UNIVERSITY OF CALIFORNIA,
IRVINE

Techniques for Almost-Asynchronous Distributed Cryptography

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Benjamin Terner

2023

DEDICATION

To Mom and Dad

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

In many ways, Karim Eldefrawy is my “true” PhD advisor. Thank you for supporting me when I needed it most. You gave me a better education than I could have ever dreamed possible. Ben Kreuter inspired my interest in cryptography, gave me the opportunities that initiated my career, and became an incredible friend and mentor throughout the process. You’ve never led me astray. Pratik Soni became my partner in learning early in this journey. Your support throughout the highs and lows has been a constant force driving me forward. We are forever partners in success.

I’ve had a number of academic advisors along the way, and have learned many different things from each of them. Special thanks to Gene Tsudik, who put me in all the right places at all the right times, opened incredible doors for me, and pushed me over the finish line. Thank you to abhi shelat, Rachel Lin, and Stefano Tessaro for opening my eyes to the richness and depth to be plundered in cryptography, and for teaching me so much.

I’ve learned tremendous amounts of cryptography from all of the mentors I’ve collected along the way, including Juan Garay, Stanislaw Jarecki, Julian Loss, Victor Shoup, Nigel Smart, and Moti Yung. In particular, Moti’s tremendous and unwavering encouragement has driven my development as a scientist. Juan took a chance on me in a challenging time and gave me a crucial boost that made the rest possible.

My labmates and friends at UVa, UCSB, UW, and UCI, a countable set that is still too large to enumerate, have kept me laughing. Kelong Cong and Sashidhar Jakkamsetti were excellent collaborators who kept me on my toes. I am so grateful for my lasting friendships with Nathan Brunelle and Binyi Chen; I still learn computing from the two of you just by watching. Thanks to all my colleagues at SRI International, especially Linda Briesemeister, Nick Genise, Bob Halley, Briland Hitaj, Tim McCarthy, and Hadas Zeilberger. I thoroughly enjoyed our time together. Thank you to Marty Humphrey, one of my favorite professors, whose challenges, counsel, and encouragement set the stage for everything that would happen after. And thank you to Elise deGoege Dorough, whose support kept me on the right path.

Finally, thanks to my brothers Steven, Jeremy, and Zachary. You are my moral compasses and my inspiration.

VITA

Benjamin Terner

EDUCATION

Doctor of Philosophy in Computer Science	**2023**
University of California, Irvine	*Irvine, California*
Master of Science in Computer Science	**2015**
University of Virginia	*Charlottesville, Virginia*
Bachelor of Science in Computer Science	**2014**
University of Virginia	*Charlottesville, Virginia*

ABSTRACT OF THE DISSERTATION

Techniques for Almost-Asynchronous Distributed Cryptography

By

Benjamin Terner

Doctor of Philosophy in Computer Science

University of California, Irvine, 2023

Gene Tsudik, Chair

This dissertation addresses "distributed cryptography" in the presence of "almost" asynchrony. Many machines collaborate to accomplish some goal – whether agreeing on a bit, secure group messaging, or computing a function of their inputs – over a network that is not guaranteed to deliver messages within a known time constraint. The applications in this dissertation explore new tradeoffs on the spectrum of synchronous to asynchronous executions, adversarial advantages, and efficiency.

First, we explore consensus protocols in both a poorly-studied permissioned model and a novel permissionless model. Consensus protocols require that all parties agree on the *same* output bit, and that the output is constrained by their inputs. In the permissioned model, we study optimal constraints on highly efficient, sublinear-round protocols, where somewhat faulty but otherwise correct parties must produce outputs consistent with fully functional, honest parties. In the permissionless model we ask under what synchronization conditions consensus is possible at all, and illustrate the power of a technique common to most Proof-of-X primitives to achieve consensus in a very weakly synchronized model.

We then explore asynchronous, concurrent group messaging and provide a framework for reasoning about group keys that simplifies their security proofs. In concurrent group messaging, parties send application messages while also continuously updating the group key;

in an asynchronous network, there may be many simultaneous latest group keys due to concurrent evolutions by different parties. Using our framework, we prove security of a novel protocol that achieves better security properties in asynchronous networks than any previous protocol.

Finally, we analyze timed cryptographic primitives and introduce a novel model for proving security of arbitrarily composed timed primitives. The model permits an adversary to operate an asynchronous network, but the adversary is bounded in depth to model realistic passage of time.

For each application, we discuss or prove how the constraints applied to the asynchronous model make the problem tractable, and explain how the resulting models represent progress towards cryptography that could be deployed.

Chapter 1

Introduction

This dissertation considers a range of applications in cryptography which use different models, different primitives, and different techniques to prove security. Chapters 4 and 5 consider scenarios for consensus in permissioned and permissionless networks, respectively, in which all participants produce the *same* output bit as a function of their inputs. Chapter 6 considers group messaging, where participants continuously agree on a set of evolving cryptographic keys used to exchange messages securely. Chapter 7 develops a theory for the composition of timed cryptographic primitives, such as protocols that include time-lock puzzles. Despite the variety in these applications, the included research share themes of powerful adversaries and adverse networks.

The asynchronous model is the most general – and most adverse – model for communication and computation used in cryptography. Asynchrony is formally modeled in Chapter 3. Depending on the application, it might mean that the parties executing a protocol are not running at the same speed, or that they receive messages out of order and after arbitrary delays, or that an adversary attacking a protocol can pause its attack to run side sessions of similar protocols, or that all of these adverse conditions are present simultaneously. Networks are modeled as asynchronous in cryptography because if a protocol is proven secure in an asynchronous network, then it will work in any real-world network that guarantees eventual message delivery. In practice, eventual delivery is guaranteed by a protocol like TCP [60] over an unreliable network. However, asynchronous protocols are challenging to design, and a number of foundational results separate the asynchronous model from synchronous models, both via impossibility results and by showing differences in tolerable corruption threshold (such as [58, 67] and many more).

The primary theme of this dissertation is that the network is "almost" asynchronous. "Almost" asynchronous means a different thing for each application, but in each case it constrains the network assumptions or adversary from a fully asynchronous environment in order to make the problem more tractable, while still admitting a very powerful adversary that models a realistic network.

– The model of Chapter 4 is synchronous, but it adds send corruptions and receive corruptions (in addition to standard byzantine corruptions) to the adversary's corruption budget, which allow the adversary to drop messages sent to otherwise honest-behaving parties. This makes the network more adverse than the standard synchronous model.

– In Chapter 5, the only form of synchronization available to the parties is an upper bound on the rate at which an abstraction designed to capture many forms of Proof-of-X (including Proof of Work and Proof of Stake) enter the network relative to the time it takes for messages to be delivered.

– Chapter 6 studies an asynchronous network, and the protocol in Section 6.4 includes an optimization that trades synchrony for security or correctness.

– Chapter 7 analyzes a setting for multi-party computation where the adversary can delay messages arbitrarily and run concurrent executions. The adversary is constrained computationally by not permitting it to arbitrarily pause an attack to run a side session.

Despite these constraints on asynchrony, the hypothetical adversary in each chapter is provided additional advantages to be as powerful or more powerful than in comparable models. Even with such a powerful adversary, the works present new feasibility results for their respective computational problems:

- In Chapter 4, the adversary is permitted to read messages that are sent over the network, and then choose to remove messages and corrupt the senders. The model allows for an even stronger adversary than comparable work because the attacker can corrupt the sender of messages *without depleting its budget for full malicious parties.* Additionally, the chapter argues that additionally allowing a "send corruption" category is in some cases (explained in Section 4.1.1) just as pathological as a fully malicious corruption. Nevertheless, Chapter 4 presents the first constant-round consensus protocol that allows a majority of online parties to be dishonest.

- In Chapter 5, the adversary has full information about all of the parties – including their internal states. It can spawn arbitrarily many parties, it can corrupt arbitrarily many parties, and it even chooses which parties send messages. It is only constrained by how many protocol messages the corrupt parties send relative to the honest parties. Despite this powerful adversary, Chapter 5 presents a consensus protocol in a model designed to abstract many forms of Proof-of-X.

- In Chapter 6, an adaptive adversary delivers messages asynchronously, and some "insider attacks" [10, 12] are allowed. The protocol advances the state of the art of concurrency for secure group messaging by allowing both concurrent updates and concurrent sessions.

- In Chapter 7, an MPC adversary is constrained in depth by a fine-grained polynomial (rather than an arbitrary polynomial, the literature standard) to model the depth-sensitivity of security arguments for timed cryptographic primitives. These constraints lead to new definitions and composition theorems for security of timed cryptographic primitives.

The final theme of this dissertation is that each contribution is born from a change of perspective on the problem from the common literature:

- Chapter 4 adopts a "realistic" corruption model (introduced by [131]), where parties that are operating honestly by following the protocol specification are hampered by network issues. The model's novelty – and its challenge – is that these somewhat-faulty parties should still be considered in the security definitions as if they are not faulty.
- Chapter 5 shifts focus from the number of *parties* participating in a protocol to the number of *messages* that are sent. In the permissioned model, security arguments bound the proportion of honest and corrupt parties, and each party sends as many messages as it wants. In the permissionless model, security arguments bound the number of messages that are sent in the protocol, but allow unlimited participants. (The number of messages is constrained by making it "hard" to send a message due to some physical constraint.)
- In Chapter 6, the definitions of forward secrecy and post-compromise secrecy – the abilities to learn the contents of previous or future keys, respectively – are unified in a novel key lattice framework, and security is proven based on the framework.
- Chapter 7 observes that the existing literature on timed cryptographic primitives do not prove security in a model consistent with their constructions. It therefore presents a falsifiable model for security of timed primitives, and proceeds through the implications of the corresponding observations all the way to composition of multi-party protocols.

Constraining Asynchrony The protocols and frameworks studied in this thesis overcome powerful adversaries and challenging networks by applying structure that the adversary cannot break or overcome.

The protocol presented in Section 4.3 uses quorum techniques to make inferences about messages that must have been sent and received by all honest parties; to complete the protocol, we adapt standard quorum techniques to handle both send- and receive- corruptions.

The protocol in Section 5.5 builds a graph structure that orders the messages received by the honest parties, and uses the known upper bound on the rate of PoX to make inferences about the ordering of events based on this graph. Intuitively, the protocol exploits the relationship between the *collective work* of all parties and the (unknown) synchronization constant of the network.

The protocol in Section 6.4 builds a lattice structure to track the evolution of the group key. The scheme permits concurrent key updates by multiple honest parties by allowing multiple valid group keys to exist simultaneously. To prevent the state tracked by every party from growing exponentially, the scheme employs a homomorphic structure over key evolutions. The scheme additionally includes a mechanism for honest parties to track which keys might be used or will never be used again, even in an asynchronous network.

The granular time control of the model in Chapter 7 constrains the total depth of the adversary's side-sessions. In comparison to the UC [39], in which the adversary can run arbitrary (polynomial) computations and can likewise pause its attack on a particular protocol, this limits the computational power of the adversary.

Chapter 2

Related Work

This chapter overviews work related to the projects featured in this dissertation. The problems discussed in this chapter are defined in Chapter 3.

2.1 Consensus

The problem of *consensus*, in which a group of parties with one-bit inputs must agree on a single-bit output, has been studied for over forty years [14, 54, 107]. The following overview of expanded synchronous models compares results that are relevant to the constructions in Chapters 4 and 5, including recent work in mixed corruption models, constant-round protocols, and permissionless consensus.

2.1.1 Mixed and Hybrid Fault Models

Toward a realistic failure model in which a majority of parties may be corrupted in some way, a line of work has explored mixed models that account for both crash faults and byzantine

Protocol	**Faults**	**# Rounds**
Modified DS (Section 4.1.1)	Send & Byz: $2t_{\mathsf{snd}} + 2t_{\mathsf{byz}} < n$	$O(n)$
GP [72]	Crash & Byz: $t_{\mathsf{cra}} + 3t_{\mathsf{byz}} < n$	$O(n)$
ZHM [131]	Receive, Send, Byz: $t_{\mathsf{snd}} + t_{\mathsf{rcv}} + 3t_{\mathsf{byz}} < n$	$O(n)$
ELT [63]	Receive, Spotty Send, Byz: $t_{\mathsf{rcv}} + t_{\mathsf{snd}} + 2t_{\mathsf{byz}} < n$	$\widehat{O(1)}$
ELT [63]	Receive, Send, Byz: $t_{\mathsf{rcv}} + 2t_{\mathsf{snd}} + 2t_{\mathsf{byz}} < n$	$\widehat{O(1)}$
LS [90]	Receive, Send, Byz: $t_{\mathsf{rcv}} + t_{\mathsf{snd}} + 2t_{\mathsf{byz}} < n$	$\widehat{O(1)}$

Table 2.1: Comparison with relevant consensus protocols in mixed corruption models.

faults. In the error-free setting, Garay and Perry [72] and Altmann, Fitzi, and Maurer [6] show that byzantine agreement is possible if and only if $n > t_{\mathsf{cra}} + 3t_{\mathsf{byz}}$, where t_{cra} bounds the number of crashed parties and t_{byz} bounds the number of byzantine parties. In the asynchronous model, Backes and Cachin [15] showed that reliable broadcast within the mixed model is possible if and only if $n > 2t_{\mathsf{cra}} + 3t_{\mathsf{byz}}$. For byzantine agreement, Kursawe [84] developed a protocol for the same bound ($n > 2t_{\mathsf{cra}} + 3t_{\mathsf{byz}}$) assuming a public key infrastructure (PKI). Recent work in the dishonest majority setting by Wan et al. [125] showed round efficient broadcast protocols in the majority dishonest setting. Expanding further into even more realistic failures, Zikas, Hauser and Maurer (ZHM) [131] gave a protocol in the error-free synchronous model for $n > t_{\mathsf{rcv}} + t_{\mathsf{snd}} + 3t_{\mathsf{byz}}$, where t_{rcv} bounds the number of receive corruptions and t_{snd} bounds the number of send corruptions. The work by ZHM introduced parties which may be faulty *but the faulty parties' outputs must be consistent with honest parties' outputs* because they otherwise behave honestly. (In all other corruption models, the output of any faulty party need not be considered towards the correctness definition.) Subsequent to the publication of the work presented in Chapter 4, Abraham et al. [4] partially repeated the same results, and expanded to multi-valued consensus and state machine replication with optimal thresholds of $n > 2t_{\mathsf{byz}} + t_{\mathsf{cra}}$.

Table 2.1 overviews the results most relevant to Chapter 4: consensus protocols in mixed corruption models. The table includes a construction by modifying Dolev-Strong broadcast (Section 4.1.1) via the reduction of consensus to broadcast. In the table, $\widehat{O(R)}$ indicates the round complexity R is given in expectation; otherwise the round complexity is worst-case.

DS denotes Dolev-Strong. Observe that Loss and Stern [90] built on the results of [63] (on which Chapter 4 is based) to provide a corruption-optimal protocol with expected-constant round complexity.

Hybrid Corruption Models

Recent work has generalized corruptions into "sleepy" [106] or "sluggish" [76] faults that combine properties of synchronous and asynchronous parties. With reference to Section 3.1.2, these works model "awake" parties as synchronous and "asleep" parties are asynchronous. They also allow parties to transition between modes subject to some constraints. In general, leader-based protocols fail in the strongly rushing model with adaptive adversaries, as the adversary can always adaptively put the leader to sleep.

In the sluggish model [76], a (mobile) sluggish party can be temporarily disconnected from honest parties due to network partition, but can later rejoin. While disconnected, messages sent by or to a party are delayed until the party is reconnected. However, in that work it is (implicitly) required that at least half of the parties are not sluggish and participate in the protocol at all times, and the adversary is static. Abraham et al. [5], also in the sluggish model, require a majority of online parties to be honest at all times.

Pass and Shi [106] introduce a model in which the adversary can make parties "fall asleep" and later wake them up (i.e., temporarily crash them) at which point all messages that they missed are delivered at once, along with potentially some adversarially-inserted messages. They show that in their model, a protocol requires only that a majority of the *awake* parties are honest at all times. Malkhi et al. [93] consider another similar mixed model of corruption, in which some corrupt parties attack only correctness of the protocol but not liveness, and require that a majority of online players behave honestly at any time.

2.1.2 Dishonest-Majority Protocols

One might expect that because dishonest-majority broadcast protocols tolerate $n > t_{\mathsf{byz}}$ corruptions, they are sufficient for building a consensus protocol tolerating $n > t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$ corruptions via folklore reduction (discussed in Section 4.1.3), which would achieve better corruption tolerance than the construction in Section 4.3. Section 4.1.2 explains why this is not the case. Wan et al. [126] provide an expected constant round protocol for dishonest majority broadcast under a weakly adaptive adversary. Another recent work [125] uses time-lock puzzles to provide a round-efficient broadcast protocol in the presence of dishonest majority and a strongly adaptive adversary. However, as explained in Section 4.1.2, these approaches fail due to partitioning attacks in the presence of send-corrupt parties.

Another recent, dishonest majority byzantine broadcast protocol by Chan et al. [41] runs in $O(\kappa)$ rounds (for a security parameter κ independent of n) and requires only $O(n^2\kappa)$ communication complexity by relying on the player replaceability paradigm. However, this protocol neither works in the strongly rushing adversary model nor achieves optimal corruption thresholds within this round complexity, as it can tolerate only any constant fraction of byzantine corruptions in n, whereas optimality requires corrupting up to $n-1$ out of n parties. (The protocol still works for smaller fractions of honest parties, but becomes as expensive as the standard Dolev-Strong protocol). Other works that use the player replaceability paradigm such as [1] suffer from similar issues.

The work of Loss and Stern [90] built on the work of [63] to provide a corruption-optimal protocol with expected-constant round complexity in the general send-corruption model.

2.1.3 Other Expected Constant-Round Protocols

A number of expected constant-round consensus protocols for the synchronous honest-majority setting consider only byzantine faults. Feldman and Micali [66] gave an expected constant round scheme for $n > 3t_{\mathsf{byz}}$. Katz and Koo [80] later gave a protocol tolerating $n > 2t_{\mathsf{byz}}$, assuming a PKI and signatures. Micali [94] gave another simple protocol assuming $n > 3t_{\mathsf{byz}}$. Abraham et al. [2] gave the most efficient scheme and tolerate a strongly rushing, adaptive adversary for $n > 2t_{\mathsf{byz}}$.

Can ZHM [131] be Adapted to an Expected Constant-Round Protocol?

A natural attempt to achieve sub-linear round consensus tolerating $n > t_{\mathsf{rcv}} + t_{\mathsf{snd}} + 3t_{\mathsf{byz}}$ is to adapt the protocol by Zikas, Hauser and Maurer (ZHM) [131] to an expected constant-round protocol using the standard construction [66, 80] via graded consensus and a common coin protocol. The protocol by ZHM is a linear-round protocol because it depends on the phase-king paradigm [70]; the protocol must run long enough to guarantee that the king is honest in at least one round. To move to an expected-constant round protocol, phase king is replaced with a common coin primitive; however, known common coin constructions require a threshold scheme. In the model of Chapter 4, threshold schemes work only when $n - t_{\mathsf{rcv}} > 2(t_{\mathsf{snd}} + t_{\mathsf{byz}})$, meaning there are more honest parties than send-corrupt or byzantine parties. In the dishonest majority setting where send-corrupt plus byzantine parties outnumber honest parties, the construction suffers from the partitioning attack described above: a group of send-corrupt parties reach the threshold independently of and without knowledge of honest parties, and honest parties therefore output a different coin than send-corrupt parties. The ZHM construction and corruption bound therefore fail in sublinear rounds.

2.1.4 Blockchains and Permissionless Consensus

Several works have studied one-bit consensus using Proof of Work (PoW) and blockchains. Among them, Miller and Laviola [100] were among the first, and showed how to achieve anonymous consensus from moderately hard puzzles when the network delay is known. GKL [71] show how to achieve byzantine agreement in synchronous networks using the "Bitcoin backbone" protocol. EFL [59] construct broadcast and consensus from Proof of Work but require clocks and knowing the network delay.

A number of other works analyze permissionless blockchains with Proof of Work or Proof of Stake, most notably GKL [71], PSs [104] (followed by Pass and Shi [25]) and their respective successors. BMTZ [17] model the Bitcoin protocol in the UC model with dynamic player sets. Ouroboros Praos [51] models a Proof of Stake blockchain with semi-synchronous communication, and Ouroboros Genesis [16] presents their version of dynamic availability. The Ouroboros protocols (weakly) synchronize their participants via a global clock functionality. Fan, Katz, Thai, and Zhou [65] provide a Proof of Stake protocol in which parties mine on multiple unpredictable chains. Among all the works studying PoX consensus protocols, [122] (on which Chapter 5 is based) is the only one in a deterministic model in which the adversary controls allocation.

Majority of Online Resources An important contribution of the line of work of blockchain consensus is that they changed the classical perspective of security arguments from assuming that all parties are always online. Instead of using quorum techniques [40, 63, 66, 80, 84, 94], they rely on other techniques to show, for example, that consensus is achievable if a majority of the *online* parties are honest [25], or if a majority of the *online* computational power is held by honest parties [71, 104].

Consensus on DAGs There are many works that implement agreement on directed acyclic graphs (DAGs). The structure of the DAG protocol presented in Chapter 5 bears some resemblance to SPECTRE [119] and PHANTOM [120], but Chapter 5 considers a much stronger adversary. The same is true for Meshcash [24], who adopt the model of [104]. The Avalanche protocol [115] also employs agreement on a DAG for high throughput of consensus instances for synchronous participants in a permissioned network.

Abraham et al. [3] proposed an incentive-compatible DAG protocol that uses techniques similar to the graph in Section 5.5, and was subsequent to the publication of the work in Chapter 5. The adversary for [3] has full information of which parties will add vertices in the future. Their network is synchronous, with unboundedly many parties. Also subsequent to the initial preprint of Chapter 5, Lewis-Pye and Roughgarden [87] showed that in a permissionless network without a bound on the total participation in the protocol, liveness implies that no protocol is secure; they did not consider bounding instead the rate of special messages, as is the contribution of Chapter 5.

2.2 Concurrent Group Messaging

Group key agreement and group messaging protocols have a long history. Early work focused on generalizing the Diffie-Hellman key exchange protocol [79, 121]. Later work extended the security guarantees (e.g., by providing authentication, forward secrecy, and post-compromise security) [33, 35, 36, 37], and improved performance and added new features (e.g., support for dynamic groups) [34]. Recent research has developed increasingly secure protocols in progressively more asynchronous networks (focusing mostly on concurrency of key updates), and the Messaging Layer Security (MLS) IETF working group[1] is defining standards for security of secure messaging.

[1]https://messaginglayersecurity.rocks/

Ratchet Trees and Propose & Commit:

The family of key agreement protocols popularized by the Message Layer Security (MLS) working group [20] is based on binary trees. These protocols are efficient and secure; they require $O(\log(n))$ public key operations to update a shared key, and they achieve both forward secrecy (FS) and post-compromise security (PCS).

The first in this line of binary-tree protocols introduced asynchronous ratcheting trees (ART) [46, 83]. In ART, the authors constructed the first asynchronous GKA protocol with FS and PCS. The group initiator selects the secret keys for nodes on the tree, and allows the group members to update the secret. TreeKEM [113] evolved ART to introduce support dynamic for groups.

Alwen et al. [9] explained that TreeKEM does not provide adequate FS. Concretely, they formalized the security model and showed that, in the worst case, FS is only achieved if every group member updates their key material, which has a cost of $O(n \log n)$. To achieve optimal FS and reduce the complexity, the authors introduced a modification to TreeKEM, called Re-randomized TreeKEM (RTreeKEM), that uses updatable public key encryption to roll the group key with every encryption and decryption. This reduced the healing cost to $O(\log n)$.

Bienstock, Dodis, and Rosler [26] give a tree-based construction that works with concurrent updates. The communication complexity varies between $O(\log n)$, when there is no concurrency, and $O(n)$, when the updates are fully concurrent. Alwen et al. [10, 12] added insider security to the family of TreeKEM protocols by considering the key schedule.

Recent evolutions of ratchet trees employ the “Propose & Commit” framework to achieve a nontrivial amount of concurrency. Specifically, the parties can concurrently propose updates, which are resolved with a serial or ordered commit in the next round. CoCoA [8] handles concurrent updates within one epoch with the help of a server. Their key idea is to apply all concurrent updates in one epoch by applying them in an order determined by an ordering function that is a system parameter.[2] Consequently, it may take up to $\log(n)$ rounds to complete all updates. DeCAF [7] improves on CoCoA's healing time, and requires a blockchain for ordering. SAIK [11] explicitly models the role of the server in group key agreement and improves on the upload cost to update the group key using multi-message multi-recipient PKE. CmPKE [77] is similar to SAIK in these regards, with tradeoffs on the communication costs compared to SAIK, and does not explicitly model the role of the server.

The closest work to Chapter 6 is by Weidner et al. [127], who introduced "decentralized" continuous group key agreement (DCGKA). DCGKA makes progress on the concurrency problems in ART and RTreeKEM so that all group members converge to the same view if they receive the same set of messages (possibly in different orders). The key primitive that enables concurrent updates is authenticated causal broadcast, defined in a similar way as Lamport's vector clocks [85]. Additionally, the authors made progress on how to manage group membership in an asynchronous network without a central server. However, their construction still requires a serial commitment.

In comparison to Weidner et al. [127], the construction in Chapter 6 does not require authenticated causal broadcast; it permits asynchronous messaging by buffering messages that are received out of order, and authenticates via authenticated encryption for GM. The construction also does not require acknowledgements, since the key lattice enables parties to change the group key without acknowledgments, and every party can always encrypt with respect to the latest group key in its own view. This substantially reduces the cost of an update because DCGKA requires $n-1$ broadcast acknowledgements for an update.

[2]This assumes a fully synchronous network; otherwise, consensus is required.

