corrupted party. (This internal state should be consistent with the past messages sent by the party and with the local input and output of that party.) In particular, the messages  $(sid, f)$ ,  $(sid, y_1, \dots, y_\ell)$  and  $(sid, b_1, \dots, b_\ell)$  fully define the input bits  $x_1, \dots, x_\ell$ . However, in the case that  $T$  is not initially corrupted,  $\mathcal{S}$  does not know  $x_1, \dots, x_\ell$  and therefore with high probability, the messages define a different set of input bits. Thus, if  $\mathcal{A}$  corrupts  $T$  after the execution has concluded,  $\mathcal{S}$  cannot provide  $\mathcal{A}$  with an internal state of  $T$  that is consis-

tent both with  $x_1, \dots, x_\ell$  and the simulated transcript that it had previously generated.

**Adaptive UC oblivious transfer.** Due to the above-described problem, we use a different protocol for UC realizing  $\mathcal{F}_{\text{OT}}^\ell$  for the case of adaptive, semi-honest adversaries. A main ingredient in this protocol are **non-committing encryptions** as defined in [CFGN96] and constructed in [CFGN96, B97, DN00]. In addition to standard semantic security, such encryption schemes have the property that ciphertexts that can be opened to both 0 and 1 can be generated. That is, a non-committing (bit) encryption scheme consists of a tuple  $(G, E, D, S)$ , where  $G$  is a key generation algorithm,  $E$  and  $D$  are encryption and decryption algorithms, and  $S$  is a simulation algorithm (for generating non-committing ciphertexts). The triple  $(G, E, D)$  satisfies the usual properties of semantically secure encryption. That is,  $G(1^k) = (e, d)$  where  $e$  and  $d$  are the respective encryption and decryption keys, and  $D_d(E_e(m)) = m$  except with negligible probability. Furthermore,  $\{E_e(1)\}$  is indistinguishable from  $\{E_e(0)\}$ . In addition, the simulator algorithm  $S$  is able to generate “dummy ciphertexts” that can be later “opened” as encryptions of either 0 or 1. More specifically, it is required that  $S(1^k) = (e, c, r_0, r_1, d_0, d_1)$  with the following properties:

- The tuple  $(e, c, r_0, d_0)$  looks like a normal encryption and decryption process for the bit 0. That is,  $(e, c, r_0, d_0)$  is indistinguishable from a tuple  $(e', c', r', d')$  where  $(e', d')$  is a randomly chosen pair of encryption and decryption keys,  $r'$  is randomly chosen, and  $c' = E_{e'}(0; r')$ . (In particular, it should hold that  $D_{d_0}(c) = 0$ .)
- The tuple  $(e, c, r_1, d_1)$  looks like a normal encryption and decryption process for the bit 1. That is,  $(e, c, r_1, d_1)$  is indistinguishable from a tuple  $(e', c', r', d')$  where  $(e', d')$  is a randomly chosen pair of encryption and decryption keys,  $r'$  is randomly chosen, and  $c' = E_{e'}(1; r')$ . (In particular, it should hold that  $D_{d_1}(c) = 1$ .)

Thus, given a pair  $(e, c)$ , it is possible to explain  $c$  both as an encryption of 0 (by providing  $d_0$  and  $r_0$ ) and as an encryption of 1 (by providing  $d_1$  and  $r_1$ ). Here, we actually use *augmented* non-committing encryption protocols that have the following two additional properties:

1. **Oblivious key generation:** It should be possible to choose a public encryption key  $e$  “without knowing” the decryption key  $d$ . That is, there should exist a different key generation algorithm  $\hat{G}$  such that  $\hat{G}(1^k) = \hat{e}$  where  $\hat{e}$  is indistinguishable from the encryption key  $e$  chosen by  $G$ , and in addition  $\{E_{\hat{e}}(0)\}$  remains indistinguishable from  $\{E_e(1)\}$  even when the entire random input of  $\hat{G}$  is known.
2. **Invertible samplability:** this property states that the key generation and oblivious key generation algorithms  $G$  and  $\hat{G}$  should be invertible. That is, we require the existence of an inverting algorithm who receives any  $e$  output by the simulator algorithm  $S$  and outputs  $r$  such that  $\hat{G}(r) = e$ .



(This algorithm may receive the coins used by  $S$  in computing  $e$  in order to find  $r$ .) We also require an algorithm that receives any pair  $(e, d_i)$  for  $i \in \{0, 1\}$  from the output of  $S$ , and outputs  $r$  such that  $G(r) = (e, d_i)$ . (As before, this algorithm may receive the coins used by  $S$ .) The idea here is that in order to “explain” the simulator-generated keys as being generated in a legal way, it must be possible to find legal random coin tosses for them.<sup>12</sup>

Augmented two-party non-committing encryption protocols exist under either one of the RSA or the DDH assumptions. The requirements are also fulfilled in any case that public keys are uniformly distributed over some public domain and secret keys are defined by the random coins input to  $G$ . See more details in [CFGN96, DN00].

The protocol for UC realizing  $\mathcal{F}_{\text{OT}}^\ell$ , denoted AOT (for Adaptive Oblivious Transfer) is presented in Figure 4.3. As in the static case, the protocol is defined for the case where each of the  $\ell$  input values is a single bit.

#### Protocol AOT

Proceed as follows, on security parameter  $k$  and using an augmented non-committing encryption scheme  $(G, \hat{G}, E, D, S)$ .

1. Given input (receiver,  $sid, i$ ), receiver  $R$  runs  $G(1^k)$  to obtain  $(e, d)$ , and runs  $\hat{G}(1^k)$   $\ell - 1$  times to obtain  $\hat{e}_1, \dots, \hat{e}_{i-1}, \hat{e}_{i+1}, \dots, \hat{e}_\ell$ . Then,  $R$  sends  $(sid, \hat{e}_1, \dots, \hat{e}_{i-1}, e, \hat{e}_{i+1}, \dots, \hat{e}_\ell)$  to  $T$ .
2. Given input (sender,  $sid, x_1, \dots, x_\ell$ ), and having received  $(sid, e_1, \dots, e_\ell)$  from  $R$ , sender  $T$  computes  $c_j = E_{e_j}(x_j)$  for every  $1 \leq j \leq \ell$ , and sends  $(sid, c_1, \dots, c_\ell)$  to  $R$ .
3. Having received  $(sid, c_1, \dots, c_\ell)$  from  $T$ , receiver  $R$  computes  $x_i = D_d(c_i)$  and outputs  $(sid, x_i)$ .

**Fig. 4.3.** The adaptive, semi-honest oblivious transfer protocol

**Claim 4.4.2** *Assume that  $(G, \hat{G}, E, D, S)$  is an augmented non-committing encryption scheme. Then, Protocol AOT UC realizes  $\mathcal{F}_{\text{OT}}^\ell$  in the presence of semi-honest, adaptive adversaries.*

**Proof.** The main difference between this proof and the proof of Claim 4.4.1 is due to the fact that the real-model adversary  $\mathcal{A}$  can corrupt parties during (or after) the simulation. When  $\mathcal{S}$  receives such a “corrupt” command, it corrupts the ideal-model party and receives its input (and possibly its output). Given this information,  $\mathcal{S}$  must produce random coins for this party such that the

<sup>12</sup> In its most general form, one can define an invertible sampling algorithm for  $G$  that receives *any* pair  $(e, d)$  in the range of  $G$  and  $S$  and outputs  $r$  such that  $G(r) = (e, d)$ . However, we actually only need the inverting algorithm to work for keys output by the simulator  $S$ .

simulated transcript generated so far is consistent with the revealed input (and output).

Let  $\mathcal{A}$  be a semi-honest, *adaptive* adversary that interacts with parties running the above protocol. We construct an adversary  $\mathcal{S}$  in the ideal process for  $\mathcal{F}_{\text{OT}}^\ell$  such that no environment  $\mathcal{Z}$  can tell with non-negligible advantage whether it is interacting with  $\mathcal{A}$  and the above protocol or with  $\mathcal{S}$  in the ideal process for  $\mathcal{F}_{\text{OT}}^\ell$ . In describing the simulation here, it is helpful to distinguish between the ideal (dummy) parties, denoted  $\tilde{T}$  and  $\tilde{R}$ , and the ITMs representing the real model parties in the internal interaction, denoted  $T$  and  $R$ .  $\mathcal{S}$  works as follows:

**Simulating the communication with  $\mathcal{Z}$ :** Every input value that  $\mathcal{S}$  receives from  $\mathcal{Z}$  is written on  $\mathcal{A}$ 's input tape (as if coming from  $\mathcal{A}$ 's environment). Likewise, every output value written by  $\mathcal{A}$  on its output tape is copied to  $\mathcal{S}$ 's own output tape (to be read by  $\mathcal{S}$ 's environment  $\mathcal{Z}$ ).

**Simulating the receiver:** We separately describe the simulation for a corrupted and uncorrupted receiver.

1. *Simulation when the receiver  $\tilde{R}$  is not corrupted:* In this case,  $\mathcal{S}$  needs to simulate the real receiver message. This is done as follows:  $\mathcal{S}$  runs the encryption simulation algorithm  $S(1^k)$  independently  $\ell$  times. For each  $j$ ,  $\mathcal{S}$  obtains a tuple  $(e_j, c_j, r_0^j, r_1^j, d_0^j, d_1^j)$ ; see the explanation above for the meaning of each element in this tuple. Then,  $\mathcal{S}$  generates  $R$ 's message to be  $(\text{sid}, e_1, \dots, e_\ell)$  and simulates  $R$  writing it on its outgoing communication tape for  $T$ .
2. *Simulation when the receiver  $\tilde{R}$  is corrupted:* In this case,  $\mathcal{S}$  holds the input  $(\text{receiver}, \text{sid}, i)$  of the ideal receiver  $\tilde{R}$  and constructs a virtual real-model receiver  $R$  as follows. The contents of the input tape of  $R$  is set to  $(\text{receiver}, \text{sid}, i)$ . In order to set the contents of  $R$ 's random tape,  $\mathcal{S}$  first runs the encryption simulation algorithm  $S(1^k)$  independently  $\ell$  times, obtaining tuples  $(e_j, c_j, r_0^j, r_1^j, d_0^j, d_1^j)$ . Next, for every  $j \neq i$ ,  $\mathcal{S}$  uses the invertible sampling algorithm in order to find  $r_j$  so that  $\hat{G}(r_j) = e_j$ , where  $\hat{G}$  is the oblivious key generator. Furthermore,  $\mathcal{S}$  uses the invertible sampling algorithm in order to find  $r_i$  so that  $G(r_i) = (e_i, d_{x_i}^i)$ . (Recall that  $x_i$  is the ideal receiver  $\tilde{R}$ 's output.) Then,  $\mathcal{S}$  sets the contents of  $R$ 's random tape to equal  $r_1, \dots, r_\ell$ .

$\mathcal{S}$  passes the simulated internal state of  $R$  (including the contents of its input and random tapes) to  $\mathcal{A}$  and waits for  $\mathcal{A}$  to activate  $R$  in the simulation. When this occurs,  $\mathcal{S}$  internally obtains the message  $(\text{sid}, e_1, \dots, e_\ell)$  that  $R$  writes on its outgoing message tape. Then, when  $\mathcal{A}$  delivers this message to  $T$  in the internal simulation,  $\mathcal{S}$  externally delivers the  $(\text{receiver}, \text{sid}, i)$  message from  $\tilde{R}$  to  $\mathcal{F}_{\text{OT}}^\ell$ . (We note that for every  $j \neq i$ , it holds that  $e_j = \hat{G}(r_j)$  is the same  $e_j$  as generated by  $S(1^k)$ . On the other hand,  $e_i$  is the encryption key output by  $G(r_i)$ , where  $G$  is the standard encryption key generator.)

**Simulating the sender:** Once again, we separately consider the case that the sender is corrupted or not corrupted.

1. *Simulation when the sender  $\tilde{T}$  is not corrupted:* After  $\mathcal{A}$  delivers the message from  $R$  to  $T$  in the internal simulation,  $\mathcal{S}$  simulates the real-model  $T$  writing  $(sid, c_1, \dots, c_\ell)$  on its outgoing communication tape, where the  $c_i$ 's were generated from  $S(1^k)$  when simulating the receiver message above. When  $\mathcal{A}$  delivers this message from  $T$  to  $\tilde{R}$  in the internal simulation, then  $\mathcal{S}$  delivers the output from  $\mathcal{F}_{\text{OT}}^\ell$  to  $\tilde{R}$  in the ideal process. This is the same whether or not  $\tilde{R}$  is corrupted.
2. *Simulation when the sender  $\tilde{T}$  is corrupted:*  $\mathcal{S}$  holds the ideal  $\tilde{T}$ 's input  $(\text{sender}, sid, x_1, \dots, x_\ell)$  and constructs a virtual real-model sender  $T$  by writing  $(\text{sender}, sid, x_1, \dots, x_\ell)$  on its input tape and a *uniformly* distributed string on its random tape. Then, as above,  $\mathcal{S}$  passes  $\mathcal{A}$  the simulated internal state of  $T$  (consisting of the contents of its input and random tapes). When  $\mathcal{A}$  activates  $T$  and delivers the outgoing message from  $T$  to  $R$ , then simulator  $\mathcal{S}$  externally delivers the message  $(\text{sender}, sid, x_1, \dots, x_\ell)$  from  $\tilde{T}$  to  $\mathcal{F}_{\text{OT}}^\ell$ .

**Dealing with “corrupt” commands:** We assume that any corruption of a party occurs after the adversary has delivered the party’s protocol message in the simulation. (Otherwise, the corruption essentially occurs before the protocol begins and the instructions above suffice.) Now, if  $\mathcal{S}$  receives a command from  $\mathcal{A}$  to corrupt the real-model  $R$ , then it corrupts the ideal model  $\tilde{R}$  and obtains its input  $i$  and its output  $x_i$ . Given  $i$  and  $x_i$ , simulator  $\mathcal{S}$  passes  $\mathcal{A}$  the decryption-key  $d_{x_i}^i$  (and thus the ciphertext  $c_i$  given to  $R$  in the simulated sender-message is “decrypted” to  $x_i$ ). Furthermore, for every  $j \neq i$ ,  $\mathcal{S}$  runs the invertible sampling algorithm on  $e_j$  in order to find  $r_j$  such that  $\hat{G}(r_j) = e_j$ . Finally,  $\mathcal{S}$  runs the invertible sampling algorithm on  $e_i$  in order to find  $r_i$  such that  $G(r_i) = (e_i, d_{x_i}^i)$ . Notice that these two invertible sampling algorithms are different.  $\mathcal{S}$  supplies  $\mathcal{A}$  with the random tape  $r_1, \dots, r_\ell$  for  $R$ .

If  $\mathcal{S}$  receives a command from  $\mathcal{A}$  to corrupt real-model  $T$ , then it first corrupts the ideal-model  $\tilde{T}$  and obtains  $x_1, \dots, x_\ell$ . Next, it prepares appropriate randomness to make it appear that for every  $j$ , it holds that  $c_j = E_{e_j}(x_j)$  (where the  $(c_j, e_j)$  pairs are taken from the simulated receiver and sender messages). Since the encryption keys are non-committing and were generated by running  $S(1^k)$ , we have that for every  $1 \leq j \leq \ell$  simulator  $\mathcal{S}$  has a string  $r_{x_j}$  such that  $c_j = E_{e_j}(x_j; r_{x_j})$ . Therefore,  $\mathcal{S}$  writes  $r_{x_1}, \dots, r_{x_\ell}$  as  $T$ 's random tape.

As argued in the proof of Claim 4.4.1, it suffices to show that  $\mathcal{A}$ 's view in the simulation is indistinguishable from its view in a real execution. (Note that in the adaptive case there is a positive correctness error. That is, there is non-zero probability that the outputs of the uncorrupted parties in the real-life interaction will differ from their outputs in the ideal process. This error probability is due to the fact that encryption schemes can “fail” with

negligible probability. Since the probability of such an event is negligible, we ignore it here.) The indistinguishability of the views is demonstrated using the properties of the augmented non-committing encryption scheme. In particular, the non-committing encryption keys, ciphertexts and decommitment strings are all indistinguishable from those appearing in a real execution. Furthermore, by the oblivious key-generation algorithm,  $\mathcal{S}$  supplies only a single decryption key (for the  $i^{\text{th}}$  encryption key) and this is what a real receiver would also have. (More precisely, given an environment that distinguishes between the real-life and ideal interactions we construct an adversary that breaks either the security of the non-committing encryption or the oblivious key generation property. We omit further details.) ■

#### 4.4.2 The General Construction

We are now ready to show how to UC realize any (adaptively) well-formed two-party functionality in the  $\mathcal{F}_{\text{OT}}$ -hybrid model, in the semi-honest case. (Adaptively well-formed functionalities are defined in Section 4.3.3. Two-party functionalities are functionalities that interact with the adversary, plus at most two parties.) The construction is essentially that of [GMW87, G02a], although as we have mentioned, our protocol is actually more general in that it also deals with reactive functionalities. We begin by formally restating Proposition 4.2.1:

**Proposition 4.4.3** (Proposition 4.2.1 – formally restated): *Assume that trapdoor permutations exist. Then, for any two-party well-formed ideal functionality  $\mathcal{F}$ , there exists a non-trivial protocol that UC realizes  $\mathcal{F}$  in the presence of semi-honest, static adversaries. Furthermore, if two-party augmented non-committing encryption protocols exist, then for any two-party adaptively well-formed ideal functionality  $\mathcal{F}$ , there exists a non-trivial protocol that UC realizes  $\mathcal{F}$  in the presence of semi-honest, adaptive adversaries.*

Recall that a protocol is non-trivial if the ideal-process adversary delivers all messages from the functionality to the parties whenever the real-life adversary delivers all messages and doesn't corrupt any parties. This restriction is included to rule out meaningless protocols such as the protocol that never generates output. (See Section 4.3.1 for more discussion.)

**The construction.** Let  $\mathcal{F}$  be an ideal two-party functionality and let  $P_1$  and  $P_2$  be the participating parties. The first step in constructing a protocol that UC realizes  $\mathcal{F}$  is to represent (the core of)  $\mathcal{F}$  via a family  $C_{\mathcal{F}}$  of boolean circuits. That is, the  $m^{\text{th}}$  circuit in the family describes an activation of  $\mathcal{F}$  when the security parameter is set to  $m$ . Following [GMW87, G02a], we use arithmetic circuits over  $\text{GF}(2)$  where the operations are addition and multiplication modulo 2.

For simplicity we assume that the input and output of each party in each activation has at most  $m$  bits, the number of random bits used by  $\mathcal{F}$  in all

activations is at most  $m$ , and at the end of each activation the local state of  $\mathcal{F}$  can be described in at most  $m$  bits. Consequently the circuit has  $3m$  input lines and  $3m$  output lines, with the following interpretation. In each activation, only one party has input. Therefore,  $m$  of the input lines are allocated for this input. The other  $2m$  input lines describe  $\mathcal{F}$ 's  $m$  random coins and the length- $m$  internal state of  $\mathcal{F}$  at the onset of an activation. The  $3m$  output lines are allocated as follows:  $m$  output lines for the output of each of the two parties and  $m$  output lines to describe the state of  $\mathcal{F}$  following an activation. The circuit is constructed so that each input from the adversary is set to 0, and outputs to the adversary are ignored.<sup>13</sup> We note that if the input or output of a party in some activation is less than  $m$  bits then this is encoded in a standard way. Also, each party initially sets its shares of the state of  $\mathcal{F}$  to 0 (this denotes the empty state).

**Protocol  $\Pi_{\mathcal{F}}$  (for UC realizing  $\mathcal{F}$ ):** We state the protocol for an activation in which  $P_1$  sends a message to  $\mathcal{F}$ ; the case where  $P_2$  sends the message is easily derived (essentially by exchanging the roles of  $P_1$  and  $P_2$  throughout the protocol). It is assumed that both parties hold the same session identifier  $sid$  as auxiliary input. When activated with input  $(sid, v)$  within  $P_1$ , the protocol first sends a message to the partner  $P_2$ , asking it to participate in a joint evaluation of the  $m^{\text{th}}$  circuit in  $C_{\mathcal{F}}$ . Next,  $P_1$  and  $P_2$  engage in a gate-by-gate evaluation of  $C_{\mathcal{F}}$ , on inputs that represent the incoming message  $v$  from  $P_1$ , the current internal state of  $\mathcal{F}$ , and a random string. This is done as follows.

### 1. Input Preparation Stage:

- *Input value:* Recall that  $v$  is  $P_1$ 's input for this activation.  $P_1$  first pads  $v$  to be of length exactly  $m$  (using some standard encoding). Next  $P_1$  “shares” its input. That is,  $P_1$  chooses a random string  $v_1 \in_R \{0, 1\}^m$  and defines  $v_2 = v_1 \oplus v$ . Then,  $P_1$  sends  $(sid, v_2)$  to  $P_2$  and stores  $v_1$ .
- *Internal state:* At the onset of each activation, the parties hold shares of the current internal state of  $\mathcal{F}$ . That is, let  $c$  denote the current internal state of  $\mathcal{F}$ , where  $|c| = m$ . Then,  $P_1$  and  $P_2$  hold  $c_1, c_2 \in \{0, 1\}^m$ , respectively, such that  $c_1$  and  $c_2$  are random under the restriction that  $c = c_1 \oplus c_2$ . (In the first activation of  $\mathcal{F}$ , the internal state is empty and so the parties' shares both equal  $0^m$ .)
- *Random coins:* Upon the first activation of  $\mathcal{F}$  only, parties  $P_1$  and  $P_2$  choose random strings  $r_1 \in_R \{0, 1\}^m$  and  $r_2 \in_R \{0, 1\}^m$ , respectively. These constitute shares of the random coins  $r = r_1 \oplus r_2$  to be used by  $C_{\mathcal{F}}$ . We stress that  $r_1$  and  $r_2$  are chosen upon the first activation only. The same  $r_1$  and  $r_2$  are then used for each subsequent activation

<sup>13</sup> Thus, we effectively prevent the ideal-model adversary from utilizing its capability of sending and receiving messages. This simplifies the construction, and only strengthens the result.

of  $\mathcal{F}$  ( $r_1$  and  $r_2$  are kept the same because the random tape of  $\mathcal{F}$  does not change from activation to activation).

At this point,  $P_1$  and  $P_2$  hold (random) shares of the input message to  $\mathcal{F}$ , the internal state of  $\mathcal{F}$  and the random tape of  $\mathcal{F}$ . That is, they hold shares of every input line into  $C_{\mathcal{F}}$ . Note that the only message sent in the above stage is the input share  $v_2$  sent from  $P_1$  to  $P_2$ .

2. **Circuit Evaluation:**  $P_1$  and  $P_2$  proceed to evaluate the circuit  $C_{\mathcal{F}}$  in a gate-by-gate manner. Let  $\alpha$  and  $\beta$  denote the values of the two input lines to a given gate. Then  $P_1$  holds bits  $\alpha_1, \beta_1$  and  $P_2$  holds bits  $\alpha_2, \beta_2$  such that  $\alpha = \alpha_1 + \alpha_2$  and  $\beta = \beta_1 + \beta_2$ . The gates are computed as follows:

- *Addition gates:* If the gate is an addition gate, then  $P_1$  locally sets its share of the output line of the gate to be  $\gamma_1 = \alpha_1 + \beta_1$ . Similarly,  $P_2$  locally sets its share of the output line of the gate to be  $\gamma_2 = \alpha_2 + \beta_2$ . (Thus  $\gamma_1 + \gamma_2 = \alpha + \beta$ .)
- *Multiplication gates:* If the gate is a multiplication gate, then the parties use  $\mathcal{F}_{\text{OT}}^4$  in order to compute their shares of the output line of the gate. That is, the parties wish to compute random shares  $\gamma_1$  and  $\gamma_2$  such that  $\gamma_1 + \gamma_2 = \alpha \cdot \beta = (\alpha_1 + \alpha_2)(\beta_1 + \beta_2)$ . For this purpose,  $P_1$  chooses a random bit  $\gamma_1 \in_R \{0, 1\}$ , sets its share of the output line of the gate to  $\gamma_1$ , and defines the following table:

Value of $(\alpha_2, \beta_2)$	Receiver input $i$	Receiver output $\gamma_2$
(0,0)	1	$o_1 = \gamma_1 + (\alpha_1 + 0) \cdot (\beta_1 + 0)$
(0,1)	2	$o_2 = \gamma_1 + (\alpha_1 + 0) \cdot (\beta_1 + 1)$
(1,0)	3	$o_3 = \gamma_1 + (\alpha_1 + 1) \cdot (\beta_1 + 0)$
(1,1)	4	$o_4 = \gamma_1 + (\alpha_1 + 1) \cdot (\beta_1 + 1)$

Having prepared this table,  $P_1$  sends the oblivious transfer functionality  $\mathcal{F}_{\text{OT}}^4$  the message (**sender**,  $\text{sid} \circ j, o_1, o_2, o_3, o_4$ ), where this is the  $j^{\text{th}}$  multiplication gate in the circuit and  $\circ$  denotes concatenation (the index  $j$  is included in order to ensure that the inputs of the parties match to the same gate).  $P_2$  sets its input value  $i$  for  $\mathcal{F}_{\text{OT}}^4$  according to the above table (e.g., for  $\alpha_2 = 1$  and  $\beta_2 = 0$ ,  $P_2$  sets  $i = 3$ ). Then,  $P_2$  sends (**receiver**,  $\text{sid} \circ j, i$ ) to  $\mathcal{F}_{\text{OT}}^4$  and waits to receive  $(\text{sid} \circ j, \gamma_2)$  from  $\mathcal{F}_{\text{OT}}^4$ . Upon receiving this output,  $P_2$  sets  $\gamma_2$  to be its share of the output line of the gate. Thus, we have that  $\gamma_1 + \gamma_2 = (\alpha_1 + \beta_1)(\alpha_2 + \beta_2)$  and the parties hold random shares of the output line of the gate.

3. **Output Stage:** Following the above stage, the parties  $P_1$  and  $P_2$  hold shares of all the output lines of the circuit  $C_{\mathcal{F}}$ . Each output line of  $C_{\mathcal{F}}$  is either an output addressed to one of the parties  $P_1$  and  $P_2$ , or belongs to the internal state of  $C_{\mathcal{F}}$  after the activation. The activation concludes as follows:

- *$P_1$ 's output:*  $P_2$  sends  $P_1$  all of its shares in  $P_1$ 's output lines.  $P_1$  reconstructs every bit of its output value by adding the appropriate shares, and writes the result on its output tape. (If the actual output generated by  $\mathcal{F}$  has less than the full  $m$  bits then this will be encoded in the output in a standard way.)
- *$P_2$ 's output:* Likewise,  $P_1$  sends  $P_2$  all of its shares in  $P_2$ 's output lines;  $P_2$  reconstructs the value and writes it on its output tape.
- *$\mathcal{S}$ 's output:* Recall that the outputs of  $\mathcal{F}$  to  $\mathcal{S}$  are ignored by  $\mathcal{C}_{\mathcal{F}}$ . Indeed, the protocol does not provide the real-life adversary with any information on these values. (This only strengthens the security provided by the protocol.)
- *Internal state:* Finally,  $P_1$  and  $P_2$  both locally store the shares that they hold for the internal state lines of  $\mathcal{C}_{\mathcal{F}}$ . (These shares are to be used in the next activation.)

Recall that there is no guarantee on the order of message delivery, so messages may be delivered “out of order”. However, to maintain correctness, the protocol must not start some evaluation of  $\mathcal{C}_{\mathcal{F}}$  before the previous evaluation of  $\mathcal{C}_{\mathcal{F}}$  has completed. Furthermore, evaluating some gate can take place only after the shares of the input lines of this gate are known. Thus, in order to guarantee that messages are processed in the correct order, a tagging method is used. Essentially, the aim of the method is to assign a *unique* tag to every message sent during all activations of  $\mathcal{F}$ . Thus, the adversary can gain nothing by sending messages in different orders. This is achieved in the following way. Recall that both parties hold the same session-identifier  $sid$ . Then, in activation  $i$ , the parties use the session-identifier  $sid \circ i$ . They also attach a tag identifying the stage which the message sent belongs to. Thus, for example, the message  $v_2$  sent by  $P_1$  in the input stage of the  $\ell^{\text{th}}$  activation is tagged with  $\langle sid \circ \ell \circ \text{input} \rangle$ . Likewise, the  $j^{\text{th}}$  call to  $\mathcal{F}_{\text{OT}}$  in the  $i^{\text{th}}$  activation is referenced with the session identifier  $sid \circ \ell \circ j$  (and not just  $sid \circ j$  as described above). Now, given the above tagging method, the ordering guarantees can be dealt with in standard ways by keeping messages that arrive too early in appropriate buffers until they become relevant (where the time that a message becomes relevant is self-evident from the labelling). By the above, it makes no difference whether or not the adversary delivers the messages according to the prescribed order. From here on we therefore assume that all messages *are* delivered in order. We also drop explicit reference to the additional tagging described here. This completes the description of  $\Pi_{\mathcal{F}}$ .

We now prove that the above construction UC realizes any adaptively well-formed functionality. (We stress that for the case of static adversaries,  $\Pi_{\mathcal{F}}$  UC realizes any well-formed functionality, and not just those that are *adaptively* well-formed. Nevertheless, here we prove the claim only for adaptively well-

formed functionalities and adaptive adversaries. The static case with security for any well-formed functionality is easily derived.)

Notice that each activation of  $\Pi_{\mathcal{F}}$  is due to an input sent by one of the participating parties. This implicitly assumes that the only messages that the functionality receives are from the parties themselves. This is indeed the case for well-formed functionalities (or, more accurately, the shells of such functionalities). However, recall that in general, functionalities also receive notification of the parties that are corrupted. The protocol does not (and cannot) deal with such messages and therefore does not UC realize functionalities that are not well-formed.

**Claim 4.4.4** *Let  $\mathcal{F}$  be a two-party adaptively well-formed functionality. Then, protocol  $\Pi_{\mathcal{F}}$  UC realizes  $\mathcal{F}$  in the  $\mathcal{F}_{\text{OT}}$ -hybrid model, in the presence of semi-honest, adaptive adversaries.*

Note that the claim holds unconditionally. In fact, it holds even if the environment and the adversary are computationally unbounded. (Of course, computational assumptions are required for UC realizing the oblivious transfer functionality.) The proof below deals with the security of reactive functionalities, in the presence of adaptive adversaries. This proof is significantly more involved than an analogous claim regarding non-reactive functionalities and static adversaries. For a warm-up, we refer the reader unfamiliar with this more simple case to [G02a, Sec. 2.2.4].

**Proof.** First note that protocol  $\Pi_{\mathcal{F}}$  “correctly” computes  $\mathcal{F}$ . That is, in each activation, if the inputs of both parties in the real-life model are identical to their inputs in the ideal process, then the outputs of the uncorrupted parties are distributed identically as their outputs in the ideal process. This fact is easily verified and follows inductively from the property that the parties always hold correct shares of the lines above the gates computed so far. (The base case of the induction relates to the fact that the parties hold correct shares of the input and internal state lines. In addition, the lines corresponding to  $\mathcal{F}$ ’s random tape contain uniformly distributed values.)

We now proceed to show that  $\Pi_{\mathcal{F}}$  UC realizes  $\mathcal{F}$ . Intuitively, the security of protocol  $\Pi_{\mathcal{F}}$  is based on the fact that all the intermediate values seen by the parties are uniformly distributed. In particular, the shares that each party receives of the other party’s input are random. Furthermore, every output that a party receives from an oblivious transfer is masked by a random bit chosen by the sender. Throughout the proof, we denote by  $x$  and  $y$  the outputs of  $P_1$  and  $P_2$ , respectively.

Let  $\mathcal{A}$  be a semi-honest, adaptive adversary that interacts with parties running Protocol  $\Pi_{\mathcal{F}}$  in the  $\mathcal{F}_{\text{OT}}$ -hybrid model. We construct an adversary  $\mathcal{S}$  for the ideal process for  $\mathcal{F}$  such that no environment  $\mathcal{Z}$  can tell whether it interacts with  $\mathcal{A}$  and  $\Pi_{\mathcal{F}}$  in the  $\mathcal{F}_{\text{OT}}$ -hybrid model, or with  $\mathcal{S}$  in the ideal process for  $\mathcal{F}$ .  $\mathcal{S}$  internally runs a simulated copy of  $\mathcal{A}$ , and proceeds as follows:



**Simulating the communication with  $\mathcal{Z}$ :** Every input value that  $\mathcal{S}$  receives from  $\mathcal{Z}$  is written on  $\mathcal{A}$ 's input tape (as if coming from  $\mathcal{A}$ 's environment). Likewise, every output value written by  $\mathcal{A}$  on its output tape is copied to  $\mathcal{S}$ 's own output tape (to be read by  $\mathcal{S}$ 's environment  $\mathcal{Z}$ ).

**Simulation of the input stage:** We first describe the simulation in the case that  $P_1$  is corrupted before the protocol begins. In this case,  $\mathcal{S}$  holds the contents of  $P_1$ 's input tape  $(sid, v)$  and therefore externally sends the value to the ideal functionality  $\mathcal{F}$ . Now, the input stage of  $\Pi_{\mathcal{F}}$  consists only of  $P_1$  sending a random string  $v_2$  to  $P_2$ . In the case that  $P_1$  is corrupted, this string is already determined by the specified uniform random tape of  $P_1$ , and thus no further simulation is required. In the case that  $P_1$  is not corrupted,  $\mathcal{S}$  chooses a uniformly distributed string  $v_2$  and simulates  $P_1$  sending this string to  $P_2$ .

**Simulation of the circuit evaluation stage:** The computation of addition gates consists only of local computation and therefore requires no simulation. In contrast, each multiplication gate is computed using an ideal call to  $\mathcal{F}_{\text{OT}}$ , where  $P_1$  plays the sender and  $P_2$  plays the receiver. We describe the simulation of these calls to  $\mathcal{F}_{\text{OT}}$  separately for each corruption case:

1. *Simulation when both  $P_1$  and  $P_2$  are not corrupted:* In this case, the only message seen by  $\mathcal{A}$  in the evaluation of the  $j^{\text{th}}$  gate is the  $(sid \circ j)$  message from the corresponding copy of  $\mathcal{F}_{\text{OT}}$ . Thus,  $\mathcal{S}$  simulates this stage by merely simulating for  $\mathcal{A}$  an  $(sid \circ j)$  message sent from  $\mathcal{F}_{\text{OT}}$  to the recipient  $P_2$ .
2. *Simulation when  $P_1$  is corrupted and  $P_2$  is not corrupted:* The simulation in this case is the same as in the previous ( $P_1$  obtains no output from  $\mathcal{F}_{\text{OT}}$  and therefore  $\mathcal{A}$  receives  $(sid \circ j)$  only).
3. *Simulation when  $P_1$  is not corrupted and  $P_2$  is corrupted:* The receiver  $P_2$  obtains a uniformly distributed bit  $\gamma_2$  as output from each call to  $\mathcal{F}_{\text{OT}}$ . Therefore,  $\mathcal{S}$  merely chooses  $\gamma_2 \in_R \{0, 1\}$  and simulates  $P_2$  receiving  $\gamma_2$  from  $\mathcal{F}_{\text{OT}}$ .
4. *Simulation when both  $P_1$  and  $P_2$  are corrupted:* Since all input and random tapes are already defined when both parties are corrupted, simulation is straightforward.

**Simulation of the output stage:**  $\mathcal{S}$  simulates  $P_1$  and  $P_2$  sending each other their shares of the output lines. As above, we separately describe the simulation for each corruption case:

1. *Simulation when both  $P_1$  and  $P_2$  are not corrupted:* In this case, all  $\mathcal{A}$  sees is  $P_1$  and  $P_2$  sending each other random  $m$ -bit strings. Therefore,  $\mathcal{S}$  chooses  $y_1, x_2 \in_R \{0, 1\}^m$  and simulates  $P_1$  sending  $y_1$  to  $P_2$  and  $P_2$  sending  $x_2$  to  $P_1$  ( $y_1$  is  $P_1$ 's share in  $P_2$ 's output  $y$  and vice versa for  $x_2$ ).
2. *Simulation when  $P_1$  is corrupted and  $P_2$  is not corrupted:* First, notice that the output shares of a corrupted party are already defined

(because  $\mathcal{A}$  holds the view of any corrupted party and this view defines the shares in all output lines). Thus, in this case, the string sent by  $P_1$  in the output stage is predetermined. In contrast,  $P_2$ 's string is determined as follows:  $P_1$  is corrupted and therefore  $\mathcal{S}$  has  $P_1$ 's output  $x$ . Furthermore,  $P_1$ 's shares  $x_1$  in its own output lines are fixed (because  $P_1$  is corrupted).  $\mathcal{S}$  therefore simulates  $P_2$  sends  $x_2 = x \oplus x_1$  to  $P_1$  (and so  $P_1$  reconstructs its output to  $x$ , as required).

3. *Simulation when  $P_1$  is not corrupted and  $P_2$  is corrupted:* The simulation here is the same as in the previous case (while reversing the roles of  $P_1$  and  $P_2$ ).
4. *Simulation when both  $P_1$  and  $P_2$  are corrupted:* The shares of all output lines of both parties are already determined and so simulation is straightforward.

**Simulation of the first corruption:** We now show how  $\mathcal{S}$  simulates the first corruption of a party. Notice that this can occur at any stage after the simulation described above begins. (If the party is corrupted before the execution begins, then the simulation is according to above.) We describe the corruption as if it occurs at the end of the simulation; if it occurs earlier, then the simulator follows the instructions here only until the appropriate point. We differentiate between the corruptions of  $P_1$  and  $P_2$ :

1.  *$P_1$  is the first party corrupted:* Upon corrupting  $P_1$ , simulator  $\mathcal{S}$  receives  $P_1$ 's input value  $v$  and output value  $x$ .  $\mathcal{S}$  proceeds by generating the view of  $P_1$  in the input preparation stage. Let  $v_2$  be the message that  $P_1$  sent  $P_2$  in the simulation of the input stage by  $\mathcal{S}$ . Then,  $\mathcal{S}$  sets  $P_1$ 's shares of its input to  $v_1$ , where  $v_1 \oplus v_2 = v$ . Furthermore,  $\mathcal{S}$  sets  $P_1$ 's  $m$ -bit input  $r_1$  to the lines corresponding to  $C_{\mathcal{F}}$ 's random tape to be a uniformly distributed string, and  $P_1$ 's shares of the internal state of  $\mathcal{F}$  to be a random string  $c_1 \in_R \{0, 1\}^m$ . (Actually, if this is the first activation of  $C_{\mathcal{F}}$ , then  $c_1$  is set to  $0^m$  to denote the empty state.) In addition,  $\mathcal{S}$  sets  $P_1$ 's random tape to be a uniformly distributed string of the appropriate length for running  $\Pi_{\mathcal{F}}$ . (Notice that this random tape defines the bits  $\gamma_1$  that  $P_1$  chooses when defining the oblivious transfer tables for the multiplication gates; recall that these bits then constitute  $P_1$ 's shares of the output lines from these gates.) In the case that  $P_1$  is corrupted before the output stage, this actually completes the simulation of  $P_1$ 's view of the evaluation until the corruption took place. This is due to the fact that  $P_1$  receives *no* messages during the protocol execution until the output stage ( $P_1$  is always the oblivious transfer sender).

We now consider the case that  $P_1$  is corrupted after the output stage is completed. In this case the output messages  $x_2$  and  $y_1$  of both parties have already been sent. Thus, we must show that  $\mathcal{S}$  can efficiently compute a random tape for  $P_1$  that is consistent with these

messages. For simplicity of exposition, we assume that only multiplication gates, and no addition gates, lead to output lines; any circuit can be easily modified to fulfill this requirement. Now, notice that the random coin  $\gamma_1$  chosen by  $P_1$  in any given multiplication gate is independent of all other coins. Therefore, the simulated output messages  $x_2, y_1$  that  $\mathcal{S}$  already sent only influence the coins of multiplication gates that lead to output lines; the coins of all other multiplication gates can be chosen uniformly and independently of  $x_2, y_1$ . The coins for multiplication gates leading to output lines are chosen as follows: For the  $i^{\text{th}}$  output line belonging to  $P_2$ 's output,  $\mathcal{S}$  sets  $P_1$ 's coin  $\gamma_1$  to equal the  $i^{\text{th}}$  bit of  $y_1$ . (Recall that  $P_1$ 's random coin  $\gamma_1$  equals its output from the gate; therefore,  $P_1$ 's output from the gate equals its appropriate share in  $P_2$ 's output, as required.) Furthermore, for the  $i^{\text{th}}$  output line belonging to  $P_1$ 's output,  $\mathcal{S}$  sets  $P_1$ 's random coin  $\gamma_1$  to equal the  $i^{\text{th}}$  bit of  $x \oplus x_2$ . (Therefore,  $P_1$ 's reconstructed output equals  $x$ , as required; furthermore, this reconstructed value is independent of the intermediary information learned by the adversary.)

2.  *$P_2$  is the first party corrupted:* Upon the corruption of  $P_2$ , simulator  $\mathcal{S}$  receives  $P_2$ 's output  $y$  ( $P_2$  has no input). Then,  $\mathcal{S}$  must generate  $P_2$ 's view of the execution.  $\mathcal{S}$  begins by choosing  $r_2 \in_R \{0, 1\}^m$  and setting  $P_2$ 's input to the lines corresponding to  $C_{\mathcal{F}}$ 's random tape to  $r_2$ . In addition, it chooses the shares of the internal state of  $\mathcal{F}$  to be a random string  $c_2$ . (As above, in the first activation of  $C_{\mathcal{F}}$ , the string  $c_2$  is set to  $0^m$ .) Next, notice that from this point on,  $P_2$  is deterministic (and thus it needs no random tape). Also, notice that the value that  $P_2$  receives in each oblivious transfer is uniformly distributed. Therefore,  $\mathcal{S}$  simulates  $P_2$  receiving a random bit for every oblivious transfer ( $\mathcal{S}$  works in this way irrespective of when  $P_2$  was corrupted). If this corruption occurs before the output stage has been simulated, then the above description is complete (and accurate). However, if the corruption occurs after the simulation of the output stage, then the following changes must be made. First, as above, the random bits chosen for  $P_2$ 's outputs from the oblivious transfers define  $P_2$ 's shares in all the output lines. Now, if the output stage has already been simulated then the string  $x_2$  sent by  $P_2$  to  $P_1$  and the string  $y_1$  sent by  $P_1$  to  $P_2$  have already been fixed. Thus, as in the previous case,  $\mathcal{S}$  chooses the output bits of the oblivious transfers so that they are consistent with these strings. In particular, let  $y$  be  $P_2$ 's output (this is known to  $\mathcal{S}$  since  $P_2$  is corrupted) and define  $y_2 = y \oplus y_1$ . Then,  $\mathcal{S}$  defines  $P_2$ 's output-bit of the oblivious transfer that is associated with the  $i^{\text{th}}$  bit of its shares of its own output to be the  $i^{\text{th}}$  bit of  $y_2$ . Likewise, the output from the oblivious transfer associated with the  $i^{\text{th}}$  bit of  $P_2$ 's share of  $P_1$ 's output is set to equal the  $i^{\text{th}}$  bit of  $x_2$ .

We note that in the above description,  $\mathcal{S}$  generates the corrupted party's view of the *current* activation. In addition, it must also generate the party's view for all the activations in the past. Observe that the only dependence between activations is that the shares of the state string input into a given activation equal the shares of the state string output from the preceding activation. Thus, the simulation of prior activations is exactly the case of simulation where the corruption occurs after the output stage has been completed. The only difference is that  $\mathcal{S}$  defines the shares of the state string so that they are consistent between consecutive activations.

**Simulation of the second corruption:** As before, we differentiate between the corruptions of  $P_1$  and  $P_2$ :

1.  *$P_2$  is the second party corrupted:* Upon corrupting  $P_2$ , simulator  $\mathcal{S}$  obtains  $P_2$ 's output in this activation and all its inputs and outputs from previous activations. Furthermore, since the functionality is adaptively well-formed,  $\mathcal{S}$  obtains the random tape used by the ideal functionality  $\mathcal{F}$  in its computation. Next,  $\mathcal{S}$  computes the internal state of  $\mathcal{F}$  in this activation, based on all the past inputs, outputs and the internal random tape of  $\mathcal{F}$  (this can be computed efficiently). Let  $c$  be this state string and let  $r$  equal  $\mathcal{F}$ 's  $m$ -bit random tape. Then,  $P_2$  sets  $c_2$  such that  $c = c_1 \oplus c_2$ , where  $c_1$  was randomly chosen upon  $P_1$ 's corruption. ( $\mathcal{S}$  also makes sure that the output state information from the previous execution equals the input state information from this execution. This is easily accomplished because output gates are always immediately preceded by multiplication gates, and so independent random coins are used.) Similarly,  $\mathcal{S}$  sets  $r_2 = r_1 \oplus r$ , where  $r$  equals  $\mathcal{F}$ 's random tape and  $r_1$  equals the random string chosen upon  $P_1$ 's corruption (for simulating  $P_1$ 's share of the random tape of  $C_{\mathcal{F}}$ ).

We now proceed to the rest of the simulation. In the case we are considering here,  $P_1$  has already been corrupted. Therefore, all the tables for the oblivious transfers have already been defined. It thus remains to show which values  $P_2$  receives from each of these gate evaluations. However, this is immediately defined by  $P_2$ 's input and the oblivious transfer tables. Thus, all the values received by  $P_2$  from this point on, including the output values, are fully defined, and  $\mathcal{S}$  can directly compute them.

2.  *$P_1$  is the second party corrupted:* The simulation by  $\mathcal{S}$  here begins in the same way as when  $P_2$  is the second party corrupted. That is,  $\mathcal{S}$  corrupts  $P_1$  and obtains the random tape of  $\mathcal{F}$ . Then,  $\mathcal{S}$  defines the appropriate state share string  $c_1$ , and random tape share string  $r_1$  (in the same way as above). In addition,  $\mathcal{S}$  obtains  $P_1$ 's input value  $v$  and defines the appropriate share  $v_1$  (choosing it so that  $v_1 \oplus v_2 = v$ ).

This defines all the inputs into the circuit  $C_{\mathcal{F}}$ . Given this information,  $\mathcal{S}$  constructs the tables for all the oblivious transfers. Recall that  $P_2$  is already corrupted. Therefore, the bits that it receives from each oblivious transfer are already defined. Now, for each gate (working from the inputs to the outputs),  $\mathcal{S}$  works as follows. Let  $\gamma_2$  be the output that  $P_2$  received from some oblivious transfer. Furthermore, assume that  $\mathcal{S}$  holds the input shares of both parties for the gate in question (this can be assumed because  $\mathcal{S}$  works bottom-up, from the input lines to the output lines). Then,  $\mathcal{S}$  checks what the real (unmasked) output bit of the gate should be, let this value be  $\gamma$ . Given that  $P_2$  received  $\gamma_2$  and the output value should be  $\gamma$ , simulator  $\mathcal{S}$  sets  $P_1$ 's random-bit in defining this table to be  $\gamma_1 = \gamma_2 \oplus \gamma$  (notice that  $\gamma_1$  and  $P_1$ 's input values to the gate fully define the table).  $\mathcal{S}$  continues in this way for all the gates evaluated in the simulation before  $P_1$  was corrupted. We note that if the corruption occurred after the output stage, then the output strings sent are defined by the outputs of the gates, as computed above.

**Output and output delivery:**  $\mathcal{S}$  delivers the output from  $\mathcal{F}$  to (an uncorrupted) party after  $\mathcal{A}$  delivers the corresponding output message to the party in the simulation. This takes care of the outputs of uncorrupted parties. For a corrupted party  $P_i$  ( $i \in \{1, 2\}$ ), simulator  $\mathcal{S}$  copies the contents of the simulated  $P_i$ 's output tape (as written by  $\mathcal{A}$ ) onto the output tape of the ideal process party  $P_i$ .

**Analysis of  $\mathcal{S}$ .** We show that no environment  $\mathcal{Z}$  can distinguish the case where it interacts with  $\mathcal{S}$  and  $\mathcal{F}$  in the ideal process from the case where it interacts with  $\mathcal{A}$  and  $\Pi_{\mathcal{F}}$  in the  $\mathcal{F}_{\text{OT}}$ -hybrid model. In fact, we demonstrate that  $\mathcal{Z}$ 's view is distributed *identically* in the two interactions.

The proof proceeds by induction on the number of activations in a run of  $\mathcal{Z}$ . Recall that in each activation,  $\mathcal{Z}$  reads the output tapes of  $P_1$ ,  $P_2$ , and the adversary, and then activates either  $P_1$ ,  $P_2$  or the adversary with some input value. (One should not confuse activations of a party, as is the intention here, with activations of the functionality and protocol.) We actually prove a somewhat stronger claim: Let  $\zeta_i^{\text{R}}$  denote the random variable describing the state of  $\mathcal{Z}$  at the onset of the  $i^{\text{th}}$  activation in a *real* (or, more precisely, hybrid) interaction with adversary  $\mathcal{A}$  and parties running  $\Pi_{\mathcal{F}}$  in the  $\mathcal{F}_{\text{OT}}$ -hybrid model, and let  $\alpha_i^{\text{R}}$  denote the random variable describing the state of  $\mathcal{A}$  at this point in the interaction. Let  $\zeta_i^{\text{I}}$  denote the random variable describing the state of  $\mathcal{Z}$  at the onset of its  $i^{\text{th}}$  activation in an interaction with adversary  $\mathcal{S}$  in the *ideal* process for functionality  $\mathcal{F}$ , and let  $\alpha_i^{\text{I}}$  denote the random variable describing the state of the simulated  $\mathcal{A}$  within  $\mathcal{S}$  at this point in the interaction. We show that for all  $i$ , the pairs  $(\zeta_i^{\text{R}}, \alpha_i^{\text{R}})$  and  $(\zeta_i^{\text{I}}, \alpha_i^{\text{I}})$  are identically distributed. More precisely, Let  $\epsilon > 0$ . Then, for any values  $a_1, a_2, b_1, b_2$  we show:

$$\begin{aligned} \Pr \left[ (\zeta_{i+1}^R, \alpha_{i+1}^R) = (b_1, b_2) \mid (\zeta_i^R, \alpha_i^R) = (a_1, a_2) \right] \\ = \Pr \left[ (\zeta_{i+1}^I, \alpha_{i+1}^I) = (b_1, b_2) \mid (\zeta_i^I, \alpha_i^I) = (a_1, a_2) \right] \end{aligned} \quad (4.2)$$

That is, assume that the states of  $\mathcal{Z}$  and  $\mathcal{A}$  at the onset of some activation of  $\mathcal{Z}$  have some arbitrary (fixed) values  $a_1$  and  $a_2$ , respectively. Then the joint distribution of the states of  $\mathcal{Z}$  and  $\mathcal{A}$  at the onset of the next activation of  $\mathcal{Z}$  is the same regardless of whether we are in the “real interaction” with  $\Pi_{\mathcal{F}}$ , or in the ideal process. (In the real interaction with  $\Pi_{\mathcal{F}}$ , the probability is taken over the random choices of the uncorrupted parties. In the ideal process the probability is taken over the random choices of  $\mathcal{S}$  and  $\mathcal{F}$ .)

Asserting Eq. (4.2), recall that in the  $i^{\text{th}}$  activation  $\mathcal{Z}$  first reads the output tapes of  $P_1, P_2$ , and the adversary. (We envision that these values are written on a special part of the incoming communication tape of  $\mathcal{Z}$ , and are thus part of its state  $\zeta_i^R = \zeta_i^I$ .) Next,  $\mathcal{Z}$  either activates some uncorrupted party with some input  $v$ , or activates the adversary with input  $v$ . We treat these cases separately:

**$\mathcal{Z}$  activates an uncorrupted party with some input value  $v$ .** In this case, in the interaction with  $\Pi_{\mathcal{F}}$ , the activated party sends out a request to the other party to evaluate an activation of  $C_{\mathcal{F}}$ , plus a random share of  $v$ . This message becomes part of the state of  $\mathcal{A}$  (who sees all messages sent). In the ideal process,  $\mathcal{S}$  (who sees that the party has written a message on its outgoing communication tape for  $\mathcal{F}$ ) generates the message that  $\mathcal{A}$  would expect to see; recall that this message is just a uniformly distributed string.

**$\mathcal{Z}$  activates the adversary or a corrupted party with an input  $v$ .** Recall that in the interaction with  $\Pi_{\mathcal{F}}$  adversary  $\mathcal{A}$  is now activated, reads  $v$ , and in addition has access to the messages sent by the parties and by the various copies of  $\mathcal{F}_{\text{OT}}$  since its last activation. (We envision that this information is written on  $\mathcal{A}$ ’s incoming communication tape.) Next,  $\mathcal{A}$  can either deliver a message to some party, modify the input/output tapes of some already corrupted party or corrupt a currently honest party. Finally,  $\mathcal{A}$  writes some value on its output tape and completes its activation. In the ideal process,  $\mathcal{S}$  forwards  $v$  to  $\mathcal{A}$  and activates  $\mathcal{A}$ . Next,  $\mathcal{S}$  provides  $\mathcal{A}$  with additional information representing the messages sent by the parties and also, in case of party corruption, the internal state of the corrupted party.

We proceed in four steps. First, we show that the contents of  $\mathcal{A}$ ’s incoming communication tape has the same distribution in both interactions. Second, we show that the effect of message delivery on the states of  $\mathcal{A}$  and  $\mathcal{Z}$  is the same in both interactions. Third, we demonstrate that the information learned by  $\mathcal{A}$  upon corrupting a party has the same distribution in both interactions. Finally, we demonstrate that  $\mathcal{A}$ ’s view regarding the states of the already corrupted parties has the same distribution in both interactions.

**New messages seen by  $\mathcal{A}$ .** Each message seen by  $\mathcal{A}$  is of one of the following possible types:

- *An input-sharing message as described above:* As mentioned above, in this case in both interactions  $\mathcal{A}$  sees an  $m$ -bit long random string, representing a share of the sender's new input value.
- *A message from a party to some copy of  $\mathcal{F}_{\text{OT}}$ :* In this case, in both interactions,  $\mathcal{A}$  only gets notified that some message was sent from the party to the copy of  $\mathcal{F}_{\text{OT}}$ .
- *A message from some copy of  $\mathcal{F}_{\text{OT}}$  to  $P_2$ :* In both interactions, if  $P_2$  is uncorrupted then  $\mathcal{A}$  does not see the contents of this message. If  $P_2$  is corrupted then this message consists of a single random bit that is independent from the states of  $\mathcal{A}$  and  $\mathcal{Z}$  so far. (This bit is  $P_2$ 's share of the corresponding line of the circuit.)
- *An output message from one party to another:* Here one party sends its share of some output line to the other party (who is designated to get the value of this line.) In both interactions, if the recipient party is uncorrupted then this message consists of a single random bit  $\alpha$  that is independent from the states of  $\mathcal{A}$  and  $\mathcal{Z}$  so far. If the recipient is corrupted then  $\mathcal{A}$  already has  $\beta$ , the recipient's share of that line. In the interaction with  $\Pi_{\mathcal{F}}$ , the value  $\gamma = \alpha \oplus \beta$  is the value of this output line in  $C_{\mathcal{F}}$ . In the ideal process,  $\gamma = \alpha \oplus \beta$  is the corresponding value generated by  $\mathcal{F}$ . The distribution of  $c$  (given the states of  $\mathcal{A}$  and  $\mathcal{Z}$  so far) is identical in both cases; this is the case because  $C_{\mathcal{F}}$  correctly represents an activation of  $\mathcal{F}$ .

**Messages delivered by  $\mathcal{A}$ .** If  $\mathcal{A}$  delivers an output message to some party in an execution of  $\Pi_{\mathcal{F}}$ , then this party outputs the (correct) value derived from the corresponding output lines of  $C_{\mathcal{F}}$ . This output value,  $\gamma^{\text{R}}$ , becomes part of the state of  $\mathcal{Z}$  (to be read by  $\mathcal{Z}$  at the onset of its next activation.) If  $\mathcal{A}$  delivers an output message to some party in the ideal process, then  $\mathcal{S}$  (who runs  $\mathcal{A}$ ) delivers the corresponding message from  $\mathcal{F}$  to this party. Consequently, this party outputs the value,  $\gamma^{\text{I}}$ , sent by  $\mathcal{F}$ . Since  $C_{\mathcal{F}}$  correctly represents the computation of  $\mathcal{F}$ , we have that  $\gamma^{\text{R}}$  and  $\gamma^{\text{I}}$  are identically distributed.

If  $\mathcal{A}$  delivers to some party  $P_i$  a message that is not an output message then  $P_i$  outputs nothing. ( $P_i$  may send other messages, but these messages will only become part of the state of  $\mathcal{A}$  in its next activation. This next activation of  $\mathcal{A}$  occurs after the next activation of  $\mathcal{Z}$ .)

**Corruption of the first party.** In the interaction with  $\Pi_{\mathcal{F}}$ , upon corrupting the first party  $\mathcal{A}$  sees all the past inputs and outputs of this party. In addition, it sees all the shares of this party from the input lines, the random input lines, the internal state input lines, and all the internal lines of the circuit  $C_{\mathcal{F}}$ ; these shares are all random values distributed independently from the states of  $\mathcal{A}$  and  $\mathcal{Z}$  so far. In the ideal process,  $\mathcal{S}$  provides  $\mathcal{A}$  with identically distributed information.

**Corruption of the second party.** In the interaction with  $\Pi_{\mathcal{F}}$ , upon corrupting the second party  $\mathcal{A}$  sees the same information as in the first corruption, namely all the past inputs and outputs of this party, as well as the shares of this party from the input lines, the random input lines, the internal state input lines, and all the internal lines of the circuit  $C_{\mathcal{F}}$ . Here, however, this information determines the actual values of all types of input lines to the circuit, plus the values of all the internal lines of the circuit. (The values of the random input lines to the circuit are uniformly distributed. All other lines are uniquely determined by the states of  $\mathcal{Z}$  and  $\mathcal{A}$  at this point.) In the ideal process,  $\mathcal{S}$  provides  $\mathcal{A}$  with identically distributed information. (This can be seen from the code of  $\mathcal{S}$ .)

This completes the analysis of  $\mathcal{S}$  and the proof of the claim. ■

Using the UC composition theorem, Proposition 4.4.3 follows from Claims 4.4.1, 4.4.2, and 4.4.4.

## 4.5 Universally Composable Commitments

We describe our new universally composable non-interactive commitment scheme that is secure against adaptive adversaries. Our construction is in the common reference string model, and assumes only the existence of trapdoor permutations. (If the common reference string must come from the uniform distribution, then we actually require trapdoor permutations with dense public descriptions [DP92].) UC commitment schemes are protocols that UC realize the multi-session ideal commitment functionality  $\mathcal{F}_{\text{MCOM}}$  that is presented in Figure 4.4. Note that  $\mathcal{F}_{\text{MCOM}}$  is in fact a re-formulation of  $\hat{\mathcal{F}}_{\text{COM}}$ , the multi-session extension of the single-session ideal commitment functionality,  $\mathcal{F}_{\text{COM}}$ , presented in [CF01].

### Functionality $\mathcal{F}_{\text{MCOM}}$

$\mathcal{F}_{\text{MCOM}}$  proceeds as follows, running with parties  $P_1, \dots, P_n$  and an adversary  $\mathcal{S}$ :

- **Commit Phase:** Upon receiving a message (`commit`,  $sid$ ,  $ssid$ ,  $P_i$ ,  $P_j$ ,  $b$ ) from  $P_i$ , where  $b \in \{0, 1\}$ , record the tuple  $(ssid, P_i, P_j, b)$  and send the message (`receipt`,  $sid$ ,  $ssid$ ,  $P_i$ ,  $P_j$ ) to  $P_j$  and  $\mathcal{S}$ . Ignore any future `commit` messages with the same  $ssid$  from  $P_i$  to  $P_j$ .
- **Reveal Phase:** Upon receiving a message (`reveal`,  $sid$ ,  $ssid$ ) from  $P_i$ : If a tuple  $(ssid, P_i, P_j, b)$  was previously recorded, then send the message (`reveal`,  $sid$ ,  $ssid$ ,  $P_i$ ,  $P_j$ ,  $b$ ) to  $P_j$  and  $\mathcal{S}$ . Otherwise, ignore.

**Fig. 4.4.** The ideal commitment functionality



Informally speaking, in order to achieve universal composability against adaptive adversaries, a commitment scheme must have the following two properties:

- *Polynomial equivocality*:<sup>14</sup> the simulator (i.e., the adversary in the ideal process) should be able to produce polynomially many commitments for which it can decommit to both 0 and 1, using the same reference string. (An additional property is actually needed for adaptive security; see below.) Of course, the real committer must be able to decommit to only a single value (as required by the binding property of commitment schemes).
- *Simulation extractability*: the simulator should be able to extract the contents of any valid commitment generated by the adversary, even after having supplied an adversary with an arbitrary number of equivocal commitments.

We remark that in the equivocal commitment protocols of [DIO98, DKOS01] each copy of the reference string can be used for only a single commitment. Furthermore, they are not extractable. In contrast, [CF01] show how to use a single copy of the reference string for multiple commitments (although they rely on specific cryptographic assumptions).

We describe our construction in phases. First we describe a new non-interactive variant of the Feige-Shamir trapdoor commitment scheme [FS89], which is at the heart of the construction. Then we show how to transform this scheme into one that is universally composable.

**Underlying standard commitment.** Our UC commitment scheme uses a non-interactive, perfectly binding commitment scheme with pseudorandom commitments; denote this scheme by  $Com$ . An example of such a scheme is the standard non-interactive commitment scheme based on a one-way permutation  $f$  and a hard-core predicate  $b$  of  $f$ . In order to commit to a bit  $\sigma$  in this scheme, one computes  $Com(\sigma) = \langle f(U_k), b(U_k) \oplus \sigma \rangle$ , where  $U_k$  is the uniform distribution over  $\{0, 1\}^k$ . The  $Com$  scheme is computationally secret and produces pseudorandom commitments: that is, the distribution ensembles  $\{Com(0)\}$ ,  $\{Com(1)\}$ , and  $\{U_{k+1}\}$  are all computationally indistinguishable.

**Non-interactive Feige-Shamir trapdoor commitments.** We briefly describe a non-interactive version of the Feige-Shamir trapdoor commitment scheme [FS89], which is based on the zero-knowledge proof for Hamiltonicity of Blum [B86]. (We are able to obtain a non-interactive version of this scheme by utilizing the common reference string.) First, we obtain a graph  $G$  (with  $q$  nodes), so that it is hard to find a Hamiltonian cycle in  $G$  within polynomial-time. This is achieved as follows: choose  $x \in_R \{0, 1\}^k$  and compute  $y = f(x)$ , where  $f$  is a one-way function. Then, use the (Cook-Levin)

<sup>14</sup> Commitments with this property have mistakenly been called “equivocal”; we fix this error and call them *equivocal*.

NP-reduction of the language  $\{y \mid \exists x \text{ s.t. } y = f(x)\}$  to that of Hamiltonicity, to obtain a graph  $G$  so that finding a Hamiltonian cycle in  $G$  is equivalent to finding the preimage  $x$  of  $y$ . The one-wayness of  $f$  implies the difficulty of finding a Hamiltonian cycle in  $G$ . This graph  $G$ , or equivalently the string  $y$ , is placed in the common reference string accessible by both parties. Now, in order to commit to 0, the committer commits to a random permutation of  $G$  using the underlying commitment scheme  $Com$  (and decommits by revealing the entire graph and the permutation). In order to commit to 1, the committer commits to a graph containing a randomly labelled  $q$ -cycle only (and decommits by opening this cycle only). Note that this commitment scheme is binding because the ability to decommit to both 0 and 1 implies that the committer knows a Hamiltonian cycle in  $G$ . The important property of the scheme of [FS89] that we use here is equivocality (or what they call the trapdoor property). That is, given a Hamiltonian cycle in  $G$ , it is possible to generate commitments that are indistinguishable from legal ones, and yet have the property that one can decommit to both 0 and 1. In particular, after committing to a random permutation of  $G$ , it is possible to decommit to 0 in the same way as above. However, it is also possible to decommit to 1 by only revealing the (known) Hamiltonian cycle in  $G$ .

**Adaptively secure commitments.** In order to explain elements of our construction that are important in obtaining security against adaptive adversaries, we begin by describing a general problem that can occur in such a setting. In the adaptive setting, the adversary can corrupt parties at any point in the computation. Specifically, for the case of commitments, this means that the adversary can corrupt a party *after* the commit stage has been completed. In such a case, the simulator must generate some “simulated commitment” string  $c$  before the committing party is corrupted, and therefore without knowing the value  $b$  that is being committed to. Then, after the corruption takes place and the committed value  $b$  becomes known to the simulator and adversary, it must be possible to “explain” the string  $c$  as a valid commitment to  $b$ . This is because in a real execution,  $c$  is indeed a commitment to  $b$  and  $\mathcal{A}$  will see this upon corrupting the committer. However, in the ideal process,  $c$  is generated by  $\mathcal{S}$  who knows nothing about the value  $b$ . A valid “explanation” of the string  $c$  is a series of coins  $r_b$  such that upon input  $b$  and random coins  $r_b$ , the *honest committer* would output the commitment string  $c$ . Consider the [FS89] commitment scheme described above, and assume that the ideal-model simulator  $\mathcal{S}$  knows a Hamiltonian cycle in  $G$ . Then, as we have described, it is possible for  $\mathcal{S}$  to generate a commitment string  $c$  that can be decommitted to both 0 and 1. However, it is *not* possible for  $\mathcal{S}$  to later provide two sets of coins  $r_0$  and  $r_1$  so that  $c$  is the result of an honest commitment to  $b$  with coins  $r_b$  (for  $b = 0$  and  $b = 1$ ). This is because the coins used in all of the  $Com$  commitments demonstrate whether a permutation of  $G$  was committed to or just a simple cycle. This

demonstrates that the trapdoor or equivocality property is not enough for obtaining adaptive security.

In order to obtain adaptive security, we modify the way that the *Com* commitments are generated in the [FS89] scheme. Specifically, commitments which are not opened upon decommitment (i.e., commitments outside of the simple cycle in a commitment to 1) are generated by just choosing random strings. Since *Com* has a pseudorandom range and these commitments are never opened, this makes no difference. Below we will show that this actually suffices for obtaining adaptive security. More precisely, the adaptively secure scheme, denoted **aHC** (for *adaptive Hamiltonian Commitment*), is defined as follows:

- To commit to a 0, the sender picks a random permutation  $\pi$  of the nodes of  $G$ , and commits to the entries of the adjacency matrix of the permuted graph one by one, using *Com*. To decommit, the sender sends  $\pi$  and decommits to every entry of the adjacency matrix. The receiver verifies that the graph it received is  $\pi(G)$ . (This is the same as in the [FS89] scheme.)
- To commit to a 1, the sender chooses a randomly labeled  $q$ -cycle, and for all the entries in the adjacency matrix corresponding to edges on the  $q$ -cycle, it uses *Com* to commit to 1 values. For all the other entries, it produces *random values* from  $U_{k+1}$  (for which it does not know the decommitment). To decommit, the sender opens only the entries corresponding to the randomly chosen  $q$ -cycle in the adjacency matrix. (This is the point where our scheme differs to that of [FS89]. That is, in [FS89] the edges that are not on the  $q$ -cycle are sent as commitments to 0. Here, random strings are sent instead.)