Other Protocols:

There are many group key agreement and group messaging protocols that do not use the above tree structure, e.g., generalized Diffie-Hellman protocols [79, 121]. These early protocols do not provide the strong security properties found in modern protocols or are not efficient (i.e., requiring $O(n)$ rounds of communication to establish a key).

Secure group messaging can be implemented by running two-party Signal between all pairs in a group [49, 116]. If a party wants to send a message to a group, it sends the message over all of its pairwise channels[3]. An advantage of this approach is that if two parties are in multiple groups, they can reuse their pairwise channel. Forward secrecy and post-compromise security are guaranteed by the underlying Signal protocol. This approach works in a concurrent environment because PCS updates are transmitted only over pairwise channels and do not need to be synchronized.

Sender Keys, currently deployed by WhatsApp [130], also builds group messaging from pairwise Signal. During initialization, each party sends a symmetric "sender" key to all the group members using the pairwise Signal protocol. This key is used for encrypting payload messages by that party. Every party keeps n "sender" keys in their state where $n-1$ keys are used for decryption and 1 is used for encryption.

Sender Keys by itself does not provide PCS since an adversary who corrupts a party will learn all the symmetric keys and decrypt all future messages. When a corrupted party sends a new sender key, its symmetric key is healed. However, because the adversary learns all of the symmetric keys, it can still decrypt messages sent by other parties. Fully healing the state therefore requires every party to update its symmetric key, which has a cost of $O(n^2)$.

The construction in Chapter 6 can be viewed as a generalization of Sender Keys with im-

[3]In practice it is not as easy as simply creating a Signal instance between every two parties. Additional steps need to be added for the users to establish the group ID and perform group management tasks.

Protocol	Update Cost		Healing Rounds	PCS	FS	Active Server	Concurrent Updates	Proof	Adaptive
	Sender	Receiver							
Original TreeKEM [113]	$O(\log n)$	$O(1)$	n	yes	yes	Ordering	no	None	n/a
Causal TreeKEM [128]	$O(\log n)$	$O(1)$	n	yes	yes	none	causal	StM	yes
RTreeKEM [9]	$O(\log n)$	$O(1)$	2	yes	yes	Ordering	no	ROM	yes
Concurrent TreeKEM [26]	$O(n)$	$O(1)$	2	yes	no	none	yes	StM	yes
Signal group [49, 116]	$O(n)$	$O(1)$	2	yes	yes	Prekeys	yes	None	n/a
Sender Keys [130, 116]	$O(n^2)$	O(n)	2	yes	yes	Prekeys	yes	None	n/a
DCGKA [127]	$O(n)$ (□)	$O(1)$	2	yes	yes	none	yes (◇)	ROM	no
CoCoA [8]	$O(\log n)$	$O(1)$	$\log(n)$	yes	yes	Process Updates (‡)	yes (◇)	ROM	yes
SAIK [11]	$O(\log n)$	$O(1)$	2	yes	yes	Process Updates (‡)	yes (△)	ROM	yes
DeCAF [7]	$O(\log t)$ (†)	$O(1)$	$\log(t)$	yes	yes	blockchain	yes (◇)	ROM	yes
CEST [47] (Chapter 6)	$O(n)$	$O(1)$	2	yes	yes	none	yes	StM	yes

Table 2.2: Overview of the latest concurrent group messaging protocols

proved security and functionality, where parties update the key lattice instead of holding symmetric keys for each party. The group session heals once a corrupted party's pairwise channels heal because the next update it sends or receives is indecipherable to the adversary. This requires $O(n)$ public key operations (also $O(n)$ communication complexity) after one corruption.

Summary

Table 2.2 summarizes a representative sample of recent literature on group key agreement and group messaging. The entry on the last line is featured in Chapter 6. "Update Cost" gives the communication complexity to update a shared or pairwise key, for the sender and the receiver, and "Healing Rounds" describes the round complexity of healing the session after a corruption. "Active Server" is a server that provides additional functionalities other than a PKI, such as ordering messages or post-processing updates. For example, the Signal servers need to store single-use pre-keys and the TreeKEM servers need to order messages. "Adaptive" means whether the adversary can adaptively pick which oracles to query during the security game. "PCS" denotes post compromise security, and "FS" denotes forward secrecy. "ROM" stands for the random oracle model, "StM" denotes the standard model. (□) an update for DCGKA requires $n-1$ broadcast acknowledgements, so the total complexity is $O(n^2)$, although the sender's computational complexity is $O(n)$. (◇) These works use the Propose & Commit paradigm, which assumes the existence of epochs and allows concurrent proposals, but a serial commitment is required. (†) t is the number of corrupt parties. (‡)

The server in CoCoA and SAIK processes an update to send an individual packet to each participant. The server also orders messages. (△) The SAIK server arbitrarily chooses one of concurrent updates to be processed.

2.3 Timed and Fine-Grained Cryptography

Time-lock puzzles have been studied since the seminal work of Rivest, Shamir, and Wagner (RSW) [114] more than twenty-five years ago. Over twenty years ago, Boneh and Naor [31] introduced timed commitments as a way to achieve fairness in MPC. More recently, inherently sequential functions motivated by large scale consensus, distributed ledgers, and blockchain applications have also precipitated considerable research in verifiable delay functions [30, 109, 129]. The interest in time-lock primitives has yielded various notions like non-malleable time-lock puzzles [69], non-malleable timed commitments [81], and UC-security [21, 22] of time-lock puzzles in the random oracle model. Recently, Wan et al. [125] used time-lock puzzles to construct more efficient broadcast with adaptive security.

Chapter 7 studies composition of timed primitives with other cryptographic primitives, and casts the time-release of information in a falsifiable model. Table 2.3 frames the work of Chapter 7 with respect to other recent research that studies composition of time-lock puzzles. In the table, RO stands for "Random Oracle," P stands for "Plain," SAGM stands for "Strong Algebraic Group Model," and RC stands for "Residual Complexity." In the "Composition" column, UC stands for "Universal Composability," NM stands for "Non-malleability", and S,C stands for "Sequential and Concurrent."

	Analytical Model			
Protocol	**Gen**	**Solve**	**Contradiction**	**Composition**
Arapinis et al. [13]	RO	RO	No	UC
Baum et al. '20b [21]	Trapdoor	RO	YES	UC
Baum et al. '20a [22]	Trapdoor	RO	YES	UC
Freitag et al. [69]	N/A	P,RO	N/A	NM
Katz et al. [81]	Trapdoor	SAGM	YES	NM
Eldefrawy et al. [62] (Chapter 7)	Trapdoor	RC	NO	S,C

Table 2.3: State of Research of Composable Time-Lock Primitives.

2.3.1 Time-lock Puzzles in the Literature

In spite of the early work in timed primitives, the work on which Chapter 7 is based [62] is the first (to the best of our knowledge) that considers composition of timed primitives with fine-grained security. Recent work (below) on composability of non-malleable time-lock puzzles and timed-commitments do not provide a full treatment of fine-grained security in a falsifiable model.

Time-lock Puzzles. The seminal work on time-lock puzzles was produced by Rivest, Shamir, and Wagner (RSW) [114]. Boneh and Naor [31] introduced timed-commitments, which progressed the study of using timed primitives for fairness in MPC. Verifiable delay functions, which are cryptographic primitives that depend on sequential work in order to delay release of information, have also been the focus of several recent research efforts [29, 109, 129]. Bitansky et al. [27] formally defined time-lock puzzles and constructed them using randomized encodings, which in turn depend on indistinguishability obfuscation. They also construct weak time-lock puzzles (with fast *parallel* generation time, although sequentially they may take longer to generate than to solve) from one-way functions. Baum et al. [21, 22] formalized time-lock puzzles in the UC model [39]. Freitag et al. [69] built publicly verifiable, non-malleable time-lock puzzles. Katz et al. [81] recently constructed non-interactive non-malleable timed-commitments that come with a proof that they can be forced open. They additionally showed that in a quantitative group model, speeding up squaring is as hard as factoring. In terms of negative results, Mahmoody et al. [91] proved that in the random oracle model, there are no time-lock puzzles with more than polynomial time gap.

Two additional works of note have addressed the assumptions underlying the repeated squaring problem in idealized models. Rotem and Segev [117] showed that speeding up repeated squaring in a generic ring is equivalent to factoring. van Baarsen and Stevens [124] address multiple hardness assumptions used for timed primitives in generic Abelian hidden-order groups.

Time-locked Cryptography and Composition. The recent work by Baum et al. [22] studies composition of time-lock puzzles in the UC model. The recent work by Freitag et al. [69] studies concurrent composition of non-malleable primitives. To our knowledge, these are the only other works that consider composition of time-based primitives. The comparison with Baum is not straightforward because there are common themes with different treatments; we generalize depth-secure computation and Baum considers concurrent composition of time-lock puzzles in UC. Baum provides an idealized version of RSW's underlying assumption, while we introduce a generic model which may leak information. Freitag et al. consider depth-bounded adversaries against concurrent composition of time-lock puzzles and imply small leakage until a puzzle is solved, but do not consider the more general composition with MPC.

2.3.2 Techniques for Timed Primitives

We next overview techniques in the literature for modeling and composing timed primitives.

Fine-grained Cryptography: A number of recent works have studied fine-grained cryptographic primitives. Degwekar et al. [53] initiated the study of fine-grained cryptographic primitives that can be built in one complexity class and are secure against adversaries in larger complexity classes. Egashira et al. [61] extended their results. Ball et al. [18, 19] built fine-grained proofs-of-work by using fine-grained worst-case to average-case reductions of hard problems. Lavigne et al. [86] studied the properties necessary to imply fine-grained public key cryptography and presented a fine-grained key exchange protocol.

Concurrently and independently, Cohen et al.[44] provided composition theorems for a subclass of resource-restricted primitives that are special versions of our composition theorems.

Homomorphic Time-lock Puzzles. Malavolta and Thyagarajan [92] provided practical homomorphic time-lock puzzles that are either additively homomorphic, multiplicatively homomorphic, or branching programs, but they require indistinguishability obfuscation in order to achieve full homomorphism. They also do not consider composition of their puzzles with other cryptographic primitives.

Contrast with the Composition of Baum et al. [21, 22] Baum et al. [22] models a new abstraction of time by allowing the adversary to control ticks of some time-keeping functionality. They define a time-lock functionality that implements the assumption by RSW [114], and provide a protocol that builds a puzzle with respect to this functionality. The functionality implements an idealized version of their assumption which does not leak information until the time-lock expires. In contrast, in Section 7.5 time is modeled by the depth of the environment's computation, and is therefore not controlled by the adversary. To enforce time-based privacy, Section 7.5 models idealized leaky functionalities that respond to environment-directed time. To model gradual release of information from a timed primitive, we discuss how to simulate an adversary's view as it extracts information from a time-lock puzzle.

The central issue for Baum's approach is a "side-door" attack in which an environment may use cycles from the concurrent execution of a different session in order to solve a time-lock puzzle in given session. The approach of Chapter 7 considers this particular attack to be infeasible because all parties are depth-bounded, including the environment. In Section 7.5 an environment should be constrained by the same depth requirements among all of its concurrent executions. An environment that expends computational resources in a concurrent session in order to solve a time-lock puzzle must also expend the same depth in the session of a given time-lock protocol; therefore, although the environment may increase its parallel computation to solve a puzzle by invoking concurrent sessions, the depth constraint remains. The depth-bounded model specifically excludes this form of attack.

Simulation of Time-Lock Puzzles in Phases. The work of Chvojka et al. [43] builds time-release encryptions and sequential time-lock puzzles using a phased simulation technique to argue the security. They define a sequential time-lock puzzle to be one where an intermediate solution is considered to be the starting point of the next puzzle in the sequence. However, the intermediate values provided to their simulators in the arguments of security treat a different issue than arises in our proof. When they consider the simulatability of a sequential puzzle, they must show how to simulate an intermediate step without computing the previous steps explicitly. Their technique is to provide a function of the previous steps as advice to the simulator for an intermediate step, but they cannot provide the true solutions for previous sequential steps as inputs to their simulator.

The simulation-in-phases technique described in Section 7.5.5 addresses an entirely different issue; it formulates the ability of an uninterrupted simulator to successfully simulate a prefix of the execution ending in a specific phase, given only the information it may learn by the end of that phase. In Section 7.5.5 all of the intermediate values of a prefix of the execution are available to the simulator; in [43] the intermediate values from a prefix of the sequential puzzle solution are not available.

2.3.3 Comparison with Other Definitions

This section compares the residual complexity model in Section 7.2.1 with popular approaches for defining time-lock puzzles in the existing literature. In both cases below, the provided treatment is insufficient for completely modeling the use of such a primitive in composition with other protocols.

Bitansky's Time-lock Definition. Bitansky et al. [27] formalized the notion that up to a certain point in time before the puzzle is solved, the solver's distribution on the puzzle solution retains very high pseudo-entropy. Their notion is reproduced in Definition 7.4. Intuitively, for any puzzle with polynomial running time t and any polynomial time solver running in time up to t^{ε}, where $\varepsilon < 1$ is called the *gap parameter*, the solver gains only negligible advantage in guessing the solution of a challenge puzzle. This definition, however, does not consider what happens *after* the solver exceeds the time-gapped running time. (The notion of a time-lock puzzle by [69] has a similar feel, where the solver runs within some polynomially smaller time than the puzzle has been tuned for, and also does not describe what happens after the time-lock puzzle expires but before the honest parties solve the puzzle.)

Blum-Blum Shub. As a more concrete and illustrative example of what happens when a time-based primitive reaches the end of its guarantees, recall the generalized Blum-Blum-Shub (BBS) assumption by Boneh and Naor [31].

Definition 2.1 ($(n, n', \delta, \varepsilon)$-Generalized BBS Assumption [31])**.** *For $g \in \mathbb{Z}$ and a positive integer $k > n'$, let $W_{g,k} = \langle g^2, g^4, g^{16}, \ldots, g^{2^{2^i}}, \ldots, g^{2^{2^k}} \rangle$. Then for any integer $n' < k < n$ and any $\delta * 2^k$-depth-bounded $\mathcal{A}$,*

$$| \Pr[\mathcal{A}(N, g, k, W_{g,k} \bmod N, g^{2^{2^{k+1}}}) = 1] - \Pr[\mathcal{A}(N, g, k, W_{g,k} \bmod N, R^2) = 1] | \leq \varepsilon$$

where the probability is taken over the random choice of an n-bit RSA modulus $N = p_1p_2$, where p_1 and p_2 are equal primes satisfying $p_1 = p_2 = 3 \mod 4$, and element $g \in \mathbb{Z}_N$, and $R \in \mathbb{Z}_N$.

The assumption was first introduced for the design of a pseudo-random generator [28], and then elegantly generalized [31] to develop timed commitments (with a trapdoor). As noted in [31], the assumption states that given the sequence of repeated squares $W_{g,k}$ of some generator g, the $k+1$st element in the sequence $g^{2^{2^{k+1}}}$ is indistinguishable from a random quadratic residue, for any party whose running time is much less than 2^k. This suffices for showing the pseudo-randomness of the BBS generator.

However, for timed protocols in which any party solves a puzzle as part of the protocol, eventually the depth of the puzzle solver *must* approach the sequence length 2^k *by definition*. At this point, the guarantee of the BBS assumption breaks down! As the solver approaches the duration of the time-lock – even before it finally learns the solution – the distribution of the solver's "best guess" on the solution becomes more refined over time.

Chapter 3

Modeling and Definitions

This chapter presents models for distributed cryptography and definitions of computational problems that will be used in the rest of the dissertation.

3.1 Computational Model

Throughout the thesis, every machine is modeled as an Interactive Turing Machine (ITM). The execution model is derived from the Universal Composability (UC) Model by Canetti [39]. For each application, the model is adapted by constraining the network and adversary.

An *execution* of a protocol is directed by an adversary, sometimes referred to as the environment. (We sometimes distinguish the adversary from the environment and sometimes combine them; we make clear which case in each section.) The adversary "activates" a party when the adversary permits that party to perform some computation, as specified by the party's transition function. When the adversary activates a party, writes a message to the party's incoming message tape, which may be the empty string. When the party has finished executing during its activation, it writes a string to its outgoing tape, which may be the empty string. The adversary reads this string, which encodes what messages the party intends to send to other parties. (If the empty string, the party sends no messages.) The adversary determines when (or if) to deliver the message(s) to the intended recipient(s).

In Chapters 4 and 7 the number of parties n in a specific execution is known and does not change. In Chapter 5, this number is unbounded. In Chapter 6, it may change dynamically.

3.1.1 Communication Models

By delivering messages between parties, the adversary implements a network. The adversary cannot modify messages sent by one party to another; this models peer-to-peer authenticated channels. However, the adversary may always write (or inject) messages to the incoming message buffers of the participants which were not sent by other parties.

In Chapter 5, the partiicpants in a protocol do not know the identities of the other parties participating. In this case, the adversary must implement a *multicast* channel. When one party sends a message via multicast, the adversary must deliver it to *all* other parties on the network. However, a party that receives a multicast message cannot tell the difference between a multicast and a message that was sent directly to it, or to only a subset of the parties. (Corrupt parties can therefore take advantage of multicast to cause inconsistencies in the views of honest parties.)

3.1.2 Synchronization Constraints (and Asynchrony)

For the sake of exposition, time is modeled an element from a totally ordered set $\mathcal{T}$.[1]

[1] Only in Chapter 4 do the parties have access to clocks with which they can read the current time. Otherwise, we use time to define an ordering of events, and parties can tell time only by the depths of the computations that they have performed; see Lamport [85] for a discussion of equivalent executions in time.

An execution proceeds in time by allowing the adversary to activate some number of parties at each time step. In general, an execution is *asynchronous* if the adversary is not constrained on the number of parties it activates at each time step; the strength of the constraint levied on the adversary determines the strength of synchronization. More formally, synchronous computation and communication are defined as follows:

Definition 3.1 (Φ-Synchronous Computation)**.** *For a constant Φ, an execution is Φ-synchronous if for every time t and every $t' > t$, if any participant is activated Φ times between t and t', then every other participant is activated at least once between t and t'.*

Definition 3.2 (Δ-Synchronous Communication)**.** *For a constant Δ, communication in an execution is Δ-synchronous if for any message m that is sent to participant p at time t, if p does not receive m before $t + \Delta$, then p receives m at its first activation at or after $t + \Delta$.*

Computation and communication are *asynchronous* if there does not exist a Φ or Δ, respectively that constrains the adversary. DLS [58] modeled environments where Φ or Δ exist but are unknown to the protocol as *partially* synchronous.

Dynamic Participation Participation is *dynamic* to in an execution if parties can join and leave an execution at any time. Specifically, the adversary can spawn new parties at any time, and conversely it can also stop activating a party altogether. In this way, dynamic participation is an extension of asynchronous computation, with the added feature that parties can be added or removed from an execution at any time. In the classical form of asynchronous computation, all parties are expected to be activated infinitely many times in any infinite execution; in a dynamic environment, an infinite execution may have parties that are activated only finitely many times due to joining and leaving, including parties that are only active long enough to send a single message.

When computation is asynchronous but communication is synchronous – or when dynamic participation is permitted in a synchronous network – parties can go through long periods of no activation, then "wake up" and receive many messages. This model was analyzed as early as [14] and [54].

3.1.3 Standard Corruption Models

Honest parties are those which specified protocol. *Corrupt* parties deviate from the protocol. Throughout this thesis, the adversary can *adaptively* corrupt parties, meaning it may choose to corrupt any party at any time, and in particular may do so after reading messages that have been sent during an execution. Once corrupted, a party is always corrupted for the remainder of the execution (recoveries are not considered).

The most adverse form of corruption is a *malicious* or *byzantine* corruption. In this case, the adversary learns the party's internal state, reads all of its incoming messages, and chooses which messages the party sends and to whom. A malicious or byzantine party may deviate arbitrarily from the protocol specification in order to break the desired security properties.

Rushing Adversary A *rushing* adversary is a strenghthening of the power of the adversary's ability to corrupt parties. A rushing adversary is permitted to read the messages that a party sends, and *after* reading a message the adversary may choose to corrupt the sending party and optionally remove the read message from the network. The adversary in Chapters 4 and 5 is rushing. (In Chapters 6 and 7, rushing adversaries do not impact the analysis.)

3.1.4 Send and Receive Corruptions

Chapter 4 additionally considers *send corruptions* and *receive corruptions* in a model introduced by ZHM [131]. Send and receive corruptions are both forms of omission faults, which were originally studied by Perry and Toueg [108], and later by Raynal and Parvedy [103, 112]. A *send* corrupt party may have its messages "dropped" by the adversary, meaning the adversary may adaptively decide not to messages sent by that party. A *receive* corrupt party may have its incoming messages similarly dropped. A party can be both send-corrupt and receive-corrupt, in which case the adversary adaptively chooses whether to deliver messages sent by or sent to the corrupt party. Note that otherwise, send-corruptions and receive corrupt parties *still follow the honest protocol*. Send corruptions and receive corruptions are both strict generalizations of crash corruptions. A crashed party ceases to both send and receive messages, but unlike send and receive corrupt parties, a crashed party ceases to participate.

Spotty Send Corruptions Chapter 4 introduces a *spotty* send corrupt party, named to describe when a party is sometimes able to send messages and sometimes cannot send any messages. For a spotty send corruption, the adversary must choose whether to deliver *all* messages sent by the corrupt party within a period of Δ time, or *none* of them. The spotty send corruption is born from the observation that when a party's sent messages are either delivered to all parties or no parties, it is less harmful to the protocol execution.

Zombies Zombie parties were introduced by ZHM [131] to describe honestly behaving parties that recognize they are corrupted and therefore drop out of the protocol. When an honest party recognizes that it is corrupted and cannot contributed to the protocol, it sends the message zombie to all parties and ceases to participate, outputting $\perp$.

3.1.5 Permissionless Executions

A *permissionless* execution is distinct from a “classical” execution in the following ways:

1. The set of parties participating in an execution is not known to the honest parties beforehand, and it may be determined by the adversary during the execution.
2. The adversary can corrupt arbitrarily many parties, and its corruption budget is not bounded as a constant proportion of the total number of parties.
3. Participation is fully dynamic: (honest) parties may join and leave an execution arbitrarily.

The permissionless model attempts to capture internet-scale protocols where parties may participate sporadically over inconsistent network links. Moreover, the adversary may launch Sybil attacks [56] by spawning fake identities. In particular, some honest parties may join an execution, send a single message, and then leave the execution forever, and this message must factor into the decision made by the online honest parties. (This was later modeled for MPC by Gentry et al. [73])

3.2 Problem Definitions

This section introduces notation and defines the computational problems addressed in the rest of this thesis.

3.2.1 Notation

Denote by $\mathbb{N}$ the natural numbers and by $\mathbb{Z}$ the integers. For a list ℓ, denote by $\ell[i]$ the ith element of ℓ. A is *negligible* function negl is a function such that for every polynomial p there exists an n^* such that for all $n > n^*$, $\mathsf{negl}(n) < \frac{1}{p(n)}$. Write $[m] = \{1, \ldots, m\}$, and $[a, b] = \{a, a+1, \ldots, b-1, b\}$ where $b > a$. For a set S, $|S|$ denotes the cardinality of S. For graphs G and G', $G \subseteq G'$ denotes that G is a subgraph of G'. Let $\mathcal{P}$ be the group of parties in an execution, and let $n = |\mathcal{P}|$.

3.2.2 Preliminaries for Graphs

The protocol presented in Chapter 5 is based upon graphs. A graph $G = (V, E)$ is a set of vertices and a set of edges between vertices. For a graph G, we denote the set of its vertices as $G.V$ and its edges as $G.E$. This work considers only directed acyclic graphs (DAGs); therefore, every reference to a graph is a DAG. A *root vertex* in a graph is a vertex with in-degree 0. In this work, every graph has exactly one root vertex. (In cryptocurrencies, this is analogous to the genesis vertex.)

Depth of a vertex and depth of a graph are defined in a non-standard way:

Definition 3.3 (Depth of a Vertex, Depth of a Graph)**.** *Let* root *be the root vertex of a graph G. The* depth *of a vertex v in G is defined as the length of the* longest *path from* root *to v. The* depth *of G is defined as the depth of its deepest vertex.*

$\mathsf{D}(G)$ denotes the depth of a graph G, and $\mathsf{D}_G(v)$ denotes the depth of a vertex v in G. When the graph is implied from context, we simply write $\mathsf{D}(v)$. The depth of a root vertex is always 0. $G|_d$ denotes the subgraph of G including only vertices with depth $\leq d$. Figure 3.1 illustrates the depths of vertices in a simple graph. A path from vertices v to u is denoted $v \to u$. A path $v \to u$ *spans d depth* if $\mathsf{D}(u) - \mathsf{D}(v) = d$. $u \in G.V$ is *reachable* from $v \in G.V$ if

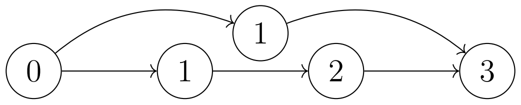


Figure 3.1: An example DAG in which each vertex is labeled with its depth.

there is a path $v \to u$. For a vertex $v \in G.V$, the *predecessor graph* of v is the subgraph of G containing v and every vertex and edge on every path from root to v. $\cup$ denotes graph union and $\subseteq$ denotes a subgraph. $\mathsf{indegree}(v)$ denotes the indegree of a vertex v and $\mathsf{outdegree}(v)$ denotes its outdegree. For applications, a vertex may also have a "payload" string that gives semantics to the vertex.

3.2.3 Computational Indistinguishability

Chapters 4 and 5 assume an idealized model that does not require cryptographic hardness; however, Chapters 6 and 7 do require cryptographic hardness. We therefore define computational indistinguishability.

Definition 3.4 (Computational Indistinguishability)**.** *Two ensembles* $X = \{X(a,n)\}_{a\in\{0,1\}^*,n\in\mathbb{N}}$ *and* $Y = \{Y(a,n)\}_{a\in\{0,1\}^*,n\in\mathbb{N}}$ *are* computationally indistinguishable*, denoted* $X \overset{c}{\equiv} Y$ *if for every probabilistic polynomial time distinguisher* D *there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for every* $a \in \{0,1\}^*$ *and every* $n \in \mathbb{N}$

$$\Pr[D(X(a,n)) = 1] - \Pr[D(Y(a,n)) = 1] \leq \mathsf{negl}(n)$$

Chapter 7 introduces granular runtime constraints on machines by their depth; we therefore require a notion of depth-bounded indistinguishability. In the following definition, a machine is d-depth bounded, where $d = d(n)$ is a polynomial function, if it runs in time no more than d. An ensemble of machines is denoted $D = \{D_n\}_{n\in\mathbb{N}}$, where the depth of D_n is bounded by $d(n)$.

Definition 3.5 (Depth-Bounded Indistinguishability)**.** *Two ensembles* $X =$

$\{X(a,n)\}_{a\in\{0,1\}^*,n\in\mathbb{N}}$ *and* $Y = \{Y(a,n)\}_{a\in\{0,1\}^*,n\in\mathbb{N}}$ *are* d-depth-indistinguishable*, denoted* $\overset{d}{\approx}$ *if for every d-depth-bounded distinguisher* $D = \{D_n\}_{n\in\mathbb{N}}$ *there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for every* $a \in \{0,1\}^*$ *and every* $n \in \mathbb{N}$

$$\Pr[D_n(X(a,n)) = 1] - \Pr[D_n(Y(a,n)) = 1] \leq \mathsf{negl}(n)$$

3.2.4 Consensus and Broadcast

In the definition of classical consensus [54, 58, 80, 107], every participant has a bit $b \in \{0,1\}$ as input, and each party outputs a bit $v \in \{0,1\}$. In the related primitive of broadcast [55, 70, 125], a dealer $D \in \mathcal{P}$ wishes to send a message $m \in \{0,1\}^*$ to the parties in $\mathcal{P}$. Each party $p \in \mathcal{P}$ outputs a message $m' \in \{0,1\}^* \cup \{\perp\}$.

Security of these protocols is defined with respect to the number of parties that the adversary may corrupt. t_{snd}, t_{rcv}, and t_{byz} denote thresholds on the number of send, receive, and byzantine corruptions, respectively, in an execution. The following definition facilitates the problem definitions by constraining how many parties are corrupted in an execution. Note that for most studies of consensus, $t_{\mathsf{snd}} = t_{\mathsf{rcv}} = 0$.

Definition 3.6 (($t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}}$)-Compliant Execution)**.** *For a protocol* Π*, an execution of* Π *is* ($t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}}$) *compliant if at most* t_{snd}*,* t_{rcv}*, and* t_{byz} *parties are send-corrupted, receive-corrupted, and byzantine-corrupted, respectively, in the execution.*

Live Parties In the following definitions, *live* parties are honest, send-corrupt, or receive-corrupt but not zombies. These parties' inputs and outputs are constrained in the security definition because they are defined to be honestly following the protocol specification. Note that in models where there are no send or receive corruptions, the live parties are exactly the honest parties, and therefore the following definition is a strict generalization of the literature standard (for example, [68, 80]).

Definition 3.7 (Consensus)**.** *Let* Π *be protocol for parties* $\mathcal{P} = \{p_1, \ldots, p_n\}$ *in which each party has an input* $b \in \{0,1\}$*.* Π *is a* Consensus *protocol if the following properties hold except with negligible probability.*

1. $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$***-Validity****:* Π *is* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-valid *if in every* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-compliant execution in which all live parties have the same input* $b \in \{0,1\}$*, all honest parties output* b*.*

2. $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$***-Consistency****:* Π *is* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-consistent *if in every* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-compliant execution in which any live party outputs* v*, every live party outputs* v*.*

3. $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$***-Termination:*** Π *is* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-terminating *if in every* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-compliant execution, every live party outputs* $v \in \{0,1\}$ *and terminates within finitely many steps.*

If Π *is* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-valid,* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-consistent, and* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-terminating then we call it* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-secure.*

Definition 3.8 (Broadcast)**.** *Let* Π *be a protocol for parties* $\mathcal{P} = \{p_1, \ldots, p_n\}$ *in which a distinguished party* $D \in \mathcal{P}$ *holds an input* $m \in \{0,1\}^*$*.* Π *is a* Broadcast with Unanimity for Send-Corruptions *protocol if the following properties hold except with negligible probability.*

1. $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$***-Validity:*** Π *is* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-valid *if in every* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-compliant execution in which* D *is honest or receive corrupt (but not send-corrupt), every live party outputs* m*.*

2. $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$***-Consistency:*** Π *is* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-consistent *if in every* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-compliant execution in which any live party outputs* $m' \in \{0,1\}^* \cup \{\perp\}$*, every live party outputs* m'*.*

3. $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$***-Termination:*** Π *is* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-terminating *if in every* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-compliant execution, every live party outputs some* $m' \in \{0,1\}^* \cup \{\perp\}$ *and terminates within finitely many steps.*

If Π *is* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-valid,* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-consistent, and* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-terminating then we call it* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-secure.*

3.2.5 Graph Consensus

In an execution of a graph consensus protocol, participants have no input. Each participant p maintains a local graph G_p based on the messages it has received so far and the protocol specification. A graph consensus protocol specifies how participants generate new vertices, and how to propose that other participants include the new vertices in their local graphs. It also specifies how a participant determines whether a new vertex, which it receives in a proposal from another participant, should be included in its local graph. For a participant p active at time t, $G_p^{(t)}$ denotes its local graph after all vertices are added at t. Each participant p additionally maintains an output graph G_p^*, which it outputs whenever it is active. The protocol must specify a deterministic way for each p to compute G_p^* as a function of its local graph G_p. $G_p^{*(t)}$ denotes the output of p at time t.

An execution of graph consensus may continue indefinitely. The goal of a protocol is for the participants' outputs to obey consistency and liveness properties across time. Graph consistency requires that if participants p active at t and q active at t', output $G_p^{*(t)}$ and $G_q^{*(t')}$, then one output graph must be a subgraph of the other.

Definition 3.9 (Graph Consistency)**.** *A protocol* Π *satisfies* graph consistency *if in every execution of* Π*, for all times* t *and* t'*, and for all honest* p *and* q *active at* t *and* t'*, respectively:* $G_p^{*(t)} \not\subseteq G_q^{*(t')} \implies G_q^{*(t')} \subseteq G_p^{*(t)}$.

A protocol can trivially satisfy graph consistency if participants always output the empty graph. Therefore, liveness requires that each participant p's output G^*_p grows in time. The following definition depends on the environment allocating *resources*, which is a novel contribution of the work described in Chapter 5 and presented formally in Section 5.3. One can roughly understand the number of resources allocated by time t as the number of vertices in $G^{(t)}_p$.[2]

Definition 3.10 (f-Liveness)**.** *Let* $f\colon \mathbb{N} \to \mathbb{N}$*. A protocol* Π *satisfies* f-liveness *if in every execution, for every time* t *and honest participant* p *active at* t*: if the environment has allocated* N *resources by time* t*, then* $|G^{*(t)}_p.V| \geq f(N)$*.*

In some applications, it is desirable to show that some proportion of the vertices in an honest participant's output must be generated by honest participants. If a vertex is generated by an honest participant, it is considered an *honest vertex*; otherwise, it is considered a *corrupt vertex*. $\mathsf{hon}(G.V)$ denotes the honest vertices in G. h-honest-vertex quantifies the guaranteed proportion of honest vertices in a participant's output graph.

Definition 3.11 (h-Honest-Vertex Liveness)**.** *Let* $h\colon \mathbb{N} \to \mathbb{N}$*. A protocol* Π *satisfies* h-honest-vertex liveness *if in every execution, for every time* t *and honest participant* p *active at* t*:* $|\mathsf{hon}(G^{*(t)}_p.V)| \geq h(|G^{(t)}_p.V|)$*.*

f-liveness and h-honest vertex liveness are sometimes referenced together by f, h-liveness. Note that because graph consensus does not place thresholds on the number of corrupt parties, thresholds are elided from the security definitions.

[2]Due to malicious withholding, this is not precisely true; however, for the purpose of intuition this suffices until the definition is presented in Section 5.3.

3.2.6 Multi-Party Computation

Chapter 7 studies secure multi-party computation with timed cryptographic primitives and therefore requires a definition for secure multi-party computation that considers fine-grained running times of the adversary and distinguisher.

The Ideal/Real Paradigm

As in the UC model [39], our model for secure computation separates the roles of the adversary and the environment. The environment directs the execution by activating parties and the adversary. The adversary chooses which parties to corrupt and participates on behalf of the corrupt parties. We prove security via the ideal/real paradigm described below.

Separating the environment and the adversary decouples the malicious effects of different roles, which is especially relevant to Chapter 7. The adversary attacks the timed primitive, and it will attempt to solve the primitive before honest parties. The environment attacks the *execution* involving a timed primitive; it distinguishes the execution of a real protocol from an idealized execution (below), and may spawn side sessions in order to help it attack the execution.

Execution in the Real Model. In the real model, participants execute a protocol Π to compute the desired functionality $\mathcal{F}$ without a trusted party. At the end of the execution, honest parties output their protocol outputs. The corrupt parties output nothing. The adversary outputs an arbitrary function of its inputs and the messages that corrupt parties have received. The environment learns every output. The random variable $\mathsf{REAL}_{\Pi,\mathcal{A}(z),\mathcal{Z}}(\overline{x})$ denotes the output of the environment in a real execution of Π with honest inputs $\overline{x}$, auxiliary input z to $\mathcal{A}$, with environment $\mathcal{Z}$.

Execution in the Ideal Model. In an ideal execution, the parties interact with a trusted party by submitting all of their inputs to the trusted party in the beginning of the execution. The trusted party computes the desired function of all parties' inputs and returns an output to every party. At the end of the execution, honest parties output whatever they have received from the trusted party. Corrupt parties output nothing, and the adversary outputs an arbitrary function of its input and the messages that corrupt parties have received from the trusted party. The environment learns every output. The random variable $\mathsf{IDEAL}_{\mathcal{F},\mathcal{A}(z),\mathcal{Z}}(\overline{x})$ denotes the output of the environment in an *ideal execution* of functionality $\mathcal{F}$ on honest inputs $\overline{x}$, auxiliary input z to $\mathcal{A}$, with environment $\mathcal{Z}$.

Defining Secure Computation

The definition of seure computation used in Chapter 7 is a strict generalization of the literature standard [75, 89]. When d_s is expected polynomial time and d_a and d_e are arbitrary polynomials, this is exactly the standard definition.

Definition 3.12 (Depth-Bounded Secure Computation: General)**.** *Let $d_a = d_a(\lambda)$, $d_s = d_s(\lambda)$, and $d_e = d_e(\lambda)$. Protocol Π (d_a, d_s, d_e)-depth securely computes $\mathcal{F}$ if there exists a d_s-depth-bounded $\mathcal{S}$ such that for every real-world d_a-depth-bounded adversary $\mathcal{A}$ and every d_e-depth-bounded environment $\mathcal{Z}$, the following two ensembles are d_e-depth indistinguishable:*

$$\{\mathsf{REAL}_{\Pi,\mathcal{A}(z),\mathcal{Z}}(\overline{x})\}_{\overline{x}\in(\{0,1\}^*)^n, z\in\{0,1\}^*}$$

$$\{\mathsf{IDEAL}_{\mathcal{F},\mathcal{S}(z),\mathcal{Z}}(\overline{x})\}_{\overline{x}\in(\{0,1\}^*)^n, z\in\{0,1\}^*}$$

3.2.7 Group Messaging

In a group messaging protocol, the parties maintain and evolve a set of group keys under which they encrypt messages. When a party wants to send a message, it encrypts it under one of the group keys in its local state. When a party wants to decrypt a message, it selects a group key from its local state for decryption. In a group messaging protocol, each party should be able to successfully decrypt every message that it receives. To aid in security, the parties also consistently change the group key via an evolution function.

Definition 3.13 (Group Messaging)**.** *A group messaging protocol is a tuple of five stateful algorithms* $(\mathsf{Init}, \mathsf{Evolve}, \mathsf{Recv}, \mathsf{Enc}, \mathsf{Dec})$ *defined as follows:*

– $\mathsf{GM.Init}(G, w)$*: Initialize the protocol with group* $G \subseteq \mathcal{P}$ *and the windows size* w*. Output a set of messages, one for each party in* G*.*

– $\mathsf{GM.Evolve}()$*: Outputs a set of update messages, one for each party in* G*.*

– $\mathsf{GM.Recv}(M)$*: Processes the message* M *(e.g., from the network), and outputs a response.*

– $\mathsf{GM.Enc}(m)$*: Encrypts a plaintext* m *and outputs a ciphertext.*

– $\mathsf{GM.Dec}(c)$*: Decrypts ciphertext* c *and outputs a plaintext.*

A secure group messaging protocol's algorithms are constrained for both correctness and security. For correctness, all messages must be decrypted correctly. In the security game, the adversary must be shown not to learn any group keys that are not explicitly held in the states of parties it corrupts. The security properties ared defined via oracle games. Because of the technical detail of the oracle game to describe security of group messaging, the formalism is deferred to Chapter 6. Next we overview oracle games and highlight how they model asynchrony.

Oracle Games

In an oracle game, the adversary directs a simulated execution of a protocol. The adversary invokes the API calls of the parties in the execution by calling oracles that maintain the state of the protocol for each party, and return the responses (i.e. messages to be sent) that an honest party would. The adversary may also corrupt the parties represented by specific oracles, in which case it learns the parties' internal state. This is analogous to how the environment directs the execution of a protocol for a multi-party computation, except that when the environment directs the multi-party computation execution, we think of it as activating the parties themselves, and not oracles that keep the state of the parties. The adversary in an oracle game may also run arbitrarily many (polynomial) side-sessions of the protocol it is attacking, by specifying to the oracles to launch new sessions.

The difference between the oracle game model and an adversary's game for multi-party computation is that in the oracle game, the adversary is not distinguishing between an ideal and real execution. In the oracle game, the adversary issues a single Test query that attacks a security property of the protocol under attack. In the fulfillment of a Test query, the oracle flips a bit that determines whether to respond faithfully to the query or whether to replace the response with some random value. The adversary must distinguish the possible executions with respect to the oracle's chosen response.

In the oracle game defined in Chapter 6, the adversary emulates an asynchronous network by choosing to deliver messages as if the network were fully asynchronous – specifically, both computation and communication may be asynchronous.

Chapter 4

Expected Constant-Round Consensus with Send and Receive Corruptions

This chapter presents the first expected-constant round protocol for single-shot consensus in which a majority of the *online* parties are dishonest. The computation and communication model for this protocol is the strongest found in this thesis: computation is fully synchronous – every party is activated in each time step – and communication is Δ-synchronous. The challenge is the variety and quantity of corruptions. Some parties are send corrupted, some are receive corrupted, and some are byzantine corrupted. The work on which this chapter is based [63] studies the different pathologies of these forms of corruption, and asks *"what are optimal thresholds in the cryptographic setting that can be tolerated with mixed corruptions?"*

For receive corruptions, the optimal threshold is achievable by applying (a simple modification of) a protocol by ZHM, which is presented in Section 4.3.1. Surprisingly, for the most general form of send corruption, we could not find a constant-round protocol that could tolerate more send corruptions than byzantine corruptions; however, we also could not prove that send corruption is as pathological to a consensus protocol as a full byzantine corruption.

Note that in the work by ZHM, a send corruption is as pathological as a receive corruption. For a weaker form of send corruption termed a *spotty* send corruption, we showed that it is less pathological to a consensus protocol than a byzantine corruption.

4.1 Pathology of a Send Corruption

This section discusses the pathology of "standard" send corruptions with respect to current techniques in the literature, and describes why send corruptions appear as deleterious as full byzantine corruptions. Although the focus is on consensus protocols, we consider techniques for both consensus and broadcast; the two are related by a (folklore) reduction, which is presented in Section 4.1.3.

1. We first provide an intuitive overview of the pathologies of our two forms of send corruptions.

2. We then recall the proof by Dolev and Strong that any deterministic broadcast protocol requires at least $t_{\mathsf{byz}} + 1$ rounds (for at most t_{byz} byzantine corruptions), and we show that the impossibility result immediately requires that when send-corrupt parties exist, any deterministic broadcast protocol requires at least $t_{\mathsf{snd}} + 1$ rounds (for at most t_{snd} send-corruptions).

3. We show that the Dolev-Strong broadcast protocol fails as written when considering send corruptions. We modify the protocol and show that without new ideas, its corruption threshold degrades from $n > t_{\mathsf{byz}}$ (in the original model) to $n > 2(t_{\mathsf{snd}} + t_{\mathsf{byz}})$.

4. We visit recent techniques for security against strongly rushing, adaptive adversaries – who have the ability to adaptively remove messages from the network after they have been sent – and show that these also fall short of a corruption threshold better

than $n > 2(t_{\mathsf{snd}} + t_{\mathsf{byz}})$ (which our construction in Section 4.3 achieves). One might expect these techniques would apply to send-corrupt parties because of the adversary's ability to adaptively drop messages from a majority of parties. However, the techniques fail when requiring send-corrupt parties' outputs to be consistent with honest parties' outputs.

Pathology of a (standard) send corruption. The standard model of a send corruption permits the adversary to selectively drop messages by send-corrupting a party. Because this behavior is a subset of a byzantine corruption, one would expect that corruption bounds follow directly from the byzantine case. This is not the case in general. The consistency property of many current protocols breaks under the specific attack that some (corrupt) party selectively sends a message to some honest parties which other honest parties may never receive. Therefore, a send-corrupt party may receive a message that would change its output *and fail to inform any honest party about the message.*

As an illustrative example (embodying a common technique), the Dolev-Strong [55] broadcast protocol requires that if some honest party – whose output is constrained by definition – receives a message, then all other honest parties will receive that message before the protocol terminates. But as shown in Section 4.1.1, Dolev-Strong breaks down in our model because a send-corrupt party may receive a message that would change its output but fail to forward it.

Generalizing this theme, an adversary can undermine current techniques by dividing the execution such that send-corrupt parties are unable to send messages to honest parties, but send-corrupt parties receive all messages sent by honest parties. Crucially, a situation results where there are two sets of parties with different sets of messages received by the end, but their outputs are constrained by consistency. For example, divide an execution into sets such that S contains all send-corrupt parties and H contains all honest parties, and let $|S| > |H|$.

Then it may be the case that *a majority of parties cannot communicate with the honest parties*, but all of their outputs must be consistent.

Although we could not prove it, it appears very difficult to tolerate more send-corrupt parties than honest parties because this partition requires the use of some threshold scheme to ensure sufficiently many parties are "aware" of a message to allow it to influence the output. Specifically, we do not know how to generate and use information that an honest party has *not* received a message sent by a send-corrupt party as part of the protocol. On the other hand, impossibility proofs that depend on partitioning techniques also fail in this model because it is impossible to completely separate the send-corrupt group from the honest group, since send-corrupt parties always receive all of the honest parties' messages.

The spotty corruption model, described below, alleviates the above issue because it enforces unanimity: if any send-corrupt party or honest party receives a message sent by a send-corrupt party, then all honest and send-corrupt parties receive the message. This is sufficient to enforce consistency of honest and send-corrupt views that permits thresholds over the number of byzantine parties.

Pathology of a spotty send corruption. Although the "spotty" send corruption is limited in some ways, it is still rich enough to cause failure of some popular techniques for synchronous consensus. In particular, it is unclear how to construct a protocol that employs leader election in order to reach constant expected round complexity in our model. Specifically, a strongly rushing adversary can wait for a leader to be elected – and even to send messages that attest to its election (e.g., based on a Verifiable Random Function, VRF, as in [95, 74]) – spotty-corrupt the party, and force it to fail as leader for the duration of its tenure, without even expending its budget towards byzantine corruptions. While in the purely byzantine model this type of attack can be easily mitigated by using threshold signatures (see, e.g., [88, 2]), this approach completely fails in our model, as electing a leader in this way would most likely elect one of the potentially t_{snd} send-corrupt parties (since t_{snd} can be much larger than the number of honest and in-synch parties). For this reason, recent protocols for dishonest majority broadcast that rely on the player-replaceable paradigm, such as [1] and [41] fail in a model that includes send corruptions.

4.1.1 Dolev and Strong's Lowerbound with Send Corruptions

The classical lowerbound for deterministic broadcast by Dolev and Strong transfers directly to the case of send corruptions, applying the observation (discussed in their paper) that all of the byzantine parties in their proof are constrained to dropping messages that should be sent, but otherwise behave honestly. A full exposition of Dolev and Strong's lowerbound is provided in Appendix B.

Theorem 4.1 (Dolev and Strong [55]). *There is no deterministic broadcast protocol tolerating t_{snd} send corruptions which terminates in fewer than $t_{\mathsf{snd}} + 1$ rounds, even assuming an idealized PKI and signatures.*

Recent work by Chan, Pass, and Shi [41] extends the lowerbound by Dolev and Strong to randomized protocols. Because their adaptation also requires only dropping sent messages, their lowerbound also directly transfers to the send-corrupt model.

Modifying Dolev-Strong Broadcast

As an example of the pathology of send-corruptions, recall the classical authenticated broadcast protocol by Dolev and Strong [55] in Figure 4.1.

The protocol uses a data structure called a *sig-chain*. A 1-sig-chain is a pair (m, σ), where σ is a signature on string m. For $i > 1$, an i-sig-chain is a pair (m, σ), where m is an $(i-1)$-sig-chain and σ is a signature on m. A *valid* i-sig-chain is a sig-chain with the property that

Protocol 1 Dolev Strong Broadcast Protocol Π^{DS}

Shared Setup: Public Key Infrastructure for a signature scheme.
Inputs: The dealer $D \in \mathcal{P}$ has an input $m \in \{0,1\}^*$.
Outputs: Each party $p \in \mathcal{P}$ outputs a value $m' \in \{0,1\}^* \cup \{\bot\}$.
Local Variable: Each party $p \in \mathcal{P}$ maintains a local variable S, which is a set initialized to $\{\}$.
Protocol: The protocol begins at time 1 and proceeds in rounds. Each round party p proceeds as follows:

1. **Round 1: Dealer's Messages** The Dealer D signs its input $\sigma \leftarrow \mathsf{sign}_{\mathsf{sk}}(m)$ and sends (m, σ) to all parties.
2. **Sig Chains** For every round i from 2 to $t_{\mathsf{byz}}+1$: For every valid $(i-1)$-sig-chain c that p received at the end of round $i-1$ in which none of the signatures were constructed by p, p computes $\sigma \leftarrow \mathsf{sign}_{\mathsf{sk}}(c)$ and sends (σ, c) to all parties. For every valid i-sig-chain c received in round i, let m' be the message contained by c. Update $S = S \cup \{m'\}$.
3. **Output** If $|S|= 1$, then p outputs the element $m' \in S$. If $|S| \neq 1$, then p outputs $\bot$.

Figure 4.1: Dolev-Strong Broadcast Π^{DS}

no two signatures in the sig-chain are computed using the same key. An i-sig-chain *contains* a message m' if m' is the message of the 1-sig-chain on which the sig-chain is built.

The protocol operates as follows: In the first round, the dealer creates a 1-sig-chain containing its input and sends the sig-chain to all parties. In every subsequent round i, any party that received a valid $i-1$ chain in the previous round that did not contain a signature that it had computed creates an $i-1$ sig-chain by appending its own signature to the chain. It then sends the i-sig-chain to all parties. In any round i, if a party receives a valid i-sig-chain, then it adds the message m contained in the sig-chain to a set of candidate outputs. If the set of candidate outputs contains only one candidate at the end, then the party outputs that message. Otherwise it outputs $\bot$.

Where Dolev-Strong Fails. In the send-corruption model, the Dolev-Strong protocol fails because it is possible for send-corrupt parties to output some message m while honest parties output $\bot$. Consider an execution in which the parties are partitioned into three sets:

H contains all of the honest parties, S contains all send-corrupt parties, and B contains all byzantine parties. (For the sake of this argument, we need not consider receive-corrupt parties.) Let the dealer be send-corrupt. It is possible that in this execution, the send-corrupt parties communicate only with parties in $S \cup B$. Then send-corrupt and byzantine parties can collectively build a $t_{\mathsf{byz}}+1$-chain containing m and no honest parties ever receives the dealer's message or any sig-chain containing the message. But this violates consistency, because all send-corrupt and honest parties are required to output the same thing.

Modifications. In order to resolve this problem, modify the protocol in two ways. First, a party must receive an $t_{\mathsf{snd}} + t_{\mathsf{byz}} + 1$-sig-chain for any message that it will output; no chain of less than $t_{\mathsf{snd}} + t_{\mathsf{byz}} + 1$ length may add a message to the set of candidate outputs. (This additionally requires that the protocol is run for $t_{\mathsf{snd}} + t_{\mathsf{byz}} + 1$ rounds.) Second, we update the bounds to require that $n > 2t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$. A majority of honest parties is necessary to ensure that honest parties can always build a $t_{\mathsf{snd}} + t_{\mathsf{byz}} + 1$-sig-chain without the assistance of byzantine or send-corrupt parties, which is necessary for validity.

The modified Dolev-Strong protocol (Π^{modDS}) is presented in Figure 4.2.