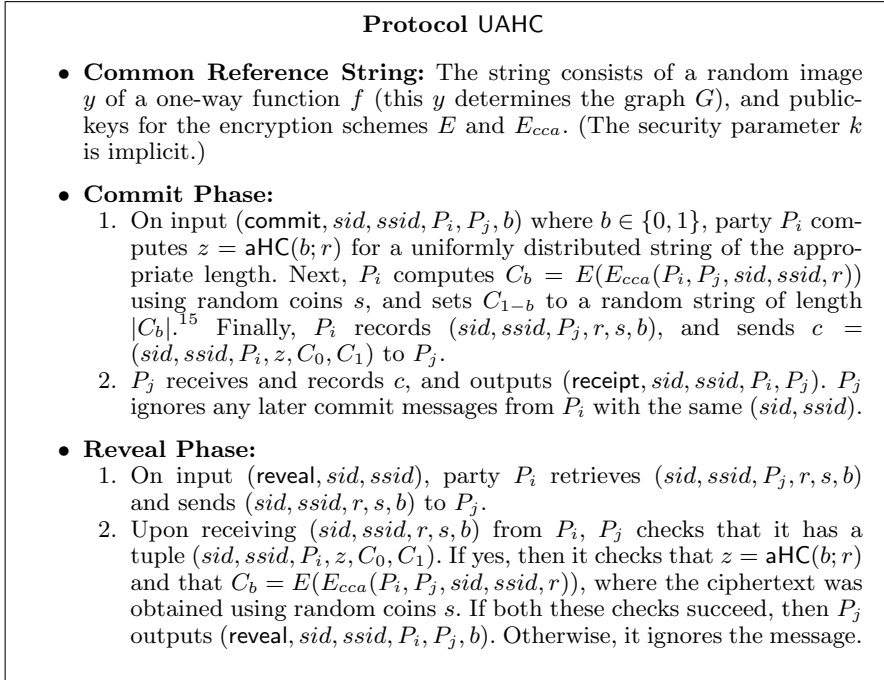
By the above description, the length of the random string used in order to commit to 0 is different from the length of the random string used in order to commit to 1. Nevertheless, we pad the lengths so that they are equal (the reason why this is needed will be explained below). We denote by **aHC**( $b; r$ ) a commitment of the bit  $b$  using randomness  $r$ , and by **aHC**( $b$ ) the distribution **aHC**( $b; U_{|r|}$ ).

This commitment scheme has the property of being computationally secret; i.e., the distribution ensembles  $\{\mathbf{aHC}(0)\}$  and  $\{\mathbf{aHC}(1)\}$  are computationally indistinguishable for any graph  $G$ . Also, given the opening of any commitment to both a 0 and 1, one can extract a Hamiltonian cycle in  $G$ . Therefore, the committer cannot decommit to both 0 and 1, and the binding property holds. Finally, as with the scheme of [FS89], given a Hamiltonian cycle in  $G$ , a simulator can generate a commitment to 0 and later open it to both 0 and 1. (This is because the simulator knows a simple  $q$ -cycle in  $G$  itself.) Furthermore, in contrast to [FS89], here the simulator can also produce a random tape for the sender, explaining the commitment as a commitment to either 0 or 1. Specifically, the simulator generates each commitment string

$c$  as a commitment to 0. If, upon corruption of the sender, the simulator has to demonstrate that  $c$  is a commitment to 0 then all randomness is revealed. To demonstrate that  $c$  was generated as a commitment to 1, the simulator opens the commitments to the edges in the  $q$ -cycle and claims that all the unopened commitments are merely uniformly chosen strings (rather than commitments to the rest of  $G$ ). This can be done since commitments produced by the underlying commitment scheme  $Com$  are pseudorandom. This therefore gives us polynomial equivocality, where the same reference string can be reused polynomially-many times.

**Achieving simulation extractability.** As discussed above, the commitment scheme  $aHC$  has the equivocality property, as required. However, a UC commitment scheme must also have the *simulation extractability* property. We must therefore modify our scheme in such a way that we add extractability without sacrificing equivocality. Simulation-extractability alone could be achieved by including a public-key for an encryption scheme secure against adaptive chosen-ciphertext attacks (CCA2) [DDN91] into the common reference string, and having the committer send an encryption of the decommitment information along with the commitment itself. A simulator knowing the associated decryption key can decrypt and obtain the decommitment information, thereby extracting the committed value from any adversarially prepared commitment. (The reason that we use a CCA2-secure encryption scheme will become evident in the proof. Intuitively, the reason is that in the simulated interaction extracting the committed value involves ciphertext *decryptions*. Thus by interacting with the simulator the adversary essentially has access to a decryption oracle for the encryption scheme.) However, just encrypting the decommitment information destroys the equivocality of the overall scheme, since such an encryption is binding even to a simulator. In order to regain equivocality, we use encryption schemes with pseudorandom ciphertexts. This is used in the following way. Given any equivocal commitment, there are two possible decommitment strings (by the binding property, only one can be efficiently found but they both exist). The commitment is sent along with two ciphertexts: one ciphertext is an encryption of the decommitment information known to the committer and the other ciphertext is just a uniformly distributed string. In this way, equivocality is preserved because a simulator knowing both decommitment strings can encrypt them both and later claim that it only knows the decryption to one and that the other was uniformly chosen. A problem with this solutions is that there is no known CCA2-secure scheme with pseudorandom ciphertexts (and assuming only trapdoor permutations). We therefore use double encryption. That is, first the value is encrypted using a CCA2-secure scheme, which may result in a ciphertext which is not pseudorandom, and then this ciphertext is re-encrypted using an encryption scheme with pseudorandom ciphertexts. (The second scheme need only be secure against chosen plaintext attacks.)

For the CCA2-secure scheme, denoted  $E_{cca}$ , we can use any known scheme based on trapdoor permutations with the (natural) property that any ciphertext has at most one valid decryption. This property holds for all known such encryption schemes, and in particular for the scheme of [DDN91]. For the second encryption scheme, denoted  $E$ , we use the standard encryption scheme based on trapdoor-permutations and hard-core predicates [GL89], where the public key is a trapdoor permutation  $f$ , and the private key is  $f^{-1}$ . Here encryption of a bit  $b$  is  $f(x)$  where  $x$  is a randomly chosen element such that the hard-core predicate of  $x$  is  $b$ . Note that encryptions of both 0 and 1 are pseudorandom. The commitment scheme, called UAHC (for UC Adaptive Hamiltonicity Commitment), is presented in Figure 4.5.



**Fig. 4.5.** The commitment protocol UAHC

Let  $\mathcal{F}_{\text{CRS}}$  denote the common reference string functionality (that is,  $\mathcal{F}_{\text{CRS}}$  provides all parties with a common, public string drawn from the distribution described in Figure 4.5). Then, we have:

<sup>15</sup> As we have mentioned, the length of the random string  $r$  is the same for the case of  $b = 0$  and  $b = 1$ . This is necessary because otherwise it would be possible to distinguish commitments merely by looking at the lengths of  $C_0$  and  $C_1$ .

**Proposition 4.5.1** *Assuming the existence of trapdoor permutations, Protocol UAHC of Figure 4.5 UC realizes  $\mathcal{F}_{\text{MCOM}}$  in the  $\mathcal{F}_{\text{CRS}}$ -hybrid model.*

**Proof.** Let  $\mathcal{A}$  be a malicious, adaptive adversary that interacts with parties running the above protocol in the  $\mathcal{F}_{\text{CRS}}$ -hybrid model. We construct an ideal process adversary  $\mathcal{S}$  with access to  $\mathcal{F}_{\text{MCOM}}$ , which simulates a real execution of Protocol UAHC with  $\mathcal{A}$  such that no environment  $\mathcal{Z}$  can distinguish the ideal process with  $\mathcal{S}$  and  $\mathcal{F}_{\text{MCOM}}$  from a real execution of UAHC with  $\mathcal{A}$ .

Recall that  $\mathcal{S}$  interacts with the ideal functionality  $\mathcal{F}_{\text{MCOM}}$  and with the environment  $\mathcal{Z}$ . The ideal adversary  $\mathcal{S}$  starts by invoking a copy of  $\mathcal{A}$  and running a simulated interaction of  $\mathcal{A}$  with the environment  $\mathcal{Z}$  and parties running the protocol. (We refer to the interaction of  $\mathcal{S}$  in the ideal process as *external interaction*. The interaction of  $\mathcal{S}$  with the simulated  $\mathcal{A}$  is called *internal interaction*.) We fix the following notation. First, the session and sub-session identifiers are respectively denoted by *sid* and *ssid*. Next, the committing party is denoted  $P_i$  and the receiving party  $P_j$ . Finally,  $C$  denotes a ciphertext generated from  $E(\cdot)$ , and  $C^{cca}$  denotes a ciphertext generated from  $E_{cca}(\cdot)$ . Simulator  $\mathcal{S}$  proceeds as follows:

**Initialization step:** The common reference string (CRS) is chosen by  $\mathcal{S}$  in the following way (recall that  $\mathcal{S}$  chooses the CRS for the simulated  $\mathcal{A}$  by itself):

1.  $\mathcal{S}$  chooses a string  $x \in_R \{0, 1\}^k$  and computes  $y = f(x)$ , where  $f$  is the specified one-way function.
2.  $\mathcal{S}$  runs the key-generation algorithm for the CCA2-secure encryption scheme, obtaining a public-key  $E_{cca}$  and a secret-key  $D_{cca}$ .
3.  $\mathcal{S}$  runs the key-generation algorithm for the CPA-secure encryption scheme with pseudorandom ciphertexts, obtaining a public-key  $E$  and a secret-key  $D$ .

Then,  $\mathcal{S}$  sets the common reference string to equal  $(y, E_{cca}, E)$  and locally stores the triple  $(x, D_{cca}, D)$ . (Recall that  $y$  defines a Hamiltonian graph  $G$  and knowing  $x$  is equivalent to knowing a Hamiltonian cycle in  $G$ .)

**Simulating the communication with  $\mathcal{Z}$ :** Every input value that  $\mathcal{S}$  receives from  $\mathcal{Z}$  is written on  $\mathcal{A}$ 's input-tape (as if coming from  $\mathcal{A}$ 's environment). Likewise, every output value written by  $\mathcal{A}$  on its own output tape is copied to  $\mathcal{S}$ 's own output tape (to be read by  $\mathcal{S}$ 's environment  $\mathcal{Z}$ ).

**Simulating “commit” when the committer is uncorrupted:** In the ideal model, when an uncorrupted committer  $P_i$  receives an input value  $(\text{commit}, \text{sid}, \text{ssid}, P_i, P_j, b)$  from the environment, it writes this message on its outgoing communication tape for  $\mathcal{F}_{\text{MCOM}}$ . Recall that by convention, the  $(\text{commit}, \text{sid}, \text{ssid}, P_i, P_j)$  part of the message (i.e., the header) is public and can be read by  $\mathcal{S}$ , whereas the actual input value  $b$  cannot be read by  $\mathcal{S}$  (see Section 4.3.1 – “the ideal process”). Now, upon seeing that  $P_i$  writes a “commit” message for  $\mathcal{F}_{\text{MCOM}}$ ,  $\mathcal{S}$  simulates a real party  $P_i$  writing the commit message of Protocol UAHC on its outgoing

communication tape for  $P_j$ . That is,  $\mathcal{S}$  computes  $z \leftarrow \text{aHC}(0)$  along with two strings  $r_0$  and  $r_1$  such that  $r_b$  constitutes a decommitment of  $z$  to  $b$ . (As we have described, since  $\mathcal{S}$  knows a Hamiltonian cycle in  $G$ , it is able to do this.) Next,  $\mathcal{S}$  computes  $C_0 \leftarrow E(E_{cca}(P_i, P_j, \text{sid}, \text{ssid}, r_0))$  using random coins  $s_0$ , and  $C_1 \leftarrow E(E_{cca}(P_i, P_j, \text{sid}, \text{ssid}, r_1))$  using random coins  $s_1$ . Then,  $\mathcal{S}$  stores  $(c, r_0, s_0, r_1, s_1)$  and simulates  $P_i$  writing  $c = (\text{sid}, \text{ssid}, P_i, z, C_0, C_1)$  on its outgoing communication tape for  $P_j$ . When  $\mathcal{A}$  delivers  $c$  from  $P_i$  to  $P_j$  in the internal simulation, then  $\mathcal{S}$  delivers the message from the ideal process  $P_i$ 's outgoing communication tape to  $\mathcal{F}_{\text{MCOM}}$ . Furthermore,  $\mathcal{S}$  also delivers the (receipt, ...) message from  $\mathcal{F}_{\text{MCOM}}$  to  $P_j$ . If  $\mathcal{A}$  corrupts  $P_i$  before delivering  $c$  and then *modifies*  $c$  before delivering it, then  $\mathcal{S}$  proceeds by following the instructions for a corrupted committer. If  $\mathcal{A}$  corrupts  $P_i$  but does not modify  $c$ , then  $\mathcal{S}$  carries out the simulation as described here.

**Simulating “reveal” when the committer is uncorrupted:** When an honest  $P_i$  receives a (reveal,  $\text{sid}, \text{ssid}$ ) input from  $\mathcal{Z}$ , it writes this on its outgoing communication tape for  $\mathcal{F}_{\text{MCOM}}$  (this entire message is a “header” and is therefore public).  $\mathcal{S}$  then delivers this message to  $\mathcal{F}_{\text{MCOM}}$  and obtains the message (reveal,  $\text{sid}, \text{ssid}, P_i, P_j, b$ ) from  $\mathcal{F}_{\text{MCOM}}$ . Given the value  $b$ ,  $\mathcal{S}$  generates a simulated decommitment message from the real-model  $P_i$ : this message is  $(\text{sid}, \text{ssid}, r_b, s_b, b)$ , where  $r_b$  and  $s_b$  are as generated in the previous item.  $\mathcal{S}$  then internally simulates for  $\mathcal{A}$  the event where  $P_i$  writes this message on its outgoing communication tape for  $P_j$ . When  $\mathcal{A}$  delivers this message from  $P_i$  to  $P_j$  in the internal interaction, then  $\mathcal{S}$  delivers the (reveal,  $\text{sid}, \text{ssid}, P_i, P_j, b$ ) message from  $\mathcal{F}_{\text{MCOM}}$  to  $P_j$ .

**Simulating corruption of parties:** When  $\mathcal{A}$  issues a “corrupt  $P_i$ ” command in the internal (simulated) interaction,  $\mathcal{S}$  first corrupts the ideal model party  $P_i$  and obtains the values of all its unopened commitments. Then,  $\mathcal{S}$  prepares the internal state of  $P_i$  to be consistent with these commitment values in the same way as shown above. That is, in a real execution party  $P_i$  stores the tuple  $(\text{sid}, \text{ssid}, P_j, r, s, b)$  for every commitment  $c$ . In the simulation,  $\mathcal{S}$  defines the stored tuple to be  $(\text{sid}, \text{ssid}, P_j, r_b, s_b, b)$  where  $b$  is the commitment value associated with  $(\text{sid}, \text{ssid})$  in  $P_i$ 's internal state, and  $r_b$  and  $s_b$  are as generated above.

**Simulating “commit” when the committer is corrupted:** When the real-model adversary  $\mathcal{A}$ , controlling corrupted party  $P_i$ , delivers a commitment message  $(\text{sid}, \text{ssid}, P_i, z, C_0, C_1)$  to an uncorrupted party  $P_j$  in the internal (simulated) interaction,  $\mathcal{S}$  works as follows. If a commitment from  $P_i$  to  $P_j$  using identifiers  $(\text{sid}, \text{ssid})$  was delivered by  $\mathcal{A}$  from  $P_i$  to  $P_j$  in the past, then  $\mathcal{S}$  ignores the message. Otherwise, informally speaking,  $\mathcal{S}$  must extract the commitment bit committed to by  $\mathcal{A}$ . Simulator  $\mathcal{S}$  begins by decrypting both  $C_0$  and  $C_1$  obtaining ciphertexts  $C_0^{cca}$  and  $C_1^{cca}$  and then decrypting each of  $C_0^{cca}$  and  $C_1^{cca}$ . There are three cases here:

1. *Case 1:* For some  $b \in \{0, 1\}$ ,  $C_b^{cca}$  decrypts to  $(P_i, P_j, sid, ssid, r)$  where  $r$  is the correct decommitment information for  $z$  as a commitment to  $b$ , and  $C_{1-b}^{cca}$  does *not* decrypt to a decommitment to  $1 - b$ . Then,  $\mathcal{S}$  sends  $(\text{commit}, sid, ssid, P_i, P_j, b)$  to  $\mathcal{F}_{\text{MCOM}}$ , delivers  $\mathcal{F}_{\text{MCOM}}$ 's receipt response to  $P_j$ , and stores the commitment string.
2. *Case 2:* Neither  $C_0^{cca}$  or  $C_1^{cca}$  decrypt to  $(P_i, P_j, sid, ssid, r)$  where  $r$  is the appropriate decommitment information for  $z$  (and  $sid$  and  $ssid$  are the correct identifiers from the commitment message). In this case,  $\mathcal{S}$  sends  $(\text{commit}, sid, ssid, P_i, P_j, 0)$  to  $\mathcal{F}_{\text{MCOM}}$  and delivers  $\mathcal{F}_{\text{MCOM}}$ 's receipt response to  $P_j$ . (The commitment string is not stored, since it will never be opened correctly.)
3. *Case 3:*  $C_0^{cca}$  decrypts to  $(P_i, P_j, sid, ssid, r_0)$  and  $C_1^{cca}$  decrypts to  $(P_i, P_j, sid, ssid, r_1)$ , where  $r_0$  is the correct decommitment information for  $z$  as a commitment to 0 and  $r_1$  is the correct decommitment information for  $z$  as a commitment to 1. Furthermore, the identifiers in the decryption information are the same as in the commitment message. In this case,  $\mathcal{S}$  outputs a special failure symbol and halts.

**Simulating “reveal” when the committer is corrupted:** When  $\mathcal{A}$ , controlling corrupted party  $P_i$ , delivers a reveal message  $(sid, ssid, r, s, b)$  to an uncorrupted party  $P_j$  in the internal (simulated) interaction,  $\mathcal{S}$  works as follows.  $\mathcal{S}$  first checks that a tuple  $(sid, ssid, P_i, z, C_0, C_1)$  is stored and that  $r$  and  $s$  constitute a proper decommitment to  $b$ . If the above holds, then  $\mathcal{S}$  sends  $(\text{reveal}, sid, ssid, P_i, P_j)$  to  $\mathcal{F}_{\text{MCOM}}$  and delivers the reveal message from  $\mathcal{F}_{\text{MCOM}}$  to  $P_j$ . Otherwise,  $\mathcal{S}$  ignores the message.

We now prove that  $\mathcal{Z}$  cannot distinguish an interaction of Protocol UAHC with  $\mathcal{A}$  from an interaction in the ideal process with  $\mathcal{F}_{\text{MCOM}}$  and  $\mathcal{S}$ . In order to show this, we examine several hybrid experiments:

- (I) *Real interaction:* This is the interaction of  $\mathcal{Z}$  with  $\mathcal{A}$  and Protocol UAHC.
- (II) *Real interaction with partially fake commitments:* This is the interaction of  $\mathcal{Z}$  with  $\mathcal{A}$  and Protocol UAHC, except that: (i) The Hamiltonian Cycle to  $G$  is provided to all honest parties, but this information is not revealed upon corruption. (ii) In honest party commitments, a commitment to  $b$  is generated by computing  $z \leftarrow \text{aHC}(0)$  and strings  $r_0, r_1$  such that  $r_0$  and  $r_1$  are correct decommitments to 0 and 1, respectively. (This is just like the simulator.) Then,  $C_b$  is computed as an encryption to  $E(E_{cca}(P_i, P_j, sid, ssid, r_b))$ . However, unlike the simulator,  $C_{1-b}$  is still chosen as a uniformly distributed string. Again, this modification is not revealed upon corruption (i.e., the honest party decommits to  $b$  as in a real interaction).
- (III) *Real interaction with completely fake commitments:* This is the same as (II), except that in commitments generated by honest parties, the ciphertext  $C_{1-b}$  equals  $E(E_{cca}(P_i, P_j, sid, ssid, r_{1-b}))$  as generated by  $\mathcal{S}$ , rather than being chosen uniformly. Commitments are opened in the same way as the simulator.



(IV) *Simulated interaction*: This is the interaction of  $\mathcal{Z}$  with  $\mathcal{S}$ , as described above.

Our aim is to show that interactions (I) and (IV) are indistinguishable to  $\mathcal{Z}$ , or in other words that  $\mathcal{Z}$ 's output at the end of interaction (I) deviates only negligibly from  $\mathcal{Z}$ 's output at the end of interaction (IV). We prove this by showing that each consecutive pair of interactions are indistinguishable to  $\mathcal{Z}$ . (Abusing notation, we use the term “distribution  $i$ ” to denote both “interaction  $i$ ”, and “ $\mathcal{Z}$ 's output from interaction  $i$ ”.)

The fact that distributions (I) and (II) are computationally indistinguishable is derived from the pseudorandomness of the underlying commitment scheme **aHC**. This can be seen as follows. The only difference between the two distributions is that even commitments to 1 are computed by  $z \leftarrow \mathbf{aHC}(0)$ . However, the distribution ensembles  $\{\mathbf{aHC}(0)\}$  and  $\{\mathbf{aHC}(1)\}$  are indistinguishable. Furthermore, these ensembles remain indistinguishable when the decommitment information to 1 is supplied. That is,  $\{\mathbf{aHC}(0), r_1\}$  and  $\{\mathbf{aHC}(1), r\}$  are also indistinguishable, where  $r_1$  is the (simulator) decommitment of  $\mathbf{aHC}(0)$  to 1, and  $r$  is the (prescribed) decommitment of  $\mathbf{aHC}(1)$  to 1. (A standard hybrid argument is employed to take into account the fact that many commitments and decommitments occur in any given execution.)

Next, distributions (II) and (III) are indistinguishable due to the pseudorandomness of encryptions under  $E$ . In particular, the only difference between the distributions is that in (II) the ciphertext  $C_{1-b}$  is uniformly chosen, whereas in (III) ciphertext  $C_{1-b}$  equals  $E(E_{cca}(P_i, P_j, sid, ssid, r_{1-b}))$ . Intuitively, CPA security suffices because in order to emulate experiments (II) and (III), no decryption oracle is needed. In order to formally prove this claim, we use the “left-right” oracle formulation of security for encryption schemes [BBM00]. In this formulation of security, there is a “left-right” oracle (LR-oracle) which has a randomly chosen and hidden value  $b \in \{0, 1\}$  built into it. When queried with a pair of plaintexts  $(a_0, a_1)$ , the oracle returns  $E(a_b)$ . Equivalently, the oracle can be queried with a single message  $a$  such that it returns  $E(a)$  if  $b = 0$  and a uniformly distributed string if  $b = 1$ . This reflects the fact that here the security lies in the pseudorandomness of the ciphertext, rather than due to the indistinguishability of encryptions. (We stress that the LR-oracle *always* uses the same bit  $b$ .) A polynomial-time attacker is successful in this model if it succeeds in guessing the bit  $b$  with a non-negligible advantage. For chosen-plaintext security, this attacker is given access to the LR-oracle for the encryption scheme  $E$ . We now construct an adversary who carries out a chosen-plaintext attack on  $E$  and distinguishes encryptions to strings of the form  $E_{cca}(P_i, P_j, sid, ssid, r_{1-b})$  from uniformly chosen strings. This adversary emulates experiments (II) and (III) by running  $\mathcal{Z}$  and all the parties. However, when an honest party is supposed to generate  $C_{1-b}$ , the attacker hands the LR-oracle the query  $E_{cca}(P_i, P_j, sid, ssid, r_{1-b})$  and receives back  $C'$  which either equals  $E(E_{cca}(P_i, P_j, sid, ssid, r_{1-b}))$  or is uniformly distributed. The attacker then sets  $C_{1-b} = C'$ . This emulation can be carried out given the encryption-key  $E$  only (i.e., no decryption key

is required). This is the case because decryption is only needed for decommitment, and the only ciphertext to be decrypted upon decommitment is  $C_b$  (and not  $C_{1-b}$ ). Now, if  $b = 1$  for the LR-oracle, then the attacker perfectly emulates experiment (II). Furthermore, if  $b = 0$  then the attacker perfectly emulates experiment (III). Finally, as we have mentioned, the above emulation is carried out using a chosen-plaintext attack on  $E$  only. Therefore, if  $\mathcal{Z}$  can distinguish experiments (II) and (III), then the attacker can guess the bit  $b$  of the LR-oracle with non-negligible advantage. This is in contradiction to the CPA-security of  $E$ .

Finally, we consider the hybrid experiments (III) and (IV). The only difference between these experiments is that in experiment (III) the checks causing  $\mathcal{S}$  to output **failure** are not carried out. That is, if  $\mathcal{S}$  never outputs **failure**, then experiments (III) and (IV) are *identical*. This is due to the fact that if  $\mathcal{S}$  never outputs **failure**, then for every commitment  $c$  generated by the real-model adversary  $\mathcal{A}$ , there is at most one possible decommitment. There is therefore no need for  $\mathcal{S}$  to carry out these checks. We conclude that it suffices to show that  $\mathcal{S}$  outputs **failure** with at most negligible probability. In order to prove this, we again consider a sequence of hybrid experiments:

- (V) *Simulation with partially real encryptions*: This is the same as (IV), except that  $\mathcal{S}$  is given (say, by  $\mathcal{F}$ ) the true values of the inputs for all uncorrupted parties. Then, when generating simulated commitments for uncorrupted parties,  $\mathcal{S}$  replaces  $C_{1-b}$  with  $E(E_{cca}(P_i, P_j, \text{sid}, \text{ssid}, 0^{|r_{1-b}|}))$ , where  $b$  is the true input value.
- (VI) *Simulation with nearly real commitments*: This is the same as (V), except that in the simulated commitments generated for uncorrupted parties,  $\mathcal{S}$  computes  $z \leftarrow \text{aHC}(b)$  where  $b$  is the true value of the commitment (instead of always computing  $z \leftarrow \text{aHC}(0)$ ).

We now claim that the probability that  $\mathcal{S}$  outputs **failure** in experiment (IV) is negligibly close to the probability that it outputs **failure** in experiment (V). The only difference between these experiments relates to the encryption value of  $C_{1-b}$  (i.e., in (IV) we have that  $C_{1-b}$  contains the actual random coins  $r_{1-b}$ , whereas in (V) this is replaced by  $0^{|r_{1-b}|}$ ). The proof relies on the chosen-ciphertext security of the scheme  $E_{cca}$ . (Chosen ciphertext security is required because the emulation of experiments (IV) and (V) requires access to a decryption oracle: Recall that  $\mathcal{S}$  must decrypt ciphertexts in the simulation of commit-activations where the committer is corrupted.) Formally, we prove this claim using the “left-right” oracle formulation of security for encryption schemes. Recall that according to this definition, an attacker is given access to an LR-oracle that has a randomly chosen bit  $b$  internally hard-wired. The attacker can then query the oracle with pairs  $(a_0, a_1)$  and receives back  $E_{cca}(a_b)$ . When considering CCA2-security, the attacker is given access to the LR-oracle as well as a decryption oracle for  $E_{cca}$  which works on all ciphertexts except for those output by the LR-oracle.

We argue that if  $\mathcal{S}$  outputs **failure** with probabilities that are non-negligibly far apart in experiments (IV) and (V), then  $\mathcal{Z}$  together with  $\mathcal{A}$

can be used to construct a successful CCA2 attacker against  $E_{cca}$  in the LR-model. We now describe the attacker. The attacker receives the public key for  $E_{cca}$ . It then simulates experiment (IV) by playing  $\mathcal{Z}$ ,  $\mathcal{A}$  and  $\mathcal{S}$  as above, except for the following differences:

1. The public key for  $E_{cca}$  is given to  $\mathcal{S}$  externally and  $\mathcal{S}$  does not have the decryption key.
2. When generating a simulated commitment for an honest party  $P_i$ , the attacker computes  $z \leftarrow \text{aHC}(0)$  and decommitment strings  $r_0$  and  $r_1$  to 0 and 1, respectively. Furthermore, the attacker computes  $C_b \leftarrow E(E_{cca}(P_i, P_j, \text{sid}, \text{ssid}, r_b))$  as  $\mathcal{S}$  does. However, for  $C_{1-b}$ , the attacker queries the LR-oracle with the plaintexts  $(P_i, P_j, \text{sid}, \text{ssid}, r_{1-b})$  and  $(P_i, P_j, \text{sid}, \text{ssid}, 0^{|r_{1-b}|})$ . When the LR-oracle responds with a ciphertext  $C^{cca}$ , the attacker sets  $C_{1-b} \leftarrow E(C^{cca})$ .
3. When  $\mathcal{S}$  obtains a commitment  $(\text{sid}, \text{ssid}, P_j, z, C_0, C_1)$  from  $\mathcal{A}$  controlling a corrupted party  $P_i$ , then the attacker decrypts  $C_0$  and  $C_1$  using the decryption key for  $E$  and obtains  $C_0^{cca}$  and  $C_1^{cca}$ . There are two cases:
  - *Case 1* – a ciphertext  $C_b^{cca}$  came from a commitment previously generated for an honest party by  $\mathcal{S}$ : If this generated commitment was not from  $P_i$  to  $P_j$ , then  $C_b^{cca}$  cannot constitute a valid decommitment because the encryption does not contain the pair  $(P_i, P_j)$  in this order. Likewise, if the previous commitment was from  $P_i$  to  $P_j$  but the sub-session identifiers are different, then it still cannot be a valid decommitment. (Recall that  $P_j$  will ignore a second commitment from  $P_i$  with the same identifiers.) In the above cases, the attacker acts just as  $\mathcal{S}$  would for ciphertexts that do not decrypt to valid decommitment information. (Notice that the attacker does not need to use the decryption oracle in this case.)

One other possible case is that the previous commitment was from  $P_i$  to  $P_j$  and the sub-session identifiers are correct. This can only happen if  $P_i$  was corrupted after the commitment message containing  $C_b^{cca}$  was written by  $\mathcal{S}$  on  $P_i$ 's outgoing communication tape, but before  $\mathcal{A}$  delivered it. However, in this case, the attacker knows the decryption of the ciphertext without querying the decryption oracle, and therefore uses this value in the simulation.

- *Case 2* – a ciphertext  $C_b^{cca}$  was not previously generated by  $\mathcal{S}$ : Then, except with negligible probability, this ciphertext could not have been output by the LR-oracle. Therefore, the attacker can query its decryption oracle and obtain the corresponding plaintext. Given this plaintext, the attacker proceeds as  $\mathcal{S}$  does.

We note that the above simulation can be carried out without knowing the decryption key for  $E_{cca}$ . This is because the attacker knows the value  $b$  that an honest party is committing to before it needs to generate the simulated commitment value. Therefore, it never needs to decrypt  $C_{1-b}$ .

Analyzing the success probability of the attacker, we make the following observations. If the LR-oracle uses Left encryptions (i.e., it always outputs a ciphertext  $C^{cca}$  that is an encryption of  $(P_i, P_j, sid, ssid, r_{1-b})$ ), then the resulting simulation is negligibly close to experiment (IV). (The only difference is in the case that a ciphertext  $C_b^{cca}$  generated by  $\mathcal{A}$  coincides with a ciphertext output by the LR-oracle. However, this occurs with only negligible probability, otherwise  $E_{cca}$  does not provide correctness.) On the other hand, if the LR-oracle uses Right encryptions (i.e., it always outputs a ciphertext  $C^{cca}$  that is an encryption of  $(P_i, P_j, sid, ssid, 0^{r_{1-b}})$ ), then the resulting simulation is negligibly close to experiment (V). Therefore, by the CCA2-security of  $E_{cca}$ , the probability that  $\mathcal{Z}$  outputs 1 from experiment (IV) must be negligibly close to the probability that it outputs 1 in experiment (V). By having  $\mathcal{Z}$  output 1 if and only if  $\mathcal{S}$  outputs a failure symbol, we have that the probability that  $\mathcal{S}$  outputs failure in the two experiments is negligibly close.

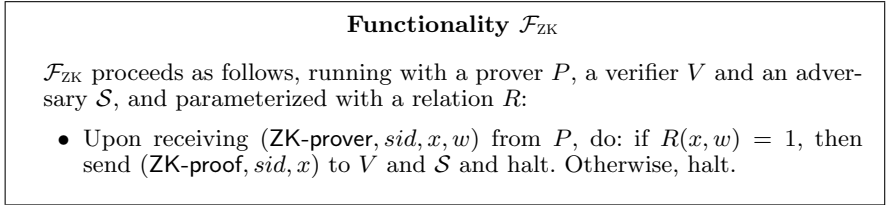
We now proceed to show that the probability that  $\mathcal{S}$  outputs failure in experiments (V) and (VI) is negligibly close. This follows from the indistinguishability of commitments  $\{\mathbf{aHC}(0)\}$  and  $\{\mathbf{aHC}(1)\}$ . (A standard hybrid argument is used to take into account the fact that many commitments are generated by  $\mathcal{S}$  during the simulation.) Here we use the fact that in both experiments (V) and (VI) the ciphertext  $C_{1-b}$  is independent from the rest of the commitment. That is, in these experiments  $C_{1-b}$  does not contain the random coins  $r_{1-b}$  which would constitute a decommitment to  $\mathbf{aHC}(1-b)$ . Now, for commitments to 0 (i.e., when  $b = 0$ ), experiments (V) and (IV) are identical. However, for commitments to 1 (i.e., when  $b = 1$ ), they are different. Nevertheless, it is hard to distinguish the case that  $z = \mathbf{aHC}(0)$  from the case that  $z = \mathbf{aHC}(1)$ , even given the random coins  $r_1$  that constitute a decommitment of  $z$  to 1.

Finally, to complete the proof, we show that in experiment (VI) the probability that  $\mathcal{S}$  outputs failure is negligible. The main observation here is that in experiment (VI),  $\mathcal{S}$  does not use knowledge of a Hamiltonian cycle in  $G$ . Now, if  $\mathcal{S}$  outputs failure when simulating commit activations for a corrupted party, then this means that it obtains a decommitment to 0 and to 1 for some commitment string  $z$ . However, by the construction of the commitment scheme, this means that  $\mathcal{S}$  obtains a Hamiltonian cycle (and equivalently a pre-image of  $y = f(x)$ ). Since  $\mathcal{S}$  can do this with only negligible probability we have that this event can also only occur with negligible probability. We conclude that  $\mathcal{S}$  outputs failure in experiment (VI), and therefore in experiment (IV), with only negligible probability. (Formally speaking, given  $\mathcal{S}$  we construct an inverter for  $f$  that proceeds as described above.) This completes the hybrid argument, demonstrating that  $\mathcal{Z}$  can distinguish experiments (I) and (IV) with only negligible probability. ■

## 4.6 Universally Composable Zero-Knowledge

We present and discuss the ideal zero-knowledge functionality  $\mathcal{F}_{\text{ZK}}$ . This functionality plays a central role in our general construction of protocols for realizing any two-party functionality. Specifically, our protocol for realizing the commit-and-prove functionality is constructed and analyzed in a hybrid model with access to  $\mathcal{F}_{\text{ZK}}$  (i.e., in the  $\mathcal{F}_{\text{ZK}}$ -hybrid model). Using the universal composition theorem, the construction can be composed with known protocols that UC realize  $\mathcal{F}_{\text{ZK}}$ , either in the  $\mathcal{F}_{\text{MCOM}}$ -hybrid model or directly in the common reference string (CRS) model, to obtain protocols for realizing any two-party functionality in the CRS model. (Actually, here we use universal composition with joint state. See more details below.)