Theorem 4.2. *Π^{modDS} is a $(t_{\mathsf{snd}}, t_{\mathsf{byz}})$-secure broadcast protocol for $n > 2t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$.*

Proof. The proof is similar to the original by Dolev and Strong, subject to modifications described above. Validity follows from the fact that when $n > 2t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$ and the dealer is honest, the honest parties build a $(t_{\mathsf{snd}} + t_{\mathsf{byz}} + 1)$ sig-chain, and that no sig-chain can exist containing some m' that the dealer did not send. Consistency follows from the fact that if a $(t_{\mathsf{snd}} + t_{\mathsf{byz}} + 1)$ sig-chain exists, then some honest party's signature must be included. It follows that if any honest party output m, then all honest parties receive a $(t_{\mathsf{snd}} + t_{\mathsf{byz}} + 1)$ sig-chain containing m. Assume that some honest party receives a $(t_{\mathsf{snd}} + t_{\mathsf{byz}} + 1)$ sig-chain containing m and another honest party receives a $(t_{\mathsf{snd}} + t_{\mathsf{byz}} + 1)$ sig-chain containing

Protocol 2 Modified Dolev Strong Broadcast Protocol Π^{modDS}

Shared Setup: Public Key Infrastructure (PKI) for a signature scheme.
Inputs: The dealer $D \in \mathcal{P}$ has an input $m \in \{0,1\}^*$.
Outputs: Each party $p \in \mathcal{P}$ outputs a value $m' \in \{0,1\}^* \cup \{\perp\}$.
Local Variable: Each party $p \in \mathcal{P}$ maintains a local variable S, which is a set initialized to $\{\}$.
Protocol: The protocol begins at time 1 and proceeds in rounds. Each round party p proceeds as follows:

1. **Round 1: Dealer's Messages** The Dealer D signs its input $\sigma \leftarrow \mathsf{sign}_{\mathsf{sk}}(m)$ and sends (m, σ) to all parties.
2. **Sig Chains:** For every round i from 2 to $t_{\mathsf{snd}} + t_{\mathsf{byz}} + 1$: For every valid $(i-1)$-sig-chain c that p received at the end of round $i-1$ in which none of the signatures were constructed by p, p computes $\sigma \leftarrow \mathsf{sign}_{\mathsf{sk}}(c)$ and sends (σ, c) to all parties.
3. **Output:** For every valid $(t_{\mathsf{snd}} + t_{\mathsf{byz}} + 1)$-sig-chain c that p received at the end of round $t_{\mathsf{snd}} + t_{\mathsf{byz}} + 1$, let m' be the message contained by c and update $S = S \cup \{m'\}$. If $|S| = 1$, then p outputs the element $m' \in S$. If $|S| \neq 1$, then p outputs $\perp$.

Figure 4.2: Modified Dolev-Strong Broadcast Protocol Π^{modDS}

m'. Then both sig-chains must include an honest signature, and therefore there must be $(t_{\mathsf{snd}} + t_{\mathsf{byz}} + 1)$ sig-chain containing m and m' in the view of every honest party. It follows that every honest and send-corrupt party outputs $\perp$. □

Can Dolev-Strong Be Fixed to Support $n > t_{\mathsf{snd}} + t_{\mathsf{byz}}$? Without new ideas, Dolev-Strong cannot be updated to tolerate $n > t_{\mathsf{snd}} + t_{\mathsf{byz}}$ (which it is easy to prove is an optimal corruption budget). However, we cannot rule out such a threshold. In the pathological execution described above, honest parties do not send any messages if they do not receive any valid sig-chains. However, honest parties may send messages in each round containing $\perp$, indicating "I have not received a message," which conveys that the party's sent message *was not dropped.* This provides more information to the protocol, but we do not know how to use such a technique to improve broadcast.

4.1.2 Recent Techniques for Adaptive, Strongly Rushing Adversaries

Recent techniques for byzantine agreement and broadcast against a strongly rushing adversary also fail when requiring consistency between send-corrupt parties' outputs and honest parties' outputs, even when the adversary is not strongly adaptive. For example, the byzantine agreement protocol by Abraham et al. [2] and the broadcast protocol by Wan et al. [125] achieve security against a strongly adaptive adversary by effectively committing to any leader's messages early in the protocol, and then revealing a leader in a later round. This thwarts strongly rushing adaptive adversaries because by the time a leader is elected, it is too late to corrupt the leader and remove the messages it has sent.

In the partitioning attack, send-corrupt parties are able to communicate with each other but not with the honest parties, and are able to reach signature thresholds on messages that no honest party ever receives. For example, in [2], messages often require $b+1$ distinct signatures (implying at least one honest party signed a message) in order to be recognized by an honest party. But when there are more send-corrupt parties than honest parties, any threshold number of signatures that honest parties must be able to attain on their own must also be attainable by send-corrupt parties only. This can cause send-corrupt parties to adopt a different leader in some step than the honest parties. Similarly, in [125], send-corrupt parties' puzzles may never be delivered to honest parties. When honest parties choose a leader based on the solutions to a set of time-lock puzzles, send-corrupt parties may make a decision based on a larger set than the honest parties, and their decisions may differ. This form of attack is prevented by the implicit echoing assumption in [125], but it does not carry into the send-corrupt model. In our model, this attack is thwarted by requiring the number of honest parties be greater than $2(t_{\mathsf{snd}} + t_{\mathsf{byz}})$, as thresholds on the number of signatures can enforce that some honest party signs a message.

4.1.3 Generic Consensus from Broadcast

This section presents the the folklore construction (also discussed by Fitzi [68]) of achieving consensus using a broadcast primitive. Recall that we do not know how to prove an optimal corruption threshold for consensus in our regime for the general form of send corruption. By this construction, the problem is reduced to finding a broadcast protocol tolerating $n > t_{\mathsf{rcv}} + t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$.

Given a broadcast protocol Π^{B}, all parties simultaneously broadcast their inputs. Each party counts the number of broadcasts for which it outputs 0, 1, and $\perp$, and it outputs whichever of 0 and 1 appears more. In our model, we require that $n > t_{\mathsf{snd}} + t_{\mathsf{rcv}} + 2t_{\mathsf{byz}}$ to enforce that even if all send-corrupt parties' broadcasts and receive-corrupt parties' broadcasts output $\perp$ (because they fail in their own respective ways), a majority of the remaining parties are honest.

Note that it is still an open problem (with closest attempt coming from Wan et al. [126]) to obtain a constant round byzantine broadcast protocol for dishonest majority, with only crash and byzantine faults. Our model is still stronger than theirs, as we consider a strongly adaptive adversary (theirs is weakly adaptive) and permit send-corruptions.

Lemma 4.1. *If there exists a $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$ secure broadcast protocol for $n > t_{\mathsf{snd}} + t_{\mathsf{rcv}} + 2t_{\mathsf{byz}}$ then there is a $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-secure consensus protocol for $n > t_{\mathsf{snd}} + t_{\mathsf{rcv}} + 2t_{\mathsf{byz}}$.*

Proof. Given a $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$ secure broadcast primitive or protocol Π^{B} for $n > t_{\mathsf{snd}} + t_{\mathsf{rcv}} + 2t_{\mathsf{byz}}$, we construct a corresponding consensus protocol Π^{C} as outlined in Figure 4.3.

We prove that Π^{C} is a $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-secure protocol. Termination follows from the fact that Π^{B} terminates. Consistency follows from consistency of Π^{B}, which requires that if any live party output $m \in \{0, 1, \perp\}$ from any instance of Π^{B}, then all live parties output the same thing. It follows that every live party has the same values of n_0, n_1, and $n_\perp$.

Protocol 3 Consensus from Broadcast Π^{C}

Inputs: Each party $p \in \mathcal{P}$ has an input $b \in \{0,1\}$.
*Outputs:*Each party $p \in \mathcal{P}$ outputs $v \in \{0,1\}$.
Protocol: Each party p proceeds as follows:

1. **Broadcast Input:** Each party broadcasts its input to all other parties using Π^{B}.
2. **Count Received Bits:** For $u \in \{0,1,\perp\}$, let n_u be the number of broadcasts in the previous step for which p output u.
3. **Output:** Output $v \in \{0,1\}$ for which $n_v > n_{1-v}$. If $n_0 = n_1$ then output 1.

Figure 4.3: Generic Consensus from Broadcast Construction

Validity follows from the fact that even if all send-corrupt parties' broadcasts output $\perp$ and all receive-corrupt parties become zombies before completing their broadcast protocols (implicitly assuming that a protocol at worst outputs $\perp$ if a receive-corrupt sender does not complete the protocol), then there are still more honest parties who successfully broadcast than byzantine parties. This follows from $n - t_{\mathsf{snd}} - t_{\mathsf{rcv}} > 2t_{\mathsf{byz}}$.

□

4.2 Building Block Primitives

This section presents the building blocks for the expected-constant round consensus protocol in Section 4.3.

4.2.1 Digital Signatures and Coin Flipping

Our constructions require the use of a digital signature scheme. In particular, we assume that parties have access to a public key infrastructure (PKI) for a digital signature scheme, meaning each party is aware of a set of public keys $\{\mathtt{pk}_1, \ldots, \mathtt{pk}_n\}$, where $\mathtt{pk}_i$ is associated with p_i for $i \in [n]$. We assume that all parties choose their own public and private keys; in particular, some parties may adversarially choose their key pairs. Our constructions additionally assume an idealized signature scheme for which signatures are perfectly unforgeable; when using signature schemes that achieve unforgeability against computationally bounded adversaries, the protocols achieve our desired properties except with negligible probability.

Additionally, our construction requires the use of an unbiasable coin flipping protocol Π^{coin}. Our protocols assume idealized access to such a primitive, which may be considered to be implemented by an ideal functionality that takes no input (or more formally, takes as input the empty string) and delivers a uniformly random bit to all parties. At a high level, we require that:

- Until at least one live party queries Π^{coin} in the r-th invocation, the output for that invocation is uniformly distributed for the adversary.
- All live parties output the same value in Π^{coin}.

Such a coin flipping protocol may be instantiated (assuming a trusted setup) by augmenting threshold signatures [88] (using threshold $t_{\mathsf{byz}} + 1$, see below) with a protocol for reliable sends in our model, such as FixReceive ([131], or ours below).

We now define a series of building block protocols that build up to broadcast and binary consensus in our model. Recall that binary consensus and broadcast are already defined in Definitions 3.7 and 3.8.

4.2.2 Weak Broadcast

The first building block is a weak broadcast primitive. In a weak broadcast protocol, a dealer $D \in \mathcal{P}$ wishes to send a message $m \in \{0,1\}^*$ to the parties in $\mathcal{P}$. Each party $p \in \mathcal{P}$ outputs a message $m' \in \{0,1\}^* \cup \{\perp\}$, subject to the following constraints:

Definition 4.1 (Weak Broadcast)**.** *Let Π be a protocol for parties $\mathcal{P} = \{p_1, \dots, p_n\}$ and a distinguished party $D \in \mathcal{P}$ holds an input $m \in \{0,1\}^*$. Π is a* Weak Broadcast *protocol if the following properties hold except with negligible probability.*

1. $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$**-*Validity:*** Π *is* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-valid *if in every $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-compliant execution in which D is honest or receive corrupt (but not send-corrupt), every live party outputs m.*

2. $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$**-*Unanimity:*** Π *is* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-unanimous *if in every $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-compliant execution in which D is live, either every live party outputs $m \in \{0,1\}^*$ or every live party outputs $\perp$.*

3. $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$**-*Consistency:*** Π *is* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-consistent *if in every $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-compliant execution in which any honest party outputs $m' \in \{0,1\}^*$, every live party outputs m' or $\perp$.*

4. $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$**-*Termination:*** Π *is* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-terminating *if in every $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-compliant execution, every live party outputs some $m' \in \{0,1\}^* \cup \{\perp\}$ and terminates within finitely many steps.*

If Π is $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-valid, $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-consistent, and $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-terminating then we call it $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-secure. If Π is additionally $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-unanimous, then we call it $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-secure with unanimity.

4.2.3 Weak Consensus

In a weak consensus protocol, all honest parties have an input $b \in \{0, 1, \perp\}$, and all honest parties are expected to output a value $v \in \{0, 1, \perp\}$, subject to the following:

Definition 4.2 (Weak Consensus). *Let* Π *be a protocol for parties* $\mathcal{P} = \{p_1, \ldots, p_n\}$ *in which every party* $p \in \mathcal{P}$ *has an input* $b \in \{0, 1\}$. Π *is a* Weak Consensus *protocol if the following properties hold except with negligible probability.*

1. $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$***-Validity:*** Π *is* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-valid *if in every* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-compliant execution in which all honest parties have the same input* b *and no live parties have input* $1 - b$*, all honest parties output* b.

2. $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$***-Consistency:*** Π *is* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-consistent *if in every* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-compliant execution in which any live party outputs* $v \in \{0, 1\}$*, no live party outputs* $1 - v$.

3. $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$***-Termination:*** Π *is* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-terminating *if in every* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-compliant execution, every live party outputs* $v \in \{0, 1, \perp\}$ *and terminates within finitely many steps.*

If Π *is* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-valid,* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-consistent, and* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-terminating then we call it* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-secure.*

4.2.4 Graded Consensus

Graded consensus was originally introduced by Feldman and Micali [66]. In a graded consensus protocol, each party has an input $b \in \{0, 1\}$. Each party is expected to output a pair $(v, g) \in \{0, 1\}^2$, where v is the output bit and g is a *grade.*

Definition 4.3 (0/1 Graded Consensus). *Let* Π *be a protocol for parties* $\mathcal{P} = \{p_1, \ldots, p_n\}$ *where each party has input* $b \in \{0, 1, \perp\}$. Π *is a* 0/1 Graded Consensus *protocol if the following properties hold except with negligible probability.*

1. $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$***-Validity:*** *$\Pi$ is* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-valid *if in every* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-compliant execution in which all honest parties have the same input $b \in \{0,1\}$ and no live parties have input $1-b$, all live parties output $(b,1)$.*

2. $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$***-Consistency:*** *$\Pi$ is* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-consistent *if in every* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-compliant execution in which any live party outputs $(v,1)$, every live party outputs $(v,g) \in \{0,1\}^2$.*

3. $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$***-Termination:*** *$\Pi$ is* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-terminating *if in every* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-compliant execution, every live party outputs $(v,g) \in \{0,1\}^2$ and terminates within finitely many steps.*

If Π is $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-valid, $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-consistent, and $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-terminating then we call it $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-secure.

4.3 Expected Constant-Round Synchronous Consensus for $n > t_{\mathsf{rcv}} + 2t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$

This section presents a protocol for consensus in synchronous networks in the presence of send corruptions, receive corruptions, and byzantine corruptions where digital signatures are available. The protocol is $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-secure for $n > t_{\mathsf{rcv}} + 2t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$. Section 4.4 shows that the same protocol is $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-secure for $n > t_{\mathsf{rcv}} + t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$ when send corruptions are *spotty*, and that corruption budget is optimal.

Towards presenting the consensus protocol, we first present protocols for weak broadcast, weak consensus, and graded consensus. Before these building blocks, we introduce another protocol for reliable sending when all parties send messages to each other. A party detects whether it is receive-corrupt based on the number of messages it receives; if so, it becomes

Protocol 4 All-To-All FixReceive Protocol $\Pi^{FR}(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$

Inputs: Each party $p \in \mathcal{P}$ has an input $m \in \{0,1\}^*$.
*Outputs:*Each party $p \in \mathcal{P}$ outputs some message for every other party in $\mathcal{P}$, or outputs zombie.
Protocol: The protocol proceeds in two rounds, in which every party sends its input m to every other party, and then parties forward the unique messages they have received, as follows:

1. **Send Messages:** Each party sends its signed input m to every other party.
2. **Replay:** Every party forwards every unique message that it received in Round 1 to every other party. If a party did not receive any unique messages in Round 1, it sends $\perp$ to every other party.
3. **Output:** If a party p does not receive more than $n - t_{\mathsf{snd}} - t_{\mathsf{byz}} > t_{\mathsf{byz}} + t_{\mathsf{rcv}}$ messages (including $\perp$) in either round, then it sends zombie to all other parties, outputs $\perp$, and becomes a zombie. Otherwise, p outputs the set of unique messages that it received in Round 2.

Figure 4.4: All-to-all FixReceive Protocol Π^{FR}

a zombie and notifies the other parties. If not, it continues to participate and outputs the messages that it received.

4.3.1 All-To-All FixReceive

We present a protocol that is similar to FixReceive from [131], tuned for the common scenario in our protocols, in which all parties attempt to send a message to all other parties. The parties all forward unique messages that they receive, in order to ensure that every party either receives message that was sent, or detects that it is receive-corrupted. The parties output all unique messages that they receive during the protocol.

We prove that a receive-corrupt party that does not become a zombie must receive a message from another honest or send-corrupt party. We then prove that if some honest party attempts to send a message m via the protocol, then every non-zombie party must receive that message.

Lemma 4.2. *Any party p becomes a zombie during Π^{FR} only when it is receive-corrupt. If p*

does not become a zombie then it received a message from at least one honest or send-corrupt party.

Proof. If p receives fewer than $n - t_{\mathsf{snd}} - t_{\mathsf{byz}}$ then it must be receive-corrupt, since at most t_{snd} send-corrupt parties and t_{byz} byzantine parties may not send messages to p. If p does not become a zombie, then it must receive at least $n - t_{\mathsf{snd}} - t_{\mathsf{byz}} > t_{\mathsf{byz}} + t_{\mathsf{rcv}}$ messages. Therefore, one of the messages it received must have been from an honest or send-corrupt (but not also receive-corrupt) party. □

Lemma 4.3. *If an honest party or receive-corrupt party (but not send-corrupt) sends a message m using Π^{FR}, then every live party receives m or becomes a zombie.*

Proof. Follows from the fact that in the first round, all honest parties and send-corrupt receive m. In the second step, if any party p does not become a zombie, then it must receive a message either some honest or send-corrupt party, which must include m. □

4.3.2 Weak Broadcast

Our protocol for weak broadcast is presented in Figure 4.5. It follows a standard construction, adapted for our corruption model by invoking Π^{FR} to distribute messages. It permits a designated *dealer* to send an arbitrary message m to all parties, with the guarantee that every party outputs either m or $\perp$.

Lemma 4.4. *Protocol* $\Pi^{\mathsf{WB}}(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$ *is a* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-secure weak broadcast protocol for* $n > t_{\mathsf{snd}} + t_{\mathsf{rcv}} + 2t_{\mathsf{byz}}$.

Proof. Termination is trivial. We prove validity and consistency.

Protocol 5 Weak Broadcast Protocol Π^{WB}

Shared Setup: Public Key Infrastructure for a signature scheme, every party knows the identity of the dealer and its public key $\mathtt{pk}$.
Inputs: The dealer $D \in \mathcal{P}$ has an input $m \in \{0,1\}^*$.
Outputs: Each party $p_i \in \mathcal{P}$ outputs a value $m' \in \{0,1\}^* \cup \{\perp\}$.
Protocol: The protocol begins at time 0 and proceeds in rounds, in which each round lasts for Δ time. Each round party p proceeds as follows:

1. **Dealer's Messages**: The Dealer D signs its input $\sigma \leftarrow \mathsf{sign}_{\mathsf{sk}}(m)$ and sends $(\mathsf{deal}, m, \sigma)$ to all parties, where σ is the signature on m using its secret signing key $\mathtt{sk}$.

2. **Echo Dealer's Value**: Parties run Π^{FR} based on the messages they received from D. If p received a message from D, let (m', σ) be the message and signature that p received. p inputs $(\mathsf{echo}, m', \sigma)$ to Π^{FR}. Otherwise, p inputs $(\mathsf{echo}, \perp, \perp)$ to Π^{FR}.

3. **Replay:** Parties again run Π^{FR} based on the messages they received in the previous round, where each party provides all of the unique messages it received in the previous Π^{FR} as input.

4. **Verification and Output**: If p did not output any messages signed with D's key from the first run of Π^{FR}, then it outputs $\perp$. If in the outputs of the second run of Π^{FR}, p receives any two pairs (m'_i, σ_i) and (m'_j, σ_j) such that $m'_i \neq m'_j$ but $\mathsf{ver}_{\mathsf{pk}}(\sigma_i) = 1$ and $\mathsf{ver}_{\mathsf{pk}}(\sigma_j) = 1$, then p outputs $\perp$. Otherwise, p outputs the unique message m' that it received in the first run of Π^{FR} whose signature verifies with D's public key.

Figure 4.5: Weak Broadcast Protocol Π^{WB}

$(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$**-Validity:** If the dealer is honest or receive-corrupt (but not send-corrupt) then every honest party receives a valid signature on the dealer's input m from the dealer. Then, by Lemma 4.3, all non-zombie parties output m from the first run of Π^{FR}. By the unforgeability of our idealized signature scheme, no signed message m' under the dealer's key can be forged. Therefore, every live party outputs m.

$(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$**-Consistency:** Assume that honest party p outputs m and live party q outputs m'. Then p forwards m and its signature to q via the second invocation of Π^{FR}. By Lemma 4.3, q must output m and its signature from the second invocation of Π^{FR}, or become a zombie. Because q is not a zombie, it received a signed message containing m that verifies with the dealer's public key, and therefore does not output m', a contradiction. $\square$

We provide an additional statement about the outputs of Π^{WB} when the sender is corrupt but not byzantine. Specifically, consistency holds over the outputs of all live parties when the dealer is send-corrupt (and not only when some honest party outputs $m \neq \bot$).

Lemma 4.5. *When the dealer is send-corrupt, if one live party outputs $m \neq \bot$, then every live party outputs $m' \in \{m, \bot\}$*

Proof. Follows directly from unforgeability of the idealized signature scheme. □

4.3.3 Weak Consensus

We present our weak consensus protocol Π^{WC} in Figure 4.6. The protocol is an adaptation of the reduction from Weak Consensus to Weak Broadcast [68], modified for our corruption setting. Specifically, the protocol proceeds in two synchronous rounds. First, in parallel, each party signs its protocol input and sends its signed input to all parties. Second, upon receiving all other parties' inputs, each party attempts to generate a *certificate* in favor of some output value. A certificate for a bit u is a set of $n - t_{\mathsf{snd}} - t_{\mathsf{rcv}} - t_{\mathsf{byz}}$ unique, valid signatures on u. If a party is able to generate a certificate, it sends the certificate to all other parties.

A party outputs a bit v only if it meets three conditions: First, it must generate a certificate in the beginning of the second round; second, it must receive at least $n - t_{\mathsf{snd}} - t_{\mathsf{rcv}} - t_{\mathsf{byz}}$ valid certificates from distinct parties; third, it must not receive a valid certificate for $1 - v$ from any other party. Otherwise it outputs $\bot$.

Intuitively, validity of the protocol is guaranteed by the fact that if all live parties have input b, then all honest parties will be able to construct a certificate for b, and there will not be enough corrupt parties to construct a certificate for $1 - b$. Consistency is guaranteed by the fact that if two live parties are able to generate certificates for opposite values, then they

Protocol 6 Weak Consensus $\Pi^{\mathsf{WC}}(t_{\mathsf{cra}}, t_{\mathsf{byz}})$

Shared Setup: Public Key infrastructure for a signature scheme.
Inputs: Each party $p \in \mathcal{P}$ has an input $b \in \{0, 1, \perp\}$ and a secret signing key for the signature scheme.
Outputs: Each party $p \in \mathcal{P}$ outputs a value $v \in \{0, 1, \perp\}$.
Protocol: The protocol begins at time 0 proceeds in rounds, in which each round lasts for Δ time. Each party p_i proceeds as follows:

1. **Sign Inputs:** In parallel, each party signs its input bit and sends its signed input to all other parties.

2. **Construct Certificates and WB:** Each party collects all of the signed input bits from the other parties. If there is a $v \in \{0, 1\}$ for which $n - t_{\mathsf{snd}} - t_{\mathsf{rcv}} - t_{\mathsf{byz}}$ valid signed messages are received, p constructs a *certificate* composed of $n - t_{\mathsf{snd}} - t_{\mathsf{rcv}} - t_{\mathsf{byz}}$ signatures from distinct parties on v. The parties then invoke n weak broadcasts in parallel, in which p_i is the dealer in the ith weak broadcast, and p_i provides its certificate as input if it has one; otherwise p_i provides $\perp$ as its input.

3. **Output:** Each party receives any certificates sent to it in Round 2. If p constructed a certificate for some v in round 2 AND p has received at least $n - t_{\mathsf{snd}} - t_{\mathsf{rcv}} - t_{\mathsf{byz}}$ certificates for v by the end of round 2 from distinct parties AND p has not received a valid certificate for $1 - v$, then p outputs v. Otherwise, p outputs $\perp$.

Figure 4.6: Weak Consensus Protocol Π^{WC}

must share their certificates with each other, and then both output $\perp$.

Lemma 4.6. *Protocol $\Pi^{\mathsf{WC}}(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$ is a $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-secure Weak Consensus protocol in synchronous networks for $n > t_{\mathsf{rcv}} + \frac{3}{2}t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$.*

Proof. Termination is trivial. We separately prove validity and consistency.

$(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$**-Validity** : If all honest parties have input b, then because there are at least $n - t_{\mathsf{byz}} - t_{\mathsf{snd}} - t_{\mathsf{rcv}}$ honest parties, every honest party receives at least $n - t_{\mathsf{byz}} - t_{\mathsf{snd}} - t_{\mathsf{rcv}}$ signatures on b in the first round. It follows that at least $n - t_{\mathsf{byz}} - t_{\mathsf{snd}} - t_{\mathsf{rcv}}$ honest parties construct valid certificates for v and weak broadcast them to all live parties. By validity of Π^{WB}, all of these weak broadcasts are received by all live parties. Moreover, because no live parties have input $1 - b$ and $n - t_{\mathsf{byz}} - t_{\mathsf{snd}} - t_{\mathsf{rcv}} > t_{\mathsf{byz}}$, no certificate can be constructed by corrupt parties for $1-b$. Therefore, every live party outputs b.

$(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$**-Consistency** : Assume live party p outputs v and live party q outputs $1-v$. Then p must have received at least $n - t_{\mathsf{byz}} - t_{\mathsf{snd}} - t_{\mathsf{rcv}}$ certificates for v and q must have received at least $n - t_{\mathsf{byz}} - t_{\mathsf{snd}} - t_{\mathsf{rcv}}$ certificates for $1-v$.

Let A be the set of parties from which p received a certificate and B be the set of parties from which q received a certificate. Note that by validity and Lemma 4.5, no live party in A may also be in B, or vice versa; otherwise, q (respectively p) received a certificate for v (respectively $1-v$), and q (respectively p) did not output $1-v$ (respectively v). Therefore, only corrupt parties may be in both p and q, and there are at most b of them.

We proceed toward contradiction by showing that in fact, there must be some honest or receive-corrupt party in both A and B. We do so by arguing about the size of the union of A and B. If $|A \cup B| > t_{\mathsf{snd}} + t_{\mathsf{byz}}$, then there must be some honest or receive-corrupt party that weak broadcasted the same certificate to p and q, and by validity both p and q received that certificate. This is sufficient to conclude the proof, because then p or q does not output v or $1-v$, as argued above.

Recall that $|A \cup B| = |A| + |B| - |A \cap B|$. We have argued that $|A| \geq n - t_{\mathsf{byz}} - t_{\mathsf{snd}} - t_{\mathsf{rcv}}$, $|B| \geq n - t_{\mathsf{byz}} - t_{\mathsf{snd}} - t_{\mathsf{rcv}}$, and $|A \cap B| \leq t_{\mathsf{byz}}$. Then $|A \cup B| \geq 2(n - t_{\mathsf{byz}} - t_{\mathsf{snd}} - t_{\mathsf{rcv}}) - t_{\mathsf{byz}}$, and when $n > t_{\mathsf{rcv}} + \frac{3}{2} t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$, $|A \cup B| > t_{\mathsf{snd}} + t_{\mathsf{byz}}$. As explained above, this is a contradiction.

□

4.3.4 Graded Consensus

Graded consensus protocol Π^{GC} is presented in Figure 4.7; it is an adaptation to our fault model of the reduction of graded consensus to weak broadcast discussed by Fitzi [68]. Specif-

Protocol 7 Graded Consensus $\Pi^{\mathsf{GC}}(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$

Inputs: Each party $p \in \mathcal{P}$ has an input $b \in \{0, 1, \perp\}$
Outputs: Each party $p \in \mathcal{P}$ outputs a pair $(v, g) \in \{0, 1\}^2$
Protocol: The protocol begins at time 0 and proceeds in synchronous rounds, labeled below, where each round lasts long enough for its corresponding subprotocol to complete. Each party p proceeds as follows:

1. **Weak Consensus:** Run Π^{WC} with b as input. Let b' denote the output of Π^{WC}.
2. **Weak Broadcast:** In parallel, all parties invoke n copies of $\Pi^{\mathsf{WB}}(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$, where p_j is the dealer in the jth copy. p_j uses the value b' as its input to Π^{WB}. For $u \in \{0, 1, \perp\}$, let n_u denote the number of weak broadcasts for which p outputs u.
3. **Output:**
   - Assign $v \leftarrow u \in \{0, 1\}$ for which $n_u > n_{1-u}$. Break ties by assigning $v \leftarrow 1$. Assign $g \leftarrow 1$ if $n_v \geq n - t_{\mathsf{byz}} - t_{\mathsf{rcv}} - t_{\mathsf{snd}}$. Else $g \leftarrow 0$. Output (v, g)

Figure 4.7: Graded Consensus Protocol Π^{GC}

ically, Π^{GC} proceeds in synchronous rounds in which two subprotocols are invoked. First, parties invoke a weak consensus protocol, using their protocol inputs as input to the weak consensus protocol. Second, in parallel, all parties weak broadcast their outputs from the weak consensus protocol. Parties determine their outputs based on the weak broadcasts they receive. First, a party sets the bit v to the value $u \in \{0, 1\}$ for which it received more weak broadcasts carrying u than $1 - u$. Second, a party sets its grade g to 1 if it receives than $n - t_{\mathsf{byz}} - t_{\mathsf{rcv}} - t_{\mathsf{snd}}$ weak broadcasts carrying bit v, and sets its grade to 0 otherwise. It then outputs (v, g). Intuitively, each party outputs a bit v based on the majority of weak broadcasts that it has received. A party outputs grade 1 if it has received a large enough majority of weak broadcasts carrying v that it is guaranteed no other honest party has received a majority of weak broadcasts carrying $1 - v$. The proof follows from a standard quorum argument.

Lemma 4.7. *Protocol* $\Pi^{\mathsf{GC}}(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$ *is a* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-secure graded consensus protocol in synchronous networks for* $n > t_{\mathsf{rcv}} + 2t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$.

Proof. Termination is trivial. We separately prove validity and consistency.

$(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$**-Validity:** By the validity of Π^{WC}, if all honest parties have the same input $b \in \{0, 1\}$ and no live parties have input $1-b$, then every live party outputs b from Π^{WC}, and no live party outputs $1-b$ from Π^{WC}. Next, every honest party weak-broadcasts b via Π^{WB}, so there are at least $n - t_{\mathsf{byz}} - t_{\mathsf{snd}} - t_{\mathsf{rcv}}$ weak broadcasts from which each honest party outputs b. Moreover, because no live party outputs $1-b$ from Π^{WC}, there are at most t_{byz} executions of weak broadcast from which any party outputs $1-b$. Because $n - t_{\mathsf{byz}} - t_{\mathsf{snd}} - t_{\mathsf{rcv}} > t_{\mathsf{byz}}$ (by assumption), each honest party outputs b as its value. Because at least $n - t_{\mathsf{byz}} - t_{\mathsf{snd}} - t_{\mathsf{rcv}}$ honest parties weak broadcasted b, each live party outputs 1 as its grade.

$(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$**-Consistency:** Suppose a live party p_i outputs $(v, 1)$ for $v \in \{0, 1\}$ and a live party p_j outputs $(1-v, g)$ for some $g \in \{0, 1\}$.

First we establish that no live party weak-broadcasted $1-v$ in Round 2. It must be the case that p_i output v from at least $n - t_{\mathsf{byz}} - t_{\mathsf{snd}} - t_{\mathsf{rcv}}$ parties in Round 2. Because $n - t_{\mathsf{byz}} - t_{\mathsf{snd}} - t_{\mathsf{rcv}} > t_{\mathsf{byz}}$, there must be some live party that sent v to p_i. Let this honest party be q. It must therefore be the case that q output v from the execution of Π^{WC}. By the consistency of Π^{WC}, no live party output $1-v$ from Π^{WC}, and therefore no live party weak-broadcasted $1-v$.

Next, consider that p_i received at least $n - t_{\mathsf{byz}} - t_{\mathsf{snd}} - t_{\mathsf{rcv}}$ weak broadcasts of v. We now consider the view of p_j

1. In at most s weak broadcasts, the dealer was send-corrupt; therefore p_j output $\perp$ from at most s of the broadcasts with live dealer from which p_i output v.
2. Let b^* be the number of weak broadcasts with byzantine dealer from which p_i output v. By consistency of weak broadcast, p_j must output v or $\perp$ from those weak broadcasts.

Because p_j output $(1-v, g)$ it must be the case that $n_{1-v} > n_v$ in p_j view. By the above statements, n_v must be at least $n - t_{\mathsf{byz}} - t_{\mathsf{snd}} - t_{\mathsf{rcv}} - s - b^*$ in p_j's view. Because only byzantine parties may have weak broadcasted $1 - v$, and because b^* corrupt parties did not weak broadcast $1 - v$, n_{1-v} is at most $t_{\mathsf{byz}} - b^*$ in p_j's view. This is a contradiction because $n - t_{\mathsf{byz}} - t_{\mathsf{snd}} - t_{\mathsf{rcv}} - s - b^* > t_{\mathsf{byz}} - b^*$ when $n > r + 2t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$, and therefore $n_v > n_{1-v}$ in p_j's view and p_j did not output $(1 - v, g)$. This is a contradiction. □

4.3.5 Expected Constant Round Consensus

Figure 4.8 presents Π^*, our expected-constant round consensus protocol. The protocol follows the standard coin-loop paradigm to go from graded consensus to byzantine agreement. The protocol ensures that when a party terminates, it holds a certificate that it can send to all parties in order to make them terminate with the same value.

Theorem 4.3 (Main Theorem). $\Pi^*(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$ *is a* $\Pi^*(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-secure consensus protocol in synchronous networks for* $n > t_{\mathsf{rcv}} + 2t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$*, where a common coin primitive is available.*

We proceed to prove the theorem via a sequence of lemmas. We start with validity.

Lemma 4.8 (Validity). $\Pi^*(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$ *is* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-valid in synchronous networks for* $n > t_{\mathsf{rcv}} + 2t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$.

Proof. If all live parties have input v, then by $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-validity of Graded Consensus, each live party outputs $(v, 1)$ from every iteration of Graded Consensus. It follows that the first time Π^{coin} outputs v, all live parties output v. □

Before proving consistency and termination, we prove the following claim of any $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-compliant execution of Π^* that will facilitate the proofs of both properties:

Protocol 8 Expected Constant Round Protocol $\Pi^*(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$

Common Setup: The parties have access to a public key infrastructure for some signature scheme.
Inputs: Each party $p \in \mathcal{P}$ has an input $b \in \{0,1\}$
Outputs: Each party $p \in \mathcal{P}$ outputs some $b' \in \{0,1\}$
Internal Variable: Each party maintains a variable $v \in \{0,1\}$ which is initialized to b. For each $u \in \{0,1\}$, each party also maintains a set D_u of distinct (decide, u) messages that it has received.
Protocol: The protocol begins at time 0 and proceeds in synchronous rounds. Each party p proceeds as follows:

– **Loop** starting with iteration $i = 0$ until terminating:

1. **Subround A (Graded Consensus):** Run $\Pi^{\mathsf{GC}}(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$ with v as input. Let (u, g) denote p's output of Π^{GC}.
2. **Subround B (Common Coin):** Invoke a common coin protocol Π^{coin} and assign to ϕ_i the output.
3. **Conditional Update:** If $g = 0$, then update $v \leftarrow \phi_i$. If $g = 1$, then update $v \leftarrow u$.
4. **Conditional Decision:** If $g = 1$ and $v = \phi_i$: sign (decide, v), send the signed message to all parties, and output v.
5. **Certificate Send:** All parties invoke Π^{FR}, where any party that has generated or received a *certificate* since the last invocation of Π^{FR} provides the certificate as input, and terminates after Π^{FR}. Any party that does not have a certificate inputs $\perp$.

– **Certificate:** Upon receiving a signed (decide, u) message from any party, add the message to D_u. When D_u contains at least $t_{\mathsf{byz}} + 1$ messages from distinct parties, construct a *certificate* of $t_{\mathsf{byz}} + 1$ (decide, u) messages from distinct parties. Upon receiving a certificate, output u (if have not already output).

Figure 4.8: Expected Constant Round Consensus Protocol Π^*

Claim 4.1. *Let i be the iteration in which the first live party p outputs u. Then in every iteration $s > i$ while no honest party has terminated, every live party outputs $(u, 1)$ from subprotocol Π^{GC}.*

Proof. We will show that at the end of iteration i, every live party updates its internal value v to u. The claim follows from the fact that in the following iteration, every live party inputs

u to Π^{GC}. By validity of Graded Consensus, this implies that every live party outputs $(u, 1)$ from Π^{GC} in that iteration and maintains the value of its internal variable v.

Inductively, as long as no honest party terminates, no live party ever changes its internal variable v after iteration i. This follows from the fact that all honest parties maintain the value of their internal variable v as long as all honest parties have input v and no live party has input $1 - v$; although send-corrupt or receive-corrupt parties may terminate before an honest party, their inputs are treated as $\perp$ in the subsequent executions Π^{GC}, and the honest parties' internal variable of v is maintained by validity.

Now we show that at the end of iteration i, every live party updates its internal value v to u. We consider the two following cases for any live party $q \neq p$, based on q's output from Π^{GC} in iteration i. By consistency of Graded Consensus, because p output $(u, 1)$, q may output either $(u, 0)$ or $(u, 1)$ from Π^{GC}:

1. q output $(u, 0)$. Then q updates its internal variable v to the value ϕ from the coin tossing in that iteration. By the fact that p outputs v in iteration i, $\phi_i = v$.
2. q output $(u, 1)$. Then q updates its internal variable v to the value u by the protocol specification.

□

Lemma 4.9 (Consistency)**.** $\Pi^*(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$ *is* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-consistent in synchronous networks for* $n > t_{\mathsf{rcv}} + 2t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$.

Proof. To prove consistency, we show that if some live party p outputs u, then no live party q ever outputs $1 - u$. Recall that are two methods by which a party may produce output. We enumerate them as follows:

1. First, a party may output in iteration i when its internal variable v matches ϕ_i.

2. Second, a party may output by receiving a certificate of signed decision messages.

Assume that in a $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-compliant execution of Π^*, some live party p outputs u and another live party q outputs $1-u$. First we claim that no two live parties may output conflicting values if both output by method 1.

Claim 4.2. *In any execution of Π^*, no two live parties r and s may output u and $1-u$, respectively, by method 1.*

Proof. Let there be an execution in which live party r outputs u by method 1 and live party s outputs $1-u$ by method 1. If r and s both output by method 1, they must have different values of their internal variable v at the moments they output. This is because when any party p outputs a value u by method 1, it must hold u in its internal variable v. However, because in every iteration, r and s both output the same value from Π^{coin}, and because each party outputs only when the output of Π^{coin} matches the value of its internal variable v, r and s may not both output in the same iteration. Therefore, r and s must produce their outputs in different iterations.

Without loss of generality, let r produce output before s, and let s be the first live party to output $1-u$ by method 1 after r outputs u. (If s was not already the first live party to output $1-u$ by method 1 after r outputs u, then proceed to derive contradiction with respect to the first such party.) By Claim 4.1, every live party must have u in the value of its internal variables v until some honest party terminates. Therefore, because s must have $1-u$ in the value of its internal variable v when it outputs $1-u$, some honest party must terminate between the time that r produces output and s produces output. Let w be the first honest party to terminate, and let it terminate in iteration i. Since w is the first honest party to terminate, by Claim 4.1 w must terminate after outputting u. But then w sends its certificate via Π^{FR} in round i, and by Lemma 4.3, s received w's certificate or s becomes a zombie at the end of Π^{FR}. Then s does not output $1-v$, which is a contradiction. □

It follows from Claim 4.2 that p and q may not both have output by method 1. Therefore, at least one party must have output by method 2.

Before concluding the proof, we claim that if any live party outputs a value u by method 2, then some live party must have output u by method 1. This follows directly from the fact that a valid certificate requires $t_{\mathsf{byz}} + 1$ signed (decide, u) messages. In particular, at least one live party's signed (decide, u) message must be included in any certificate, and parties only produce signed (decide, u) messages when producing output by method 1.

We conclude the proof using this claim. Without loss of generality, let p output u by method 1 and q output $1 - u$ by method 2. Then by the previous claim, there must be two honest parties that output conflicting values by method 1, which is a contradiction with Claim 4.2. We derive a similar contradiction with Claim 4.2 if two live parties output conflicting values by method 2. □

Lemma 4.10 (Termination). *For all $\kappa \geq 1$ and $n > t_{\mathsf{rcv}} + 2t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$, with probability at least $1 - \frac{1}{2^\kappa}$, every live party running $\Pi^*(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$ terminates in at most $2\kappa + 1$ iterations.*

Proof. To analyze the probability that the all live parties terminate after m iterations, we separate the analysis into two steps. First, we define a *unanimous iteration* as an iteration at the end of which all live parties set their internal variables v to the same value. We will denote the first unanimous iteration of the protocol by i^* and analyze how many iterations the protocol requires until its first unanimous iteration. Second, we analyze how many iterations the protocol requires until all honest parties terminate after i^*.

There are two possible ways that a unanimous iteration may occur:

1. If every live party outputs $(\cdot, 0)$ from Π^{GC} in iteration i, then at the end of iteration i, every live party updates its internal variable v to ϕ_i.

2. If some live party outputs $(u, 1)$ from Π^{GC} in iteration i and $\phi_i = u$, then at the end of the iteration, all live parties update their internal variable v to u.

Therefore, i^* is the first iteration in which either all live parties output $g = 0$ from Π^{GC} or in which some live party outputs $(v, 1)$ from Π^{GC} and Π^{coin} outputs v. Conditioned on the fact that some live party outputs $(\cdot, 1)$ from Π^{GC} in each iteration, it follows from the fact that Π^{coin} is not biasable that each iteration i is unanimous with probability $\frac{1}{2}$. It follows that i^* occurs by iteration ℓ with probability at least $1 - \frac{1}{2^\ell}$. Let all live parties set their internal variable v to some $v^* \in \{0, 1\}$ at the end of iteration i^*. Next we claim that every live party outputs a bit no later than the next iteration $s > i^*$ in which $\phi_s = v^*$. The claim follows as a direct consequence of Claim 4.1, since all live parties are guaranteed to output $(v^*, 1)$ from Π^{GC} in every iteration after i^*, and in the case that $\phi_s = v^*$, every party that has not yet produced output must output v^*. Because Π^{coin} is not biasable, $\phi_s = v^*$ with probability $\frac{1}{2}$ in each iteration $s > i^*$. It follows that every live party outputs v^* by iteration $i^* + \ell$ with probability at least $1 - \frac{1}{2^\ell}$. It follows from linearity of expectations that all live parties output a bit from Π^* within 2κ iterations with probability $1 - \frac{1}{2^\kappa}$. When all honest parties output a bit, it follows that they all sign and send (decide, v) messages to each other. At latest, each party receives a certificate at the end of the iteration in which all honest parties output. Therefore, all live parties terminate Π^* within 2κ iterations with probability $1 - \frac{1}{2^\kappa}$. □

4.4 Optimal Synchronous Consensus for Spotty Send Corruptions

In this section, we present slight modifications to the proofs in Section 4.3 that show Π^* achieves better corruption bounds when send-corruptions are spotty. We then prove that protocol Π^* is optimal in the number of corruptions it tolerates when send-corruptions are spotty.

4.4.1 Analysis for Spotty Send Corruptions

We now show that Π^* achieves better bounds for send-corruptions when such send failures are spotty.

Theorem 4.4. *When send-corruptions are spotty and a common coin primitive is available, Π^* is $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-secure for $n > t_{\mathsf{rcv}} + t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$.*

The proof follows in the remainder of this section, by updating the bounds and proofs of the underlying building block protocols.

Weak Broadcast With Unanimity. First we show that Π^{WB} is $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-secure *with unanimity* when send failures are spotty, for $n > t_{\mathsf{rcv}} + t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$. Because the construction does not change, we simply append the proof of unanimity.

Lemma 4.11. *Π^{WB} is $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-unanimous for $n > t_{\mathsf{rcv}} + t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$ when send corruptions are spotty.*

Proof. We show that if the dealer is live and broadcasts m, then every live party either outputs m or every live party outputs $\perp$. By the unforgeability of our idealized signature scheme, no signed message m' under the dealer's key can be forged. Therefore, whether every party outputs m or every live party outputs $\perp$ is determined completely by whether D's send succeeds in the first round of Π^{FR}. If D is honest, receive-corrupt, or if its send succeeds, then every honest and send-corrupt party receives m in the first round of Π^{FR}. If any receive-corrupt party does not receive m in the first round of Π^{FR}, then it must receive m in the second round, or become a zombie. $\square$

Weak Consensus We show that Π^{WC} is $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-secure for $n > t_{\mathsf{rcv}} + t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$ when send corruptions are spotty. We do not need to update the protocol, and we update only the proof of consistency.

Lemma 4.12. Π^{WC} *is* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-consistent for* $n > t_{\mathsf{rcv}} + t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$ *when send corruptions are spotty.*

Proof. The proof is identical to the one for Lemma 4.6, except that if send-failures are spotty, then we require only that $|A \cup B| > b$. This is because if any send-corrupt party sends a weak-broadcast that is received by any honest party, it must be received by all others by unanimity of Weak Broadcast, Lemma 4.11. □

Graded Consensus We next show that Π^{GC} is $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-secure for $n > t_{\mathsf{rcv}}+t_{\mathsf{snd}}+2t_{\mathsf{byz}}$ when send corruptions are spotty. Once again, we do not need to update the protocol, and for this protocol we only update the proof of consistency.

Lemma 4.13. Π^{GC} *is* $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$*-consistent for* $n > t_{\mathsf{rcv}}+t_{\mathsf{snd}}+2t_{\mathsf{byz}}$ *when send corruptions are spotty.*

Proof. The proof is identical to the one in Lemma 4.7, except that we need not consider send-corrupt parties whose weak broadcasts are delivered to p_i but not to p_j.

Because p_j output $(1-v, g)$ it must be the case that $n_{1-v} > n_v$ in p_j view. Then n_v must be at least $n - t_{\mathsf{byz}} - t_{\mathsf{snd}} - t_{\mathsf{rcv}} - b^*$ in p_j's view. Because only byzantine parties may have weak broadcasted $1-v$, and because b^* corrupt parties did not weak broadcast $1-v$, n_{1-v} is at most $t_{\mathsf{byz}} - b^*$ in p_j's view. We reach a contradiction because $n - t_{\mathsf{byz}} - t_{\mathsf{snd}} - t_{\mathsf{rcv}} - b^* > t_{\mathsf{byz}} - b^*$ when $n > r + t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$, and therefore $n_v > n_{1-v}$ in p_j's view and p_j did not output $(1-v, g)$. □

And it follows that Π^{GC} is $(t_{\mathsf{snd}}, t_{\mathsf{rcv}}, t_{\mathsf{byz}})$-secure because the proof of validity requires only $n > t_{\mathsf{rcv}} + t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$.

Expected Constant Round Consensus The protocol and proof of Π^* do not need to be updated, as they inherit the bounds of the building block protocols, all of which are secure for $n > t_{\mathsf{rcv}} + t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$.

4.4.2 Optimality with Respect to Spotty Send and Byzantine Corruptions

We now prove that Π^* is optimal in the number of corruptions it tolerates with respect to send corruptions and byzantine corruptions when send-corruptions are *spotty*. The proof does not consider a model with receive-corrupt parties. Recall that this model generalizes the crash failure model.

Theorem 4.5 (Optimal Send- and Byzantine Fault Tolerance)**.** *There is no protocol for synchronous consensus in the mixed fault model which permits zombie processes that tolerates t_{snd} send corruptions and t_{byz} byzantine corruptions for $n \leq t_{\mathsf{rcv}} + t_{\mathsf{snd}} + 2t_{\mathsf{byz}}$*

Proof. The proof considers only send-corrupt and byzantine parties. It can be trivially extended to include a factor for t_{rcv} parties simply by adding an additional group of receive corrupt parties and forcing them to become zombies as the protocol begins.

Assume there is a consensus protocol Π resilient to t_{snd} and t_{byz} faults for $t_{\mathsf{snd}} + 2t_{\mathsf{byz}} \geq n$. We proceed by analyzing three separate executions of Π. As a tool towards analyzing executions, we first divide the set of parties $\mathcal{P}$ into three groups: A, B, and S. Group A has $n - t_{\mathsf{snd}} - t_{\mathsf{byz}}$ parties, Group B has $t_{\mathsf{byz}} \geq n - t_{\mathsf{snd}} - t_{\mathsf{byz}}$ parties, and Group S has t_{snd} parties. In the following three executions, the schedules of messages sent by and delivered to all parties are identical. Only the corruption status of parties across groups A, B, and S differ.

1. **Execution 1:** In this execution, all of the parties in Group A are honest and all

have input 1. All the parties in Group B are byzantine and act as if they were honest parties with input 0. All of the parties in Group S are send-corrupt and do not send any messages, but have input 1. By validity, the honest parties (Group A) must output 1.

2. **Execution 2:** In this execution, all of the parties in Group A are byzantine and act as if they have input 1. All of Group B are honest and have input 0. All of the parties in S are send corrupt and do not send messages, but have input 0. By validity, all of the honest parties (Group B) must output 0.

3. **Execution 3:** In this execution, all parties in Groups A and B are honest. Group A all have input 1 and Group B all have input 0. All of the parties in S are send corrupt and do not send messages, but have input 1.

We now analyze the outputs of the parties in Execution 3. Notice that to the parties in Group A, this execution is identically distributed to Execution 1, so they must output 1. To the parties in Group B, this execution is identically distributed to Execution 2, so they must output 0. This violates consistency. □

Chapter 5

Permissionless Consensus

In this chapter, we depart from the classical regime of consensus protocols and consider a radically different regime, called *permissionless* consensus. In a traditional consensus protocol, as in Chapter 4, the set of parties participating in the protocol is known beforehand, and protocols are proven secure with respect to thresholds on the proportion of total parties which are corrupted. In the permissionless regime, the set of participating parties is unknown to the protocol and it may be unbounded; therefore, classical quorum and signature-based techniques fail.

The work on which this chapter is based [122] proposes a general abstraction for a special, elite category of message, whose supply is constrained by the protocol, and which is implemented via constraining physical resources. We argue that Proof of Work, Proof of Stake, and many other Proof of X (PoX) implement this abstraction, which is simply termed *resources*. We show that by constraining the adversary's ability to control these special messages, consensus in the permissionless regime is possible. Specifically, as long as the honest parties know a rough upperbound on the rate at which resources enter the system, and that the honest parties receive a majority of resources *in the long term*, consensus is possible, independent of the specific number or ratio of parties controlled by the adversary.

On the Power of Resources: An Adversarial Environment

Weak Network Assumptions. This model assumes a highly adversarial network environment, with weaker synchronization assumptions than comparable studies of permissionless consensus. Parties do not know the network delay, do not have clocks, and there is no bound on the number of participants. To date, all other works we know for the permissionless model require either knowledge of the network delay [25, 50, 59] or (weakly synchronized) clocks [16, 17, 51, 74], plus some assumption about the number of active participants.

Separating PoX Layers. The "competition process" for resources is modeled implicitly (and more generally) by deferring it to the (adversarial) environment. This decouples the resource-producing process (e.g. mining) from the resource-consuming process (e.g. a graph protocol). Moreover, the environment has *full information* about the states of all honest parties, it can corrupt parties adaptively, and it chooses which parties receive resources and when they receive them. (Note that in the full information model, there are no secure digital signatures; this work shows that resources imply consensus without signatures.)