In the zero-knowledge functionality, parameterized by a relation  $R$ , the prover sends the functionality a statement  $x$  to be proven along with a witness  $w$ . In response, the functionality forwards the statement  $x$  to the verifier if and only if  $R(x, w) = 1$  (i.e., if and only if it is a correct statement). Thus, in actuality, this is a proof of knowledge in that the verifier is assured that the prover actually *knows*  $w$  (and has explicitly sent  $w$  to the functionality), rather than just being assured that such a  $w$  exists. The zero-knowledge functionality,  $\mathcal{F}_{\text{ZK}}$ , is presented in Figure 4.6.



**Fig. 4.6.** The single-session  $\mathcal{F}_{\text{ZK}}$  functionality

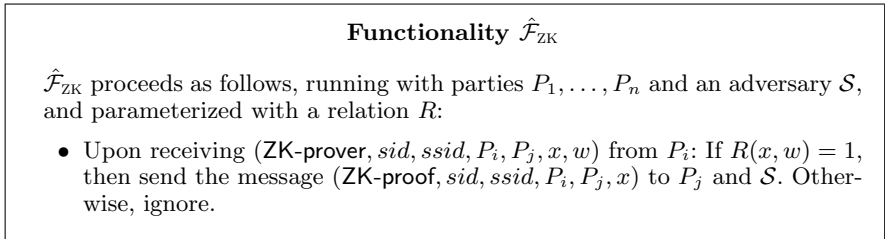
Let us highlight several technical issues that motivate the present formalization. First, notice that the functionality is parameterized by a single relation (and thus a different copy of  $\mathcal{F}_{\text{ZK}}$  is used for every different relation required). Nonetheless, the relation  $R$  may actually index any polynomial number of predetermined relations for which the prover may wish to prove statements. This can be implemented by separating the statement  $x$  into two parts:  $x_1$  that indexes the relation to be used and  $x_2$  that is the actual statement. Then, define  $R((x_1, x_2), w) \stackrel{\text{def}}{=} R_{x_1}(x_2, w)$ . (Note that in this case the set of relations to be indexed is fixed and publicly known.)<sup>16</sup>

<sup>16</sup> Another possibility is to parameterize  $\mathcal{F}_{\text{ZK}}$  by a polynomial  $q(\cdot)$ . Then,  $P_i$  sends the functionality a triple  $(x, w, C_R)$ , where  $C_R$  is a two-input binary circuit of size at most  $q(|x|)$ . (This circuit defines the relation being used.) The ideal functionality then sends  $P_j$  the circuit  $C_R$  and the bit  $C_R(x, w)$ . This approach has the advantage that the relations to be used need not be predetermined and fixed.

Second, the functionality is defined so that only correct statements (i.e., values  $x$  such that  $R(x, w) = 1$ ) are received by  $P_2$  in the prove phase. Incorrect statements are ignored by the functionality, and the receiver  $P_2$  receives no notification that an attempt at cheating in a proof took place. This convention simplifies the description and analysis of our protocols. We note, however, that this is not essential. Error messages can be added to the functionality (and realized) in a straightforward way. Third, we would like to maintain the (intuitively natural) property that a prover can always cause the verifier to reject, even if for *every*  $w$  it holds that  $R(x, w) = 1$  (e.g., take  $R = \{0, 1\}^* \times \{0, 1\}^*$ ). This technicality is solved by defining a special witness input symbol “ $\perp$ ” such that for every relation  $R$  and every  $x$ ,  $R(x, \perp) = 0$ .

Note that each copy of the functionality handles only a single proof (with a given prover and a given verifier). This is indeed convenient for protocols in the  $\mathcal{F}_{\text{ZK}}$ -hybrid model, since a new copy of  $\mathcal{F}_{\text{ZK}}$  can be invoked for each new proof (or, each “session”). However, directly realizing  $\mathcal{F}_{\text{ZK}}$  in the  $\mathcal{F}_{\text{CRS}}$ -hybrid model and using the universal composition theorem would result in an extremely inefficient composed protocol, where a new instance of the reference string is needed for each proof. Instead, we make use of universal composition with joint state, as follows. We start by defining functionality  $\hat{\mathcal{F}}_{\text{ZK}}$ , the multi-session extension of  $\mathcal{F}_{\text{ZK}}$ , and recall known protocols that UC realize  $\hat{\mathcal{F}}_{\text{ZK}}$  using a single short instance of the common string. We then use the JUC theorem (Theorem 4.3.4) to compose protocols in the  $\mathcal{F}_{\text{ZK}}$ -hybrid model with protocols that UC realize  $\hat{\mathcal{F}}_{\text{ZK}}$ .

The definition of  $\hat{\mathcal{F}}_{\text{ZK}}$ , the multi-session extension of  $\mathcal{F}_{\text{ZK}}$ , follows from the definition of  $\mathcal{F}_{\text{ZK}}$  and the general definition of multi-session extensions (see Section 4.3.2). Nonetheless, for the sake of clarity we explicitly present functionality  $\hat{\mathcal{F}}_{\text{ZK}}$  in Figure 4.7. An input to  $\hat{\mathcal{F}}_{\text{ZK}}$  is expected to contain two types of indices: the first one, *sid*, is the SID that differentiates messages to  $\hat{\mathcal{F}}_{\text{ZK}}$  from messages sent to other functionalities. The second index, *ssid*, is the sub-session ID and is unique per “sub-session” (i.e., per input message).



**Fig. 4.7.** The multi-session zero-knowledge functionality

In the case of static adversaries, there exists a protocol that UC realizes  $\hat{\mathcal{F}}_{\text{ZK}}$  for any NP relation, in the common reference string (CRS) model [DDOPS01]. The protocol of [DDOPS01] assumes the existence of trapdoor

one-way permutations. Furthermore, the protocol is “non-interactive”, in the sense that it consists of a single message from the prover to the verifier. In the case of adaptive adversaries, Canetti and Fischlin show a three-round protocol that UC realizes  $\hat{\mathcal{F}}_{\text{ZK}}$  in the  $\mathcal{F}_{\text{MCOM}}$ -hybrid model, where  $\mathcal{F}_{\text{MCOM}}$  is the multi-session universally composable commitment functionality (see Section 4.5 below). The protocol uses a single copy of  $\mathcal{F}_{\text{MCOM}}$ .<sup>17</sup>

## 4.7 The Commit-and-Prove Functionality $\mathcal{F}_{\text{CP}}$

In this section we define the “commit-and-prove” functionality,  $\mathcal{F}_{\text{CP}}$ , and present protocols for UC realizing it. As discussed in Section 4.2, this functionality, which is a generalization of the commitment functionality, is central for constructing the protocol compiler. As in the case of  $\mathcal{F}_{\text{ZK}}$ , the  $\mathcal{F}_{\text{CP}}$  functionality is parameterized by a relation  $R$ . The first stage is a commit phase in which the receiver obtains a commitment to some value  $w$ . The second phase is more general than plain decommitment. Rather than revealing the committed value, the functionality receives some value  $x$  from the committer, sends  $x$  to the receiver, and asserts whether  $R(x, w) = 1$ . To see that this is indeed a generalization of a commitment scheme, take  $R$  to be the identity relation and  $x = w$ . Then, following the prove phase, the receiver obtains  $w$  and is assured that this is the value that was indeed committed to in the commit phase.

In fact,  $\mathcal{F}_{\text{CP}}$  is even more general than the above description, in the following ways. First it allows the committer to commit to multiple secret values  $w_i$ , and then have the relation  $R$  depend on all these values in a single proof. (This extension is later needed for dealing with reactive protocols, where inputs may be received over time.) Second, the committer may ask to prove multiple statements with respect to the same set of secret values. These generalizations are dealt with as follows. When receiving a new (commit,  $\text{sid}, w$ ) request from the committer,  $\mathcal{F}_{\text{CP}}$  adds the current  $w$  to the already existing list  $\bar{w}$  of committed values. When receiving a (CP-prover,  $\text{sid}, x$ ) request,  $\mathcal{F}_{\text{CP}}$  evaluates  $R$  on  $x$  and the current list  $\bar{w}$ . Functionality  $\mathcal{F}_{\text{CP}}$  is presented in Figure 4.8.

As in the case of  $\mathcal{F}_{\text{ZK}}$ , the  $\mathcal{F}_{\text{CP}}$  functionality is defined so that only correct statements (i.e., values  $x$  such that  $R(x, w) = 1$ ) are received by  $V$  in the prove phase. Incorrect statements are ignored by the functionality, and the receiver  $V$  receives no notification that an attempt at cheating in a proof took place.

<sup>17</sup> Actually, the zero-knowledge functionality in [CF01] is only “single session” (and has some other technical differences from  $\hat{\mathcal{F}}_{\text{ZK}}$ ). Nonetheless, it is easy to see that by using  $\mathcal{F}_{\text{MCOM}}$  and having the prover first check that its input  $x$  and  $w$  is such that  $(x, w) \in R$ , their protocol UC realizes  $\hat{\mathcal{F}}_{\text{ZK}}$ .

**Functionality  $\mathcal{F}_{\text{CP}}$** 

$\mathcal{F}_{\text{CP}}$  proceeds as follows, running with a committer  $C$ , a receiver  $V$  and an adversary  $\mathcal{S}$ , and is parameterized by a value  $k$  and a relation  $R$ :

- **Commit Phase:** Upon receiving a message (`commit`,  $sid$ ,  $w$ ) from  $C$  where  $w \in \{0, 1\}^k$ , append the value  $w$  to the list  $\bar{w}$ , and send the message (`receipt`,  $sid$ ) to  $V$  and  $\mathcal{S}$ . (Initially, the list  $\bar{w}$  is empty.)
- **Prove Phase:** Upon receiving a message (`CP-prover`,  $sid$ ,  $x$ ) from  $C$ , where  $x \in \{0, 1\}^{\text{poly}(k)}$ , compute  $R(x, \bar{w})$ : If  $R(x, \bar{w}) = 1$ , then send  $V$  and  $\mathcal{S}$  the message (`CP-proof`,  $sid$ ,  $x$ ). Otherwise, ignore the message.

**Fig. 4.8.** The commit-and-prove functionality

#### 4.7.1 UC Realizing $\mathcal{F}_{\text{CP}}$ for Static Adversaries

We present protocols for UC realizing the  $\mathcal{F}_{\text{CP}}$  functionality in the  $\mathcal{F}_{\text{ZK}}$ -hybrid model, for both static and adaptive adversaries. We first concentrate on the case of static adversaries, since it is significantly simpler than the adaptive case, and therefore serves as a good warm-up.

The commit phase and the prove phase of the protocol each involve a single invocation of  $\mathcal{F}_{\text{ZK}}$ . (The relation used in each phase is different.) In the commit phase the committer commits to a value using a standard commitment scheme, and proves knowledge of the decommitment value through  $\mathcal{F}_{\text{ZK}}$ . Thus we obtain a “commit-with-knowledge” protocol, in which the simulator can extract the committed value.

Specifically, let  $\text{Com}$  be a perfectly binding commitment scheme, and denote by  $\text{Com}(w; r)$  a commitment to a string  $w$  using a random string  $r$ . For simplicity, we use a non-interactive commitment scheme. Such schemes exist assuming the existence of 1–1 one-way functions, see [G01]. (Alternatively, we could use the Naor scheme [N91] that can be based on *any* one-way function, rather than requiring 1–1 one-way functions. In this scheme, the receiver sends an initial message and then the committer commits. This changes the protocol and analysis only slightly. We note that in fact, the use of perfect binding is not essential and computational binding actually suffices, as will be the case in Section 4.7.2.) Loosely speaking, the protocol begins by the committer  $C$  sending  $c = \text{Com}(w; r)$  to  $V$ , and then proving knowledge of the pair  $(w, r)$ . In our terminology, this consists of  $C$  sending (`ZK-prover`,  $sid_C$ ,  $c = \text{Com}(w; r)$ ,  $(w, r)$ ) to  $\mathcal{F}_{\text{ZK}}$ , which is parameterized by the following relation  $R_C$ :

$$R_C = \{(c, (w, r)) \mid c = \text{Com}(w; r)\} \quad (4.3)$$

That is,  $R_C$  is the relation of pairs of commitments with their decommitment information. In addition, the committer  $C$  keeps the list  $\bar{w}$  of all the values  $w$  committed to. It also keeps the lists  $\bar{r}$  and  $\bar{c}$  of the corresponding random values and commitment values.



When the receiver  $V$  receives  $(\text{ZK-proof}, \text{sid}_C, c)$  from  $\mathcal{F}_{\text{ZK}}$ , it accepts  $c$  as the commitment string and adds  $c$  to its list  $\bar{c}$  of accepted commitments. (Note that in the  $\mathcal{F}_{\text{ZK}}$ -hybrid model,  $V$  is guaranteed that  $C$  “knows” the decommitment, in the sense that  $C$  explicitly sent the decommitment value to  $\mathcal{F}_{\text{ZK}}$ .)

The prove phase of the protocol also involves invoking  $\mathcal{F}_{\text{ZK}}$  where the relation  $R_P$  parameterizing the  $\mathcal{F}_{\text{ZK}}$  functionality is defined as follows. Let  $R$  be the relation parameterizing  $\mathcal{F}_{\text{CP}}$ . Then,  $R_P$  is defined by:

$$R_P \stackrel{\text{def}}{=} \{((x, \bar{c}), (\bar{w}, \bar{r})) \mid \forall i, c_i = \text{Com}(w_i; r_i) \ \& \ R(x, \bar{w}) = 1\} \quad (4.4)$$

That is,  $R_P$  confirms that  $\bar{c}$  is the vector of commitments to  $\bar{w}$ , and that  $R(x, \bar{w}) = 1$ . Thus, the prove phase consists of the committer proving some NP-statement regarding the values committed to previously. (The value  $x$  is the NP-statement and the values committed to, plus the randomness used, comprise the “witness” for  $x$ ). Upon receiving the message  $(\text{ZK-proof}, \text{sid}_P, (x, \bar{c}))$  from  $\mathcal{F}_{\text{ZK}}$ , the receiver accepts if  $\bar{c}$  equals the list of commitments that it had previously received. (The receiver must check  $\bar{c}$  because this is what ensures that the witness being used is indeed the list of values previously committed to, and nothing else.) Finally, note that if  $R \in \mathcal{NP}$ , then so too is  $R_P$ .

We denote by  $\mathcal{F}_{\text{ZK}}^{\text{C}}$  and  $\mathcal{F}_{\text{ZK}}^{\text{P}}$  the copies of  $\mathcal{F}_{\text{ZK}}$  from the commit phase and prove phase respectively (i.e.,  $\mathcal{F}_{\text{ZK}}^{\text{C}}$  is parameterized by  $R_C$  and  $\mathcal{F}_{\text{ZK}}^{\text{P}}$  is parameterized by  $R_P$ ). Formally, the two copies of  $\mathcal{F}_{\text{ZK}}$  are differentiated by using session identifiers  $\text{sid}_C$  and  $\text{sid}_P$ , respectively. (E.g., one can define  $\text{sid}_C = \text{sid} \circ \text{C}$  and  $\text{sid}_P = \text{sid} \circ \text{P}$ , where  $\text{sid}$  is the session identifier of the protocol for realizing  $\mathcal{F}_{\text{CP}}$  and “ $\circ$ ” denotes concatenation.) The protocol, using a perfectly binding non-interactive commitment scheme  $\text{Com}$ , is presented in Figure 4.9.

**Proposition 4.7.1** *Assuming that  $\text{Com}$  is a secure (perfectly binding) commitment scheme,<sup>18</sup> Protocol SCP of Figure 4.9 UC realizes  $\mathcal{F}_{\text{CP}}$  in the  $\mathcal{F}_{\text{ZK}}$ -hybrid model, for static adversaries.*

**Proof.** Let  $\mathcal{A}$  be a static adversary who operates against Protocol SCP in the  $\mathcal{F}_{\text{ZK}}$ -hybrid model. We construct an ideal-process adversary (or simulator)  $\mathcal{S}$  such that no environment  $\mathcal{Z}$  can tell with non-negligible probability whether it is interacting with  $\mathcal{A}$  and parties running Protocol SCP in the  $\mathcal{F}_{\text{ZK}}$ -hybrid model or with  $\mathcal{S}$  in the ideal process for  $\mathcal{F}_{\text{CP}}$ . As usual,  $\mathcal{S}$  will run a simulated copy of  $\mathcal{A}$  and will use  $\mathcal{A}$  in order to interact with  $\mathcal{Z}$  and  $\mathcal{F}_{\text{CP}}$ . For this purpose,  $\mathcal{S}$  will “simulate for  $\mathcal{A}$ ” an interaction with parties running Protocol SCP, where the interaction will match the inputs and outputs seen by  $\mathcal{Z}$  in its interaction with  $\mathcal{S}$  in the ideal process for  $\mathcal{F}_{\text{CP}}$ . We use the term

<sup>18</sup> When we refer to “secure perfectly binding commitments” here, we mean secure according to standard definitions (see [G01, Section 4.4.1] for a formal definition).

**Protocol SCP**

- **Auxiliary Input:** A security parameter  $k$ .
- **Commit phase:**
  1. On input  $(\text{commit}, \text{sid}, w)$ , where  $w \in \{0, 1\}^k$ ,  $C$  chooses a random string  $r$  of length sufficient for committing to  $w$  in scheme  $Com$ , and sends  $(\text{ZK-prover}, \text{sid}_C, Com(w; r), (w, r))$  to  $\mathcal{F}_{\text{ZK}}^C$ , where  $\mathcal{F}_{\text{ZK}}^C$  is parameterized by the relation  $R_C$  defined in Eq. (4.3). In addition,  $C$  stores in a vector  $\bar{w}$  the list of all the values  $w$  that it has sent to  $\mathcal{F}_{\text{ZK}}^C$ , and in vectors  $\bar{r}$  and  $\bar{c}$  the corresponding lists of random strings and commitment values.
  2. When receiving  $(\text{ZK-proof}, \text{sid}_C, c)$  from  $\mathcal{F}_{\text{ZK}}^C$ , the receiver  $V$  outputs  $(\text{receipt}, \text{sid})$ , and adds  $c$  to its list of commitments  $\bar{c}$ . (Initially,  $\bar{c}$  is empty.)
- **Prove phase:**
  1. On input  $(\text{CP-prover}, \text{sid}, x)$ ,  $C$  sends  $(\text{ZK-prover}, \text{sid}_P, (x, \bar{c}), (\bar{w}, \bar{r}))$  to  $\mathcal{F}_{\text{ZK}}^P$ , where  $\bar{w}, \bar{r}, \bar{c}$  are the above-define vectors and  $\mathcal{F}_{\text{ZK}}^P$  is parameterized by the relation  $R_P$  defined in Eq. (4.4).
  2. When receiving  $(\text{ZK-proof}, \text{sid}_P, (x, \bar{c}))$  from  $\mathcal{F}_{\text{ZK}}^P$ ,  $V$  proceeds as follows. If its list of commitments equals  $\bar{c}$ , then it outputs  $(\text{CP-proof}, \text{sid}, x)$ . Otherwise, it ignores the message.

**Fig. 4.9.** A protocol for realizing  $\mathcal{F}_{\text{CP}}$  for static adversaries

*external communication* to refer to  $\mathcal{S}$ 's communication with  $\mathcal{Z}$  and  $\mathcal{F}_{\text{CP}}$ . We use the term *internal communication* to refer to  $\mathcal{S}$ 's communication with the simulated  $\mathcal{A}$ .

Recall that  $\mathcal{A}$  is a static adversary and therefore the choice of which parties are under its control (i.e., corrupted) is predetermined. When describing  $\mathcal{S}$ , it suffices to describe its reaction to any one of the possible external activations (inputs from  $\mathcal{Z}$  and messages from  $\mathcal{F}_{\text{CP}}$ ) and any one of the possible outputs or outgoing messages generated internally by  $\mathcal{A}$ . This is done below. For clarity, we group these activities according to whether or not the committing party  $C$  is corrupted. Simulator  $\mathcal{S}$  proceeds as follows:

**Simulating the communication with the environment:** Every input value coming from  $\mathcal{Z}$  (in the external communication) is forwarded to the simulated  $\mathcal{A}$  (in the internal communication) as if coming from  $\mathcal{A}$ 's environment. Similarly, every output value written by  $\mathcal{A}$  on its output tape is copied to  $\mathcal{S}$ 's own output tape (to be read by the external  $\mathcal{Z}$ ).

**Simulating the case that the committer is uncorrupted:** In this case,  $\mathcal{A}$  expects to see the messages sent by  $\mathcal{F}_{\text{ZK}}^C$  and  $\mathcal{F}_{\text{ZK}}^P$  to  $V$ . (Notice that the only messages sent in the protocol are to and from  $\mathcal{F}_{\text{ZK}}^C$  and  $\mathcal{F}_{\text{ZK}}^P$ ; therefore, the only messages seen by  $\mathcal{A}$  are those sent by these functionalities. This holds regardless of whether the receiver  $V$  is corrupted or not.) In the ideal process,  $\mathcal{S}$  receives the  $(\text{receipt}, \dots)$  and  $(\text{CP-proof}, \dots)$

messages that  $V$  receives from  $\mathcal{F}_{\text{CP}}$ . It constructs the appropriate  $\mathcal{F}_{\text{ZK}}^{\text{C}}$  messages given the **receipt** messages from  $\mathcal{F}_{\text{CP}}$ , and the appropriate  $\mathcal{F}_{\text{ZK}}^{\text{P}}$  messages given the **CP-proof** messages from  $\mathcal{F}_{\text{CP}}$ . This is done as follows:

- Whenever  $\mathcal{S}$  receives a message (**receipt**,  $sid$ ) from  $\mathcal{F}_{\text{CP}}$  where  $C$  is uncorrupted,  $\mathcal{S}$  computes  $c = \text{Com}(0^k; r)$  for a random  $r$  and (internally) passes  $\mathcal{A}$  the message (**ZK-proof**,  $sid_C, c$ ), as  $\mathcal{A}$  would receive from  $\mathcal{F}_{\text{ZK}}^{\text{C}}$  in a real protocol execution. Furthermore,  $\mathcal{S}$  adds the value  $c$  to its list of simulated-commitment values  $\bar{c}$ . (It is stressed that the commitment here is to an unrelated value, however by the hiding property of commitments and the fact that all commitments are of length  $k$ , this is indistinguishable from a real execution.)
- Whenever  $\mathcal{S}$  receives a message (**CP-proof**,  $sid, x$ ) from  $\mathcal{F}_{\text{CP}}$  where  $C$  is uncorrupted,  $\mathcal{S}$  internally passes  $\mathcal{A}$  the message (**ZK-proof**,  $sid_P, (x, \bar{c})$ ), as  $\mathcal{A}$  would receive from  $\mathcal{F}_{\text{ZK}}^{\text{P}}$  in a protocol execution, where  $\bar{c}$  is the current list of commitment values generated in the simulation of the commit phase.

**Simulating the case that the committer is corrupted:** Here,  $\mathcal{A}$  controls  $C$  and generates the messages that  $C$  sends during an execution of Protocol SCP.<sup>19</sup> Intuitively, in this case  $\mathcal{S}$  must be able to extract the decommitment value  $w$  from  $\mathcal{A}$  during the commit phase of the protocol simulation. This is because, in the ideal process,  $\mathcal{S}$  must explicitly send the value  $w$  to  $\mathcal{F}_{\text{CP}}$  (and must therefore know the value being committed to). Fortunately, this extraction is easy for  $\mathcal{S}$  to do because  $\mathcal{A}$  works in the  $\mathcal{F}_{\text{ZK}}$ -hybrid model, and any message sent by  $\mathcal{A}$  to  $\mathcal{F}_{\text{ZK}}$  is seen by  $\mathcal{S}$  during the simulation. In particular,  $\mathcal{S}$  obtains the **ZK-proof** message sent by  $\mathcal{A}$  to  $\mathcal{F}_{\text{ZK}}^{\text{C}}$ , and this message is valid only if it explicitly contains the decommitment. The simulation is carried out as follows:

- Whenever the simulated  $\mathcal{A}$  internally delivers a message of the form (**ZK-prover**,  $sid_C, c, (w, r)$ ) from a corrupted  $C$  to  $\mathcal{F}_{\text{ZK}}^{\text{C}}$ , simulator  $\mathcal{S}$  checks that  $c = \text{Com}(w; r)$ . If this holds, then  $\mathcal{S}$  externally sends (**commit**,  $sid, w$ ) to  $\mathcal{F}_{\text{CP}}$  and internally passes (**ZK-proof**,  $sid_C, c$ ) to  $\mathcal{A}$  (as if coming from  $\mathcal{F}_{\text{ZK}}^{\text{C}}$ ). Furthermore,  $\mathcal{S}$  adds  $c$  to its list of received commitments  $\bar{c}$ . Otherwise,  $\mathcal{S}$  ignores the message.
- Whenever the simulated  $\mathcal{A}$  internally generates a message of the form (**ZK-prover**,  $sid_P, (x, \bar{c}), (\bar{w}, \bar{r})$ ) going from  $C$  to  $\mathcal{F}_{\text{ZK}}^{\text{P}}$ , simulator  $\mathcal{S}$  acts as follows. First,  $\mathcal{S}$  checks that  $\bar{c}$  equals its list of commitments and that  $((x, \bar{c}), (\bar{w}, \bar{r})) \in R_P$ . If yes, then  $\mathcal{S}$  internally passes (**ZK-proof**,  $sid_P, (x, \bar{c})$ ) to  $\mathcal{A}$  (as if received from  $\mathcal{F}_{\text{ZK}}^{\text{P}}$ ) and externally sends the prover message (**CP-prover**,  $sid, x$ ) to  $\mathcal{F}_{\text{CP}}$ . If no, then  $\mathcal{S}$  does nothing.

<sup>19</sup> We assume without loss of generality that the receiver  $V$  is uncorrupted, since carrying out an interaction where both participants are corrupted bears no effect on the view of  $\mathcal{Z}$ .

**Message delivery:** It remains to describe when (if at all)  $\mathcal{S}$  delivers the messages between the dummy parties  $C$  and  $V$ , and the functionality  $\mathcal{F}_{\text{CP}}$ . Simulator  $\mathcal{S}$  delivers **commit** and **CP-prover** messages from  $C$  to  $\mathcal{F}_{\text{CP}}$  when  $\mathcal{A}$  delivers the corresponding **ZK-prover** messages from  $C$  to  $\mathcal{F}_{\text{ZK}}^{\text{C}}$  and  $\mathcal{F}_{\text{ZK}}^{\text{P}}$  in the internal simulation. Likewise,  $\mathcal{S}$  delivers the **receipt** and **CP-proof** messages from  $\mathcal{F}_{\text{CP}}$  to  $V$  in the ideal process, when  $\mathcal{A}$  delivers the corresponding **ZK-proof** messages from  $\mathcal{F}_{\text{ZK}}^{\text{C}}$  and  $\mathcal{F}_{\text{ZK}}^{\text{P}}$  to  $V$  in the simulation.

We show that for every environment  $\mathcal{Z}$  it holds that:

$$\left\{ \text{IDEAL}_{\mathcal{F}_{\text{CP}}, \mathcal{S}, \mathcal{Z}}(k, z) \right\} \stackrel{\text{c}}{=} \left\{ \text{HYBRID}_{\text{SCP}, \mathcal{A}, \mathcal{Z}}^{\mathcal{F}_{\text{ZK}}} (k, z) \right\} \quad (4.5)$$

We first assert the following claim regarding the case where the committer is corrupted: the receiver  $V$  accepts a proof in the protocol execution if and only if in the ideal model simulation,  $V$  receives (**CP-proof**,  $\text{sid}, x$ ) from  $\mathcal{F}_{\text{CP}}$ . This can be seen as follows. First, note that if  $\mathcal{A}$  (controlling  $C$ ) sends a **ZK-prover** message containing a different vector of commitments to that sent in previous **commit** activations, then  $\mathcal{S}$  does not send any **CP-prover** message to  $\mathcal{F}_{\text{CP}}$ . Likewise, in such a case,  $V$  ignores the **ZK-proof** message. Simulator  $\mathcal{S}$  also checks that  $((x, \bar{c}), (\bar{w}, \bar{r})) \in R_P$  before sending any **CP-prover** message to  $\mathcal{F}_{\text{CP}}$ . Thus, if this does not hold, no **CP-proof** message is received by  $V$ . Likewise, in a protocol execution, if  $((x, \bar{c}), (\bar{w}, \bar{r})) \notin R_P$ , then  $V$  receives no **CP-proof** message. Finally, we note that by the (perfect) binding property of the commitment scheme, if  $\mathcal{A}$  tries to use a different vector of witnesses than that committed to in the commit phase, then this is detected by  $V$  and  $\mathcal{S}$ , and the message is ignored. (By the perfect binding of the commitment scheme, the vector  $\bar{c}$  defines a unique witness vector  $\bar{w}$  that can be used.) We conclude that when  $\mathcal{S}$  sends a **CP-prover** message to  $\mathcal{F}_{\text{CP}}$  the following holds:  $R(x, \bar{w}) = 1$  if and only if  $R_P((x, \bar{c}), (\bar{w}, \bar{r})) = 1$ , where  $\bar{c}$  is the vector of commitments sent by the corrupted committer. Thus,  $V$  outputs (**CP-proof**,  $\text{sid}, x$ ) in a protocol execution if and only if  $\mathcal{F}_{\text{CP}}$  sends (**CP-proof**,  $\text{sid}, x$ ) to  $V$  in the ideal model simulation.

We proceed to demonstrate Eq. (4.5). Since  $\mathcal{S}$  obtains the messages sent by  $\mathcal{A}$  to both the  $\mathcal{F}_{\text{ZK}}^{\text{C}}$  and  $\mathcal{F}_{\text{ZK}}^{\text{P}}$  functionalities, most of the simulation is perfect and the messages seen by  $\mathcal{A}$  are exactly the same as it would see in a hybrid execution of Protocol SCP. There is, however, one case where the simulation *is* different from a real execution. When the committer is uncorrupted,  $\mathcal{S}$  receives a (**receipt**,  $\text{sid}$ ) message from  $\mathcal{F}_{\text{CP}}$  and must generate the message that  $\mathcal{A}$  would see from  $\mathcal{F}_{\text{ZK}}^{\text{C}}$  in the protocol. Specifically,  $\mathcal{S}$  sends (**ZK-proof**,  $\text{sid}, c$ ) to  $\mathcal{A}$ , where  $c = \text{Com}(0^k; r)$ . That is,  $\mathcal{S}$  passes  $\mathcal{A}$  a commitment to a value that is unrelated to  $C$ 's input. In contrast, in a real execution of Protocol SCP, the value  $c$  seen by  $\mathcal{A}$  is  $c = \text{Com}(w; r)$ , where  $w$  is  $C$ 's actual input. Intuitively, by the hiding property of the commitment scheme  $\text{Com}$ , these two cases are indistinguishable. Formally, assume that there exists an adversary

$\mathcal{A}$ , an environment  $\mathcal{Z}$  and an input  $z$  to  $\mathcal{Z}$ , such that the IDEAL and HYBRID distributions can be distinguished. Then, we construct a distinguisher  $D$  for the commitment scheme  $\text{Com}$ . That is, the distinguisher  $D$  chooses some challenge  $w$ , receives a commitment  $c$  that is either to  $0^k$  or to  $w$ , and can tell with non-negligible probability which is the case.

Distinguisher  $D$  invokes the environment  $\mathcal{Z}$ , the party  $C$  and the simulator  $\mathcal{S}$  (which runs  $\mathcal{A}$  internally) on the following simulated interaction. First, a number  $i$  is chosen at random in  $\{1, \dots, t\}$ , where  $t$  is a bound on the running time of  $\mathcal{Z}$ . Then, for the first  $i - 1$  commitments  $c$  generated by  $\mathcal{S}$ , distinguisher  $D$  sets  $c = \text{Com}(0^k; r)$ . When  $\mathcal{S}$  is about to generate the  $i^{\text{th}}$  commitment,  $D$  declares the corresponding value  $w$  to be the challenge value, and obtains a test value  $c^*$ . (This  $w$  is the value that the simulated  $\mathcal{Z}$  hands the uncorrupted committer  $C$ .) Then,  $\mathcal{S}$  uses  $c^*$  as the commitment value for the  $i^{\text{th}}$  commitment. The rest of the commitments in the simulation are generated as normal commitments to the corresponding input values provided by  $\mathcal{Z}$ . When  $\mathcal{Z}$  halts with an output value,  $D$  outputs whatever  $\mathcal{Z}$  outputs and halts.

Analysis of the success probability of  $D$  is done via a standard hybrid argument and is omitted. We obtain that  $D$  succeeds in breaking the commitment with advantage  $p/t$ , where  $p$  is the advantage in which  $\mathcal{Z}$  distinguishes between an interaction in the hybrid model and an interaction in the ideal process (and  $t$  is the bound on  $\mathcal{Z}$ 's running time). ■

**On sufficient assumptions for realizing  $\mathcal{F}_{\text{CP}}$ :** For simplicity, Protocol SCP uses a non-interactive commitment scheme, which can be based on 1–1 one-way functions. However, as we have mentioned, the commit-phase of Protocol SCP can be modified to use Naor's commitment scheme [N91] (which in turn can use *any* one-way function). In this case,  $V$  begins by sending the receiver message of the [N91] scheme, and then  $C$  sends the commit message, using  $\mathcal{F}_{\text{ZK}}^{\text{C}}$  as in Protocol SCP. Thus, we have that  $\mathcal{F}_{\text{CP}}$  can be UC realized in the  $\mathcal{F}_{\text{ZK}}$ -hybrid model, assuming the existence of *any* one-way function.

#### 4.7.2 UC Realizing $\mathcal{F}_{\text{CP}}$ for Adaptive Adversaries

We now present a protocol for UC realizing functionality  $\mathcal{F}_{\text{CP}}$  in the  $\mathcal{F}_{\text{ZK}}$ -hybrid model, in the presence of adaptive adversaries. The difference between this protocol and Protocol SCP for static adversaries is in the properties of the underlying commitment scheme  $\text{Com}$  in use. Essentially, here we use a commitment scheme that is “adaptively secure”. That is, a simulator (having access to some trapdoor information) can generate “dummy commitments” that can later be opened in several ways.<sup>20</sup> In order to achieve this, the

<sup>20</sup> The property actually required is that the simulator can generate a “commitment”  $c$  such that given any  $w$  at a later stage, it can find randomness  $r_w$  such that  $c = \text{Com}(w; r_w)$ . This is needed for the adaptive corruption of the committing party. See Section 4.5 for more discussion.

commit phase of the protocol will now involve *two* invocations of  $\mathcal{F}_{\text{zk}}$ . As in the case of Protocol SCP, the relations used by the invocations of  $\mathcal{F}_{\text{zk}}$  in the commit phase are different from the relation used in the prove phase. Thus, for sake of clarity we use three different copies of  $\mathcal{F}_{\text{zk}}$ , two for the commit messages and one for the prove messages.

The specific commitment scheme  $C$  used in the commit phase here is the aHC commitment that lies at the core of the universally composable commitment scheme of Section 4.5. Recall that this scheme uses a common reference string containing an image  $y$  of a one-way function  $f$ . However, here we work in the  $\mathcal{F}_{\text{zk}}$ -hybrid model and do not have access to a common reference string. Thus, the common reference string is “replaced” by interaction via the  $\mathcal{F}_{\text{zk}}$  functionality. That is, the basic commitment scheme is as follows. The receiver  $V$  chooses a random value  $t$  and sends the committer  $C$  the value  $s = f(t)$ . Next,  $C$  uses  $s$  to commit to its input value, as defined in the aHC commitment scheme. That is,  $C$  first obtains a Hamiltonian graph  $G$  such that finding a Hamiltonian cycle in  $G$  is equivalent to computing the preimage  $t$  of  $s$ . (This is obtained by reducing the NP-language  $\{s \mid \exists t \text{ s.t. } s = f(t)\}$  to Hamiltonicity.) Then, in order to commit to 0,  $C$  chooses a random permutation  $\pi$  of the nodes of  $G$  and commits to the edges of the permuted graph one-by-one, using a non-interactive commitment scheme  $Com$  with pseudorandom commitments. (Such a scheme can be obtained using one-way permutations, see Section 4.5.) On the other hand, in order to commit to 1,  $C$  chooses a randomly labeled cycle over the same number of nodes as in  $G$ . Then,  $C$  uses  $Com$  to commit to these entries and produces random values for the rest of the adjacency matrix. As was shown in Section 4.5, this commitment scheme is both hiding and binding, and also has the property that given a preimage  $t$  of  $s$ , it is possible to generate a “dummy commitment” that can be later explained as a commitment to both 0 and 1. (Thus,  $t$  is essentially a trapdoor.) We denote a commitment of this type by  $\text{aHC}_s(w; r)$ , where  $s$  is the image of the one-way function being used,  $w$  is the value being committed to, and  $r$  is the randomness used in generating the commitment.

The commitment scheme aHC as described above requires that the underlying one-way function be a permutation. However, by using interaction, we can implement  $Com$  using the commitment scheme of Naor [N91] (this scheme also has pseudorandom commitments). We thus obtain that aHC can be implemented using any one-way function. For simplicity, the protocol is written for  $Com$  that is non-interactive (and therefore assumes one-way permutations). However, it is not difficult to modify it so that the [N91] scheme can be used instead.

As in the case of Protocol SCP, the first stage of the adaptive protocol (denoted ACP for adaptive commit-and-prove) involves carrying out the commit phase of the above-described commitment scheme via invocations of  $\mathcal{F}_{\text{zk}}$ . This use of  $\mathcal{F}_{\text{zk}}$  enables the simulator to extract the committed value from the committing party. In addition, here  $\mathcal{F}_{\text{zk}}$  is also used to enable the

simulator to obtain the trapdoor information needed for carrying out the adaptive simulation. Thus the protocol begins by the receiver first choosing a random string  $t$  and computing  $s = f(t)$ . Next, it sends  $s$  to the committer and, in addition, proves that it knows the preimage  $t$ . Loosely speaking, in the  $\mathcal{F}_{\text{ZK}}$ -hybrid model, this involves sending a (ZK-prover,  $\text{sid}, s, t$ ) message to  $\mathcal{F}_{\text{ZK}}$  and having the functionality send  $C$  the message (ZK-proof,  $\text{sid}, s$ ) if  $s = f(t)$ . We note that this step is carried out only once, even if many values are later committed to. Thus, the same  $s$  is used for many commitments.

Let  $\mathcal{F}_{\text{ZK}}^{\text{T}}$  denote the copy of  $\mathcal{F}_{\text{ZK}}$  used for proving knowledge of the trapdoor/preimage. Then,  $\mathcal{F}_{\text{ZK}}^{\text{T}}$  is parameterized by the relation  $R_T$  defined as follows:

$$R_T \stackrel{\text{def}}{=} \{(s, t) \mid s = f(t)\} \quad (4.6)$$

$\mathcal{F}_{\text{ZK}}$  is used twice more; once more in the commit phase and once in the prove phase. These copies of  $\mathcal{F}_{\text{ZK}}$  are denoted  $\mathcal{F}_{\text{ZK}}^{\text{C}}$  and  $\mathcal{F}_{\text{ZK}}^{\text{P}}$ , respectively. The uses of  $\mathcal{F}_{\text{ZK}}$  here are very similar to the static case (Protocol SCP). We therefore proceed directly to defining the relations  $R_C$  and  $R_P$  that parameterize  $\mathcal{F}_{\text{ZK}}^{\text{C}}$  and  $\mathcal{F}_{\text{ZK}}^{\text{P}}$ , respectively:

$$R_C \stackrel{\text{def}}{=} \{((s, c), (w, r)) \mid c = \text{aHC}_s(w; r)\} \quad (4.7)$$

$$R_P \stackrel{\text{def}}{=} \{((x, s, \bar{c}), (\bar{w}, \bar{r})) \mid \forall i, c_i = \text{aHC}_s(w_i; r_i) \ \& \ R(x, \bar{w}) = 1\} \quad (4.8)$$

The only difference between the definition of  $R_C$  and  $R_P$  here and in the static case is that here the value  $s$  is included as well. This is because a *pair*  $(s, c)$  binds the sender to a single value  $w$ , whereas  $c$  by itself does not. The protocol for the adaptive case is presented in Figure 4.10. (As in the static case, we formally differentiate the copies of  $\mathcal{F}_{\text{ZK}}$  by session identifiers  $\text{sid}_T$ ,  $\text{sid}_C$  and  $\text{sid}_P$ .)

**Proposition 4.7.2** *Assuming the existence of one-way functions, Protocol ACP of Figure 4.10 UC realizes  $\mathcal{F}_{\text{CP}}$  in the  $\mathcal{F}_{\text{ZK}}$ -hybrid model, in the presence of adaptive adversaries.*

**Proof (sketch).** The proof of the above proposition follows similar lines to the proof of Proposition 4.7.1. However, here the adversary  $\mathcal{A}$  can adaptively corrupt parties. Therefore, the simulator  $\mathcal{S}$  must deal with instructions from  $\mathcal{A}$  to corrupt parties during the simulation. When given such a “corrupt” command,  $\mathcal{S}$  corrupts the ideal model party and receives its input (and possibly its output). Then, given these values,  $\mathcal{S}$  must provide  $\mathcal{A}$  with random coins such that the simulated transcript generated so far is consistent with this revealed input and output. (An additional “complication” here is that the binding property of the underlying commitment scheme aHC is only computational. Thus, the validity of the simulation will be demonstrated by a reduction to the binding property of aHC.)

More precisely, let  $\mathcal{A}$  be an adaptive adversary who operates against Protocol ACP in the  $\mathcal{F}_{\text{ZK}}$ -hybrid model. We construct a simulator  $\mathcal{S}$  such that

**Protocol ACP**

- **Auxiliary Input:** A security parameter  $k$ , and a session identifier  $sid$ .
- **Initialization phase:**  
 The first time that the committer  $C$  wishes to commit to a value using the identifier  $sid$ , parties  $C$  and  $V$  execute the following before proceeding to the commit phase:
  1.  $C$  sends  $sid$  to  $V$  to indicate that it wishes to initiate a commit activation.
  2. Upon receiving  $sid$  from  $C$ , the receiver  $V$  chooses  $t \in_R \{0, 1\}^k$ , computes  $s = f(t)$  (where  $f$  is a one-way function), and sends  $(\text{ZK-prover}, sid_T, s, t)$  to  $\mathcal{F}_{\text{ZK}}^T$ , where  $\mathcal{F}_{\text{ZK}}^T$  is parameterized by the relation  $R_T$  defined in Eq. (4.6).  $V$  records the value  $s$ .
  3. Upon receiving  $(\text{ZK-proof}, sid_T, s)$  from  $\mathcal{F}_{\text{ZK}}^T$ ,  $C$  records the value  $s$ .
- **Commit phase:**
  1. On input  $(\text{commit}, sid, w)$  (where  $w \in \{0, 1\}^k$ ),  $C$  computes  $c = \text{aHC}_s(w; r)$  for a random  $r$  and using the  $s$  it received in the initialization phase.  $C$  then sends  $(\text{ZK-prover}, sid_C, (s, c), (w, r))$  to  $\mathcal{F}_{\text{ZK}}^C$ , where  $\mathcal{F}_{\text{ZK}}^C$  is parameterized by the relation  $R_C$  defined in Eq. (4.7). In addition,  $C$  stores in a vector  $\bar{w}$  the list of all the values  $w$  that were sent, and in vectors  $\bar{r}$  and  $\bar{c}$  the corresponding lists of random strings and commitment values.
  2. Upon receiving  $(\text{ZK-proof}, sid_C, (s', c))$  from  $\mathcal{F}_{\text{ZK}}^C$ ,  $V$  verifies that  $s'$  equals the string  $s$  that it sent in the initialization phase, outputs  $(\text{receipt}, sid)$  and adds the value  $c$  to its list  $\bar{c}$ . (Initially,  $\bar{c}$  is empty.) If  $s' \neq s$ , then  $V$  ignores the message.
- **Prove phase:**
  1. On input  $(\text{CP-prover}, sid, x)$ , the committer/prover  $C$  sends  $(\text{ZK-prover}, sid_P, (x, s, \bar{c}), (\bar{w}, \bar{r}))$  to  $\mathcal{F}_{\text{ZK}}^P$ , where  $\bar{c}$ ,  $\bar{w}$  and  $\bar{r}$  are the vectors described above, and  $\mathcal{F}_{\text{ZK}}^P$  is parameterized by the relation  $R_P$  defined in Eq. (4.8).
  2. Upon receiving  $(\text{ZK-proof}, sid, (x, s', \bar{c}))$  from  $\mathcal{F}_{\text{ZK}}^P$ ,  $V$  verifies that  $s'$  is the same string  $s$  that it sent in the initialization phase, and that its list of commitments equals  $\bar{c}$ . If so, then it outputs  $(\text{CP-proof}, sid, x)$ . Otherwise, it ignores the message.

**Fig. 4.10.** A protocol for realizing  $\mathcal{F}_{\text{CP}}$  for adaptive adversaries

no environment  $\mathcal{Z}$  can tell with non-negligible probability whether it is interacting with  $\mathcal{A}$  and parties running Protocol ACP in the  $\mathcal{F}_{\text{ZK}}$ -hybrid model or with  $\mathcal{S}$  in the ideal process for  $\mathcal{F}_{\text{CP}}$ . Simulator  $\mathcal{S}$  will operate by running a simulated copy of  $\mathcal{A}$  and will use  $\mathcal{A}$  in order to interact with  $\mathcal{Z}$  and  $\mathcal{F}_{\text{CP}}$ .  $\mathcal{S}$  works in a similar way to the simulator in the static case (see the proof of Proposition 4.7.1), with the following changes:

1.  $\mathcal{S}$  records the pair  $(s, t)$  from the initialization phase of an execution. In the case where the receiver is uncorrupted, this pair is chosen by  $\mathcal{S}$  itself. In the case where the receiver is corrupted this pair is chosen by



the simulated  $\mathcal{A}$ , and  $\mathcal{S}$  obtains both  $s$  and  $t$  from the message that the corrupted receiver sends to  $\mathcal{F}_{\text{ZK}}^{\text{T}}$ .

2. Whenever an uncorrupted party  $C$  commits to an unknown value  $w$ , simulator  $\mathcal{S}$  hands  $\mathcal{A}$  a commitment to  $0^k$  as the commitment value. More precisely, whenever an uncorrupted  $C$  writes a **commit** message on its outgoing transcript for  $\mathcal{F}_{\text{CP}}$ , simulator  $\mathcal{S}$  simulates the hybrid-model  $C$  writing a **ZK-prover** message on its outgoing communication tape for  $\mathcal{F}_{\text{ZK}}^{\text{C}}$  (recall that only the headers in these messages are public). Then, when  $\mathcal{A}$  delivers this message to  $\mathcal{F}_{\text{ZK}}^{\text{C}}$  in the simulated interaction,  $\mathcal{S}$  computes  $c = \text{aHC}_s(0^k; r)$  for a random  $r$ , and simulates  $\mathcal{A}$  receiving the message (**ZK-proof**,  $\text{sid}_C, (s, c)$ ) from  $\mathcal{F}_{\text{ZK}}^{\text{C}}$ . Likewise,  $\mathcal{S}$  delivers the **commit** message from  $C$  to  $\mathcal{F}_{\text{CP}}$ , and receives back (**receipt**,  $\text{sid}$ ) from  $\mathcal{F}_{\text{CP}}$ . Then, when  $\mathcal{A}$  delivers the **ZK-proof** from  $\mathcal{F}_{\text{ZK}}^{\text{C}}$  to  $V$ , simulator  $\mathcal{S}$  delivers the **receipt** message from  $\mathcal{F}_{\text{CP}}$  to  $V$ . The “prove phase” is simulated in an analogous way.

(Recall that by the **aHC** scheme, the commitment  $c$  to  $0^k$  that is generated by  $\mathcal{S}$  can be later opened as any string in  $\{0,1\}^k$ , given the trapdoor information  $t$ ; see Section 4.5. This will be needed below.)

3. When the simulated  $\mathcal{A}$  internally corrupts  $C$ , simulator  $\mathcal{S}$  first externally corrupts  $C$  in the ideal process for  $\mathcal{F}_{\text{CP}}$  and obtains the vector of values  $\bar{w}$  that  $C$  committed to so far. Next,  $\mathcal{S}$  prepares for  $\mathcal{A}$  a simulated internal state of  $C$  in Protocol ACP as follows. Apart from the vector of committed values  $\bar{w}$ , the only hidden internal state that  $C$  keeps in Protocol ACP is a vector of random strings  $\bar{r}$  that were used to commit to each  $w_i$  in  $\bar{w}$ . That is, for each input value  $w_i$  in  $\bar{w}$ , adversary  $\mathcal{A}$  expects to see a value  $r_i$  such that  $c_i = \text{aHC}_s(w_i, r_i)$ , where  $c_i$  is the corresponding commitment value that  $\mathcal{S}$  generated and handed to  $\mathcal{A}$  in the simulation of commitments by an uncorrupted  $C$  (see step 2 above). Thus, for every  $i$ ,  $\mathcal{S}$  generates the appropriate value  $r_i$  using the trapdoor  $t$ , and then hands the list  $\bar{r}$  to  $\mathcal{A}$ . (See Section 4.5 for a description of exactly how this randomness is generated.)
4. When the simulated  $\mathcal{A}$  internally corrupts  $V$ ,  $\mathcal{S}$  provides  $\mathcal{A}$  with a simulated internal state of  $V$ . This state consists of the preimage  $t$ , plus the messages that  $V$  receives from  $\mathcal{F}_{\text{ZK}}$ . All this information is available to  $\mathcal{S}$ .

The analysis of the above simulator is very similar to the static case (Proposition 4.7.1). The main difference is that here the commitment is only computationally binding. Thus the following bad event is potentially possible: When the committer  $C$  is corrupted, the simulated  $\mathcal{A}$  commits to a series of values  $\bar{w}$  with corresponding commitment values  $\bar{c}$ . Later, in the prove phase,  $\mathcal{A}$  then generates a message (**ZK-prover**,  $\text{sid}_P, (x, \bar{c}), (\bar{w}', \bar{r}')$ ) to send to  $\mathcal{F}_{\text{ZK}}^{\text{P}}$ , where  $\bar{w}' \neq \bar{w}$  and yet for every  $i$ , it holds that  $c_i = \text{aHC}_s(w'_i, r'_i)$ . Furthermore,  $R(x, \bar{w}') = 1$  and  $R(x, \bar{w}) = 0$ . In other words, the bad event corresponds to a case where in the ideal process  $\mathcal{F}_{\text{CP}}$  does not send a (**CP-proof**,  $\text{sid}, x$ ) message (because  $R(x, \bar{w}) = 0$ ), whereas  $V$  does output such a message in a real

execution of Protocol ACP (because  $R_P((x, s, \bar{c})(\bar{w}', \bar{r}')) = 1$  and the vector of commitments  $\bar{c}$  is as observed by  $V$ ). (We note that given that this event does not occur, the correctness of the simulation carried out by  $\mathcal{S}$  follows the same argument as in the proof of Proposition 4.7.1.)

We conclude the proof by showing that this bad event occurs with negligible probability, or else  $\mathcal{Z}$  and  $\mathcal{A}$  can be used to construct an algorithm that breaks the binding property of the aHC commitment scheme. It suffices to show that  $\mathcal{A}$  cannot generate a message  $(\text{ZK-prover}, \text{sid}_P, (x, \bar{c}), (\bar{w}', \bar{r}'))$  where  $\bar{w}' \neq \bar{w}$  and yet for every  $i$ , it holds that  $c_i = \text{aHC}_s(w'_i, r'_i)$ . Intuitively, this follows from the binding property of aHC (see Section 4.5). In particular, let  $\mathcal{Z}$  and  $\mathcal{A}$  be such that the bad event occurs with non-negligible probability during the above ideal-process simulation by  $\mathcal{S}$ . Then, we construct a machine  $M$  who receives  $s$  and with non-negligible probability outputs a commitment  $c$  along with  $(w_1, r_1)$  and  $(w_2, r_2)$ , where  $c = \text{aHC}_s(w_1; r_1) = \text{aHC}_s(w_2; r_2)$  and  $w_1 \neq w_2$ .

$M$  invokes  $\mathcal{S}$  on  $\mathcal{Z}$  and  $\mathcal{A}$ , and emulates the ideal process, while playing the roles of the ideal functionality and the uncorrupted parties  $C$  and  $V$ . Simulator  $\mathcal{S}$  is the same as described above, with the following two important differences:

- Instead of  $\mathcal{S}$  choosing the pair  $(s, t)$  itself in step 1 of its instructions above, it uses the value  $s$  that  $M$  receives as input. (Recall that  $M$  receives  $s$  and is attempting to contradict the binding property of the commitment relative to this  $s$ .)
- If  $C$  commits to any values before it is corrupted, the simulation is modified as follows. Instead of  $\mathcal{S}$  providing  $\mathcal{A}$  with a commitment  $c = \text{aHC}_s(0^k; r)$ , machine  $M$  provides  $\mathcal{S}$  with the input  $w$  being committed to and then  $\mathcal{S}$  provides  $\mathcal{A}$  with  $c = \text{aHC}_s(w; r)$ . Upon corruption of  $C$ , simulator  $\mathcal{S}$  then provides  $\mathcal{A}$  directly with the list of random values  $\bar{r}$  used in generating the commitments.  $M$  can give  $\mathcal{S}$  these values because it plays the uncorrupted  $C$  in the emulation and therefore knows the  $w$  values.<sup>21</sup>

If during the emulation by  $M$ , the above-described bad event occurs, then  $M$  outputs  $c$  and the two pairs  $(w_1, r_1)$  and  $(w_2, r_2)$ . In order to analyze the success probability of  $M$ , first notice that the views of  $\mathcal{Z}$  and  $\mathcal{A}$  in this simulation by  $M$  are indistinguishable from their views in an ideal process execution. The only difference between the executions is that  $\mathcal{A}$  does not receive “dummy commitments” to  $0^k$ , but real commitments to  $w$ . By the hiding property of the commitments, these are indistinguishable. Therefore, the probability that  $\mathcal{A}$  generates the messages constituting a bad event in  $M$ ’s emulation is negligibly close to the probability that the bad event occurs in

<sup>21</sup> A more “natural” definition of  $M$  would be to have it run  $\mathcal{S}$  in the same way as in the simulation, even before  $C$  is corrupted. In such a case,  $M$  would need to know the trapdoor in order to proceed when  $C$  is corrupted. However, it is crucial here that  $M$  not know the trapdoor (because the binding property only holds when the trapdoor is not known).

the ideal process simulation by  $\mathcal{S}$ . The key point here is that  $\mathcal{S}$  does *not need to know the trapdoor  $t$*  in order to carry out the emulation. In particular,  $M$  carries out its emulation with  $s$  only, and without knowing  $t$ . Therefore the binding property of the commitment scheme  $\text{aHC}_s$  must hold with respect to  $M$ . However, by the contradicting hypothesis, the bad event occurs in  $M$ 's emulation with non-negligible probability. This contradicts the binding property of the commitment scheme. ■

## 4.8 Two-Party Secure Computation for Malicious Adversaries

In this section, we show how to obtain universally composable general secure computation in the presence of *malicious* adversaries. Loosely speaking, we present a protocol compiler that transforms any protocol that is designed for the semi-honest adversarial model into a protocol that guarantees essentially the same behaviour in the presence of malicious adversaries. The compiler is described in Section 4.8.1. Then, Section 4.8.2 ties together all the components of the construction, from Sections 4.4, 4.6, 4.7 and 4.8.1.

### 4.8.1 The Protocol Compiler

As discussed in Section 4.2.2, the  $\mathcal{F}_{\text{CP}}$  functionality is used to construct a protocol compiler that transforms any non-trivial protocol that UC realizes some two-party functionality  $\mathcal{F}$  in the presence of semi-honest adversaries (e.g., the protocol of Section 4.4), into a non-trivial protocol that UC realizes  $\mathcal{F}$  in the presence of malicious adversaries. In this section we present the compiler (the same compiler is valid for both static and adaptive adversaries). Now, let  $\Pi$  be a two-party, reactive protocol. Without loss of generality, we assume that  $\Pi$  works by a series of activations, where in each activation, only one of the parties has an input. This is consistent with our description of general two-party functionalities, see Sections 4.3.1 and 4.3.3. For the sake of simplicity, we also assume that the lengths of the random tapes specified by  $\Pi$  for all activations is  $k$ .

The compiled protocol  $\text{Comp}(\Pi)$  is described in Figure 4.11 below. It uses two copies of  $\mathcal{F}_{\text{CP}}$ : one for when  $P_1$  is the committer and one for when  $P_2$  is the committer. These copies of  $\mathcal{F}_{\text{CP}}$  are denoted  $\mathcal{F}_{\text{CP}}^1$  and  $\mathcal{F}_{\text{CP}}^2$ , respectively, and are formally identified by session identifiers  $\text{sid}_1$  and  $\text{sid}_2$  (where  $\text{sid}_i$  can be taken as  $\text{sid} \circ i$ ). The description of the compiler is from the point of view of party  $P_1$ ;  $P_2$ 's instructions are analogous. We note that  $P_1$  and  $P_2$  only consider messages that are associated with the specified identifier  $\text{sid}$  (this is implicit in the protocol specification).

Loosely speaking, the effect of the compiler on the adversary's capabilities, is that the (malicious) adversary must exhibit semi-honest behaviour,

**Comp( $\Pi$ )**

Party  $P_1$  proceeds as follows (the code for party  $P_2$  is analogous):

1. **Random tape generation:** When activating **Comp( $\Pi$ )** for the first time with session identifier  $sid$ , party  $P_1$  proceeds as follows:
  - a) *Choosing a random tape for  $P_1$ :*
    - i.  $P_1$  chooses  $r_1^1 \in_R \{0, 1\}^k$  and sends (**commit**,  $sid_1$ ,  $r_1^1$ ) to  $\mathcal{F}_{CP}^1$ . ( $P_2$  receives a (**receipt**,  $sid_1$ ) message, chooses  $r_1^2 \in_R \{0, 1\}^k$  and sends ( $sid$ ,  $r_1^2$ ) to  $P_1$ .)
    - ii. When  $P_1$  receives a message ( $sid$ ,  $r_1^2$ ) from  $P_2$ , it sets  $r_1 \stackrel{\text{def}}{=} r_1^1 \oplus r_1^2$  ( $r_1$  serves as  $P_1$ 's random tape for the execution of  $\Pi$ ).
  - b) *Choosing a random tape for  $P_2$ :*
    - i.  $P_1$  waits to receive a message (**receipt**,  $sid_2$ ) from  $\mathcal{F}_{CP}^2$  (this occurs after  $P_2$  sends a **commit** message (**commit**,  $sid_2$ ,  $r_2^2$ ) to  $\mathcal{F}_{CP}^2$ ). It then chooses  $r_2^1 \in_R \{0, 1\}^k$  and sends ( $sid$ ,  $r_2^1$ ) to  $P_2$ . ( $P_2$  sets  $r_2 = r_2^1 \oplus r_2^2$  to be its random tape for the execution of  $\Pi$ .)
2. **Activation due to new input:** When activated with input  $(sid, x)$ , party  $P_1$  proceeds as follows.

- a) *Input commitment:*  $P_1$  sends (**commit**,  $sid_1$ ,  $x$ ) to  $\mathcal{F}_{CP}^1$  and adds  $x$  to the list of inputs  $\bar{x}$  (this list is initially empty and contains  $P_1$ 's inputs from all the previous activations of  $\Pi$ ). Note that at this point  $P_2$  receives the message (**receipt**,  $sid_1$ ) from  $\mathcal{F}_{CP}^1$ .
- b) *Protocol computation:* Let  $\bar{m}_1$  be the series of  $\Pi$ -messages that  $P_1$  received from  $P_2$  in all the activations of  $\Pi$  until now ( $\bar{m}_1$  is initially empty).  $P_1$  runs the code of  $\Pi$  on its input list  $\bar{x}$ , messages  $\bar{m}_1$ , and random tape  $r_1$  (as generated above).
- c) *Outgoing message transmission:* For any outgoing message  $m$  that  $\Pi$  instructs  $P_1$  to send to  $P_2$ ,  $P_1$  sends (**CP-prover**,  $sid_1$ ,  $(m, r_1^2, \bar{m}_1)$ ) to  $\mathcal{F}_{CP}^1$  where the relation  $R_\Pi$  for  $\mathcal{F}_{CP}^1$  is defined as follows:

$$R_\Pi = \{((m, r_1^2, \bar{m}_1), (\bar{x}, r_1^1)) \mid m = \Pi(\bar{x}, r_1^1 \oplus r_1^2, \bar{m}_1)\}$$

In other words,  $P_1$  proves that  $m$  is the correct next message generated by  $\Pi$  when the input sequence is  $\bar{x}$ , the random tape is  $r_1 = r_1^1 \oplus r_1^2$  and the series of incoming  $\Pi$ -messages equals  $\bar{m}_1$ . (Recall that  $r_1^1$  and all the elements of  $\bar{x}$  were committed to by  $P_1$  in the past using commit invocations of  $\mathcal{F}_{CP}^1$ , and that  $r_1^2$  is the random string sent by  $P_2$  to  $P_1$  in Step 1(a)ii above.)

3. **Activation due to incoming message:** When activated with incoming message (**CP-proof**,  $sid_2$ ,  $(m, r_2^1, \bar{m}_2)$ ) from  $\mathcal{F}_{CP}^2$ ,  $P_1$  first verifies that the following conditions hold (we note that  $\mathcal{F}_{CP}^2$  is parameterized by the same relation  $R_\Pi$  as  $\mathcal{F}_{CP}^1$ ):

- a)  $r_2^1$  is the string that  $P_1$  sent to  $P_2$  in Step 1(b)i above.
- b)  $\bar{m}_2$  equals the series of  $\Pi$ -messages received by  $P_2$  from  $P_1$  (i.e.,  $P_1$ 's outgoing messages) in all the activations until now.

If either condition fails,  $P_1$  ignores the message. Otherwise,  $P_1$  appends  $m$  to its list of incoming  $\Pi$ -messages  $\bar{m}_1$  and proceeds to Steps 2b and 2c.

4. **Output:** Whenever  $\Pi$  generates an output value, **Comp( $\Pi$ )** generates the same output value.

**Fig. 4.11.** The compiled protocol **Comp( $\Pi$ )**

or else its cheating will be detected. Recall that a semi-honest adversary follows the protocol instructions exactly, according to a fixed input and a uniformly distributed random input. The following proposition asserts that for every malicious adversary  $\mathcal{A}$  participating in an execution of the compiled protocol (in the  $\mathcal{F}_{\text{CP}}$ -hybrid model), there exists a semi-honest adversary  $\mathcal{A}'$  that interacts with the original protocol in the plain real-life model such that for every environment  $\mathcal{Z}$ , the output distributions in these two interactions are *identical*. Thus, essentially, a malicious adversary is reduced to semi-honest behaviour. We note that the compiler does not use any additional cryptographic construct other than access to  $\mathcal{F}_{\text{CP}}$ . Consequently, the following proposition holds unconditionally, and even if the adversary and environment are computationally unbounded.

**Proposition 4.8.1** *Let  $\Pi$  be a two-party protocol and let  $\text{Comp}(\Pi)$  be the protocol obtained by applying the compiler of Figure 4.11 to  $\Pi$ . Then, for every malicious adversary  $\mathcal{A}$  that interacts with  $\text{Comp}(\Pi)$  in the  $\mathcal{F}_{\text{CP}}$ -hybrid model there exists a semi-honest adversary  $\mathcal{A}'$  that interacts with  $\Pi$  in the plain real-life model, such that for every environment  $\mathcal{Z}$ ,*

$$\left\{ \text{REAL}_{\Pi, \mathcal{A}', \mathcal{Z}}(k, z) \right\}_{k \in \mathbf{N}, z \in \{0,1\}^*} \equiv \left\{ \text{HYBRID}_{\text{Comp}(\Pi), \mathcal{A}, \mathcal{Z}}^{\mathcal{F}_{\text{CP}}}(k, z) \right\}_{k \in \mathbf{N}, z \in \{0,1\}^*}$$

An immediate corollary of this proposition is that any protocol that UC realizes some two-party functionality  $\mathcal{F}$  in the semi-honest model can be compiled into a protocol that UC realizes  $\mathcal{F}$  in the malicious model. This holds both for static and adaptive adversaries.

**Corollary 4.8.2** *Let  $\mathcal{F}$  be a two-party functionality and let  $\Pi$  be a non-trivial protocol that UC realizes  $\mathcal{F}$  in the real-life model and in the presence of semi-honest adversaries. Then  $\text{Comp}(\Pi)$  is a non-trivial protocol that UC realizes  $\mathcal{F}$  in the  $\mathcal{F}_{\text{CP}}$ -hybrid model and in the presence of malicious adversaries.*

We note that the proposition and corollary hold both for the case of adaptive adversaries and for the case of static adversaries. Here we prove the stronger claim, relating to adaptive adversaries. We now prove the proposition.

**Proof of Proposition 4.8.1.** Intuitively, a malicious adversary cannot cheat because the validity of each message that it sends is verified using the  $\mathcal{F}_{\text{CP}}$  functionality. Therefore, it has no choice but to play in a semi-honest manner (or be detected cheating).

More precisely, let  $\mathcal{A}$  be a malicious adversary interacting with  $\text{Comp}(\Pi)$  in the  $\mathcal{F}_{\text{CP}}$ -hybrid model. We construct a semi-honest adversary  $\mathcal{A}'$  that interacts with  $\Pi$  in the plain real-life model, such that no environment  $\mathcal{Z}$  can tell whether it is interacting with  $\text{Comp}(\Pi)$  and  $\mathcal{A}$  in the  $\mathcal{F}_{\text{CP}}$ -hybrid model, or with  $\Pi$  and  $\mathcal{A}'$  in the plain real-life model. As usual,  $\mathcal{A}'$  works by running

a simulated copy of  $\mathcal{A}$  and using the messages sent by  $\mathcal{A}$  as a guide for its interaction with  $\Pi$  and  $\mathcal{Z}$ . We use the term *external communication* to refer to the communication of  $\mathcal{A}'$  with  $\mathcal{Z}$  and  $\Pi$ . The term *internal communication* is used to refer to the communication of  $\mathcal{A}'$  with the simulated  $\mathcal{A}$ . Before describing  $\mathcal{A}'$ , we note the difference between this proof and all previous ones in this paper. Until now, we constructed an *ideal process* adversary  $\mathcal{S}$  from a hybrid or real model adversary  $\mathcal{A}$ . In contrast, here we construct a *real model* adversary  $\mathcal{A}'$  from a hybrid model adversary  $\mathcal{A}$ . Furthermore, previously both  $\mathcal{S}$  and  $\mathcal{A}$  were malicious adversaries, whereas here  $\mathcal{A}$  is malicious and  $\mathcal{A}'$  is semi-honest. We now describe  $\mathcal{A}'$ :

First,  $\mathcal{A}'$  runs a simulated copy of  $\mathcal{A}$  and simulates for  $\mathcal{A}$  the  $\text{Comp}(\Pi)$  messages relating to the generation of the random string of both parties. Next,  $\mathcal{A}'$  translates each message externally sent in  $\Pi$  to the corresponding message (or sequence of messages) in  $\text{Comp}(\Pi)$ . Each message sent by the simulated  $\mathcal{A}$  (in the name of a corrupted party running  $\text{Comp}(\Pi)$ ) is translated back to a  $\Pi$ -message and sent externally. The rationale of this behaviour is that if the simulated  $\mathcal{A}$  (controlling the corrupted party) deviates from the protocol, then this would have been detected by the partner in  $\text{Comp}(\Pi)$ , and thus  $\mathcal{A}'$  can ignore that message. If  $\mathcal{A}$  does not deviate from the protocol, then  $\mathcal{A}'$  can forward the messages sent by  $\mathcal{A}$  to the other party as this is allowed behaviour for a semi-honest party. More precisely,  $\mathcal{A}'$  proceeds as follows.

**Simulating the communication with the environment:** Every input value coming from  $\mathcal{Z}$  (in the external communication) is forwarded to the simulated  $\mathcal{A}$  (in the internal communication) as if coming from  $\mathcal{A}$ 's environment. Similarly, every output value written by  $\mathcal{A}$  on its output tape is copied to  $\mathcal{A}'$ 's own output tape (to be read by the external  $\mathcal{Z}$ ).