Player-Replaceability. Any protocol that is secure in our model must achieve consensus even when every honest participant sends *at most one message* before it leaves the execution, and even when every honest participant is only active for a (very) short period of time from the moment it joins to the moment it leaves. (In the extreme, just long enough to receive the state of the execution and send a single message.)

On Knowing the Rate Limit

It is easy to show via partitioning attack (Section 5.4) that some constraint on the network is necessary in order to achieve consensus. However, knowing a bound on the resource rate relative to the network delay is a *weaker* assumption than knowing the network delay. Given knowledge of the network delay, participants can execute synchronous protocols that proceed in rounds. This is possible because the adversary cannot manipulate honest participants' timekeeping abilities. A bound on just the resource rate does not directly yield synchronization, since the adversary may induce a large difference in two honest processors' views at the same moment in time by selectively delivering corrupt resources to one honest participant but not to another.

The question remains of why it suffices to constrain the rate at which resources enter a system relative to the maximum network delay. In their seminal work, DLS [58] showed that consensus is possible in *partially synchronous* environments in which there exists any relationship between processor synchronization and the maximum network delay. The protocol in Section 5.5 uses the rate at which resources are allocated to measure *aggregate* processor activity. Even when parties are constantly joining and leaving an execution, it is possible to aggregate the amount of computational work they do over time. This relationship between aggregate activity and the network delay suffices for consensus.

In practice, it has been reasonable to assume knowing the rate of resource allocations relative to the network delay *despite the fact that the true network delay is unknown.* For example, a PoW system is parameterized by estimating the time required to propagate a block through the network (as by [52]) and then tuning a hardness parameter to bound the rate of puzzle solutions per "safely estimated" network delay. PoS systems also tune their parameters to achieve a certain number of PoS solutions per round.

On Determinism

Our work is the first we know that attempts to model PoX in a deterministic model that gives the adversary the ability to determine which parties get resources and when resources are allocated. This is significant for more than just the power of the adversary. FLP [67] and Ben-Or [23] showed a separation between the feasibility of (classical) consensus protocols in deterministic and randomized fully asynchronous models. In comparison, our work is the first to show deterministic consensus in the permissionless model, and although our network is not completely asynchronous, we prove in Section 5.4 that *some* network assumption is necessary in the permissionless model.

Although the protocols in Sections 5.5 and 5.6.2 are deterministic, our model is still capable of capturing nondeterministic lottery-style protocols. In most PoX protocols, the sources of nondeterminism in the execution are the lottery selection and message delivery over the network. Our model strengthens this this nondeterminism by replacing it with an adversarial allocation.

5.1 Resources: An Abstraction for PoX

Resources are an abstraction of a special protocol message which is supply-constrained by some factor external to the protocol. In most protocols that use PoX, the PoX carry additional, unique semantics (meaning two parties cannot point to the same PoX and claim different semantics), and a party that receives a PoX can verify its validity with respect to the external process that produced it. PoX can also refer to other PoX. We next explain the properties chosen as for the resources abstraction at a high level, then formalize them via a syntactic model in Section 5.1.2.

5.1.1 Resources Intuition

The following informal rules govern how resources appear in an execution:

1. **Unforgeability** No participant can "fake" the fact that it has a resource. In practice, PoX schemes enforce this requirement by requiring that PoX solutions must be found by solving some puzzle, and the solutions are verifiable by other participants. An execution satisfies *resource unforgeability* if no resource appears in the execution before its allocation event. This enforces that parties are constrained to obtaining resources *only* by receiving them from the environment, which abstracts the resource-producing process.

2. **Binding** Each resource can be *bound* with one and only one string, which gives the resource semantics. The string must be chosen at the moment that the resource is generated. This models that in PoX schemes, parties attempt to solve puzzles with respect to a specific message they wish to send. (In some implementations, this message includes a public key that boostraps special status to future messages signed with that key.) A string m bound to a resource ψ is encoded as $\psi||m||\psi$ [1], where $||$ denotes concatenation. An execution satisfies *resource binding* if for any two encodings $\psi||m||\psi$ and $\psi'||m'||\psi'$: $\psi = \psi'$ implies $m = m'$.

Constraining the Supply of Resources

Constraints on the supply of resources over arbitrary periods of time enforce their scarcity:

1. **Long Term Honest Majority** Over any period of time in which n resources are allocated, we require that $\alpha n - \varepsilon$ are allocated to honest participants and at most

[1] This is a standard encoding technique. By encompassing the message with its resource, it is clear where the string bound to the resource begins and ends

$\beta n + \varepsilon$ are allocated to corrupt participants, where $\beta = 1 - \alpha$. When $\alpha > \beta$, we say that honest participants receive a *long term* majority of resources. ε represents a short-term corrupt advantage, which models an adversary which pools its physical resources in order to achieve a short "burst" of resources.

2. **Rate Limit** We let ρ upperbound the number of resources that may be generated per Δ time, where Δ is the (unknown) maximum network delay.

5.1.2 Execution Constraints

This section describes formal constraints on an execution that give resources semantics and constrain their supply, as discussed above.

Resources are a set of symbols Ψ that augment the alphabet used by the protocol to compose strings. The environment *allocates a resource* to some party by writing the resource to the party's incoming message tape. When a party is allocated a resource, it must choose the semantics of the resource by *binding* a string to it.

Definition 5.1 (Resource Binding)**.** *An execution satisfies* resource binding *if*

1. *every bound resource ψ_m is properly encoded as $\psi||m||\psi$, and*
2. *for any two bound resource encodings $\psi||m||\psi$ and $\psi'||m'||\psi'$, if $\psi = \psi'$ then $m = m'$.*

Just as no participant can solve a PoX puzzle without evaluating some function, no participant may send a bound resource to another participant if the resource has not been allocated. This property is *unforgeability.*

Definition 5.2 (Resource Unforgeability)**.** *An execution respects resource* unforgeability *if no resource is sent in any message before it is allocated by the environment.*

An *admissible* execution satisfies the previous two constraints. All of the theorems we prove are with respect to admissible executions.

Definition 5.3 (Admissible Execution)**.** *An execution is* admissible *if it respects resource allocation and resource unforgeability.*

Resource Allocations and Corruption The following notation constrains resource allocations by both the rate of allocation and by the proportion of honest allocations.

Definition 5.4 (ρ-Rate-Limiting)**.** *Let $\rho \in \mathbb{N}$ and let Δ be the communication synchronization constant, or the network delay. An execution with network delay Δ is* ρ-rate-limited *if for all t, there are at most ρ resources allocations between t and $t + \Delta$.*

The following notation denotes how many resources are allocated to honest and to corrupt participants over some span of time.

Definition 5.5 ($\Psi^{(t,t')}$,$\Psi^{(t,t')}_{\mathsf{hon}}$,$\Psi^{(t,t')}_{\mathsf{cor}}$)**.** *We denote the sets of resources allocated by the environment to all participants, honest participants, and corrupt participants between times t and t' in an execution as follows:*

- $\Psi^{(t,t')}$ *is the set of all resources allocated between t and t'.*
- $\Psi^{(t,t')}_{\mathsf{hon}}$ *is the set of resources allocated to honest participants between t and t'.*
- $\Psi^{(t,t')}_{\mathsf{cor}}$ *is the set of resources allocated to corrupt participants between t and t'.*

The following notation constrains the proportions of resources allocated to honest and corrupt parties:

Definition 5.6 ((α, ε)-honest resource allocation, (β, ε)-corrupt resource allocation)**.** *Let $\alpha \in [0, 1]$ and let $\varepsilon \in \mathbb{N}$. An execution satisfies (α, ε)-honest resource allocation if for all* *times* $t, t' > t$: $|\Psi^{(t,t')}_{\mathsf{hon}}| \geq \alpha|\Psi^{(t,t')}| - \varepsilon$. *Equivalently, let* $\beta = 1 - \alpha$. *An execution satisfies* (β, ε)*-corrupt resource allocation if for all times* $t, t' > t$: $|\Psi^{(t,t')}_{\mathsf{cor}}| \leq \beta|\Psi^{(t,t')}| + \varepsilon$.

Intuitively, α and β capture the long-term ratios of honest and corrupt resource allocations, respectively, and ε represents a small amount of "slack" in the ratios. ε also captures the short term advantage that corrupt participants may obtain in receiving resources. Note that for *any* t and t' for which $|\Psi^{(t,t')}| < \frac{\varepsilon}{\alpha}$, all of the resources allocations may be to corrupt parties.

5.2 Real-World Implementations of Resources

What Makes a Pox? We now intuitively explain how the most popular forms of PoX puzzles implement resources. For each scheme *every puzzle solution constitutes a resource.* PoX are made hard to obtain by the fact that physical resources in the system are constrained, and cryptographic puzzles enforce that each PoX is semantically bound to a single string. Protocols then base their security guarantees on the constraint that a majority of the special messages must be generated by honest participants.

To formally show that any PoX scheme implements resources, one would proceed through the intermediate step of defining a "fair allocation" of resources, then prove that the PoX implementation achieves fair allocation. Proving a fair allocation should require with overwhelming probability that over a long enough period of time, honest parties compute a majority of puzzle solutions, and that the rate at which solutions are found be upper-bounded. Because full formal proofs of PoX schemes require careful analysis in each scheme's syntactic model, such proofs are out of scope of this thesis.

5.2.1 Proof of Work

In a Proof of Work (PoW) scheme, parties attempt to find solutions to a hash puzzle for a cryptographic hash function H. A "solution" to a hash puzzle is a string x for which $\mathsf{H}(x) < D$, where D is a difficulty parameter. In most PoW schemes, the input x is composed of a nonce, a payload, and a pointer to a previous puzzle solution. When a puzzle solution is found, we consider the input x to be the string that is bound to the resource. Note that to strictly implement resources, it is not necessary that an input to a hash puzzle include a pointer to a previous hash puzzle; however, this property is used by many Proof-of-Work protocols to prevent precomputation and to enforce a graph structure.

Unforgeability of a resource in a PoW scheme follows from verifiability of the hash function. Honest parties can easily verify that a string x is a valid solution to the hash puzzle $\mathsf{H}(\cdot) < D$. **Binding** of a resource follows from the collision resistance of the hash function. Given a resource bound to string x, in order to claim the resource has been bound to another string, a corrupt party must find an x' such that $\mathsf{H}(x') = \mathsf{H}(x)$. (See [99] for a discussion.) **Honest majority** of PoW schemes follows from the assumption that honest parties maintain a majority of active computational power at all times.

Rate limiting of PoW schemes is enforced in practice by regularly retargeting the difficulty parameter. The difficulty parameter is set based on the total hash power of the network (measured as the number of hash function evaluations per second, this is an estimate of physical computing resources) in order to target a particular rate of puzzle solutions. For Bitcoin, the difficulty parameter is set such that a puzzle solution is found about every 10 minutes [102]; in Ethereum the difficulty parameter is set such that a puzzle solution is found about every 13 seconds [64]. In order to show that a proof-of-work scheme implements resources, one would have to identify realistic assumptions from which to show that difficulty calibration of the hash puzzle effectively upperbounds the rate at which hash puzzles are found.

5.2.2 Proof of Stake

In Proof of Stake (PoS), during each time step each participant evaluates some number of virtual lottery tickets to determine if it is the leader in the protocol at that time step. The number of lottery tickets each party can evaluate at any time step is proportional to its stake in the system at that time. In every PoS, a lottery ticket evaluates some function $F(x, \texttt{pk})$, where pk is a public key associated with stake in the system, and x encodes a time slot and some protocol state *which must contain entropy.* For a "winning ticket," the message bound to the corresponding resource is therefore $(x, \texttt{pk})$; this ties the public key pk to the global state of the system at the time it becomes the leader.

Rate-limiting is imposed by parameterizing each Proof of Stake protocol to upperbound the number of winning lottery tickets that are evaluated per time step. (This is analogous to parameterizing the number of proof of work solutions per network delay, or upperbounding the number of resources that are allocated per span of time.) We remark that each of the schemes that we overview relies on either knowing the maximum communication delay of the network or on loosely synchronized clocks in order to synchronize the rounds of the lottery.

Lottery by VRF In the PoS schemes of both Ouroboros [51] and Algorand [74], lottery tickets are implemented using a verifiable random function (VRF) [51, 96]. A participant evaluates a lottery ticket by computing $\mathsf{vrf.prove}_{\texttt{sk}}(x) \to \pi$, where sk is the secret key associated with a stake in the system that the participant owns a particular time, x is the state of the system, and π is the output of the vrf. (We elide details about Algorand's cryptographic sortition.) A lottery ticket is a "winner" if $\pi < D$, for some tunable parameter D. To verify the role of a claimed leader, other participants must verify the VRF via

$\mathsf{vrf.vfy}_{\mathtt{pk}}(x, \pi)$, where $\mathtt{pk}$ is the public key associated with $\mathtt{sk}$.

Binding follows from the unpredictability of the VRF. Given one solution, it should be hard to find another input to the VRF that evaluates to the same proof. **Unforgeability** follows from the verifiability of the VRF, because one cannot fake that a puzzle solution has been found.

Lottery by Hash Function The PoS mechanisms of Snow White [50] and FKTZ [65] use a cryptographic hash function that is seeded by a stateful nonce which depends on the previous hash puzzle solutions. Specifically, the PoS is evaluated as $\mathsf{H}(\mathsf{st}, \mathtt{pk}, t) < D$, where D is a difficulty parameter, st is a protocol state, $\mathtt{pk}$ is a public key for a digital signature scheme, and t is a timestamp ([65] includes a signature σ on st in the input to H). If $\mathsf{H}(\mathsf{st}, \mathtt{pk}, t) < D$ then the participant with public key $\mathtt{pk}$ becomes a leader. **Binding** follows from collision resistance of the hash function, and **unforgeability** follows from the verifiability of the hash function.

Enforcing Honest Majority: Preventing Grinding PoS constructions must argue that the adversary cannot increase its share of puzzle solutions to be more than roughly its proportion of stake in the system. This requires proving that the adversary cannot efficiently "predict" keys which will be leaders in any particular time slot. (If it could, it might create a long adversarial branch to compete with the honest branch.) Existing PoS constructions require that the state encoded in the input x contains enough entropy that the outputs are unpredictable, making pre-computation attacks and "grinding" attacks computationally infeasible. For a full treatment, refer to the discussions in each of the PoS implementations we have referenced. In [51] refer to the discussion on VRF Unpredictability under Malicious Key Generation (Section 3.2). In [74] refer to the discussions on choosing the VRF seed in each round and setting secret keys well before each round (Sections 5.1, 5.2). In [50], refer to the discussion on security under adversarially biased hashes (Sections 2 and G). The recent work of Fan, Katz, Thai, and Zhou [65] show a PoS scheme with maximal unpredictability in which all parties simultaneously mine PoS on multiple chains.

5.2.3 Other Cryptographic and Non-Cryptographic PoX

There have been many additional cryptographic PoX variants proposed, for example Proof of Spacetime [101] and Proof of Retrievability [98]. We do not analyze them all here. However, we remark that PoX need not necessarily be implemented cryptographically. For example, Proof of Elapsed Time [118, 42] elects leaders in a consensus protocol via verifiable timer. Additionally, resources could be implemented in low-power environments in which participants seldom have enough energy to send a message. In this case, every message would be associated with a resource, as the resource represents physical energy. Future research could study ways to move resource allocation to the environment (e.g. by random lottery based on external factors), rather than by solving hash puzzles.

An Example: Satellites and Base-Stations For the sake of exposition, we give an extended example of how resources may be implemented without PoX puzzles. Imagine that a number of "base stations" on the surface of Earth wish to perform consensus, and the base stations communicate with some known number of satellites whose orbits around the earth follow randomized paths. A base station may communicate with a satellite only when the satellite is "overhead," meaning within the line of sight of the base station. When a satellite is overhead, a base stations may request a resources from the satellite by requesting that it issue a signature on a message that the base station requests. Each satellite is programmed to respond to each request it receives with probability $\frac{1}{2}$, subject to the constraint that a satellite signs *at most one message per 24 hours*, where time is kept in the UTC time zone. If the number of satellites is known to the base stations, all of the satellites' public keys are known to the base stations, and the satellites can be trusted to only sign messages subject to the constraints above, then a satellite's signature on a message constitutes a resource. Each signature is bound to the message it signed. The rate limit is computable because the rate of resources per day is bounded by the total number of satellites. Unforgeability of resources follows from unforgeability of the signature scheme.

5.3 Termination and Liveness Based on Resources

Our definition of termination for permissionless consensus and our definition of liveness (Definition 3.10) in graph consensus depend on resources. We discuss them in detail here, with the context of both intuition and definitions of resources.

Participants are required to terminate only if sufficiently many resources have been allocated. In comparison, classical definitions require participants to terminate after finitely many steps. Intuitively, because our model features asynchronous parties that do not have clocks and do not know the network delay, they have no way to tell how many other parties are in an execution who haven't yet sent messages. The only constraint by which termination can be enforced is the number of resources in the execution.[2]

Definition 5.7 (Termination)**.** *An execution satisfies* termination *if there exists a positive integer R^* such that if R^* resources have been allocated by the environment at time t, then for every honest participant p active at any time $t' > t$: p outputs* 0 *or* 1*.*

For graph consensus, we desire that a protocol is live with respect to the total number of resources that have been allocated by the environment, and not those simply the number in a party's view. This definition forces the claim of a protocol's liveness to account for the

[2]Chapter 5 argues that an upperbound on the resource rate is a weaker form of synchronization than knowing the network delay.

possibility of withholding by corrupt parties who do not send messages to honest parties using the resource they receive.

We can now restate the definition of f-Liveness:

Definition 3.10 (f-Liveness)**.** *Let $f\colon \mathbb{N} \to \mathbb{N}$. A protocol* Π *satisfies* f-liveness *if in every execution, for every time t and honest participant p active at t: if the environment has allocated N resources by time t, then $|G_p^{*(t)}.V| \geq f(N)$.*

5.4 Necessary Assumptions for Consensus in the Permissionless Model

This section incluides proofs that both (1) a long term majority of honest resources, and (2) some constraint on the network delay, are necessary for consensus. (Recall that our model bounds the network delay relative to the resource rate, which Section 5 argues is weaker than directly bounding the network delay.)

Theorem 5.1. *There is no consensus protocol in the permissionless regime that does not require a long-term honest majority of resources.*

Proof. Assume there is a protocol Π that achieves consensus in the permissionless regime without an honest majority of resources. We proceed by describing several similar executions, and bring contradiction at the end.

In each execution, we divide the participants into two groups, A and B. In Execution 1, all participants in A are honest and have input $b \in \{0, 1\}$. All participants in B are corrupt, and act as if they were honest with input $1 - b$. Group A collectively receives fewer resources than Group B, but a sufficient proportion of resources for Π to guarantee consensus. By validity, all honest participants must output b.

In Execution 2, we divide the same participants in to the same groups, A and B. All participants in A are corrupt and act as if they are honest with input b. All participants in B are honest and have input $1 - b$. The activation schedule, including the allocation of resources, in Execution 2 is identical to Execution 1. Again by validity, all honest participants must output $1 - b$.

Now consider a third execution, Execution 3. In Execution 3 we divide the same participants into the same groups, A and B. However, all parties are honest. In Group A all parties have input b, and in Group B all parties have input $1 - b$. The activation schedule, including the allocation of resources, in Execution 3 is identical to Executions 1 and 2. Because the view of every participant in A is the same as in Execution 1, each participant in A must output b. Similarly, each participant in B must output $1 - b$. This violates consistency. $\square$

Theorem 5.2. *There is no consensus protocol in the permissionless regime that does not require a constraint on the network delay.*

Proof. If there is no constraint on the network delay known to the honest parties, then the proof follows from a standard partitioning attack, similar to that of Pass and Shi [105]. For completeness, we present a full proof here.

Consider an execution in which all participants are honest, and an adversary that can partition the honest parties into two groups, A and B, such that all honest parties in group A have input $b \in \{0, 1\}$ and all honest parties in group B have input $1 - b$. By validity and termination, there must be some execution in which A output b if no messages sent by parties in B are received by A, and similarly there must be some execution in which B output $1 - b$ if no messages sent by A are received by parties in B. If there is no constraint on the network delay, then an adversary can delay messages sent by parties in A until after the parties in B have output $1 - b$, and similarly the adversary can delay messages sent by parties in B until after the parties in A have output b. This violates consistency.

Note that the proof holds if the network is asynchronous by our definition of asynchrony, or if the network is partially synchronous but the parties do not know any constraint on Δ (relative to any known parameter). Specifically, the protocol cannot depend on Δ, and therefore there must be values of Δ for which groups A and B output their values before Δ time has elapsed. $\square$

5.5 A Permissionless Consensus Protocol

Protocol Π^G, presented in Figure 5.1, is a graph consensus protocol. It is parameterized by α and ε, which describe the proportion of honest resources which are allocated (Definition 5.6), and the maximum rate of resource allocation ρ (Definition 5.4).

5.5.1 Intuition

At a high level, our graph protocol uses the properties of resources to build a directed acyclic graph (DAG) which captures the (partial) ordering in which the honest parties receive resources. Importantly, every resource received by an honest party is associated with a vertex in the global DAG (much like every PoX is associated with a vertex in a blockchain). The honest participants embed structure into the DAG that can be used to infer when corrupt parties attempt to cheat by "withholding" their resources, i.e. not immediately multicasting a vertex they have added to the graph. The unforgeability and binding properties of resources enforce that corrupt participants cannot manipulate the graph structure other than choosing where to add their vertices.

The structure that honest participants build into the global DAG is *reachability*. Every honest vertex which is added to the global DAG is guaranteed to gain an honest successor, and to always be a predecessor of one of the deepest vertices in the global DAG. Importantly, we require that the honest participants can build deeper branches on the DAG than the corrupt participants. If honest participants can build longer paths in the global DAG over time than corrupt participants, then if corrupt participants withhold their vertices for too long, their withheld branches will eventually fall behind the depth of the global DAG.

The technical challenge is to compute how long it takes – measured in depth – for a withheld branch to fall short of the honest parties' branch. Honest participants extract their outputs by selecting vertices in their local views of the global DAG which are predecessors of the deepest vertices in their views, excising all corrupt vertices on branches which have fallen short.

5.5.2 Formal Description

Each participant p maintains a local DAG G_p in which every vertex except the root is a resource. The graph G_p is initialized to $(\{\mathsf{root}\}, \emptyset)$, and grows from the root toward high depths throughout the execution as participants are allocated resources and receive messages. Whenever p is allocated a resource, it adds the resource to its graph as a new vertex, and then immediately multicasts its local graph including the new vertex to all honest participants. When an honest participant receives a message containing a graph, it updates its local graph to include new vertices and edges not previously in its local graph. We must show how a participant p chooses the predecessors of each vertex that it adds to its graph, and p computes its output G_p^* from its local graph G_p.

Resources are described as vertices as follows. When any participant is allocated resource ψ, we let v_ψ denote the vertex corresponding to ψ. When describing an arbitrary vertex, we denote it as v or u, eliding its respective resource.

When any honest participant p adds a new vertex to its graph, it adds the vertex to its graph as the new deepest vertex. Specifically, when p is allocated a resource ψ and adds vertex v_ψ to its local graph G_p, p adds an inbound edge to v_ψ from every vertex u in G_p which (a) has no outbound edges in G_p, and (b) is close in depth to G_p. When p is allocated ψ, it must also choose v_ψ's edges *immediately*, as p must bind the inbound edges of v_ψ to ψ. Because each vertex's inbound edges are bound to the vertex's respective resource, it may not gain additional predecessors.

Over time, some vertices will gain successors and some vertices may be "orphaned" and stop gaining successors. Each participant computes its output G_p^* as a subgraph of its G_p consisting of vertices which are both far from the end of its graph (measured in the difference in depth between the vertex and the graph) and are still gaining successors.

Encoding a Graph Using Resources

We model a resource as a black box object which is *bound to a string* that conveys its semantics *at the moment* it is allocated. In Π^G, the string bound to each resources encodes the direct predecessors of its respective vertex; when a participant is allocated a resource ψ, it binds to ψ the encoding of each vertex which has an outbound edge to v_ψ. If no edges are bound to ψ, then v_ψ is defined to have an edge from root. In this way, each vertex is uniquely committed to its predecessors at the moment it is allocated.

Event Responses

We now detail how participants respond when they are allocated resources and when they receive messages, and we explain how participants compute their outputs from their local graphs.