**Simulating the “random tape generation” phase:** When the first activation of  $\Pi$  takes place,  $\mathcal{A}'$  internally simulates the “random tape generation” phase of  $\text{Comp}(\Pi)$ . Here we separately deal with each corruption case:

1. *Both parties are not corrupted:*  $\mathcal{A}'$  simulates both parties' messages from this stage. That is, in order to simulate the generation of  $P_1$ 's random tape,  $\mathcal{A}'$  internally passes  $\mathcal{A}$  the message (`receipt`,  $sid_1$ ), as if coming from  $\mathcal{F}_{\text{cp}}^1$ . Furthermore,  $\mathcal{A}'$  chooses a random  $r_1^2$ , records the value, and simulates  $P_2$  sending  $P_1$  the message ( $sid$ ,  $r_1^2$ ) of Step 1(a)ii in Figure 4.11. The simulation of  $P_2$ 's random tape is analogous.
2.  *$P_1$  is not corrupted and  $P_2$  is corrupted:* We begin with the generation of  $P_1$ 's random tape. As above,  $\mathcal{A}'$  begins by internally passing  $\mathcal{A}$  the message (`receipt`,  $sid_1$ ), as if coming from  $\mathcal{F}_{\text{cp}}^1$ . Then,  $\mathcal{A}'$  obtains and records the message ( $sid$ ,  $r_1^2$ ) from the corrupted  $P_2$  (controlled by  $\mathcal{A}$  in  $\text{Comp}(\Pi)$ ).

We now proceed to the generation of  $P_2$ 's random tape.  $\mathcal{A}'$  obtains from  $\mathcal{A}$  the message (`commit`,  $sid_2$ ,  $r_2^2$ ), as sent by  $P_2$  to  $\mathcal{F}_{\text{cp}}^2$  in an execution of  $\text{Comp}(\Pi)$ . Now, let  $r_2$  equal the random tape of the

corrupted  $P_2$  in the external execution of  $\Pi$  ( $\mathcal{A}'$  knows this value because it can read all of the corrupted  $P_2$ 's tapes). Then,  $\mathcal{A}'$  sets  $r_2^1 = r_2 \oplus r_2^2$  and internally passes  $\mathcal{A}$  the message  $(\text{sid}, r_2^1)$ , as if sent by  $P_1$  to  $P_2$ . (Recall that  $\mathcal{A}'$  is semi-honest and thus it cannot modify  $P_2$ 's random tape  $r_2$  for  $\Pi$ .  $\mathcal{A}'$  therefore “forces”  $\mathcal{A}$  to use this exact same random tape for  $P_2$  in the simulated execution of  $\text{Comp}(\Pi)$ .)

3.  $P_1$  is corrupted and  $P_2$  is not corrupted: The simulation of this case is analogous to the previous one. In particular, for the generation of the corrupted  $P_1$ 's random tape,  $\mathcal{A}'$  first receives a message  $(\text{commit}, \text{sid}_1, r_1^1)$  from  $\mathcal{A}$  and simulates  $P_2$  sending  $(\text{sid}, r_1^2)$  to  $P_1$ , where  $r_1^2 = r_1 \oplus r_1^1$  and  $r_1$  equals the random tape of the real party  $P_1$  executing  $\Pi$ .
4. Both parties are corrupted: When both parties are corrupted, the entire simulation is straightforward. ( $\mathcal{A}'$  simply runs both malicious parties and at the end, copies the contents of their output tapes to the output tapes of the semi-honest parties running  $\Pi$ .) We therefore ignore this case from now on.

**Simulating an activation due to new input:** Recall that the input commitment phase consists only of  $P_1$  sending a **commit** message to  $\mathcal{F}_{\text{CP}}^1$ . We deal with the case that  $P_1$  is not corrupted separately from the case that  $P_1$  is corrupted. First, in the case that party  $P_1$  is not corrupted, then  $\mathcal{A}'$  learns that the external  $P_1$  received new input from the fact that it sends its first message of the execution of  $\Pi$ . In response,  $\mathcal{A}'$  simulates the input commitment step by internally passing  $(\text{receipt}, \text{sid}_1)$  to  $\mathcal{A}$  (as  $\mathcal{A}$  expects to receive from  $\mathcal{F}_{\text{CP}}^1$  in a real execution of  $\text{Comp}(\Pi)$ ).

If  $P_1$  is corrupted, then  $\mathcal{A}'$  receives a message  $(\text{commit}, \text{sid}_1, x)$  from  $\mathcal{A}$  (who controls  $P_1$  in  $\text{Comp}(\Pi)$ ). Then,  $\mathcal{A}'$  adds  $x$  to the list  $\bar{x}$  of inputs committed to by  $P_1$  and passes  $\mathcal{A}$  the string  $(\text{receipt}, \text{sid}_1)$ , as if coming from  $\mathcal{F}_{\text{CP}}^1$ . Furthermore,  $\mathcal{A}'$  sets  $P_1$ 's input tape to equal  $x$ . (Recall that a semi-honest adversary is allowed to modify the input values that the environment writes on the input tape of a corrupted party. Formally, when the environment  $\mathcal{Z}$  notifies the semi-honest  $\mathcal{A}'$  of the value that it wishes to write on  $P_1$ 's input tape,  $\mathcal{A}'$  simulates for  $\mathcal{A}$  the malicious model where  $\mathcal{Z}$  writes directly to  $P_1$ 's input tape. Then, when  $\mathcal{A}$  sends the message  $(\text{commit}, \text{sid}_1, x)$  in the simulation,  $\mathcal{A}'$  externally instructs  $\mathcal{Z}$  to write the value  $x$  (as committed to by  $\mathcal{A}$ ) on  $P_1$ 's input tape. See Section 4.3.1 for an exact description of how values are written to the parties' input tapes in the semi-honest model.)

**Dealing with  $\Pi$  messages sent externally by uncorrupted parties:**

If an uncorrupted party  $P_1$  externally sends  $P_2$  a message  $m$  in the execution of  $\Pi$ , then  $\mathcal{A}'$  internally passes  $\mathcal{A}$  the analogous message that it expects to see in  $\text{Comp}(\Pi)$ :  $(\text{CP-proof}, \text{sid}_1, (m, r_1^2, \bar{m}_1))$ , where  $r_1^2$  is the value recorded by  $\mathcal{A}'$  in the simulated generation of  $P_1$ 's random tape above, and  $\bar{m}_1$  is the series of all  $\Pi$ -messages received by  $P_1$  so far. Simi-

larly, if an uncorrupted party  $P_2$  sends  $P_1$  a message  $m$  in the execution of  $\Pi$ , then  $\mathcal{A}'$  internally passes  $\mathcal{A}$  the message  $(\text{CP-proof}, \text{sid}_2, (m, r_2^1, \bar{m}_2))$ , where  $r_2^1$  and  $\bar{m}_2$  are the analogous values to the previous case.

Next, the messages sent from  $P_1$  to  $P_2$  (resp., from  $P_2$  to  $P_1$ ) in the real execution of  $\Pi$  are delivered externally by  $\mathcal{A}'$ , when  $\mathcal{A}$  delivers the corresponding  $(\text{CP-proof}, \dots)$  messages from  $\mathcal{F}_{\text{CP}}^1$  to  $P_2$  (resp., from  $\mathcal{F}_{\text{CP}}^2$  to  $P_1$ ) in the simulated execution of  $\text{Comp}(\Pi)$ .

**Dealing with  $\text{Comp}(\Pi)$  messages sent by corrupted parties:** Assume that  $P_1$  is corrupted. If adversary  $\mathcal{A}$ , controlling  $P_1$ , sends a message  $(\text{CP-prover}, \text{sid}_1, (m, r_1'^2, \bar{m}_1'))$ , then  $\mathcal{A}'$  works as follows. First,  $\mathcal{A}'$  has seen all the messages  $\bar{m}_1$  received by  $P_1$  and can check that  $\bar{m}_1' = \bar{m}_1$ . Likewise,  $\mathcal{A}'$  checks that  $r_1'^2 = r_1^2$  (recall that  $r_1^2$  is the value recorded by  $\mathcal{A}'$  in the simulated generation of  $P_1$ 's random tape above). Finally,  $\mathcal{A}'$  checks that  $m = \Pi(\bar{x}, r_1^1 \oplus r_1^2, \bar{m}_1)$ . (Notice that since  $P_1$  is corrupted,  $\mathcal{A}'$  has all the necessary information to carry out these checks.) If all of the above is true, then  $\mathcal{A}'$  internally passes  $\mathcal{A}$  the message  $(\text{CP-proof}, \text{sid}_1, (m, r_1'^2, \bar{m}_1'))$ , as  $\mathcal{A}$  expects to receive from  $\mathcal{F}_{\text{CP}}^1$ . Then, when  $\mathcal{A}$  delivers this  $(\text{CP-proof}, \dots)$  message from  $\mathcal{F}_{\text{CP}}^1$  to  $P_2$  in the simulation,  $\mathcal{A}$  externally delivers the message that  $P_1$ , running  $\Pi$ , has written on its outgoing communication tape for  $P_2$ .<sup>22</sup> If any of these checks fail, then  $\mathcal{A}'$  does nothing. (That is, no message is externally delivered from  $P_1$  to  $P_2$  at this point.) The case of  $\mathcal{A}$  sending a CP-proof message in the name of a corrupt  $P_2$  is analogous.

**Dealing with corruption of parties:**<sup>23</sup> When the simulated  $\mathcal{A}$  internally corrupts a party  $P_1$ ,  $\mathcal{A}'$  first externally corrupts  $P_1$  and obtains all of  $P_1$ 's past inputs and outputs, and its random tape. Next,  $\mathcal{A}'$  prepares for  $\mathcal{A}$  a simulated internal state of  $P_1$  in protocol  $\text{Comp}(\Pi)$ . This is done as follows. The only additional internal state that  $P_1$  keeps in  $\text{Comp}(\Pi)$  is the random string  $r_1^1$  (this is the string that  $P_1$  commits to in the random tape generation phase of  $\text{Comp}(\Pi)$ ). Therefore,  $\mathcal{A}'$  sets  $r_1^1 = r_1 \oplus r_1^2$ , where  $r_1$  is  $P_1$ 's random string for  $\Pi$  and  $r_1^2$  is the string that  $P_2$  sent to  $P_1$  in the internal simulated interaction with  $\mathcal{A}$  of  $\text{Comp}(\Pi)$ . In this way,  $\mathcal{A}'$  prepares a simulated internal state of  $P_1$  in  $\text{Comp}(\Pi)$  and internally passes it to  $\mathcal{A}$ .  $\mathcal{A}'$  works in an analogous way upon the corruption of  $P_2$ .

We argue that  $\mathcal{Z}$ 's view of the interaction with  $\mathcal{A}'$  and parties running  $\Pi$  in the real-life semi-honest model is *identical* to its view of the interaction with  $\mathcal{A}$  and parties running  $\text{Comp}(\Pi)$  in the  $\mathcal{F}_{\text{CP}}$ -hybrid model. (In particular, the

<sup>22</sup> This point requires some elaboration. Notice that if all checks were successful, then the message that  $P_1$  would send in an execution of  $\Pi$  equals  $m$ . This is because external  $P_1$  in  $\Pi$  and internal  $P_1$  in  $\text{Comp}(\Pi)$  both have the same inputs, random tapes and series of incoming messages. Therefore, their outgoing messages are also the same.

<sup>23</sup> In the case of static adversaries the simulation remains the same with the exception that this case is ignored.



view of the simulated  $\mathcal{A}$  within  $\mathcal{A}'$  is identical to its view in a real interaction with the same  $\mathcal{Z}$  and  $\text{Comp}(\Pi)$  in the  $\mathcal{F}_{\text{CP}}$ -hybrid model.) This can be seen by observing the computational steps in an interaction of  $\mathcal{Z}$  with  $\mathcal{A}'$  and  $\Pi$ . The cases where an uncorrupted party sends a message are immediate. To see that this holds also in the case that  $\mathcal{A}'$  delivers messages sent by corrupted parties, observe the following facts:

1. *Random tape:*  $\mathcal{A}'$  forces the random tape of a corrupted  $P_1$  in the internal execution of  $\text{Comp}(\Pi)$  with  $\mathcal{A}$  to be the random tape of the semi-honest party  $P_1$  externally executing  $\Pi$ .
2. *Input:*  $\mathcal{A}'$  modifies the input tape of the external party  $P_1$  so that it is the same input as committed to by  $\mathcal{A}$ .

We therefore have that the input and random tapes that the malicious  $\mathcal{A}$  committed to for the internal  $P_1$  are exactly the same as the input and random tapes used by the external, semi-honest  $P_1$ .

3. *Message generation and delivery:* In the simulation with malicious  $\mathcal{A}$ , semi-honest  $\mathcal{A}'$  obtains all the inputs committed to by a corrupted  $P_1$ . Consequently,  $\mathcal{A}'$  is able to verify at every step if the message  $m$  sent by  $\mathcal{A}$ , in the name of corrupted  $P_1$ , is according to the protocol specification. If yes, then it is guaranteed that  $P_1$  generates the exact same message  $m$  in the external execution of  $\Pi$ . Thus,  $P_2$  receives the same  $\Pi$ -message in the execution of  $\Pi$  (where the adversary  $\mathcal{A}'$  is semi-honest) and in the execution of  $\text{Comp}(\Pi)$  (where the adversary  $\mathcal{A}$  is malicious). Furthermore, it is guaranteed that whenever  $\mathcal{A}'$  delivers a message  $m$  in the external execution of  $\Pi$ , the simulated  $\mathcal{A}$  generated and delivered a valid corresponding message to  $\mathcal{F}_{\text{CP}}$ .
4. *Corruptions:* The internal state that  $\mathcal{A}$  receives from  $\mathcal{A}'$  upon corrupting a party is exactly the same as it receives in a real execution of  $\text{Comp}(\Pi)$ . In particular, observe that in the simulation of the random tape generation phase when  $P_1$  is not corrupted,  $\mathcal{A}$  receives no information about  $r_1^1$  (it only sees a (receipt,  $\text{sid}_1$ ) message). Therefore,  $\mathcal{A}'$  can choose  $r_1^1$  as any value that it wishes upon the corruption of  $P_1$ , and in particular it can set it to equal  $r_1 \oplus r_1^2$  (recall that  $P_1$  indeed uses the random tape  $r_1$ ; therefore this is consistent with its true internal state).

We conclude that the ensembles REAL and HYBRID are identical. ■

### 4.8.2 Conclusions

Combining the semi-honest protocol of Proposition 4.4.3 with the compilation obtained in Corollary 4.8.2, we have that for any two-party ideal functionality  $\mathcal{F}$ , there exists a protocol that UC realizes  $\mathcal{F}$  in the  $\mathcal{F}_{\text{CP}}$ -hybrid model (in the presence of malicious adversaries). Combining this with the fact that  $\mathcal{F}_{\text{CP}}$  can be UC realized in the  $\mathcal{F}_{\text{ZK}}$ -hybrid model (Proposition 4.7.1), and using the

UC composition theorem (Theorem 4.3.3), we obtain universally composable general two-party computation in the  $\mathcal{F}_{\text{zk}}$ -hybrid model. That is,

**Theorem 4.8.3** (Theorem 4.2.2 – formally restated): *Assume that trapdoor permutations exist. Then, for any well-formed two-party ideal functionality  $\mathcal{F}$ , there exists a non-trivial protocol that UC realizes  $\mathcal{F}$  in the  $\mathcal{F}_{\text{zk}}$ -hybrid model in the presence of malicious, static adversaries. Furthermore, if two-party augmented non-committing encryption protocols exist, then for any adaptively well-formed two-party ideal functionality  $\mathcal{F}$ , there exists a non-trivial protocol that UC realizes  $\mathcal{F}$  in the  $\mathcal{F}_{\text{zk}}$ -hybrid model in the presence of malicious, adaptive adversaries.*

Recall that, under the assumption that trapdoor permutations exist, functionality  $\hat{\mathcal{F}}_{\text{zk}}$  (the multi-session extension of  $\mathcal{F}_{\text{zk}}$ ) can be UC realized in the  $\mathcal{F}_{\text{crs}}$ -hybrid model by protocols that uses a single copy of the reference string. We can thus use the universal composition with joint state theorem (Theorem 4.3.4) to obtain the following corollary:

**Corollary 4.8.4** *Assume that trapdoor permutations exist. Then, for any well-formed two-party ideal functionality  $\mathcal{F}$ , there exists a non-trivial protocol that UC realizes  $\mathcal{F}$  in the  $\mathcal{F}_{\text{crs}}$ -hybrid model in the presence of malicious, static adversaries. Furthermore, if two-party augmented non-committing encryption protocols exist, then for any adaptively well-formed two-party functionality  $\mathcal{F}$ , there exists a non-trivial protocol that UC realizes  $\mathcal{F}$  in the  $\mathcal{F}_{\text{crs}}$ -hybrid model in the presence of malicious, adaptive adversaries. In both cases, the protocol uses a single copy of  $\mathcal{F}_{\text{crs}}$ .*

## 4.9 Multi-party Secure Computation

This section extends the two-party constructions of Sections 4.5–4.8 to the multi-party setting, thereby proving Theorem 4.2.3. The results here relate to a multi-party network where subsets of the parties wish to realize arbitrary (possibly reactive) functionalities of their local inputs. Furthermore, there is an adaptive adversary that can corrupt *any* number of the parties (in particular, no honest majority is assumed). Throughout, we continue to assume a completely asynchronous network without guaranteed message delivery.

This section is organized as follows. We start by showing how to obtain UC multi-party computation in the presence of semi-honest adversaries. Next we define a basic broadcast primitive which will be used in all our protocols in the case of malicious adversaries. We then generalize the UC commitment, zero-knowledge and  $\mathcal{F}_{\text{cp}}$  functionalities to the multi-party case. Finally, we construct a multi-party protocol compiler using the generalized  $\mathcal{F}_{\text{cp}}$ , and obtain UC multi-party computation in the malicious adversarial model. In our presentation below, we assume familiarity with the two-party constructions.

### 4.9.1 Multi-party Secure Computation for Semi-honest Adversaries

In this section, we sketch the construction of non-trivial protocols that UC realize any adaptively well-formed functionality  $\mathcal{F}$  for semi-honest adversaries. (Recall the definition of adaptively well-formed functionalities in Section 4.3.3.) The construction is a natural extension of the construction for the two-party case. We assume that the set  $\mathcal{P}$  of participating parties in any execution is fixed and known; let this set be  $P_1, \dots, P_\ell$ . Then, the input lines to the circuit (comprising of the input value, random coins and internal state of the functionality) are shared amongst all  $\ell$  parties. That is, for every input bit  $\alpha$  to the circuit, the parties hold random bits  $\alpha_1, \dots, \alpha_\ell$ , respectively, under the constraint that  $\alpha = \bigoplus_{i=1}^{\ell} \alpha_i$ . Next, the parties compute the circuit inductively from the inputs to outputs so that at every step, they hold shares of the lines already computed. Once the circuit is fully computed, the parties reconstruct the outputs, as required. We now proceed to prove the following proposition:

**Proposition 4.9.1** *Assume that trapdoor permutations exist. Then, for any well-formed (multi-party) ideal functionality  $\mathcal{F}$ , there exists a non-trivial protocol that UC realizes  $\mathcal{F}$  in the presence of semi-honest, static adversaries. Furthermore, if two-party augmented non-committing encryption protocols exist, then for any adaptively well-formed (multi-party) functionality  $\mathcal{F}$ , there exists a non-trivial protocol that UC realizes  $\mathcal{F}$  in the presence of semi-honest, adaptive adversaries.*

As in the two-party case, for adaptive adversaries we assume the existence of *two-party* augmented non-committing encryption protocols. Indeed, as in the two-party case this assumption is needed only to UC realize the two-party functionality  $\mathcal{F}_{\text{OT}}^4$ , which plays a central role even in the multi-party case.

We begin our proof of Proposition 4.9.1 by presenting a non-trivial multi-party protocol  $\Pi_{\mathcal{F}}$  that UC realizes any adaptively well-formed functionality  $\mathcal{F}$  in the  $\mathcal{F}_{\text{OT}}$ -hybrid model. (We prove the proposition for the adaptive case only, the static case is easily derived.) We start by defining a boolean circuit  $C_{\mathcal{F}}$  that represents an activation of  $\mathcal{F}$ . The circuit  $C_{\mathcal{F}}$  has  $3m$  input lines:  $m$  lines represent the input value sent to  $\mathcal{F}$  in this activation (i.e., this is the input held by one of the parties). The additional  $2m$  input lines are used for  $\mathcal{F}$ 's random coins and for holding  $\mathcal{F}$ 's state at the onset of the activation. The circuit also has  $m$  output lines for each party and  $m$  output lines for final state of  $\mathcal{F}$  after the activation (a total of  $m\ell + m$  lines). For more details on how  $\mathcal{F}$  and  $C_{\mathcal{F}}$  are defined, see the description for the two-party case in Section 4.4.2 (the extensions to the multi-party case are straightforward).

**Protocol  $\Pi_{\mathcal{F}}$  (for UC realizing  $\mathcal{F}$ ):** Let the set of participating parties equal  $\mathcal{P} = \{P_1, \dots, P_\ell\}$ . We state the protocol for an activation in which  $P_1$  sends a message to  $\mathcal{F}$ . When activated with input  $(sid, v)$  for  $P_1$  where

$|v| \leq m$ , the protocol first pads  $v$  to length  $m$  (according to some standard encoding), and sends a message to all the parties in  $\mathcal{P}$ , asking them to participate in a joint evaluation of  $C_{\mathcal{F}}$ . Next, the parties do the following:

### 1. Input Preparation Stage:

- *Input value:*  $P_1$  starts by sharing its input  $v$  with all parties. That is,  $P_1$  chooses  $\ell$  random strings  $v_1, \dots, v_\ell \in_R \{0, 1\}^m$  with the constraint that  $\oplus_{i=1}^\ell v_i = v$ . Then,  $P_1$  sends  $(sid, v_i)$  to  $P_i$  for every  $2 \leq i \leq \ell$ , and stores  $v_1$ .
- *Internal state:* At the onset of each activation, the parties hold shares of the current internal state of  $\mathcal{F}$ . That is, let  $c$  denote the current internal state of  $\mathcal{F}$ , where  $|c| = m$  and  $m$  is an upper bound on the size of the state string stored by  $\mathcal{F}$ . Then, party  $P_i$  holds  $c_i \in \{0, 1\}^m$  and all the  $c_i$ 's are random under the restriction that  $\oplus_{i=1}^\ell c_i = c$ . (In the first activation of  $\mathcal{F}$ , the internal state is empty and so the parties hold fixed shares 0 that denote the empty state.)
- *Random coins:* Upon the *first* activation of  $\mathcal{F}$  only, each party  $P_i$  locally chooses a random string  $r_i \in_R \{0, 1\}^m$ . The strings  $r_1, \dots, r_\ell$  then constitute shares of the random coins  $r = \oplus_{i=1}^\ell r_i$  to be used by  $C_{\mathcal{F}}$  in all activations.

At this point, the parties hold (random) shares of every input line of  $C_{\mathcal{F}}$ .

- ### 2. Circuit Evaluation:
- The parties proceed to evaluate the circuit  $C_{\mathcal{F}}$  in a gate-by-gate manner. Let  $\alpha$  and  $\beta$  denote the bit-values of the input lines to a given gate. Then every  $P_i$  holds bits  $\alpha_i, \beta_i$  such that  $\alpha = \sum_{i=1}^\ell \alpha_i$  and  $\beta = \sum_{i=1}^\ell \beta_i$ . The gates are computed as follows:

- *Addition gates:* If the gate is an addition gate, then each  $P_i$  locally sets its share of the output line of the gate to be  $\gamma_i = \alpha_i + \beta_i$ . (Thus  $\sum_{i=1}^\ell \gamma_i = \sum_{i=1}^\ell (\alpha_i + \beta_i) = \alpha + \beta = \gamma$ .)
- *Multiplication gates:* If the gate is a multiplication gate, then the parties need to compute their shares of  $\gamma = \left(\sum_{i=1}^\ell \alpha_i\right) \left(\sum_{i=1}^\ell \beta_i\right)$ . The key to carrying out this computation is the following equality:

$$\left(\sum_{i=1}^\ell \alpha_i\right) \left(\sum_{i=1}^\ell \beta_i\right) = \ell \cdot \sum_{i=1}^\ell \alpha_i \beta_i + \sum_{1 \leq i < j \leq \ell} (\alpha_i + \alpha_j) \cdot (\beta_i + \beta_j)$$

(See [G02a, Section 3.2.2] for a justification of this equality.) Notice that each party can compute a share of the first sum locally (by simply computing  $\alpha_i \cdot \beta_i$  and multiplying the product by  $\ell$ ). Shares of the second sum can be computed using activations of the *two-party* oblivious transfer functionality  $\mathcal{F}_{\text{OT}}^4$ . (That is, for each pair  $i$  and  $j$ , parties  $P_i$  and  $P_j$  compute shares of  $(\alpha_i + \alpha_j) \cdot (\beta_i + \beta_j)$ . This is exactly the same computation as in the two-party case and can be carried out

using  $\mathcal{F}_{\text{OT}}^4$ .) After computing all of the shares, each party  $P_i$  locally sums its shares into a value  $\gamma_i$ , and we have that  $\sum_{i=1}^{\ell} \gamma_i = \gamma$ , as required.

**3. Output stage:** Following the above stage, the parties hold shares of all the output lines of the circuit  $C_{\mathcal{F}}$ . Each output line of  $C_{\mathcal{F}}$  is either an output addressed to one of the parties  $P_1, \dots, P_{\ell}$ , or belongs to the internal state of  $C_{\mathcal{F}}$  after the activation. The activation concludes as follows:

- *$P_i$ 's output (for every  $i$ ):* For every  $j \neq i$ , party  $P_j$  sends  $P_i$  all of its shares in  $P_i$ 's output lines.  $P_i$  then reconstructs every bit of its output value by adding the appropriate shares, and writes the result on its output tape.
- *Internal state:*  $P_1, \dots, P_{\ell}$  all locally store the shares that they hold for the internal state lines of  $C_{\mathcal{F}}$ . (These shares are to be used in the next activation.)

Recall that since we are working in an asynchronous network, there is no guarantee on the order of message delivery and messages may be delivered “out of order”. In contrast, to maintain correctness the protocol must be executed according to its prescribed order (e.g., new activations must begin only after previous ones have completed and gates may be evaluated only after the shares of the input lines are known). As in the two-party case, this is dealt with by assigning unique identifiers to every message sent during all activations. A full description of how this can be achieved appears in Section 4.4.2. By having the parties store messages that arrive before they are relevant in appropriate buffers (where the time that a message becomes relevant is self-evident from the unique tags), we have that all honest parties process the messages in correct order. Thus, it makes no difference whether or not the adversary delivers the messages according to the prescribed order and we can assume that all messages *are* delivered in order.

This completes the description of  $\Pi_{\mathcal{F}}$ . We now sketch the proof that  $\Pi_{\mathcal{F}}$  UC realizes any adaptively well-formed multi-party functionality  $\mathcal{F}$ :

**Claim 4.9.2** *Let  $\mathcal{F}$  be an adaptively well-formed multi-party functionality. Then, protocol  $\Pi_{\mathcal{F}}$  UC realizes  $\mathcal{F}$  in the  $\mathcal{F}_{\text{OT}}$ -hybrid model, in the presence of semi-honest, adaptive adversaries.*

**Proof (sketch).** The proof of this claim is very similar to the two-party case (i.e., Claim 4.4.4). First, it is clear that  $\Pi_{\mathcal{F}}$  correctly computes  $\mathcal{F}$  (i.e., all parties receive outputs that are distributed according to  $\mathcal{F}$ ). Next, we show the existence of a simulator for  $\Pi_{\mathcal{F}}$ . The basis for the simulator's actions is the fact that, as long as there is at least one uncorrupted party, all the intermediary values seen by the parties are uniformly distributed.

Let  $\mathcal{A}$  be a semi-honest, adaptive adversary; we construct a simulator  $\mathcal{S}$  for the ideal process  $\mathcal{F}$ . Simulator  $\mathcal{S}$  internally invokes  $\mathcal{A}$  and works as follows:

**Simulating the communication with  $\mathcal{Z}$ :** The input values received by  $\mathcal{S}$  from  $\mathcal{Z}$  are written on  $\mathcal{A}$ 's input tape, and the output values of  $\mathcal{A}$  are copied to  $\mathcal{S}$ 's own output tape.

**Simulation of the input stage:** Recall that in this stage, the only messages sent are random strings  $v_2, \dots, v_\ell$  that  $P_1$  sends to  $P_2, \dots, P_\ell$ . Thus, the simulation of this stage involves simulating  $P_1$  sending  $\ell - 1$  random strings  $v_2, \dots, v_\ell$  to  $P_2, \dots, P_\ell$ . (If  $P_1$  is corrupted, then  $v_2, \dots, v_\ell$  are chosen according to  $P_1$ 's random tape. Otherwise,  $\mathcal{S}$  chooses each  $v_i$  uniformly.)

**Simulation of the circuit evaluation stage:** The addition gates require no simulation since they constitute local computation only. The multiplication gates involve simulation of pairwise oblivious transfer calls to  $\mathcal{F}_{\text{OT}}$ . We describe the simulation of these oblivious transfers separately for each corruption case.

1. *Oblivious transfers run with an uncorrupted receiver:* In the case that the receiver is not corrupted, the only message seen by  $\mathcal{A}$  in a call to  $\mathcal{F}_{\text{OT}}$  is the session-identifier used. This is therefore easily simulated by  $\mathcal{S}$ . (If the sender is corrupted, then its input table to  $\mathcal{F}_{\text{OT}}$  is seen by  $\mathcal{A}$ . However, this is already defined because it is a function of the sender's view which is known to  $\mathcal{A}$ .)
2. *Oblivious transfers run with an uncorrupted sender and a corrupted receiver:* In this case, the receiver obtains a uniformly distributed bit  $\gamma_2$  as output from the oblivious transfer. Therefore,  $\mathcal{S}$  merely chooses  $\gamma_2$  uniformly.
3. *Oblivious transfers run with a corrupted sender and receiver:* Simulation is straightforward when both participating parties are corrupted (all input values and random tapes are already defined).

**Simulation of the output stage:**  $\mathcal{S}$  simulates the parties sending strings in the output stage in order to reconstruct their outputs. First, we note that the shares of the output lines are already defined for any party  $P_j$  that is already corrupted. (This is because  $\mathcal{A}$  holds the view of  $P_j$  and this view defines the shares that  $P_j$  holds of all the output lines.) This means that the strings that  $P_j$  sends in the output stage are also defined. Now,  $\mathcal{S}$  defines the strings received by a party  $P_i$  in the output stage as follows. If  $P_i$  is not corrupted, then  $\mathcal{S}$  simulates all the other uncorrupted parties sending  $P_i$  uniformly distributed strings. If  $P_i$  is corrupted, then  $\mathcal{S}$  has  $P_i$ 's output  $y_i$ .  $\mathcal{S}$  uses this to choose random strings for the honest parties so that the exclusive-or of these strings along with the defined output strings sent by the corrupted parties equals  $y_i$ . (Thus,  $P_i$ 's output is reconstructed to  $y_i$ , as required.) Simulator  $\mathcal{S}$  carries out this simulation for all parties  $P_1, \dots, P_\ell$ .

**Simulation of corruptions while honest parties remain:** When some party  $P_i$  is corrupted,  $\mathcal{S}$  should provide  $\mathcal{A}$  with the internal state of  $P_i$  for all the activations of  $\mathcal{F}$  (i.e., for all the evaluations of  $C_{\mathcal{F}}$ ) so far. All the evaluations are dealt with independently from each other, except that  $P_i$ 's output shares of  $\mathcal{F}$ 's internal state from one evaluation equals its input shares of  $\mathcal{F}$ 's internal state in the following evaluation. Also all evaluations, except perhaps for the current one, are complete. Here we describe how  $\mathcal{S}$  deals with a single, complete activation. (If the current activation is not complete then  $\mathcal{S}$  follows its instructions until the point where  $P_i$  is corrupted.)

Upon the corruption of party  $P_i$ , simulator  $\mathcal{S}$  receives  $P_i$ 's input  $x_i$  and output  $y_i$ , and should generate  $P_i$ 's view of the simulated protocol execution. This view should be consistent with the messages sent in the simulation so far. We begin with the simulation of  $P_i$ 's view of the input stage. If  $i = 1$  (i.e.,  $P_1$  is the party that is corrupted), then  $\mathcal{S}$  obtains the input value  $v$ . Let  $v_2, \dots, v_\ell$  be the random strings that  $P_1$  sent  $P_2, \dots, P_\ell$  in the simulated interaction. Then,  $\mathcal{S}$  defines  $P_1$ 's share of the input to equal  $v_1$  so that  $\bigoplus_{i=1}^\ell v_i = v$ .  $\mathcal{S}$  continues for any  $P_i$  (i.e., not just for  $i = 1$ ) as follows.  $\mathcal{S}$  chooses random strings  $r_i \in_R \{0, 1\}^m$  and  $c_i \in_R \{0, 1\}^s$  and sets  $P_i$ 's inputs to  $C_{\mathcal{F}}$ 's random-coins and internal state to be  $r_i$  and  $c_i$ , respectively.

Having completed the simulation of  $P_i$ 's view of the input stage,  $\mathcal{S}$  proceeds to simulate  $P_i$ 's view in the oblivious transfers of the protocol execution. Below we describe the simulation for all the multiplication gates except for those immediately preceding output lines (these will be dealt with separately below). We distinguish four cases (when referring to corrupted and uncorrupted  $P_j$  below, we mean the current corruption status and not the status at the time that the given oblivious transfer was executed):

1. *Oblivious transfers run with  $P_i$  as sender and an uncorrupted  $P_j$  as receiver:* Recall that in every oblivious transfer, the sender inputs a random-bit  $\gamma_1$  to mask the outcome. In this case,  $\mathcal{S}$  simply chooses  $\gamma_1$  uniformly. (This is the random bit that  $P_i$  supposedly chose upon computing this gate.)
2. *Oblivious transfers run with an uncorrupted  $P_j$  as sender and  $P_i$  as receiver:* In this case, in the execution of  $\Pi_{\mathcal{F}}$ , party  $P_i$  receives a uniformly distributed bit  $\gamma_2$  as output from the oblivious transfer. Therefore,  $\mathcal{S}$  chooses  $\gamma_2$  uniformly.
3. *Oblivious transfers run with  $P_i$  as sender and an already corrupted  $P_j$  as receiver:*  $P_j$  is already corrupted and therefore the value  $\gamma_2$  that it received from this oblivious transfer has already been fixed in the simulation. Furthermore, both  $P_i$  and  $P_j$ 's circuit inputs  $v_i, v_j, r_i, r_j$  and  $c_i, c_j$  have been fixed, as too have their views for all the multiplication gates leading to this one. ( $\mathcal{S}$  computes the view inductively

from the inputs to the outputs.) Thus, the input lines to this oblivious transfer are fixed, as too is  $P_j$ 's output from the oblivious transfer. This fully defines the oblivious transfer table that  $P_1$  constructs in the protocol execution (as well as its “random” bit  $\gamma_1$ ). Therefore,  $\mathcal{S}$  constructs the table according to the protocol instructions.

4. *Oblivious transfers run with an already corrupted  $P_j$  as sender and  $P_i$  as receiver:* As in the previous case, the input lines and the random-bit  $\gamma_1$  that  $P_j$  inputs into the oblivious transfer are fixed. Since  $P_i$ 's input into this oblivious transfer is also already fixed, this fully defines the bit  $\gamma_2$  that  $P_i$  receives as output.

As we have mentioned, there is a difference regarding the simulation of multiplication gates that precede output lines. (As in the two-party case, we assume for simplicity that every output line is preceded by a multiplication gate.) We describe the simulation of these gates together with the output stage. During the simulation of the output stage,  $P_i$  received uniformly distributed strings  $y_i^j$  from every party  $P_j$  (the strategy for choosing these values is described above in the item on “simulation of the output stage”). Note that all the  $y_i^j$ 's (for  $j \neq i$ ) are defined and fixed.<sup>24</sup> Upon the corruption of  $P_i$ , simulator  $\mathcal{S}$  receives  $P_i$ 's output string  $y_i$ . The aim of  $\mathcal{S}$  is to have  $P_i$ 's output lines define shares  $y_i^j$  such that  $\bigoplus_{j=1}^{\ell} y_i^j = y_i$  (and thus  $P_i$ 's output reconstruction will be as required). This is done as follows. Recall that the evaluation of each multiplication gate is comprised of a series of oblivious transfers between all pairs of parties. Since  $P_i$  is not the last honest party to be corrupted, there exists at least one honest party  $P_l$  with which  $P_i$  runs a pairwise oblivious transfer in the computation of this gate. All the oblivious transfers of this gate apart from this one are simulated as described above. These simulations all provide bit-shares to  $P_i$ : let  $b$  denote the sum of these shares. It remains to simulate this last oblivious transfer between  $P_i$  and  $P_l$ . Let  $\gamma_i$  be the bit of  $y_i^i$  that  $P_i$  is supposed to receive as its share of the output line that follows from this multiplication gate. Now, the specific oblivious transfer between  $P_i$  and  $P_l$  defines one share of the output bit  $\gamma_i$ , and all the other shares have already been fixed and sum to  $b$ . Thus, the aim of  $\mathcal{S}$  is to have  $P_i$ 's output from the oblivious transfer with  $P_l$  equal  $\gamma_i + b$  (and thus  $P_i$ 's overall output from the gate will be  $\gamma_i$  as required). However,  $P_l$  is not corrupted. Therefore, whether  $P_i$  is the sender or  $P_l$  is the sender,  $P_i$ 's output can be chosen arbitrarily by  $\mathcal{S}$ . (See the first 2 of the 4 simulation cases above; in those cases,  $\mathcal{S}$  merely chooses the output bit randomly.) Thus,  $\mathcal{S}$  sets the output bit to be  $\gamma_i + b$  and  $P_i$  receives the correct bit. This completes the simulation for the corruption of  $P_i$ .

<sup>24</sup> Actually, corruption can happen in the middle of the output stage and in such a case only some of the output strings may be fixed. In such a case, first (internally) fix all the output strings and then continue as here.



**Simulation of the corruption of the last honest party:** Let  $P_i$  be the party that is corrupted last. If  $i = 1$ , then  $\mathcal{S}$  obtains the input-value  $v$  into this activation. As above, in this case,  $\mathcal{S}$  defines  $P_1$ 's share of the input  $v_1$  to be so that  $\oplus_{j=1}^{\ell} v_j = v$ . In all cases,  $\mathcal{S}$  obtains the output-value  $y_i$  that  $P_i$  receives. Furthermore, since  $\mathcal{F}$  is adaptively well-formed,  $\mathcal{S}$  obtains the random tape of  $\mathcal{F}$ . Given this information,  $\mathcal{S}$  computes the internal state of  $\mathcal{F}$  in the beginning of this activation;  $\mathcal{S}$  can do this because it now holds  $\mathcal{F}$ 's random tape and the inputs of all parties in all the activations of  $\mathcal{F}$ . Let  $c$  be the computed state string and let  $r$  equal  $\mathcal{F}$ 's length- $m$  random tape. Now, apart from  $P_i$ , all the other parties have already been corrupted. Therefore, the shares of the random tape  $r_j$  are already fixed for every  $j \neq i$ .  $\mathcal{S}$  takes  $P_i$ 's share of the random tape to be  $r_i$  so that  $\oplus_{j=1}^{\ell} r_j = r$ . Likewise, except for  $c_i$ , all the shares of the state input  $c_j$  are fixed.  $\mathcal{S}$  thus defines  $c_i$  so that  $\oplus_{j=1}^{\ell} c_j = c$ . This completes the simulation of the input stage.

Next  $\mathcal{S}$  simulates the circuit evaluation stage, working from the input gates to the output gates, in the same way as described above (i.e., for the case that  $P_i$  is not the last honest party corrupted). Notice above that when  $P_i$  runs an oblivious transfer with an already corrupted party, then all the inputs and outputs are essentially fixed. Thus,  $\mathcal{S}$  merely computes the bit that  $P_i$  should see in each oblivious transfer as in the above cases. We therefore have that the simulation of this stage is a deterministic process. This simulation also defines the output shares that  $P_i$  receives, thus concluding the simulation.

**Output delivery:**  $\mathcal{S}$  delivers the output from  $\mathcal{F}$  to an uncorrupted party  $P_i$  when  $\mathcal{A}$  delivers all the output shares  $y_i^j$  that parties  $P_j$  send  $P_i$  in the simulation.

It remains to show that no environment  $\mathcal{Z}$  can distinguish the case that it interacts with  $\mathcal{S}$  and  $\mathcal{F}$  in the ideal process or with  $\mathcal{A}$  and  $\Pi_{\mathcal{F}}$  in the  $\mathcal{F}_{\text{OT}}$ -hybrid model. The analysis is similar to the one for the two-party case and is omitted. ■

#### 4.9.2 Authenticated Broadcast

In order to obtain our result, we assume that each set of parties that engage in a protocol execution have access to an *authenticated* broadcast channel. The broadcast channel is modelled by the ideal broadcast functionality,  $\mathcal{F}_{\text{BC}}$ , as defined in Figure 4.12. In our protocols for malicious adversaries, all communication among the parties is carried out via  $\mathcal{F}_{\text{BC}}$ .

Note that the  $\mathcal{F}_{\text{BC}}$ -hybrid model does not guarantee delivery of messages, nor does it provide any synchrony guarantees for the messages that are delivered. It only guarantees that no two uncorrupted parties in  $\mathcal{P}$  will receive two different message with the same *sid*. We remark that in contrast to all the

**Functionality  $\mathcal{F}_{\text{BC}}$** 

$\mathcal{F}_{\text{BC}}$  proceeds as follows, running with parties  $P_1, \dots, P_n$  and an adversary  $\mathcal{S}$ :

- Upon receiving a message (**broadcast**,  $\text{sid}, \mathcal{P}, x$ ) from  $P_i$ , where  $\mathcal{P}$  is a set of parties, send (**broadcast**,  $\text{sid}, P_i, \mathcal{P}, x$ ) to all parties in  $\mathcal{P}$  and to  $\mathcal{S}$ , and halt.

**Fig. 4.12.** The ideal broadcast functionality

other functionalities defined in this chapter, the *entire* (**broadcast**,  $\text{sid}, \mathcal{P}, x$ ) message is included in the *public header* (i.e.,  $x$  is not private and can be read by the adversary). This is consistent with the fact that the adversary can read any message  $x$  sent by one party to another party, using its outgoing communication tape. (That is, we view the broadcast to be like regular communication between parties and not like private communication that takes place between parties and ideal functionalities.)

We now show that  $\mathcal{F}_{\text{BC}}$  can be UC realized, for any number of corrupted parties and without any setup assumptions.<sup>25</sup>

**Protocol 4.9.3** (universally composable broadcast):

- **Input:**  $P_i$  has input (**broadcast**,  $\text{sid}, \mathcal{P}, x$ ).
- **The Protocol:**
  1.  $P_i$  sends ( $\text{sid}, \mathcal{P}, x$ ) to all parties.
  2. Denote by  $(\mathcal{P}^j, x^j)$  the message received by  $P_j$  in the previous round. Then, every party  $P_j$  (for  $j \neq i$ ) sends its message ( $\text{sid}, \mathcal{P}^j, x^j$ ) to all other parties.
  3. Denote the message received by  $P_j$  from  $P_k$  in the previous round by  $(\mathcal{P}_k^j, x_k^j)$  (recall that  $x^j$  denotes the value  $P_j$  received from  $P_i$  in the first round). Then,  $P_j$  outputs (**broadcast**,  $\text{sid}, P_i, \mathcal{P}, x^j$ ) if it belongs to the set  $\mathcal{P}$ , if it received all the messages  $(\mathcal{P}_k^j, x_k^j)$  and this is the only message that it saw (i.e., if it received  $(\mathcal{P}_k^j, x_k^j)$  from every  $P_k$  and it holds that all these messages are the same). Otherwise, it outputs nothing.  
We note that if  $P_j$  did not receive any value in the first round, then it does not output anything.

**Proposition 4.9.4** Protocol 4.9.3 UC realizes  $\mathcal{F}_{\text{BC}}$  in the presence of malicious, adaptive adversaries.

**Proof.** Let  $\mathcal{A}$  be a real-model adversary attacking Protocol 4.9.3. We construct an ideal model adversary  $\mathcal{S}$  for  $\mathcal{A}$  that interacts with  $\mathcal{F}_{\text{BC}}$ . We differentiate between two cases: in the first the dealer is corrupted (and thus is

<sup>25</sup> For the sake of accuracy regarding credits, we note that the fact that  $\mathcal{F}_{\text{BC}}$  can be UC realized in a point-to-point network was shown in [GL02] (and not in [CLOS02] where all the other results in this chapter were shown).

controlled by  $\mathcal{A}$ ), and in the second it is honest. Let  $P_i$  denote the dealer in the current execution:

- *Case 1 –  $P_i$  is corrupt:* In the first round,  $\mathcal{A}$  (controlling  $P_i$ ) sends messages to  $\ell$  of the honest parties for some  $\ell$ ; denote these messages by  $(\mathcal{P}_{i_1}, x_{i_1}), \dots, (\mathcal{P}_{i_\ell}, x_{i_\ell})$ . Simulator  $\mathcal{S}$  receives all these messages and then simulates the messages sent by the honest parties in the second round. Furthermore,  $\mathcal{S}$  obtains all the messages sent by  $\mathcal{A}$  in the second round.

Now, if there exist  $j$  and  $k$  ( $1 \leq j, k \leq \ell$ ) such that  $(\mathcal{P}_{i_j}, x_{i_j}) \neq (\mathcal{P}_{i_k}, x_{i_k})$ , then  $\mathcal{S}$  sends nothing to  $\mathcal{F}_{\text{BC}}$ . Otherwise, let  $x$  be the message sent by  $\mathcal{A}$ . Then,  $\mathcal{S}$  sends  $x$  to the ideal functionality  $\mathcal{F}_{\text{BC}}$ . Next,  $\mathcal{S}$  defines the set of honest parties to whom to deliver the output (**broadcast**,  $\text{sid}, P_i, \mathcal{P}, x$ ) from  $\mathcal{F}_{\text{BC}}$ . This set of parties is defined to be those parties in  $\mathcal{P}$  to whom  $\mathcal{A}$  delivers all the second round messages and whose messages all contain the same  $\mathcal{P}$  and  $x$ .  $\mathcal{S}$  concludes by delivering the messages from the  $\mathcal{F}_{\text{BC}}$  functionality to these parties, and only to these parties.

- *Case 2 –  $P_i$  is honest:*  $\mathcal{S}$  receives (**broadcast**,  $\text{sid}, P_i, \mathcal{P}, x$ ) from  $\mathcal{F}_{\text{BC}}$  and simulates  $P_i$ 's sending  $x$  to all the parties controlled by  $\mathcal{A}$ . Then,  $\mathcal{S}$  receives back messages sent by  $\mathcal{A}$  to the honest parties.  $\mathcal{S}$  defines the set of parties to whom to deliver output as those in  $\mathcal{P}$  who receive all the second round messages and who only see  $(\mathcal{P}, x)$ . Then,  $\mathcal{S}$  delivers the messages from the  $\mathcal{F}_{\text{BC}}$  functionality to these and only to these parties.

It remains to describe  $\mathcal{S}$ 's actions for corruptions. This is straightforward here because honest parties hold no private information whatsoever (the protocol is deterministic and so they don't even have random tapes). Therefore, when  $\mathcal{A}$  corrupts a party, it learns nothing beyond what was known before. Thus,  $\mathcal{S}$  need not provide any information to  $\mathcal{A}$  and can just corrupt the appropriate ideal-process party.

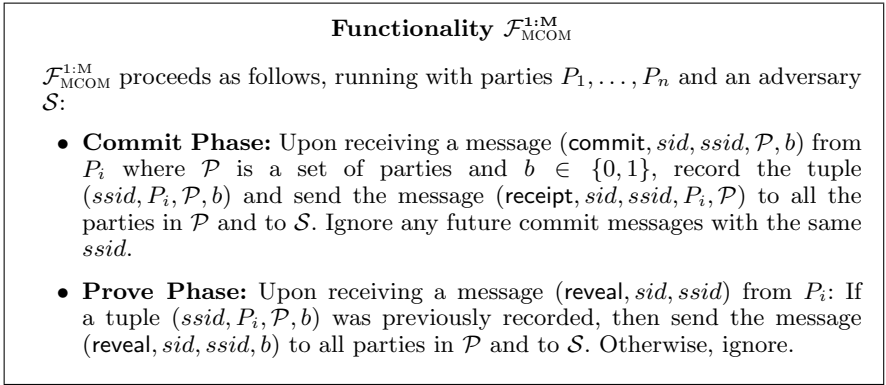
We claim that the global output of an ideal execution with  $\mathcal{S}$  is *identically distributed* to the global output of a real execution with  $\mathcal{A}$ . We first deal with the case that  $P_i$  is corrupt. If  $\mathcal{A}$  sends two different messages in round 1 (i.e., if there exist  $j$  and  $k$  such that  $(\mathcal{P}^j, x^j) \neq (\mathcal{P}^k, x^k)$ ), then by the protocol definition, all honest parties will see both  $(\mathcal{P}^j, x^j)$  and  $(\mathcal{P}^k, x^k)$ . (Here it is important that parties do not output  $(\mathcal{P}, x)$  unless seeing the round 2 messages of all parties.) Therefore, in a real execution all honest parties will output nothing. This is identical to the case that  $\mathcal{S}$  does not send anything to  $\mathcal{F}_{\text{BC}}$  in an ideal execution. In contrast, if  $\mathcal{A}$  sends the same message pair  $(\mathcal{P}, x)$  to all honest parties in the first round, then the outputs depend on what  $\mathcal{A}$  sends in the second round. Since  $\mathcal{S}$  receives all these messages from  $\mathcal{A}$ , it can see which parties would output  $(\mathcal{P}, x)$  and which parties would output nothing.  $\mathcal{S}$  thus delivers the (**broadcast**, ...) messages from  $\mathcal{F}_{\text{BC}}$  only to the parties which would output  $x$  in the real model. We conclude that the output is identical.

In the case that  $P_i$  is honest,  $\mathcal{A}$  can cause honest parties to output nothing (rather than  $(\mathcal{P}, x)$ ) by sending them messages  $(\mathcal{P}', x') \neq (\mathcal{P}, x)$  in the second round or by not delivering messages. As above,  $\mathcal{S}$  receives all these messages and therefore its delivery of (**broadcast**, ...) messages from  $\mathcal{F}_{\text{BC}}$  accurately represents exactly what happens in a real execution. ■

#### 4.9.3 One-to-Many Extensions of $\mathcal{F}_{\text{MCOM}}$ , $\mathcal{F}_{\text{ZK}}$ , and $\mathcal{F}_{\text{CP}}$

In this section, we present *one-to-many* extensions of the commitment, zero-knowledge and commit-and-prove functionalities. The extensions are called “one-to-many” because in all of them, a *single* party commits/proves to *many* receivers/verifiers.

**One-to-many UC commitments.** We begin by defining a one-to-many extension of the UC commitment functionality, denoted  $\mathcal{F}_{\text{MCOM}}^{1:\text{M}}$ . In this functionality, the committing party commits to a value to many receivers. The formal definition appears in Figure 4.13. Similarly to the two-party case, the commitment functionality is presented as a *multi-session functionality*. From here on, the JUC theorem of [CR03] is applied and we consider only single-session functionalities (see Section 4.3.2 for more explanation). We denote the single session analog to  $\mathcal{F}_{\text{MCOM}}^{1:\text{M}}$  by  $\mathcal{F}_{\text{COM}}^{1:\text{M}}$ .



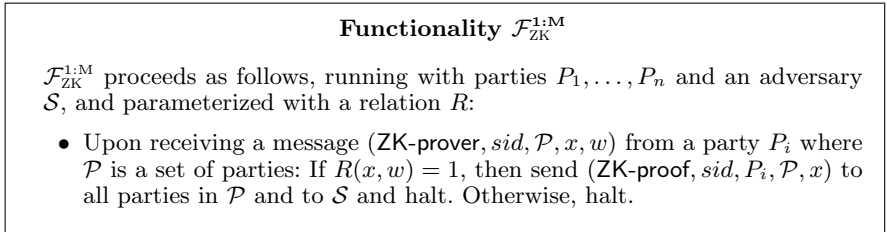
**Fig. 4.13.** One-to-Many multi-session commitment

The key observation in realizing the  $\mathcal{F}_{\text{MCOM}}^{1:\text{M}}$  functionality is that Protocol UAHC (of Section 4.5) that UC realizes the two-party commitment functionality  $\mathcal{F}_{\text{MCOM}}$  is *non-interactive*. Therefore, the one-to-many extension is obtained by simply having the committer broadcast the commitment string of Protocol UAHC to all the participating parties on the broadcast channel. The proof that this protocol UC realizes  $\mathcal{F}_{\text{MCOM}}^{1:\text{M}}$  is almost identical to the proof that Protocol UAHC UC realizes  $\mathcal{F}_{\text{MCOM}}$ , and is omitted. We do, however, mention one important point. The commitment string is broadcast using

the  $\mathcal{F}_{\text{BC}}$  functionality which ensures that only one message is broadcast using a given session identifier. This is important because otherwise the adversary could broadcast two different commitment strings  $c_1$  and  $c_2$ , where it delivers  $c_1$  to some of the honest parties and  $c_2$  to the others. This is, of course, not allowed by the  $\mathcal{F}_{\text{MCOM}}^{1:M}$  functionality that ensures that all parties receive the same commitment for the same identifier pair  $(sid, ssid)$ . We therefore have the following:

**Proposition 4.9.5** *Assuming the existence of trapdoor permutations, there exists a (non-interactive) protocol that UC realizes  $\mathcal{F}_{\text{MCOM}}^{1:M}$  in the  $(\mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{BC}})$ -hybrid model<sup>26</sup>, in the presence of malicious, adaptive adversaries. Furthermore, this protocol uses only a single copy of  $\mathcal{F}_{\text{CRS}}$ .*

**One-to-many UC zero-knowledge.** Similarly to the one-to-many extension of commitments, we define a one-to-many functionality where one party proves a statement to some set of parties. The definition of the (single-session) one-to-many zero-knowledge functionality, denoted  $\mathcal{F}_{\text{ZK}}^{1:M}$ , appears in Figure 4.14. (For simplicity, in the multi-party case we concentrate on single-session zero-knowledge, constructed using a single-session version of  $\mathcal{F}_{\text{MCOM}}^{1:M}$ . These protocols will later be composed, using universal composition with joint state, to obtain protocols that use only a single copy of the reference string when realizing all the copies of  $\mathcal{F}_{\text{ZK}}^{1:M}$ .)



**Fig. 4.14.** Single-session, One-to-Many zero-knowledge

As with the case of commitments, a non-interactive protocol that UC realizes the two-party zero-knowledge functionality  $\mathcal{F}_{\text{ZK}}$  could be directly used to realize  $\mathcal{F}_{\text{ZK}}^{1:M}$ . For the case of static adversaries, the protocol of [DDOPS01] can be used. However, for the case of adaptive adversaries, no non-interactive protocol is known. Rather, we base the one-to-many extension on the interactive UC zero-knowledge protocol of [CF01]. Their protocol is basically that of parallel Hamiltonicity (cf. [B86]), except that the commitments used are universally composable. Our extension of this protocol to the one-to-many

<sup>26</sup> In the  $(\mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{BC}})$ -hybrid model, all parties have ideal access to both the common reference string functionality  $\mathcal{F}_{\text{CRS}}$  and the ideal broadcast functionality  $\mathcal{F}_{\text{BC}}$ .

case follows the methodology of [G02a] and is presented in the proof of the following proposition:

**Proposition 4.9.6** *There exists a protocol that UC realizes  $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$  in the  $(\mathcal{F}_{\text{COM}}^{1:\text{M}}, \mathcal{F}_{\text{BC}})$ -hybrid model, in the presence of malicious, adaptive adversaries.*

**Proof (sketch).** The protocol for realizing  $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$  works by having the prover separately prove the statement in question to all parties. The protocol used in each of these pairwise proofs is exactly the two-party protocol of [CF01], with the exception that the messages of each proof are broadcast to all parties. (This also means that all commitments and decommitments are run using  $\mathcal{F}_{\text{COM}}^{1:\text{M}}$ , rather than the two-party  $\mathcal{F}_{\text{COM}}$ . ( $\mathcal{F}_{\text{COM}}^{1:\text{M}}$  is the single-session parallel to  $\mathcal{F}_{\text{MCOM}}^{1:\text{M}}$ .) Also, the protocol must make sure that each invocation of broadcast will have a unique session ID. This can be done in standard ways, given the unique session ID of the zero-knowledge protocol.) Then, a party accepts the proof, outputting  $(\text{ZK-proof}, \text{sid}, P_i, \mathcal{P}, x)$ , if and only if all the pairwise proofs are accepting. Note that other than the use of  $\mathcal{F}_{\text{COM}}^{1:\text{M}}$ , no cryptographic primitives are used. Indeed, security of this protocol in the  $\mathcal{F}_{\text{COM}}^{1:\text{M}}$ -hybrid model is unconditional.

Next, note that it is indeed possible for the parties to know whether all the pairwise proofs are accepting. This is because all the commitments and messages are seen by all the parties and the zero-knowledge proof of Hamiltonicity used by [CF01] is publicly verifiable (i.e., it is enough to see the transcript of prover/verifier messages to know whether or not the proof was accepted by the verifier).

Now, recall that in order to prove the universal composability of  $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$ , we must present an ideal-process adversary (i.e., a simulator) that simulates proofs for the case that the prover is not corrupted and verifiers are corrupted, and is also able to extract the witness from an adversarially generated proof (for the case that the prover is corrupted). When simulating a proof for a corrupted verifier, the simulator for  $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$  works simply by running the simulator of the two-party protocol of [CF01] for every pairwise proof. On the other hand, in order to extract the witness from a corrupted prover, first note that it is possible to run the two-party extractor for any pairwise proof in which the verifier is not corrupted. Now, the scenario in which we need to run the extractor here is where the prover is corrupted and at least one verifier is not (otherwise, all parties are corrupted and simulation is straightforward). Therefore, there exists one pairwise proof in which the verifier is not corrupted. The simulator for  $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$  thus runs the extractor for the protocol of [CF01] for this proof. Finally, we note that the simulator delivers the output of  $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$  to the verifiers if and only if all verifiers accept in the simulation. (Thus, the parties' outputs in the ideal process are the same as in a real execution.) This concludes the proof sketch. ■

**One-to-many UC commit-and-prove.** The one-to-many extension of the commit-and-prove functionality, denoted  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ , is presented in Figure 4.15.

The functionality handles a single session only, and requires that all commitments and proofs are to the *same* set  $\mathcal{P}$ . (This set is fixed the first time a **commit** is sent with a given *sid*.)

**Functionality  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$**

$\mathcal{F}_{\text{CP}}^{1:\text{M}}$  proceeds as follows, running with parties  $P_1, \dots, P_n$  and an adversary  $\mathcal{S}$ , and parameterized by a value  $k$  and a relation  $R$ :

- **Commit Phase:** Upon receiving a message (**commit**, *sid*,  $\mathcal{P}$ ,  $w$ ) from  $P_i$  where  $\mathcal{P}$  is a set of parties and  $w \in \{0, 1\}^k$ , append the value  $w$  to the list  $\bar{w}$ , record  $\mathcal{P}$ , and send the message (**receipt**, *sid*,  $P_i$ ,  $\mathcal{P}$ ) to the parties in  $\mathcal{P}$  and  $\mathcal{S}$ . (Initially, the list  $\bar{w}$  is empty. Also, if a **commit** message has already been received, then check that the recorded set of parties is  $\mathcal{P}$ . If it is a different set, then ignore this message.)
- **Prove Phase:** Upon receiving a message (**CP-prover**, *sid*,  $x$ ) from  $P_i$ , where  $x \in \{0, 1\}^{\text{poly}(k)}$ , compute  $R(x, \bar{w})$ : If  $R(x, \bar{w}) = 1$ , then send the message (**CP-proof**, *sid*,  $x$ ) to the parties in  $\mathcal{P}$  and to  $\mathcal{S}$ . Otherwise, ignore.

**Fig. 4.15.** One-to-Many commit-and-prove

Our protocol for UC realizing the one-to-many commit-and-prove functionality  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$  is constructed in the  $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$ -hybrid model. The protocol, denoted  $\text{ACP}^{1:\text{M}}$ , is very similar to Protocol **ACP** for the two-party case. Recall that Protocol **ACP** begins with the receiver choosing a pair  $(s, t)$ , where  $s = f(t)$  and  $f$  is a one-way function. The value  $s$  is then used by the committer who commits to  $w$  by sending  $c = \text{aHC}_s(w; r)$ . This is generalized in the natural way by having *every* receiving party  $P_j$  choose a pair  $(s_j, t_j)$ , and the committer then sending  $c_j = \text{aHC}_{s_j}(w; r_j)$  for all values of  $s_j$ . In addition, the committer proves that all these commitments are to the same  $w$  (this is done to prevent the committer from committing to different  $w$ 's for different  $s_j$ 's). We define a *compound* commitment scheme as follows. Let  $\mathbf{s} = (s_1, \dots, s_\ell)$  and  $\mathbf{r} = (r_1, \dots, r_\ell)$ . Then, define  $\mathbf{c} = \text{aHC}_{\mathbf{s}}(w; \mathbf{r}) = (\text{aHC}_{s_1}(w; r_1), \dots, \text{aHC}_{s_\ell}(w; r_\ell))$ . Restating the above, the commit phase consists of the committer committing to  $w$  using the compound scheme  $\text{aHC}_{\mathbf{s}}$  and proving that the commitment was generated correctly.

The multi-party protocol  $\text{ACP}^{1:\text{M}}$  uses three different copies of  $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$ , where each copy is parameterized by a different relation. The copies are denoted  $\mathcal{F}_{\text{ZK}, \text{T}}^{1:\text{M}}$  (for the initialization phase),  $\mathcal{F}_{\text{ZK}, \text{C}}^{1:\text{M}}$  (for the commit stage) and  $\mathcal{F}_{\text{ZK}, \text{P}}^{1:\text{M}}$  (for the prove stage). These functionalities are differentiated by session identifiers  $\text{sid}_T$ ,  $\text{sid}_C$  and  $\text{sid}_P$ , respectively. These identifiers should be unique, as long as the session ID of the current instance of  $\text{ACP}^{1:\text{M}}$  is unique. One way to guarantee this is setting  $\text{sid}_T = \text{sid} \circ T$ ,  $\text{sid}_C = \text{sid} \circ C$  and  $\text{sid}_P = \text{sid} \circ P$ , where  $\text{sid}$  is the session ID of the current instance of  $\text{ACP}^{1:\text{M}}$ . The protocol is presented in Figure 4.16.

**Protocol ACP<sup>1:M</sup>**

- **Auxiliary Input:** A security parameter  $k$ , and a session identifier  $sid$ .

- **Initialization phase:**

The first time that the committer  $P_i$  wishes to commit to a value to the set of parties  $\mathcal{P}$  using the identifier  $sid$ , the parties in  $\mathcal{P}$  execute the following initialization phase. (To simplify notation, assume that  $\mathcal{P} = \{P_1, \dots, P_\ell\}$  for some  $\ell$ , and that the sender belongs to  $\mathcal{P}$ . Also, the parties ignore incoming messages that are addressed to a set  $\mathcal{P}'$  that is different than the set  $\mathcal{P}$  specified in the first message.)

1.  $P_i$  sends a (**broadcast**,  $sid, \mathcal{P}$ , begin-commit) message to  $\mathcal{F}_{BC}$  to indicate that it wishes to initiate a commit activation.
2. Upon receiving (**broadcast**,  $sid, P_i, \mathcal{P}$ , begin-commit), each party  $P_j \in \mathcal{P}$  records the triple  $(sid, P_i, \mathcal{P})$ . From here on, the parties only relate to messages with identifier  $sid$  if they are associated with the committer/prover  $P_i$  and set of parties  $\mathcal{P}$ .

Then, every  $P_j$  chooses  $t_j \in_R \{0, 1\}^k$ , computes  $s_j = f(t_j)$  (where  $f$  is a one-way function), and sends (**ZK-prover**,  $sid_T, \mathcal{P}, s_j, t_j$ ) to  $\mathcal{F}_{ZK,T}^{1:M}$ , where  $\mathcal{F}_{ZK,T}^{1:M}$  is parameterized by the relation  $R_T$  defined by:

$$R_T \stackrel{\text{def}}{=} \{(s, t) \mid s = f(t)\}$$

3. Upon receiving (**ZK-proof**,  $sid_T, P_j, \mathcal{P}, s_j$ ) from  $\mathcal{F}_{ZK,T}^{1:M}$ , all the parties in  $\mathcal{P}$  (including the committer  $P_i$ ) record the value  $s_j$ . This phase concludes when all parties in  $\mathcal{P}$  have sent the appropriate **ZK-proof** message, and thus when all the parties hold the vector  $\mathbf{s} = (s_1, \dots, s_\ell)$ .

The parties now proceed to the commit phase.

- **Commit phase:** ( $P_i$ 's input is (commit,  $sid, \mathcal{P}, w$ ), where  $w \in \{0, 1\}^k$ .)

1.  $P_i$  computes the compound commitment  $\mathbf{c} = \text{aHC}_s(w; \mathbf{r})$  where the vector  $\mathbf{s}$  is the one obtained in the initialization phase, and the  $r_j$ 's in  $\mathbf{r}$  are uniformly chosen.

$P_i$  then sends (**ZK-prover**,  $sid_C, \mathcal{P}, (\mathbf{s}, \mathbf{c}), (w, \mathbf{r})$ ) to  $\mathcal{F}_{ZK,C}^{1:M}$ , where  $\mathcal{F}_{ZK,C}^{1:M}$  is parameterized by the relation  $R_C$  defined by:

$$R_C \stackrel{\text{def}}{=} \{(\mathbf{s}, \mathbf{c}), (w, \mathbf{r}) \mid \mathbf{c} = \text{aHC}_s(w; \mathbf{r})\}$$

(That is,  $R_C$  verifies that  $\mathbf{c}$  is a valid compound commitment to the value  $w$ , using  $\mathbf{s}$ .) In addition,  $P_i$  stores in a list  $\bar{w}$  all the values  $w$  that were sent, and in lists  $\bar{\mathbf{c}}$  and  $\bar{\mathbf{r}}$  the corresponding commitment values  $\mathbf{c}$  and random strings  $\mathbf{r} = (r_1, \dots, r_\ell)$ .

2. Upon receiving (**ZK-proof**,  $sid_C, P_i, \mathcal{P}, (\mathbf{s}', \mathbf{c})$ ) from  $\mathcal{F}_{ZK,C}^{1:M}$ , every party  $P_j \in \mathcal{P}$  verifies that  $\mathbf{s}' = \mathbf{s}$  (where  $\mathbf{s}$  equals the list of strings that it recorded in the initialization phase). If yes, then  $P_j$  outputs (receipt,  $sid$ ) and adds the commitment  $\mathbf{c}$  to its list  $\bar{\mathbf{c}}$ . (Initially,  $\bar{\mathbf{c}}$  is empty.) Otherwise, the parties in  $\mathcal{P}$  ignore the message.

- **Prove phase:** See Figure 4.17.