Protocol 9 DAG Protocol for Graph Consensus $\Pi^G(\alpha, \varepsilon, \rho)$

Parameters: $\alpha, \varepsilon, \rho$

Derived Constants:

1. $\beta = 1 - \alpha$
2. $\gamma = (1+\beta)\rho + \varepsilon + \frac{\varepsilon}{\rho} + 1$
3. $c = \gamma + \rho + \frac{\varepsilon}{\alpha}$
4. $\ell_1 = \gamma + \rho$
5. $\ell_2 = c(\varepsilon+1) + \rho + \frac{c\beta}{\frac{\alpha}{\rho} - c\beta}(c(\varepsilon+1) + (2+\beta)\rho + \frac{\varepsilon}{\alpha} + 2\frac{\varepsilon}{\rho} + 2)$
6. $\ell^* = \ell_1 + \ell_2$

Internal Variables:

1. $G_p = (V_p, E_p)$ is a participant's local state. Initially, $G_p = (\{\mathsf{root}\}, \emptyset)$
2. $G_p^* = (V_p^*, E_p^*)$ is a participant's output graph. Initially, $G_p^* = (\emptyset, \emptyset)$

Event Responses:

```
1. On Receiving a Graph ($G'$)
   – $G_p \leftarrow G_p \cup \mathsf{validateGraph}(G')$
   – $G_p^* \leftarrow \mathsf{extract}(G_p)|_{\mathsf{D}(G_p)-\ell^*}$
2. On Being Allocated a Resource $\psi$
   – $G_p \leftarrow \mathsf{addVert}(G_p, \psi)$
   – multicast $G_p$
   – $G_p^* \leftarrow \mathsf{extract}(G_p)|_{\mathsf{D}(G_i)-\ell^*}$
```

Internal Functions:

```
1. $\mathsf{addVert}(G, \psi)$:
   – $V' \leftarrow \{u \in G.V : \mathsf{D}(G) - \mathsf{D}(u) < c$ and $\mathsf{outdegree}(u) = 0\}$
   – return new graph $G'$ such that
      – $G'.V \leftarrow G.V \cup \{v_\psi\}$
      – $G'.E \leftarrow G.E \cup \{(u, v_\psi) : u \in V'\}$
2. $\mathsf{extract}(G)$:
   – $S \leftarrow \{v \in G.V : \mathsf{D}(G) - \mathsf{D}(v) \le c + \rho\}$ // “starting vertices”
   – return $S \cup \{v \in G.V : \exists u \in S$ such that $u$ is reachable from $v\}$
3. $\mathsf{validateGraph}(G')$:
   – if
      (a) $\exists (u, v) \in G'.E$ such that $\mathsf{D}(u) - \mathsf{D}(v) > c$, or
      (b) $\exists (u, v) \in G'.E$ such that $u \notin G'.V$
      then return $(\emptyset, \emptyset)$
   – return $G'$
```

Figure 5.1: Protocol Π^G for graph consensus

On Resource Allocation When an honest participant p is allocated a resource ψ, we say that it *generates* a vertex v_ψ that it adds to its local graph G_p. Participant p chooses the inbound edges of v_ψ based on its current graph G_p by adding an edge to v_ψ from each vertex u in G_p for which both $\mathsf{outdegree}(u) = 0$ and $\mathsf{D}(G_p) - \mathsf{D}(u) < c$, where c is a constant computed from the protocol parameters and is the maximum depth spanned by an honestly chosen edge. Immediately after generating v_ψ, p multicasts its entire local graph containing v_ψ and its inbound edges.

On Receipt of a Message Every message sent between participants is an encoding of a graph. (Any other message is ignored.) When a participant p receives a graph G' in a message, it verifies that G' is a valid graph. If G' is valid, then p updates its local graph as $G_p \leftarrow G_p \cup G'$. If G' is not valid, then p ignores G'.

G' may be invalid in two ways. First, G' may contain an edge (v, u) which spans more than c depth. Second, G' may be "missing a vertex," meaning there is a vertex v in $G'.V$ for which not all of v's predecessors are in $G'.V$. (This means the graph G is incomplete in the party's view.)

Computing Output An honest participant p computes its output G_p^* from its local graph G_p by first extracting a subgraph of G_p into an intermediate graph, and then outputting all but the deepest vertices in the intermediate graph. More precisely, p extracts a subgraph of G_p using the procedure $\mathsf{extract}(G_p)$, as follows. First, p selects a set of "starting vertices" as the set $S = \{v \in G_p \colon \mathsf{D}(G_p) - \mathsf{D}(v) < c + \rho\}$. Next, p extracts every starting vertex and every vertex from which any starting vertex is reachable. Finally, p outputs $G_p^* \leftarrow \mathsf{extract}(G_p)|_{\mathsf{D}(G_p)-\ell^*}$, which contains all the vertices in its extracted subgraph with depth less than $\mathsf{D}(G_p) - \ell^*$, where ℓ^* is derived from the protocol parameters.

Remark 5.1 (Sending a Whole Graph)**.** *Whenever a participant generates a new vertex, it* *multicasts its entire graph. This is unrealistic in practice, and (like most blockchain protocols) we believe with additional analysis it is possible to reduce this to sending the latest vertex.*

5.5.3 Theorem Statement

Our main theorem states that protocol Π^G satisfies graph consensus for appropriate parameters.

Theorem 5.3. *For all N, all ρ, and all ε, and for all $\alpha > \rho(1-\alpha)((3-\alpha)\rho+\frac{\varepsilon}{\alpha}+\frac{\varepsilon}{\rho}+\varepsilon+1)$ every (α,ε)-honest, ρ-rate-limited, admissible execution of $\Pi^G(\alpha,\varepsilon,\rho)$ satisfies graph consistency and f,h-liveness for $f(N) = h(N) = \alpha N - \varepsilon - \rho(\ell^* + 1)$, where ℓ^* is a derived constant defined as in the protocol.*

Recall that in Π^G, each participant computes its output by extracting a subgraph from its local graph and then chopping off the deepest vertices in the extracted subgraph, where the chop-off threshold is the derived constant ℓ^*. Liveness follows from the fact that as a participant's local graph increases in depth, the depth of the graph which it outputs also increases. The main objective of the proof is to show that the protocol achieves graph consistency.

The main desideratum of the proof of graph consistency follows:

Proposition 5.1. *Let $c = (3-\alpha)\rho + \frac{\varepsilon}{\alpha} + \frac{\varepsilon}{\rho} + \varepsilon + 1$ (as in Protocol Π^G). If $\alpha > \rho\beta c$, then for all k, times t and t', and honest participants p and q active at t and t', respectively, if $\mathsf{D}(G_p^{(t)}) > k + \ell^*$ and $\mathsf{D}(G_q^{(t')}) > k + \ell^*$, then $\mathsf{extract}(G_p^{(t)})|_k = \mathsf{extract}(G_q^{(t')})|_k$.*

where c and ℓ^* are defined as in Π^G.

Graph consistency follows directly from assigning $G_p^* \leftarrow \mathsf{extract}(G_p)|_{\mathsf{D}(G_p)-\ell^*}$, since when two honest participants output graphs, then the less deep output graph must always be a subgraph of the deeper (if the output graphs have the same depth, then they must be the same graph).

5.5.4 Proof Overview

We now overview the proof of Proposition 5.1. The full proofs of Proposition 5.1 and Theorem 5.3 are in Appendix C.

Building a Virtual Global Graph We consider that the participants collectively build a virtual global graph $\mathbb{G}$ throughout an execution. When the execution begins, $\mathbb{G}$ is initialized to a graph with only a root vertex. Whenever *any* participant is allocated a resource, the vertex that it generates is immediately added to $\mathbb{G}$. In particular, even if a corrupt participant generates a vertex and "withholds" the vertex by not sending it to any honest participant, the vertex is still added to $\mathbb{G}$ at the moment that it is generated. We denote by $\mathbb{G}^{(t)}$ the state of $\mathbb{G}$ after all vertices are added at time t.

$\mathbb{G}$ represents the global state of the execution. Consider that $G_p^{(t)}$ is p's its local view of $\mathbb{G}^{(t)}$, and it is easy to see that $G_p^{(t)}$ must be a subgraph of $\mathbb{G}^{(t)}$. Moreover, for every vertex $v \in \mathbb{G}^{(t)}.V$, if v is in $G_p^{(t)}$, then $\mathsf{D}_{\mathbb{G}^{(t)}}(v) = \mathsf{D}_{G_p^{(t)}}(v)$. Henceforth, when we refer to the depth of a vertex, we simply write $\mathsf{D}(v)$ because its depth is uniquely defined.

Outputting Predecessors and Omitting Orphans Recall that an honest participant p active at time t outputs a vertex v from its local graph $G_p^{(t)}$ if and only if $v \in \mathsf{extract}(G_p^{(t)})|_{\mathsf{D}(G_p^{(t)})-\ell^*}$. By applying $\mathsf{extract}()$ and chopping off the deepest vertices, the protocol enforces two requirements in order to output a vertex. First v must be far from the end of a participant's graph ($\mathsf{D}(G_p^{(t)}) > \mathsf{D}(v) + \ell^*$). Second, v must be a predecessor of one of the starting vertices in $G_p^{(t)}$.

Intuitively, one can consider that every participant p decides whether each vertex v in its view should be output or not. However, p "waits" before making a decision until v is sufficiently far from the end of its graph. At that point, p does not output v only if v has been "orphaned." A vertex is "orphaned" if it is more than ℓ^* depth from the end of a graph but not a predecessor of one of the graph's starting vertices.

To achieve graph consistency, p must make the same decision on v as every other honest participant. We show that by the time the depth of G_p exceeds ℓ^* more than the depth of v, v's status as an orphan or not an orphan has been determined in $\mathbb{G}$ and will not change; moreover, v's orphan status in G_p must mirror its status in $\mathbb{G}$. If v is not a predecessor of one of the starting vertices in G_p, then v will never be a predecessor of a starting vertex in any honest participant's local graph which is deep enough to decide on v. However, if v is a predecessor of one of the starting vertices in G_p, then v will never be orphaned in any honest participant's local graph.

Consistency of Honest Vertices

We first show consistency of the honest vertices which honest participants output. We do so by showing that no honest vertex is ever orphaned, and therefore *all* honest vertices are eventually output by honest participants. Our high-level lemma towards this statement actually says something stronger. It says that every honest vertex in $\mathbb{G}$ which is more than $\ell_1 < \ell^*$ distance from the end of an honest participant's graph must be extracted from the graph when it computes its output from its local graph.

Lemma 5.1 (Honest Vertex Extraction)**.** *For every time t, honest participant p active at t, and honest vertex $v \in \mathbb{G}^{(t)}$: $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) > \ell_1 \implies v \in \mathsf{extract}(G_p^{(t)})$.*

Lemma 5.1, consistency of honest vertices in participants' outputs, follows trivially from composition of Lemmas 5.2 and 5.3, described below. Lemma 5.2 shows that by the time

$\mathsf{D}(G_p) > \mathsf{D}(v) + \ell_1$ for any honest participant's graph G_p and honest vertex v, enough time must have passed since v was originally multicast that v is in G_p. Lemma 5.3 shows that every such honest vertex in an honest participant's graph must be a predecessor of a starting vertex in the graph.

Consistency of Honest Vertices in Honest Views For the first step, we show that if an honest participant's local graph G_p is deeper than an honest vertex v by more than a fixed distance ℓ_1, then $v \in G_p$.

Lemma 5.2 (Depth-Based Indicator for Honest Vertices)**.** *For all t, honest p active at t, and honest vertex $v \in \mathbb{G}^{(t)}$: $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) > \ell_1 \implies v \in G_p^{(t)}$.*

Intuitively, ℓ_1 is derived as follows. Let t_v be the time that some honest vertex v is generated by honest participant q. Naively, one would like to claim that if $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) > \rho$, then ρ vertices must have been generated after v, and it follows from the rate limit on resource allocations (Definition 5.4) that $t > t_v + \Delta$. However, the naive attempt makes the unfounded assumption that at t_v, v must be the deepest vertex in $\mathbb{G}^{(t_v)}$. Instead, we derive a constant γ that gives the maximum difference between $\mathbb{G}^{(t)}$ and an honest view $G_p^{(t)}$ at any time t. We then derive $\ell_1 = \gamma + \rho$ and show that if $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) > \ell_1$, then Δ time must have elapsed since v was generated and multicast. It follows that $v \in G_p^{(t)}$.

Extracting Every Honest Vertex Recall that an honest participant extracts the starting vertices in its graph and all their predecessors, and then outputs only the vertices which are far from the end of its graph. We show that an honest participant always extracts *every* honest vertex in its graph.

Lemma 5.3 (Extracting All Honest Vertices in a Local Graph)**.** *For every time t, honest participant p active at t, and honest vertex $v \in \mathbb{G}^{(t)}$: $v \in G_p^{(t)} \implies v \in \mathsf{extract}(G_p^{(t)})$.*

The lemma follows by showing that every honest vertex v eventually gains at least one honest successor which is not too far from v, measured in terms of depth. Intuitively, after an honest vertex v is generated, the first vertex generated by an honest participant with v in its view must be a successor of v. It follows that for every honest vertex v which is not a starting vertex in an honest participant's graph, there must be a path from v to a starting vertex in the graph.

Consistency of Corrupt Vertices

We show that consistency of corrupt vertices follows from consistency of their honest successors (or lack thereof). If every vertex is honestly generated and immediately multicast, then no vertex is ever orphaned. Only if a corrupt participant withholds a vertex can the vertex be orphaned. We show that after a corrupt vertex is generated, there is a limited time during which it must gain an honest successor or it will be orphaned. Imagine that starting at some time in an execution, corrupt participants use all of their resources to build a "withheld branch" B of $\mathbb{G}$ which includes no honest vertices, while honest participants continue to build $\mathbb{G}$ as per the protocol. Intuitively, if $\frac{\alpha}{\rho} > \beta$ (as we require), then the corrupt participants cannot keep pace with the honest participants, and eventually B will fall behind the depth of $\mathbb{G}$. We can compute for how long a withheld branch B can remain close in depth to $\mathbb{G}$. We derive a constant ℓ_2 for which any vertex which is ℓ_2 depth from the end of an honest participant's local graph and is a predecessor of a starting vertex must have an honest successor.

Lemma 5.4 (Honest Reachability Requirement for Extraction)**.** *For all t, participant p active at t, and vertex $v \in \mathsf{extract}(G_p^{(t)})$: $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) > \ell_2$ implies there exists an honest vertex u reachable from v such that $\mathsf{D}(u) - \mathsf{D}(v) \leq \ell_2$.*

Recall that an honest participant decides whether to output a vertex v only once v is $\ell^* =$

$\ell_1 + \ell_2$ depth from the end of its local graph. If v is a predecessor of a starting vertex, then it must have an honest successor which is more than ℓ_1 depth from the end of the graph. This honest successor must be in every honest participant's local graph with depth sufficient to output v; therefore, because u must be extracted from every honest view in which it exists, every honest participant with local graph deep enough to output v must do so.

5.6 From Graph Consensus To One-Bit Consensus

5.6.1 A Generic Transformation

We now show that one-bit consensus is implied by any graph consensus protocol which guarantees a long-term majority of honest vertices are accepted by honest parties. Specifically, we show that for any protocol that satisfies (a) graph consistency and (b) h-liveness such that there exists some N^* for which for all $N \geq N^*$: $h(N) > \frac{N}{2}$, there must exist a one-bit consensus protocol secure under the same parameters.

Theorem 5.4. *For any graph consensus protocol Π that satisfies both graph consistency and h-liveness for which there exists some N^* for which for all $N \geq N^*$: $h(N) > \frac{N}{2}$, there exists a one-bit consensus protocol that satisfies agreement, termination, and nontriviality under the same parameters.*

Proof. The proof transforms Π into a one-bit consensus protocol. We let Π^b represent the transformed protocol. The transformation works as follows. Whenever a participant generates a vertex, it binds an additional one-bit label, which is the participant's input bit, to the vertex. The participants run Π^b without producing output until a majority of the vertices output by the underlying Π must be honest vertices, and then they compute a majority of the bit labels of the graph output by Π. Termination follows because any honest participant with enough vertices in its graph can output a bit. Nontriviality follows because a majority of the parties' extracted vertices must be honest vertices. Agreement follows because honest participants compute the majority bit of vertex labels in *the same* output graph.

Honest participants run Π^b until they can output from their local graphs the smallest graph containing at least $\frac{N^*}{2}$ vertices. By h-liveness, there must be some point at which honest participants can output a graph with at least $\frac{N^*}{2}$ vertices. If not, then there would not be there exists an N^* such that for any honest participant p's local graph $G_p^{(t)}$ at time t for which $|G_p^{(t)}.V| > N^*$, that $|\mathsf{hon}(G_p^{*(t)}.V)| \geq \frac{|G_p^{(t)}.V|}{2} \geq \frac{N^*}{2}$.

We must argue that honest participants identify the *same* smallest graph containing at least $\frac{N^*}{2}$ vertices. We argue that in every execution, each honest participant's output graph must be partially ordered, and that any two participants' graphs must obey the same partial ordering. Assume that in some execution there is no such a partial ordering of vertices of honest participants' output graphs. Then it may be the case that for two honest participants p and q active at t and t', it is possible that that $G_p^{*(t)} \nsubseteq G_q^{*(t')}$ and that $G_q^{*(t')} \nsubseteq G_p^{*(t)}$. But this is a contradiction with the fact that Π satisfies graph consistency. However, it may be the case that some vertices in the participants' output graphs cannot be ordered relative to each other, (i.e. there are vertices u,v such that $u \not\prec v$ and $v \not\prec u$) so there may not be an output graph containing exactly $\frac{N^*}{2}$ vertices. Therefore, honest participants identify the smallest graph containing at least $\frac{N^*}{2}$ vertices by the partial ordering of their outputs. $\square$

5.6.2 Permissionless One-Bit Consensus Protocol Π^{bit}

We now show how to achieve one-bit consensus by slightly modifying Π^G. Our protocol Π^{bit} differs slightly from the generic transformation provided in Section 5.6.1 for simplicity of presentation and proof.

We modify the graph consensus protocol as follows. Whenever a participant generates a vertex, it binds an additional one-bit label, which is simply the participant's input bit, to the vertex along with the vertex's edges. The participants run Π^G without producing output until their local graphs reach depth $k^* + \ell^*$, where ℓ^* is the same as in Π^G and k^* is an additional constant derived from the protocol parameters. For any participant p active at time t for which $\mathsf{D}(G_p^{(t)}) \geq k^* + \ell^*$, the participant outputs $\mathsf{extract}(G_p^{(t)})|_{k^*}$ from the graph consensus subprotocol. As its one-bit consensus output, p computes the one-bit label that is bound to a majority of extracted vertices. Even after a participant produces its output bit, it must continue to participate in the underlying execution of Π^G indefinitely; we explain why in a remark below.

Figure 5.2 describes protocol Π^{bit} for one-bit consensus. Π^{bit} is parameterized by α, ε, and ρ, which describe the ratio of honest resources and the maximum rate of resource allocation.

Remark 5.2 (Indefinite Execution)**.** *Note that although honest participants may produce their outputs when their local graphs reach a fixed depth, it is important that honest participants continue to run the underlying graph consensus protocol indefinitely, until the execution ends. The reason is straightforward: if ever honest participants stop running the underlying graph protocol, then corrupt participants can, with enough time, run an execution on their own which builds a deeper graph, with the property that the labels bound to vertices in the second graph would induce a decision of the opposite bit. This could cause disagreement with any honest participant that "wakes up" long after honest participants stop building the original DAG, and is presented with the two competing graphs.*

Theorem 5.5. *For all ρ and all ε, and for all $\alpha > \rho(1-\alpha)((3-\alpha)\rho + \frac{\varepsilon}{\alpha} + \frac{\varepsilon}{\rho} + \varepsilon + 1)$ every every (α, ε)-honest, ρ-rate-limited admissible execution of $\Pi^{\mathsf{bit}}(\alpha, \varepsilon, \rho)$ satisfies termination, consistency, and validity.*

Protocol 10 DAG Protocol for One-Bit Consensus $\Pi^{\mathsf{bit}}(\alpha, \varepsilon, \rho)$

Parameters $\alpha, \varepsilon, \rho$

Derived Constants

1. $\beta = 1 - \alpha$
2. $\gamma = (1+\beta)\rho + \varepsilon + \frac{\varepsilon}{\rho} + 1$
3. $c = \gamma + \rho + \frac{\varepsilon}{\alpha}$
4. $x = c\varepsilon + c + \rho + \frac{\varepsilon}{\rho} + 1$
5. $\omega = \frac{\beta\rho}{\alpha}(x + \gamma + \frac{\varepsilon}{\rho} + 1) + \varepsilon$
6. $k^* = \frac{\omega + 2\varepsilon}{\alpha - \beta}$

Input

1. Each participant has a 1-bit input b

Internal Variable

1. $G_p = (V_p, E_p)$ is a participant's local state. Initially, $G_p = (\{\mathsf{root}\}, \emptyset)$

Protocol

1. **Framework** Run Protocol Π^G
2. **Labeling Vertices** Whenever a participant is allocated a resource, it additionally binds a one-bit label to the vertex it generates, where the label is the participant's input b
3. **Output** If $\mathsf{D}(G_p) > k^* + \ell^*$, output the majority bit in the labels of all vertices in $\mathsf{extract}(G_p)|_{k^*}$. Ties are broken by outputting 1.

Figure 5.2: Protocol for one-bit consensus using graph consensus

Proof Overview The proof of Theorem 5.5 inherits heavily from the proof of Theorem 5.3. In fact, termination and agreement follow directly from the liveness and graph consistency of Π^G.

– **Consistency:** By Proposition 5.1, all honest participants output *exactly the same graph*. Therefore, to achieve one-bit consistency, the one-bit consensus output can be any fixed function of the labels that the participants output from the underlying graph protocol.

– **Termination:** By Lemma C.1, honest participants' graphs grow as long as honest vertices are perpetually added. Therefore, if enough resources are allocated to honest participants, then honest participants' graphs grow to sufficient depth for them to output a bit, and Π^{bit} terminates.

To prove Theorem 5.5, only nontriviality remains. The intuition for the proof of nontriviality follows. We leverage the (assumed) property that honest participants have a long-term advantage in generating vertices over the corrupt participants, and run the graph consensus protocol until the graph is deep enough to guarantee that there must be substantially more honest vertices in $\mathbb{G}$ than corrupt vertices. We also use the property that each participant extracts *all* of the honest vertices in its view to guarantee that the long-term advantage in generating honest vertices translates to the fact that a majority of vertices output from each honest participant's local graph are honest. Validity follows from outputting the bit that comprises the majority of one-bit labels embedded in the extracted vertices. If all honest participants have the same input b, then b is guaranteed to be the label on a majority of the extracted vertices.

The only tricky part of the proof is due to the fact that honest participants stop adding vertices below depth k^* once their local graphs become deeper than k^*, but the corrupt participants may continue to add vertices at depth k^* even after the honest participants stop adding vertices at that depth. This gives the corrupt participants extra time to add vertices with depth k^*.

We use the following technique to overcome this difficulty. Intuitively, at some time t^*, $\mathsf{D}(\mathbb{G}^{(t^*)}) - k^*$ will be so large that no vertex added at any $t > t^*$ with depth k^* will ever be extracted by any honest participant. Therefore, the extra time for corrupt participants to add extra vertices with depth k^* is limited to the range of time between t_{k^*}, defined as the moment when $\mathbb{G}$ reaches k^* depth, and t^*. Therefore, in order to ensure that the majority of vertices extracted by honest participants up to depth k^* are honest, it suffices to bound the number of corrupt vertices that can be generated in the window of time between t_{k^*} and t^*.

The proof proceeds as follows. Fist, we show that there is a distance x such that if some (corrupt) vertex v is generated at t_v and $\mathsf{D}(G_{\mathcal{H}}^{(t_v)}) - \mathsf{D}(v) > x$ then v can never be extracted from any honest participant's graph. Second, we upperbound how many corrupt vertices may be generated in any execution between the time that $\mathbb{G}$ reaches an arbitrary depth k and $G_{\mathcal{H}}$ reaches depth $k + x$, and let this number be ω. Finally, we use the honest participants' known long-term advantage to set k^* to guarantee that from the beginning of the execution until the moment when $\mathsf{D}(\mathbb{G}) = k^*$, the difference between the number of honest vertices that have been generated and the number of corrupt vertices that have been generated exceeds ω. This guarantees that when an honest participant eventually computes its output, a majority of the vertices up to depth k^* in its extracted subgraph must be honest.

The full proof of Theorem 5.5 is in Appendix D.

Chapter 6

Asynchronous Secure Group Messaging

This chapter presents a novel protocol for asynchronous group messaging. Our group messaging (GM) (Section 6.4) protocol consists of three building blocks: (1) an initial group key agreement (GKA) protocol (Section 6.2), (2) a group randomness messaging (GRM) protocol used to transport key updates (Section 6.3), and (3) a key lattice (Section 6.1). The key lattice is the main contribution of the work on which this chapter is based [47].

Adversarial Model

This chapter considers the strongest adversary of any in this thesis. In the security game, the adversary orchestrates a simulated execution by invoking oracles that emulate protocol actions taken by all parties. The adversary additionally may delay and rearrange the order of messages arbitrarily, as in an asynchronous network. However, to simplify the presentation, proper ordering of messages *within a subprotocol* is enforced by sequence numbers on our updates and encrypted messages, and therefore in the exposition we assume that each subprotocol's messages are ordered, but messages sent by different subprotocols (such as GKA, GRM, and GM application messages) are not ordered with respect to each other.

In addition, rather than corrupting parties explicitly in order to send messages on their behalves, the adversary is permitted to call its oracles on messages that have not been sent by honest parties. However, because all messages in our constructions are authenticated, successfully changing the state of an oracle without knowledge of a party's underlying key would break the security of a cryptographic authentication mechanism.

The adversary can corrupt parties to learn protocol keys, and in some cases may inject messages based on those keys. For example, learning a group key allows the adversary to inject application messages, but these injections do not affect the security of other keys.[1]

Insider Security The state of the art in group messaging includes achieving a limited amount of "insider security" [10, 12]. The adversary can "take over" a party by first learning its GRM key (via a different query than leaks the group keys) and then and evolving the group key on the party's behalf. This kind of corruption is generally referred to as an insider attack, as the party has become impersonated, and it is not considered recoverable *in any known scheme* if the same party ever issues a competing key update. However, if the adversary only uses the discovered state to send a message early which would have been sent later by the party as in [10], then the attack is naturally covered by our security framework "for free," as this attack is equivalent in our game to calling an oracle's evolution function early and delaying delivery of the message to other parties.

The techniques of [12] for insider security require incorporating another protocol key into the key schedule, which might not be revealed alongside a party's other local state, as well as the simulator's ability to learn RO calls. The former is beyond the scope of the key lattice

[1]Some authentication schemes require parties to sign messages with their long-term keys [57] but adapting this to concurrent group messaging is non-trivial, and not the focus of this work.

but could be included in a comprehensive system, and it is unclear that the latter is possible in the standard model.[2]

A Concurrent and Fast-Healing Construction

The key lattice framework extends the state of the art in distributed group messaging by achieving very strong concurrency – both for parties making concurrent updates within a protocol session, and for concurrent protocol sessions. Our construction also allows us to achieve a very fast healing mechanism measured in rounds and messages, although the work required by each updating party is $O(n)$, which is higher than the goal of $O(\log n)$ by the MLS working group. We briefly highlight concurrency and our healing mechanism.