**Fig. 4.16.** A protocol for UC realizing  $\mathcal{F}_{CP}^{1:M}$  for adaptive adversaries

**Proposition 4.9.7** *Assuming the existence of one-way functions, Protocol ACP<sup>1:M</sup> of Figure 4.16 UC realizes  $\mathcal{F}_{CP}^{1:M}$  in the  $(\mathcal{F}_{BC}, \mathcal{F}_{ZK}^{1:M})$ -hybrid model, in the presence of adaptive adversaries.*



**Protocol ACP<sup>1:M</sup>** (continued from Figure 4.16)

- **Prove phase:** ( $P_i$ 's input is (CP-prover,  $sid, x$ ).)
  1.  $P_i$  sends (ZK-prover,  $sid_P, P_i, \mathcal{P}, (x, \mathbf{s}, \bar{\mathbf{c}}), (\bar{w}, \bar{\mathbf{r}})$ ) to  $\mathcal{F}_{\text{ZK}, \mathcal{P}}^{1:M}$ , where  $\bar{\mathbf{c}}$ ,  $\bar{w}$  and  $\bar{\mathbf{r}}$  are the lists described above. Let  $\bar{w} = (w_1, \dots, w_m)$ ,  $\bar{\mathbf{c}} = (\mathbf{c}_1, \dots, \mathbf{c}_m)$  and  $\bar{\mathbf{r}} = (\mathbf{r}_1, \dots, \mathbf{r}_m)$ . Then,  $\mathcal{F}_{\text{ZK}, \mathcal{P}}^{1:M}$  is parameterized by the relation  $R_P$  defined by:
 
$$R_P \stackrel{\text{def}}{=} \{((x, \mathbf{s}, \bar{\mathbf{c}}), (\bar{w}, \bar{\mathbf{r}})) \mid R(x, \bar{w}) = 1 \ \& \ \forall j \ \mathbf{c}_j = \text{aHC}_s(w_j; \mathbf{r}_j)\}$$
 That is,  $R_P$  verifies that  $R(x, \bar{w}) = 1$  and that  $\bar{\mathbf{c}}$  contains commitments to the previously committed values  $\bar{w}$ .
  2. Upon receiving (ZK-proof,  $sid_P, P_i, \mathcal{P}, (x, \mathbf{s}', \bar{\mathbf{c}})$ ) from  $\mathcal{F}_{\text{ZK}, \mathcal{P}}^{1:M}$ , every party in  $\mathcal{P}$  verifies that  $\mathbf{s}' = \mathbf{s}$  and that its list of stored commitments equals  $\bar{\mathbf{c}}$ . If yes, then it outputs (CP-proof,  $sid, x$ ). Otherwise, it ignores the message.

**Fig. 4.17.** A protocol for UC realizing  $\mathcal{F}_{\text{CP}}^{1:M}$  for adaptive adversaries (continued)

**Proof (sketch).** The proof of this proposition is very similar to the proof of Proposition 4.7.2 for the two-party case. Let  $\mathcal{A}$  be an adaptive adversary who operates against Protocol ACP<sup>1:M</sup> in the  $\mathcal{F}_{\text{ZK}}^{1:M}$ -hybrid model. We construct a simulator  $\mathcal{S}$  such that no environment  $\mathcal{Z}$  can tell with non-negligible probability whether it is interacting with  $\mathcal{A}$  and parties running Protocol ACP<sup>1:M</sup> in the  $\mathcal{F}_{\text{ZK}}^{1:M}$ -hybrid model or with  $\mathcal{S}$  in the ideal process for  $\mathcal{F}_{\text{CP}}^{1:M}$ . Simulator  $\mathcal{S}$  operates by running a simulated copy of  $\mathcal{A}$  and using  $\mathcal{A}$  in order to interact with  $\mathcal{Z}$  and  $\mathcal{F}_{\text{CP}}^{1:M}$ .  $\mathcal{S}$  works as follows.

**Simulating the initialization phase:**  $\mathcal{S}$  records all the pairs  $(s_1, t_1), \dots, (s_\ell, t_\ell)$  from the initialization phase of an execution, and defines  $\mathbf{s} = (s_1, \dots, s_\ell)$ . For every uncorrupted receiving party  $P_j$ , simulator  $\mathcal{S}$  chooses the pair  $(s_j, t_j)$  by itself. For corrupted receiving parties, the pairs are chosen by the simulated  $\mathcal{A}$ , and  $\mathcal{S}$  obtains the  $t_j$ 's from  $\mathcal{A}$ 's messages to  $\mathcal{F}_{\text{ZK}, \mathcal{T}}^{1:M}$ .

**Simulating the case where the committer is corrupted:** We first describe how to simulate the commit phase. Whenever  $\mathcal{A}$  (controlling party  $P_i$ ) wishes to commit to a value, the simulator  $\mathcal{S}$  obtains the message (ZK-prover,  $sid_C, \mathcal{P}, (\mathbf{s}, \mathbf{c}), (w, \mathbf{r})$ ) that  $\mathcal{A}$  sends to  $\mathcal{F}_{\text{ZK}, \mathcal{C}}^{1:M}$ .  $\mathcal{S}$  checks that  $\mathbf{s}$  is as generated in the initialization phase and that  $\mathbf{c} = \text{aHC}_s(w; \mathbf{r})$ . If yes, then  $\mathcal{S}$  internally passes  $\mathcal{A}$  the message (ZK-proof,  $sid_C, \mathcal{P}, (\mathbf{s}, \mathbf{c})$ ) and externally sends (commit,  $sid, \mathcal{P}, w$ ) to  $\mathcal{F}_{\text{CP}}^{1:M}$ . Furthermore,  $\mathcal{S}$  adds the commitment  $\mathbf{c}$  to its list of commitments  $\bar{\mathbf{c}}$ .

We now describe the simulation of the prove phase. Whenever  $\mathcal{A}$  wishes to prove a statement,  $\mathcal{S}$  receives a message (ZK-prover,  $sid_P, (x, \mathbf{s}, \bar{\mathbf{c}}), (\bar{w}, \bar{\mathbf{r}})$ ) that  $\mathcal{A}$  sends to  $\mathcal{F}_{\text{ZK}, \mathcal{P}}^{1:M}$ .  $\mathcal{S}$  then checks that  $\mathbf{s}$  is as generated in the initialization phase, that the list  $\bar{\mathbf{c}}$  is as stored above, and that  $R(x, \bar{w}) = 1$ .

If yes, then  $\mathcal{S}$  internally passes  $(\text{ZK-proof}, \text{sid}_P, (x, s, \bar{c}))$  to  $\mathcal{A}$  and externally sends  $(\text{CP-prover}, \text{sid}, x)$  to  $\mathcal{F}_{\text{CP}}^{1:M}$ . Otherwise, it ignores the message.

**Simulating the case where the committer is not corrupted:** Whenever an uncorrupted party  $P_i$  commits to an unknown value  $w$ , simulator  $\mathcal{S}$  hands  $\mathcal{A}$  a commitment to  $0^k$  as the commitment value. More precisely, whenever  $P_i$  writes a commit message on its outgoing communication tape for  $\mathcal{F}_{\text{CP}}$ , simulator  $\mathcal{S}$  internally simulates  $P_i$  writing an appropriate ZK-prover message on its outgoing communication tape (recall that only the commit header is viewed by  $\mathcal{S}$ , and that it only needs to write the ZK-prover header). Then, when  $\mathcal{A}$  delivers the ZK-prover message to  $\mathcal{F}_{\text{ZK,C}}^{1:M}$ , simulator  $\mathcal{S}$  delivers the commit message from  $P_i$  to  $\mathcal{F}_{\text{CP}}^{1:M}$ . Furthermore,  $\mathcal{S}$  computes  $c = \text{aHC}_s(0^k; r)$  and hands  $\mathcal{A}$  the message  $(\text{ZK-proof}, \text{sid}_C, P_i, \mathcal{P}, (s, c))$ , as if coming from  $\mathcal{F}_{\text{ZK,C}}^{1:M}$ . Now, when  $\mathcal{A}$  delivers this ZK-proof message from  $\mathcal{F}_{\text{ZK,C}}^{1:M}$  to a party  $P_j \in \mathcal{P}$  in the internal simulation, then  $\mathcal{S}$  delivers the receipt message from  $\mathcal{F}_{\text{CP}}^{1:M}$  to  $P_j$ . (Recall that by the aHC scheme, given the trapdoor information  $t = (t_1, \dots, t_\ell)$ ,  $\text{aHC}_s(0)$  can be opened as either 0 or 1; see Section 4.5.)

The simulation of the prove phase is carried out as follows. Whenever an uncorrupted  $P_i$  writes a CP-prover message on its outgoing communication tape for  $\mathcal{F}_{\text{CP}}^{1:M}$ , simulator  $\mathcal{S}$  internally simulates  $P_i$  writing the appropriate ZK-prover message on its outgoing communication tape for  $\mathcal{F}_{\text{ZK,P}}^{1:M}$ . When  $\mathcal{A}$  delivers this ZK-prover message in the internal simulation, then  $\mathcal{S}$  delivers  $P_i$ 's CP-prover message to  $\mathcal{F}_{\text{CP}}^{1:M}$  and receives back the message  $(\text{CP-proof}, \text{sid}, x)$ . Then,  $\mathcal{S}$  internally passes  $\mathcal{A}$  the message  $(\text{ZK-proof}, \text{sid}_P, (x, s, \bar{c}))$  as if it came from  $\mathcal{F}_{\text{ZK,P}}^{1:M}$ , where  $\bar{c}$  is the list of simulated commitments generated above. When  $\mathcal{A}$  delivers this ZK-proof message from  $\mathcal{F}_{\text{ZK,P}}^{1:M}$  to a party  $P_j \in \mathcal{P}$  in the internal simulation, then  $\mathcal{S}$  delivers the CP-proof message from  $\mathcal{F}_{\text{CP}}^{1:M}$  to  $P_j$ .

**Dealing with the corruption of parties:** The only private information held by a receiving party  $P_j$  is the trapdoor information  $t_j$  that it chooses in the initialization phase. As we have seen in the simulation of the initialization phase above,  $\mathcal{S}$  knows all of the trapdoors in the simulated execution. Therefore, when  $\mathcal{A}$  corrupts a receiving party  $P_j$ , simulator  $\mathcal{S}$  internally passes  $t_j$  to  $\mathcal{A}$ .

The committing party  $P_i$ 's private state in an execution of Protocol  $\text{ACP}^{1:M}$  consists of the list of committed values  $\bar{w}$  and the list of vectors of random strings  $\bar{r}$  (that contain the decommitment information of the list  $\bar{c}$ ). Therefore, when  $\mathcal{A}$  corrupts the committer  $P_i$ , simulator  $\mathcal{S}$  first externally corrupts  $P_i$  in the ideal process and obtains the list  $\bar{w}$ . Next,  $\mathcal{S}$  generates the list  $\bar{r}$  so that the simulated list of commitments  $\bar{c}$  is “explained” as a list of commitments to  $\bar{w}$ .  $\mathcal{S}$  can do this because it has all of the trapdoor information  $t_1, \dots, t_\ell$  (this case is identical to in the proof of Proposition 4.7.2).

The analysis of the correctness of the simulation is analogous to in the two-party case and is omitted. ■

#### 4.9.4 Multi-party Secure Computation for Malicious Adversaries

As in the two-party case, multi-party secure computation in the presence of malicious adversaries is obtained by constructing a protocol compiler that transforms protocols for the semi-honest model into protocols for the malicious model. This compiler is then applied to Protocol  $\Pi_{\mathcal{F}}$  of Section 4.9.1. The compiler is constructed in the  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ -hybrid model and in a very similar way to the two-party compiler. The compiler itself is described in Figure 4.18. We note that each party has to commit and prove statements to all other parties during the protocol execution. In order to do this, each party  $P_i$  uses a separate invocation of  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ , with session ID  $\text{sid}_i$ . (Also here, the protocol should make sure that these session ID's are unique as long as the session ID of the current copy of  $\text{Comp}(\Pi)$  is unique. This can be done by setting  $\text{sid}_i = \text{sid} \circ i$  where  $\text{sid}$  is the session ID of the current copy of  $\text{Comp}(\Pi)$ ). The relations parameterizing these functionalities are natural extensions of the relations parameterizing the relations  $\mathcal{F}_{\text{CP}}^1$  and  $\mathcal{F}_{\text{CP}}^2$  in the two-party compiler of Figure 4.11. In addition to commit-and-prove, the multi-party compiler here uses “standard” UC commitments for coin-tossing. However, in order to remain in the  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ -hybrid model (and not the  $(\mathcal{F}_{\text{CP}}^{1:\text{M}}, \mathcal{F}_{\text{MCOM}}^{1:\text{M}})$ -hybrid model), these commitments are implemented by  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$  where the relation used is the identity relation (and so a proof is just a decommitment). Once again, the different invocations of  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$  are distinguished by unique identifiers (in the coin-tossing used for generating  $P_j$ 's random tape,  $P_i$  uses  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$  with session ID  $\text{sid}_{i,j} = \text{sid} \circ i \circ j$ ). Notice that protocol  $\text{Comp}(\Pi)$  essentially broadcasts (via  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ ) each message that was sent in  $\Pi$ , even if this message was originally sent only to a single party. This is done to provide all parties with consistent views of the execution; it clearly has no negative effect on the security of the protocol (since the adversary anyway sees all messages).

Implicit in the protocol specification is the fact that all parties only consider messages that are associated with the specified session identifiers and referring to the same set of parties  $\mathcal{P}$ . All other messages are ignored. As in the two-party case, we assume that  $\Pi$  is such that the parties copy their input tape onto an internal work tape when first activated.

**Comp( $\Pi$ )**

Party  $P_i$  proceeds as follows (the code for all other parties is analogous):

1. **Random tape generation:** When activating **Comp( $\Pi$ )** for the first time with session identifier  $sid$  and set  $\mathcal{P}$  or parties, party  $P_i$  proceeds as follows. For every party  $P_j$ , the parties run the following procedure in order to choose a random tape for  $P_j$ :
  - a)  $P_i$  chooses  $r_i^j \in_R \{0, 1\}^k$  and sends (**commit**,  $sid_{i,j}$ ,  $\mathcal{P}$ ,  $r_i^j$ ) to  $\mathcal{F}_{CP}^{1:M}$ .
  - b)  $P_i$  receives (**receipt**,  $sid_{k,j}$ ,  $P_k$ ,  $\mathcal{P}$ ) for every  $P_k \in \mathcal{P}$ .  $P_i$  also receives (**receipt**,  $sid_j$ ,  $P_j$ ,  $\mathcal{P}$ ), where  $P_j$  is the party for whom the random tape is being chosen.  $P_i$  then uses  $\mathcal{F}_{CP}^{1:M}$  to decommit to its value  $r_i^j$ . That is,  $P_i$  sends (**CP-prover**,  $sid_{i,j}$ ,  $r_i^j$ ) to  $\mathcal{F}_{CP}^{1:M}$ , where the relation parameterizing the  $\mathcal{F}_{CP}^{1:M}$  functionality with identifier  $sid_{i,j}$  is the identity relation (i.e.,  $\mathcal{F}_{CP}^{1:M}$  sends (**CP-proof**,  $sid_{i,j}$ ,  $r_i^j$ ) if  $r_i^j$  was the value previously committed to; it thus serves as a regular commitment functionality).
  - c)  $P_i$  receives (**CP-proof**,  $sid_{k,j}$ ,  $r_k^j$ ) messages for every  $k \neq j$  and defines the string  $s_j = \bigoplus_{k \neq j} r_k^j$ . (The random tape for  $P_j$  is defined by  $r_j = r_j^j \oplus s_j$ .)

When choosing a random tape for  $P_i$ , the only difference for  $P_i$  is that it sends its random string  $r_i^i$  to  $\mathcal{F}_{CP}^{1:M}$  indexed by session-identifier  $sid_i$  and it does not decommit (as is understood from  $P_j$ 's behaviour above).

2. **Activation due to new input:** When activated with input  $(sid, x)$ , party  $P_i$  proceeds as follows.
  - a) *Input commitment:*  $P_i$  sends (**commit**,  $sid_i$ ,  $\mathcal{P}$ ,  $x$ ) to  $\mathcal{F}_{CP}^{1:M}$  and adds  $x$  to the list of inputs  $\bar{x}_i$  (this list is initially empty and contains  $P_i$ 's inputs from all the previous activations of  $\Pi$ ). (At this point all other parties  $P_j$  receive the message (**receipt**,  $sid_i$ ,  $P_i$ ,  $\mathcal{P}$ ) from  $\mathcal{F}_{CP}^{1:M}$ .  $P_i$  then proceeds to the next step.)
  - b) *Protocol computation:* Let  $\bar{m}$  be the series of  $\Pi$ -messages that were broadcast in all the activations of  $\Pi$  until now ( $\bar{m}$  is initially empty).  $P_i$  runs the code of  $\Pi$  on its input list  $\bar{x}_i$ , messages  $\bar{m}$ , and random tape  $r_i$  (as generated above). If  $\Pi$  instructs  $P_i$  to broadcast a message,  $P_i$  proceeds to the next step (Step 2c).
  - c) *Outgoing message transmission:* For each outgoing message  $m$  that  $P_i$  sends in  $\Pi$ ,  $P_i$  sends (**CP-prover**,  $sid_i$ ,  $(m, s_i, \bar{m})$ ) to  $\mathcal{F}_{CP}^{1:M}$  with a relation  $R_\Pi$  defined as follows:

$$R_\Pi = \{((m, s_i, \bar{m}), (\bar{x}_i, r_i^i)) \mid m = \Pi(\bar{x}_i, r_i^i \oplus s_i, \bar{m})\}$$

In other words,  $P_i$  proves that  $m$  is the correct next message generated by  $\Pi$  when the input sequence is  $\bar{x}_i$ , the random tape is  $r_i = r_i^i \oplus s_i$  and the series of broadcast  $\Pi$ -messages equals  $\bar{m}$ . (Recall that  $r_i^i$  and all the elements of  $\bar{x}_i$  were committed to by  $P_i$  in the past using commit activations of  $\mathcal{F}_{CP}^{1:M}$  with identifier  $sid_i$ , and that  $s_i$  is the random-string derived in the random tape generation for  $P_i$  above.)

3. **Activation due to incoming message:** See Figure 4.19.
4. **Output:** See Figure 4.19.

**Fig. 4.18.** The compiled protocol **Comp( $\Pi$ )**

**Comp( $\Pi$ )**(continued from Figure 4.18)

3. **Activation due to incoming message:** Upon receiving a message (CP-proof,  $sid_j$ ,  $(m, s_j, \overline{m})$ ) that is sent by  $P_j$ , party  $P_i$  first verifies that the following conditions hold (note that  $\mathcal{F}_{CP}^{1:M}$  with  $sid_j$  is parameterized by the same relation  $R_\Pi$  as  $\mathcal{F}_{CP}^{1:M}$  with  $sid_i$  above):
  - $s_j$  is the random string that is derived in the random tape generation for  $P_j$  above.
  - $\overline{m}$  equals the series of  $\Pi$ -messages that were broadcast in all the activations until now. ( $P_i$  knows these messages because all parties see all messages sent.)

If any of these conditions fail, then  $P_i$  ignores the messages. Otherwise,  $P_i$  appends  $m$  to  $\overline{m}$  and proceeds as in Steps 2b and 2c above.
4. **Output:** Whenever  $\Pi$  generates an output value, **Comp( $\Pi$ )** generates the same output value.

**Fig. 4.19.** The compiled protocol **Comp( $\Pi$ )** (continued)

We now prove that the compiler of Figure 4.18 achieves the desired result:

**Proposition 4.9.8** (multi-party protocol compiler): *Let  $\Pi$  be a multi-party protocol and let **Comp( $\Pi$ )** be the protocol obtained by applying the compiler of Figure 4.18 to  $\Pi$ . Then, for every malicious adversary  $\mathcal{A}$  that interacts with **Comp( $\Pi$ )** in the  $\mathcal{F}_{CP}^{1:M}$ -hybrid model there exists a semi-honest adversary  $\mathcal{A}'$  that interacts with  $\Pi$  in the plain real-life model, such that for every environment  $\mathcal{Z}$ ,*

$$\left\{ \text{REAL}_{\Pi, \mathcal{A}', \mathcal{Z}}(k, z) \right\}_{k \in \mathbf{N}, z \in \{0,1\}^*} \equiv \left\{ \text{HYBRID}_{\text{Comp}(\Pi), \mathcal{A}, \mathcal{Z}}^{\mathcal{F}_{CP}^{1:M}}(k, z) \right\}_{k \in \mathbf{N}, z \in \{0,1\}^*}$$

**Proof (sketch).** The proof sketch is very similar to the proof of Proposition 4.8.1 for the two-party case. We construct a semi-honest adversary  $\mathcal{A}'$  from the malicious adversary  $\mathcal{A}$ . Adversary  $\mathcal{A}'$  runs the protocol  $\Pi$  while internally simulating an execution of **Comp( $\Pi$ )** for  $\mathcal{A}$ . The key point in the simulation is that  $\mathcal{A}'$  is able to complete the simulation in spite of the fact that, being semi-honest, it cannot diverge from the protocol specification. This is so since  $\mathcal{A}$  is forced to send all messages via  $\mathcal{F}_{CP}^{1:M}$  that verifies their correctness. Thus, essentially,  $\mathcal{A}$  must behave in a semi-honest way and can be simulated by a truly semi-honest party  $\mathcal{A}'$ . (Of course,  $\mathcal{A}$  is not semi-honest and can send arbitrary messages. However, since all invalid messages are ignored by  $\mathcal{F}_{CP}^{1:M}$  in **Comp( $\Pi$ )**, they do not cause any problem.)  $\mathcal{A}'$  runs a simulated copy of  $\mathcal{A}$ , and proceeds as follows:

**Simulating the communication with  $\mathcal{Z}$ :** The input values received by  $\mathcal{A}'$  from  $\mathcal{Z}$  are written on  $\mathcal{A}'$ 's input tape, and the output values of  $\mathcal{A}$  are copied to  $\mathcal{A}'$ 's own output tape.

**Simulating the “random tape generation” phase:** When the first activation of  $\Pi$  takes place,  $\mathcal{A}'$  simulates the random tape generation phase of  $\text{Comp}(\Pi)$  for  $\mathcal{A}$ . We describe  $\mathcal{A}'$ 's simulation of the random tape generation for a party  $P_j$  (this simulation is repeated for every  $j$ ). We differentiate between the case that  $P_j$  is honest and  $P_j$  is corrupted:

1. *Party  $P_j$  is not corrupted:* Adversary  $\mathcal{A}'$  internally hands  $\mathcal{A}$  the message  $(\text{receipt}, \text{sid}_j, P_j, \mathcal{P})$  from  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$  (this refers to  $P_j$ 's commitment). In addition,  $\mathcal{A}'$  simulates all the  $(\text{receipt}, \text{sid}_{k,j}, P_k, \mathcal{P})$  messages that  $\mathcal{A}$  expects to receive from  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ .  $\mathcal{A}'$  completes the simulation by passing  $\mathcal{A}$  the “decommit” messages  $(\text{CP-proof}, \text{sid}_{k,j}, r_k^j)$  for every uncorrupted party  $P_k$ .  $\mathcal{A}'$  also obtains  $(\text{commit}, \text{sid}_{k,j}, r_k^j)$  messages from  $\mathcal{A}$  for the corrupted parties  $P_k$  as well as the respective decommit messages  $(\text{CP-prover}, \text{sid}_{k,j}, r_k^j)$ .  $\mathcal{A}'$  computes  $s_j$  as in the protocol and records this value.
2. *Party  $P_j$  is corrupted:* Let  $r_j$  be the random tape of the semi-honest party  $P_j$  in protocol  $\Pi$ . Now, as above, for every uncorrupted party  $P_k$ , adversary  $\mathcal{A}'$  passes  $\mathcal{A}$  the  $(\text{receipt}, \text{sid}_{k,j}, P_k, \mathcal{P})$  messages that  $\mathcal{A}$  expects to receive from  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ .  $\mathcal{A}'$  also obtains  $(\text{commit}, \text{sid}_{k,j}, r_k^j)$  messages from  $\mathcal{A}$  (for every corrupt  $P_k$ ) and corrupted  $P_j$ 's commitment  $(\text{commit}, \text{sid}_j, r_j^j)$ . Notice that at this point,  $\mathcal{A}$  is bound to all the  $r_k^j$  values of the corrupted parties, whereas  $\mathcal{A}'$  is still free to choose the analogous values for the uncorrupted parties. Therefore, in the “decommitment” part of the phase,  $\mathcal{A}'$  chooses the uncorrupted parties' values so that  $\bigoplus_{k=1}^{\ell} r_k^j = r_j$  where  $r_j$  is the random tape of the external  $P_j$  in  $\Pi$ . (Thus,  $\mathcal{A}'$  forces  $\mathcal{A}$  into using  $r_j$  for the malicious  $P_j$  in  $\text{Comp}(\Pi)$  as well.)

**Simulating an activation due to a new input:** When the first message of an activation of  $\Pi$  is sent,  $\mathcal{A}'$  internally simulates for  $\mathcal{A}$  the appropriate stage in  $\text{Comp}(\Pi)$ . This is done as follows. Let  $P_i$  be the activated party with a new input. If  $P_i$  is not corrupted, then  $\mathcal{A}'$  internally passes  $\mathcal{A}$  the message  $(\text{receipt}, \text{sid}_i, P_i, \mathcal{P})$  that  $\mathcal{A}$  expects to receive from  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ . If  $P_i$  is corrupted, then  $\mathcal{A}'$  receives a message  $(\text{commit}, \text{sid}_i, \mathcal{P}, x)$  from  $\mathcal{A}$  (who controls  $P_i$ ).  $\mathcal{A}'$  adds  $x$  to its list  $\bar{x}_i$  of inputs received from  $P_i$  and passes  $\mathcal{A}$  the string  $(\text{receipt}, \text{sid}_i, P_i, \mathcal{P})$ . Furthermore,  $\mathcal{A}'$  sets  $P_i$ 's input tape to equal  $x$ . (Recall that in the semi-honest model,  $\mathcal{A}'$  can modify the input that the environment writes on a corrupted party's input tape.)

**Dealing with messages sent by honest parties:** If an uncorrupted party  $P_i$  sends a message  $m$  in  $\Pi$  to a corrupted party (controlled by  $\mathcal{A}'$ ), then  $\mathcal{A}'$  prepares a simulated message of  $\text{Comp}(\Pi)$  to give to  $\mathcal{A}$ . Specifically,  $\mathcal{A}'$  passes  $\mathcal{A}$  the message  $(\text{CP-proof}, \text{sid}_i, (m, s_i, \bar{m}))$  as expected from  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ .

**Dealing with messages sent by corrupted parties:** When  $\mathcal{A}$  sends a  $\text{Comp}(\Pi)$ -message from a corrupted party,  $\mathcal{A}'$  translates this to the appropriate message in  $\Pi$ . That is,  $\mathcal{A}'$  obtains  $(\text{CP-prover}, \text{sid}_i, (m, s'_i, \bar{m}))$

from  $\mathcal{A}$ , in the name of a corrupted party  $P_i$ . Then,  $\mathcal{A}'$  checks that the series of broadcasted  $\Pi$ -messages is indeed  $\overline{m}$ .  $\mathcal{A}'$  also checks that  $s'_i = s_i$ , where  $s_i$  is the value defined in the random tape generation phase. Finally,  $\mathcal{A}'$  checks that  $m = \Pi(\overline{x}_i, s_i \oplus r_i^i, \overline{m})$ . If yes, then it delivers the message written on semi-honest party  $P_i$ 's outgoing communication tape in  $\Pi$ . Otherwise,  $\mathcal{A}'$  does nothing.

**Dealing with corruption of parties:** When the simulated  $\mathcal{A}$  internally corrupts a party  $P_i$ , adversary  $\mathcal{A}'$  externally corrupts  $P_i$  and obtains all of its past inputs, outputs and random tapes in  $\Pi$ . Then,  $\mathcal{A}'$  prepares a simulated internal state of  $P_i$  in  $\text{Comp}(\Pi)$ . The only additional state that  $P_i$  has in  $\text{Comp}(\Pi)$  is the random string  $r_i^i$  for the random tape generation phase. Since the string  $s_i$  is public and fixed,  $\mathcal{A}'$  sets  $r_i^i$  so that  $r_i = s_i \oplus r_i^i$ , where  $r_i$  is  $P_i$ 's random tape in  $\Pi$ .

We now claim that  $\mathcal{Z}$ 's view of an interaction with  $\mathcal{A}'$  and  $\Pi$  is distributed identically to its view of an interaction with  $\mathcal{A}$  and  $\text{Comp}(\Pi)$ . This follows from the same observations as in the two-party case. The key points are as follows. For every corrupted  $P_i$ , the semi-honest adversary  $\mathcal{A}'$  can force  $\mathcal{A}$  into using the exact random tape of  $P_i$  in  $\Pi$ . Furthermore, any modification of the inputs made by  $\mathcal{A}$  can also be carried out by  $\mathcal{A}'$ . We therefore have that if  $\mathcal{A}$  follows the protocol specification with respect to these inputs and random tapes, then the semi-honest parties in  $\Pi$  will send exactly the same messages as the malicious parties in  $\text{Comp}(\Pi)$ . The proof is concluded by observing that  $\mathcal{A}$  must follow the protocol specification because the  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$  functionality enforces this. Thus,  $\mathcal{A}'$ 's checks of correctness in the simulation perfectly simulate the behaviour of  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$  in a hybrid execution. Finally, the internal state revealed to  $\mathcal{A}$  in the case of a corruption is exactly as it expects to see. This completes the proof. ■

**Conclusions.** Combining the semi-honest protocol of Proposition 4.9.1 with the compilation obtained in Proposition 4.9.8, we have that for any multi-party ideal functionality  $\mathcal{F}$ , there exists a protocol that UC realizes  $\mathcal{F}$  in the  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$ -hybrid model (in the presence of malicious adversaries). Combining this with the fact that  $\mathcal{F}_{\text{CP}}^{1:\text{M}}$  can be UC realized in the  $(\mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{ZK}}^{1:\text{M}})$ -hybrid model (Proposition 4.9.7), and using the UC composition theorem (Theorem 4.3.3), we obtain universally composable general two-party computation in the  $(\mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{ZK}}^{1:\text{M}})$ -hybrid model. That is:

**Theorem 4.9.9** *Assume that trapdoor permutations exist. Then, for any well-formed multi-party ideal functionality  $\mathcal{F}$ , there exists a non-trivial protocol that UC realizes  $\mathcal{F}$  in the  $(\mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{ZK}}^{1:\text{M}})$ -hybrid model in the presence of malicious, static adversaries. Furthermore, if two-party augmented non-committing encryption protocols exist, then for any adaptively well-formed multi-party ideal functionality  $\mathcal{F}$ , there exists a non-trivial protocol that UC realizes  $\mathcal{F}$  in the  $(\mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{ZK}}^{1:\text{M}})$ -hybrid model in the presence of malicious, adaptive adversaries.*

By Proposition 4.9.6, the zero-knowledge functionality  $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$  can be UC realized in the  $(\mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{COM}}^{1:\text{M}})$ -hybrid model. In addition, by Proposition 4.9.5, the multi-session  $\mathcal{F}_{\text{MCOM}}^{1:\text{M}}$  can be UC realized in the  $(\mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{BC}})$ -hybrid model. Applying the universal composition with joint state theorem (Theorem 4.3.4), we obtain that  $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$  can be UC realized in the  $(\mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{BC}})$ -hybrid model (with a single copy of the CRS). Now, recall that by Proposition 4.9.4, the broadcast functionality  $\mathcal{F}_{\text{BC}}$  can be UC realized in a point-to-point network. Therefore, Theorem 4.9.9 can be restated in the  $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$ -hybrid model only, and  $\mathcal{F}_{\text{ZK}}^{1:\text{M}}$  can in turn be UC realized in the  $\mathcal{F}_{\text{CRS}}$ -hybrid model only. In conclusion, we obtain:

**Theorem 4.9.10** (Theorem 4.2.3 – restated): *Assume that trapdoor permutations exist. Then, for any well-formed multi-party ideal functionality  $\mathcal{F}$ , there exists a non-trivial protocol that UC realizes  $\mathcal{F}$  in the  $\mathcal{F}_{\text{CRS}}$ -hybrid model in the presence of malicious, static adversaries. Furthermore, if two-party augmented non-committing encryption protocols exist, then for any adaptively well-formed multi-party ideal functionality  $\mathcal{F}$ , there exists a non-trivial protocol that UC realizes  $\mathcal{F}$  in the  $\mathcal{F}_{\text{CRS}}$ -hybrid model in the presence of malicious, adaptive adversaries. In both cases, the protocol uses only a single copy of  $\mathcal{F}_{\text{CRS}}$ .*



# Index

- Abort
  - not unanimous 49, 57, 58
  - unanimous 48, 54–56
- Adaptive adversary 5, 83
- Asynchronous 59, 81, 94
- Augmented non-committing encryption 83, 112
- Authenticated Byzantine Agreement
  - definition 26
  - parallel self composition 29–35
  - sequential self composition 35–38
  - with unique identifiers 38–43
- Bounded concurrency 9, 11, 12, 79
- Broadcast with abort 61–62
- Byzantine Agreement 21
  - composition 28
  - definition 26
- Byzantine Generals 21
- Circuit evaluation protocol 117–119, 163–169
- Commit and prove 143–155
  - one-to-many extension 174–179
- Commitments
  - adaptively secure 130
  - equivocal 128
  - extractability 128
  - Feige-Shamir trapdoor scheme 129
  - UC one-to-many extension 172–173
  - universally composable 128–140
- Common reference string model 13
- Composition
  - feasibility 10–15
  - scheduling 9
  - stateless 8
  - taxonomy 8–10
- Computational model 6
  - stand-alone definition 60
- Crusader Agreement 62
- Environmental security *see* Universal composability
- Fairness 2, 47
  - complete 48, 54
  - none 48, 56, 58
  - partial 48, 55, 57
- Functionalities
  - multi-session extension 104
  - reactive 108
  - well-formed 106
- General composition
  - feasibility 13–15
  - informal definition 9
- Ideal/real model paradigm 4
- Information-theoretic model 6
  - stand-alone definition 60
- Invertible samplability 112
- Malicious adversary 6
- Non-malleability 12
- Non-trivial protocols 99
- Oblivious key generation 112
- Oblivious transfer
  - self composition 11
  - universally composable 108–116
- Plain model 10
- Private channels model 6
- Protocol compilation 155–161, 179–183
- Security properties 2–3
- Self composition
  - feasibility 10–13
  - informal definition 9

- Semi-honest adversary 5, 99
- Session identifiers 24, 102
- Signature schemes
  - definition 39
  - generalized 39
- Stand-alone model of computation 7
- Static adversary 5, 99
- Synchronous 25, 58, 59
  
- Universal composability 13, 81–184
  - composition theorem 100–102
  - definition 93–100
  - joint state 102–106
  - overview 84–86
  
- Universally composable broadcast 169–172
  
- Weak Byzantine Agreement 52, 62
  
- Zero-knowledge
  - feasibility under composition 10
  - UC one-to-many extension 173–174
  - universally composable 141–143