Full Concurrency by Eliminating Propose-and-Commit: Because we don't require the propose-and-commit framework to complete an update, we reduce the round complexity of every group update operation. Propose-and-commit requires that parties first propose a group addition or key update and then another party commits. If simultaneous commits occur, then in lieu of an explicit consensus protocol, some infrastructure must determine the winner. Even DCGKA [127], the decentralized work closest to ours by eliminating the central server, requires that a dominating commitment is made in order to heal after compromises, but in the event of concurrent commitments there is no solution. Additionally, if multiple updating or committing parties encrypt group messages with respect to their own commitments but before receiving the competing concurrent update (or commit), their message is not guaranteed to be decryptable by any party that receives the competing commit before the encrypted message.

[2]When [10] provide a construction without their RO, they also achieve only static security.

Partnering and Concurrent Sessions: In comparison to other recent work on group messaging [7, 8, 9, 26, 127], our construction achieves security of concurrent *sessions* by considering *partnering*. Partnering [37, 38, 82] (also called matching) states that parties participating in concurrent sessions of group key agreement commonly distinguish the separate sessions. It may be the case that the infrastructure server in the recent work also assigns unique session identifiers to distinguish sessions, as alluded in the PhD thesis of [45]. Other works explicitly model that only one CreateGroup instruction may be called [10].

Fast Healing: Our key lattice and modular framework achieves a fast and intuitive healing mechanism. In fact, even if *all* parties are corrupted simultaneously – meaning the adversary learns all of the keys in all parties' local states – this means all of the keys in its local lattice and all updates in its local state are learned by the adversary. as soon as *any* successive, uncompromised key update is generated, the resulting group key is not compromised. Specifically, if any party is compromised, it must first heal its local GRM execution by calling its evolution function once (this refreshes the state of the channel as well as updates the group key). The next update that it receives from an uncompromised party yields an uncompromised key (including if the recovered party performs the second evolution itself.) This means that healing requires 2 GRM messages.

6.1 The Key Lattice

The key lattice is our central idea for managing concurrent key updates. In our key lattice framework, every group key in a group messaging protocol is associated with a coordinate in a discrete n-dimensional space, where n is the number of players in the group. When parties update the group key (at some index), the new key produced is mapped to a larger index. For example, for $n = 2$, a key $\mathsf{k}_{1,0}$ at coordinate $(1, 0)$ may be updated to a new key with an associated coordinate $\mathsf{k}_{1,1}$. We also provide a graphical explanation of a key lattice in which the indices in the discrete n-dimensional space are vertices, and each vertex is labeled with a key. In the graph, edges between vertices represent key updates.

Because the key lattice tracks the set of group keys generated during a group messaging execution, we additionally define security of group messaging with respect to the key lattice. We now formally define a key lattice.

Definition 6.1 (Key Lattice)**.** *We define $\mathbb{K}$ to be the space of keys, and we define $\mathbb{L}$ to be the lattice of $\mathbb{N}^n$ where the ordering is defined by $\mathbf{i}_a \leq \mathbf{i}_b$ if all elements in $\mathbf{i}_a$ are less or equal to $\mathbf{i}_b$, and $\mathbf{i} \in \mathbb{N}^n$ denotes a point on the lattice. A* key lattice *$L = \{(\mathbf{i}, \mathsf{k}_\mathbf{i})\}_{\mathbf{i} \in \mathbb{L}}$ where $\mathsf{k}_\mathbf{i} \in \mathbb{K} \cup \{\perp\}$ is a discrete lattice for which every point $\mathbf{i} \in \mathbb{L}$ is associated with either a single key or $\perp$.*

We denote the association by letting $\mathsf{k}_\mathbf{i}$ be the key associated with $\mathbf{i}$. We also say that the key for an index $\mathbf{i}$ is *defined* if $\mathsf{k}_\mathbf{i} \neq \perp$. Intuitively, parties will compute and agree on many pairs $(\mathbf{i}, \mathsf{k}_\mathbf{i})$.

Given a key lattice, a key $\mathsf{k}_\mathbf{i}$ is j-maximal if there is no $\mathbf{j} \in \mathbb{N}^n$ for which $\mathbf{j}^{(j)} > \mathbf{i}^{(j)}$ and $\mathsf{k}_\mathbf{j} \neq \perp$. If a key is j maximal for all $j \in [n]$, we say the key is maximal in the lattice. Looking ahead, in each party's local lattice there is always a maximal key, computed by all applying all updates that the party knows.

6.1.1 Key Evolution

When a party evolves the group key, it adds a new key (or, as in our construction in Section 6.1.4, a group of keys), to the key lattice. Key evolution is described by a function $\mathsf{KeyRoll} : \mathbb{K} \times \mathcal{X} \rightarrow \mathbb{K}$, where $\mathbb{K}$ is the key space and $\mathcal{X}$ is the *update space,* which encodes the data applied to the key during evolution. In our construction, we will require a few properties of the $\mathsf{KeyRoll}$ function. First, we require that $\mathsf{KeyRoll}$ is commutative, i.e. $\mathsf{KeyRoll}(\mathsf{KeyRoll}(\mathsf{k}, x), x') = \mathsf{KeyRoll}(\mathsf{KeyRoll}(\mathsf{k}, x'), x)$ for all $\mathsf{k} \in \mathbb{K}$ and $x, x' \in \mathcal{X}$.

In addition to commutativity, we require that $\mathsf{KeyRoll} : \mathbb{K} \times \mathcal{X} \to \mathbb{K}$ is *unpredictable* in its second input. Intuitively, knowing only the first input (a key from $\mathbb{K}$), no adversary can "predict" the output (another key from $\mathbb{K}$), if the second input (an update from $\mathcal{X}$) is sampled at random. Similarly, we say that $\mathsf{KeyRoll}$'s inverse is unpredictable if given only $\mathsf{k}' \leftarrow \mathsf{KeyRoll}(\mathsf{k}, x)$, no adversary can "guess" the input k. More formally, we have the following.

Definition 6.2 (Unpredictability)**.** *A family of functions $\mathcal{F} = \{F_\lambda\}_\lambda$ where $F_\lambda \colon \mathbb{K}_\lambda \times \mathcal{X}_\lambda \to \mathbb{K}_\lambda$ is* unpredictable *in its second input if there exists a negligible function* negl *such that for every probabilistic polynomial time adversary $\mathcal{A}$ and every λ:*

$$\Pr[y = F_\lambda(k, x) \colon k \leftarrow \mathbb{K}_\lambda, x \leftarrow \mathcal{X}_\lambda, y \leftarrow \mathcal{A}(1^\lambda, k)] \leq \mathsf{negl}(\lambda)$$

$\mathcal{F}$'s inverse is unpredictable if there exists a negligible function negl *such that for any polynomial time adversary $\mathcal{A}$ and every λ:*

$$\Pr[k' = k \colon k \leftarrow \mathbb{K}_\lambda, x \leftarrow \mathcal{X}_\lambda, k' \leftarrow \mathcal{A}(1^\lambda, F_\lambda(k, x))] \leq \mathsf{negl}(\lambda)$$

where in each experiment, k and x are sampled at random from their respective domains.

There are many families of unpredictable functions. For instance, $\mathsf{KeyRoll}(k, x) = k \oplus x$ satisfies the unpredictability definition, as well as $\mathsf{KeyRoll}(k, x) = \mathsf{PRF}_x(k)$[3]. In both cases, it is not possible to predict the output without knowing the key. The difference between the first construction and the second is that in the first case, knowing the first input and the output completely leaks the update material x. This property is not critical to our construction; we can prove security for our main protocol assuming only that $\mathsf{KeyRoll}$ is

[3]In practice we cannot use the PRF construction because it is not commutative.

unpredictable. However, for completeness (and for situations where unpredictability is not enough), one can define one-wayness similarly to the traditional version.

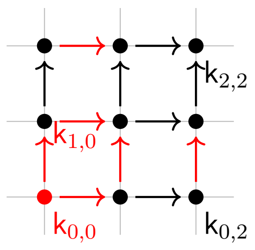
One-wayness. Intuitively, a function is one-way on a challenge (first or second) input if, given $F(k, x)$ and the other input, it is hard for any adversary to compute the challenge input. Below we provide definitions of one-wayness on the second input. Although we do not use it in our construction, it is also possible to define one-way-ness in the first input analogously to one-way-ness in the second input. Intuitively, given x and $F(k, x)$, it should be hard to compute k. If KeyRoll is one-way in the first input, then the construction inherits additional useful properties, which we describe in Section 6.4.4. We now present our definitions for one-wayness on the second input.

Definition 6.3 (One-Wayness (on the Second Input)). *A family of functions $\mathcal{F} = \{F_\lambda\}_\lambda$ where $F_\lambda \colon \mathbb{K}_\lambda \times \mathcal{X}_\lambda \to \mathbb{K}_\lambda$ is one-way on its second input if there exists a negligible function* negl *such that for every probabilistic polynomial-time adversary $\mathcal{A}$ and every λ*

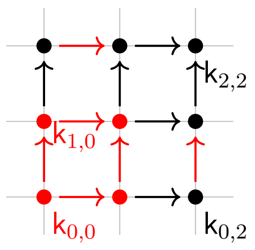
$$\Pr[x' = x \colon k \leftarrow \mathbb{K}_\lambda, x \leftarrow \mathcal{X}_\lambda, x' \leftarrow \mathcal{A}(1^\lambda, k, F_\lambda(k, x))] \leq \mathsf{negl}(\lambda).$$

where k and x are sampled randomly from their respective domains.

ℓ-Point One-Wayness. The definition above can be generalized to the setting where $\mathcal{A}$ obtains polynomially many (in the security parameter) samples of $(k, F_\lambda(k, x))$ pairs for different randomly sampled k but the same x. This additional property allows us to further constrain the power of the adversary. We defer the definition and discussion to Section 6.4.4.



(a) The red vertices and edges are explicitly revealed to the adversary.



(b) The full set of information that an adversary can compute from 6.1a.

Figure 6.1: Illustration of explicitly and implicitly revealed information in a Key Lattice.

6.1.2 The Key Graph

In our construction, parties track the group key(s) by assigning each key to a point on the lattice. When a party evolves the group key, it defines the transition from one point on the lattice to another. In fact, our construction defines the transitions from a family of points to another family of points. Therefore, it is useful to describe the key lattice as a graph, where the vertices are labeled with keys, and the edges encode key evolutions. Specifically, we define a key graph $\mathcal{G}$, where each lattice point $\mathbf{i} \in \mathbb{N}^n$ is a vertex, and each vertex is labeled with a single key or with $\perp$. In our discussion, we refer to vertices by the lattice points they represent. There exists an edge from vertex $\mathbf{i}$ to $\mathbf{j}$ if $\mathbf{j} = \mathsf{increment}(\mathbf{i}, k)$ for some $k \in [n]$, and we say that a pair of vertices $\mathbf{i}$ and $\mathbf{j}$ are *neighbors* if there is an edge from $\mathbf{i}$ to $\mathbf{j}$. Edges in a key graph are labeled with the key evolutions that they represent. We say there exists a *path* ρ of length ℓ between two vertices $\mathbf{i}$ and $\mathbf{i}'$ if there exists a sequence of edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{\ell-1}, v_\ell)$ such that (a) $v_1 = \mathbf{i}$, (b) $v_\ell = \mathbf{i}'$, and (c) v_{j-1} and v_j are neighbors for all $j \in [2, \ell]$. Cycles are not allowed in a path.

FS & PCS: Our key graph allows us to discuss FS & PCS in a unified and simple manner, as directional variants of the same abstraction. We color a vertex or edge black if it is not revealed to the adversary, and we color a vertex or edge red if it is revealed to the adversary. A party that "knows" both the key corresponding to a vertex and an edge leaving that vertex will also "know" the vertex's neighbor. FS & PCS mean that the *only* way the adversary can learn a key k^* at some target vertex v^* is by starting with a red vertex on the graph and following a path of red edges to v^*. In the traditional definition of FS, this would mean that given a vertex v, without following (in reverse) a path of red edges, the adversary cannot learn a predecessor of v. In the traditional definition of PCS, this would mean that given a vertex v, without following a path of red edges, the adversary cannot learn a successor of v.

We illustrate this paradigm in Figure 6.1; every key is mapped to a point on the graph, and updates are mapped to edges in the graph. In Figure 6.1a, the red vertices and edges are explicitly revealed to the adversary. If PCS holds, then the adversary cannot compute the key $\mathsf{k}_{2,2}$ because there is no path of red edges from a red vertex to $\mathsf{k}_{2,2}$. In Figure 6.1b, the adversary can compute the keys $\mathsf{k}_{0,1}$, and $\mathsf{k}_{0,1}$, and $\mathsf{k}_{1,1}$ by starting at $\mathsf{k}_{0,0}$ and following a path of red edges. FS can analogously be visualized by traversing the directed graph "backwards" in order to reveal vertices at smaller indices than a compromised vertex.

Computable Lattice:

The description of a key lattice L may not be "complete" in the sense that given a set $L = \{(\mathbf{i}, \mathsf{k})\}$ representing a key lattice, it may be possible to infer the keys assigned to other indices on the lattice (i.e., points not in L). Below we illustrate the possible inferences depend on the choice of the $\mathsf{KeyRoll}$ function. Consider the case where $\mathsf{KeyRoll}$ is defined using XOR, then knowing the key at $\mathbf{i}$ and a neighboring key at $\mathbf{i}' = \mathsf{increment}(\mathbf{i}, d)$ allows us to derive the update σ, which may allow us to derive the keys at other lattice points $\mathbf{j}$ such that $\mathbf{j}^{(d)} = \mathbf{i}^{(d)}$.

We introduce a function $\mathsf{Computable}(L, E) \rightarrow L'$ to output all the computable lattice points L' given the original lattice L and a set of updates $E = \{(d, j, x)\}$, where $d \in [n]$ is the dimension, j is an index and x is the argument to $\mathsf{KeyRoll}$.

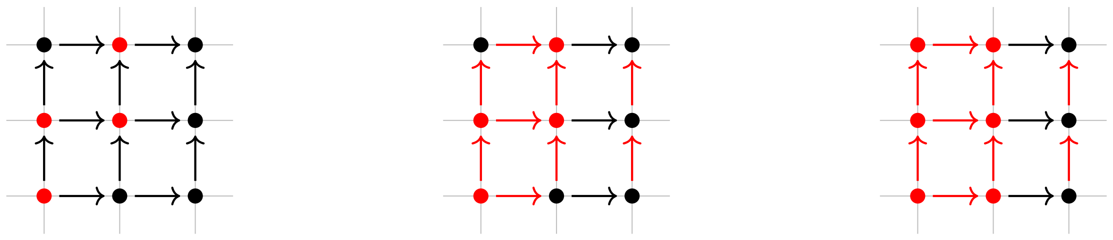


Figure 6.2: Illustration of computable information from a key lattice where KeyRoll is not one-way.

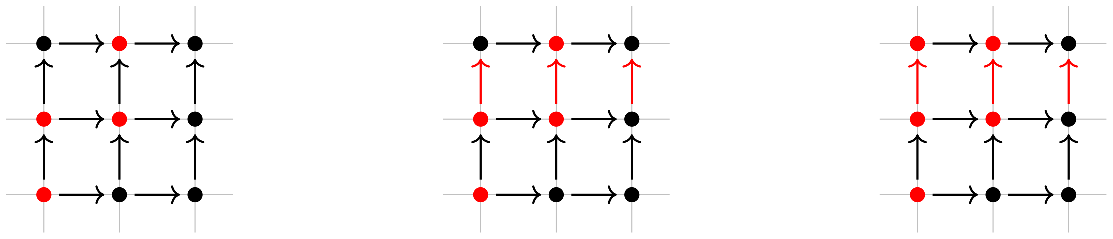


Figure 6.3: Illustration of computable information from a key lattice where KeyRoll is one-way.

The examples in Figures 6.2 and 6.3 illustrate the dependence of Computable on the properties of KeyRoll. Figure 6.2 illustrates how Computable works if a KeyRoll function is not one-way. Suppose the red keys in the figure on the left are revealed in a key lattice. If the KeyRoll function is unpredictable but not one-way, then knowledge of a pair of adjacent keys would reveal all edges (updates) in the corresponding row or column, as shown in the middle figure. These inferred edges lead to additional computable keys (colored in red) in the right figure.

Figure 6.3 illustrates the difference when KeyRoll is one-way. The figure begins with the same lattice as in Figure 6.2 but assumes KeyRoll is one-way. The lattice points in the left figure do not allow us to compute a new lattice with more keys. However, given additional information on the edges in the middle figure, it is possible to compute one additional lattice point (top left in the right figure).

The function Computable(L, E) can be realized as follows:

1. Interpret the lattice L as a directed graph $\mathcal{G}$. Initially this graph has no edges, only vertices from L.

2. Add every edge from E to the graph. Recall that every edge in E corresponds to

multiple edges in $\mathcal{G}$. Specifically, $e = (d, j, x)$ describes all edges that begin with a vertex $(\ldots, j, \ldots)$ and end with a vertex $(\ldots, j+1, \ldots)$ where j and $j+1$ are on the dth position, and each edge is labeled with the update x.

3. Traverse $\mathcal{G}$ from the origin. For every pair of neighboring vertices (u, v) where $u \neq \perp$ and $v = \perp$, if there exists an edge labeled with x connecting u to v, then compute $k_v \leftarrow \mathsf{KeyRoll}(k_u, x)$.

4. Similar to above, but traverse $\mathcal{G}$ backwards and if two neighboring vertices (u, v) where $k_u = \perp$ and $k_v \neq \perp$ then compute $k_u \leftarrow \mathsf{KeyRoll}^{-1}(k_v, x)$, where x is the label on the edge between u and v. Note that, if KeyRoll is one-way on its first input, then this step is omitted, as it is hard to compute u given x and v.

6.1.3 Updating a Key Lattice

We next explain how parties add and remove keys, and the maintenance they perform in order to prevent too much state expansion.

Adding Keys: Parties may update the key lattice using the function $\mathsf{Update}(L, e) \rightarrow L'$ which takes a key lattice L and an update $e = (d, j, x)$ and a returns a new key lattice L' as follows:

– Let $D = \{\mathbf{i}_m\}$ be all d-maximal index vectors in L.

– Output a new lattice L' with additional points defined by the tuple $(\mathsf{increment}(\mathbf{i}), \mathsf{KeyRoll}(\mathsf{k}_{\mathbf{i}}, x))$ for all $\mathbf{i} \in D$.

Note that since the lattice points included in D are d-maximal, all keys in $\mathsf{increment}(\mathbf{i}, d+1)$ are $\perp$ in the original lattice L. One can think of this operation as (possibly) adding keys to

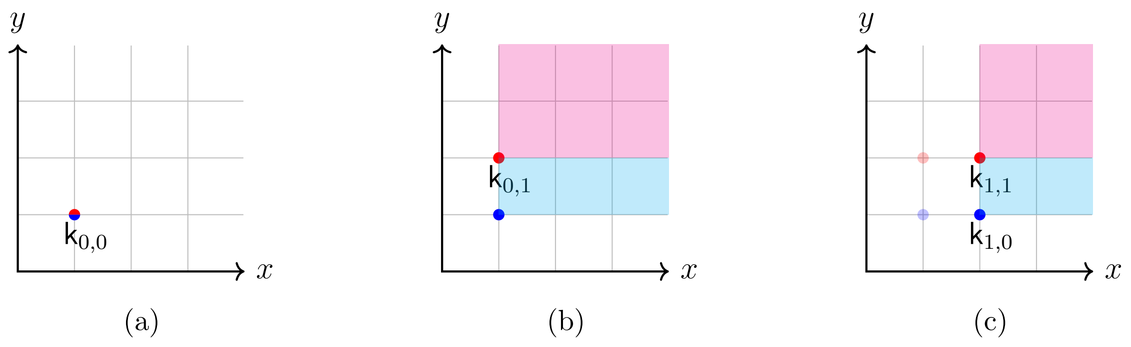


Figure 6.4: An example of a local key lattice in an execution with two players (blue and red) from the perspective of the red party.

the lattice based on e.

Forgetting Keys: A key lattice is an infinite object. In order to manage memory requirements, (and looking ahead, to provide FS) we need a way to remove keys from a party's local version of the key lattice. We next explain how parties track the keys of other parties and introduce the function Forget to formally describe how keys are removed.

Parties will maintain a local key lattice in order to track the group keys, but they do not (necessarily) need to maintain a full view of the key lattice. Each party tracks only the keys that it may need in the future in order to decrypt a message that it has not yet received. This permits the construction to achieve the best possible FS while also achieving correctness; as soon as some party knows it no longer needs the key, it deletes the key from its view (in order to prevent an adversary from learning the key after it has become deprecated).

The function $\mathsf{Forget}(L, \mathbf{i}) \rightarrow L'$ takes a key lattice L and an index vector $\mathbf{i}$, and returns a new lattice L' such that all keys all keys in index vectors $\mathbf{i}'$ such that $\mathbf{i}' < \mathbf{i}$, are set to $\perp$.

We illustrate our approach in Figure 6.4. For simplicity, we only consider two parties labelled with the colors red and blue. The shaded regions, assigned by color, indicate the set of points towards which the corresponding party may define a new group key in the future. Any point in a totally unshaded region represents an index of a key that can be deleted. In our construction, when any party updates the key, it moves the latest group key towards a point in the n-dimensional space along an axis that has been assigned uniquely to it. Blue and red update the key towards higher indices on the x axis and y axis, respectively.

1. In Figure 6.4a, the red and blue parties initialize their local key lattices with $\mathsf{k}_{0,0}$.
2. In Figure 6.4b, red evolves the group key, which moves red's latest key to $\mathsf{k}_{0,1}$.
3. In Figure 6.4c, suppose red received an update message from blue. Red applies the update and evolves its own index from $\mathsf{k}_{0,1}$ to $\mathsf{k}_{1,1}$. Because red knows that blue evolved its key, red updates its view of blue's index $\mathsf{k}_{0,0}$ to $\mathsf{k}_{1,0}$. Specifically, red's perspective of the latest key for blue becomes $\mathsf{k}_{1,0}$. Since $\mathsf{k}_{0,0}$ and $\mathsf{k}_{0,1}$ are outside the shaded region, these keys are removed.

Windowing to Limit State Expansion: In addition to the state reduction described above, we can apply a state "window" to prevent the state from blowing up when encrypted messages are delayed over the network, at the expense of the ability to decrypt long-delayed messages. Consider that if one party makes m updates to the shared group key, resulting in m possible different group keys, then parties must keep $O(m)$ state in case another party sends a message using one of those m keys. In our windowing scheme, each party maintains *at most* the latest w key evolutions from every other party, which provides the ability to compute at most w^n total keys on the key lattice at any time.

We write $\mathsf{Forget}(L, w) \rightarrow L'$ to provide a window parameter w to Forget, which works as follows.

– For every dimension $d \in [n]$, let i_d the maximum j such that there is a key defined in L at index j in dimension d.

– Let $\mathbf{i}_w$ be an index vector such that for every $d \in [n]$, $\mathbf{i}_w^{(d)} = \max(0, i_d - w)$.

– Execute $\mathsf{Forget}(L, \mathbf{i}_w)$ and return the new lattice L'.

When using this scheme, there are situations in which parties may send messages such that some application messages are not decryptable. Suppose sender S sends an application message m encrypted under key k, and then suppose S updates the group key w times starting with k. If S's message m is delayed until after receiver R receives S's key updates, then R will delete the key material describing how to decrypt m. In synchronous networks, the window can be set such that parties update their keys once per epoch, and the window can be set large enough (by setting w is equal to the number of epochs that measure the network delay) for sent messages to always be received in time to be decrypted. In the general asynchronous case, the window can be set to ∞ in order to always guarantee decryption, but this approach loses FS.[4] Thus, the w parameter allows us to trade between security and correctness.

6.1.4 Instantiation

We now describe how our group messaging protocol, which is presented in Section 6.4.2, allows parties to manipulate a key lattice.

Generating a Set of Key Evolutions. In our construction, each party updates the group key in its own "direction" in L; the dth party ($U \in \mathcal{P}$ for which $\phi(U) = d$) always updates the group key towards larger indices in the dth dimension on the lattice. A key update $\sigma \in \Sigma$ sent by one party to another is therefore a tuple (d, j, x), where d is a dimension in the key lattice (generated by the party U such that $\phi(U) = d$), $j \in \mathbb{N}$ is an index that annotates how many times the updating party has updated the group key, and $x \in \mathcal{X}$ is data that

[4]This tradeoff was similarly explored by [110]; our asynchronous security model specifically accounts for the attacks they describe by withholding some ciphertexts and corrupting a party days later to recover the messages.

describes how to update the key (for $\mathsf{KeyRoll}$). In other words, $\Sigma = [n] \times \mathbb{N} \times \mathcal{X}$. The jth key evolution generated by any party therefore defines the transition from *every* index $\mathbf{i}$ to index $\mathbf{i}'$ such that $\mathbf{i}^{(d)} = j$ and $\mathbf{i}' = \mathsf{increment}(\mathbf{i}, d)$, and it defines the evolution to use update data x. In our construction, the space $\mathcal{X}$ is the same as described in Definitions 6.2 and 6.3.

Observe that each key update in our construction defines a group of key evolutions, which can be described in our graphical representation as a group of edges. We require commutativity of $\mathsf{KeyRoll}$ to guarantee that when transitioning from key k to key k' (over one or more edges), where k is represented by vertex u, k' is represented by vertex v, and there are multiple paths between u and v in some party's key lattice, it does not matter which path is taken.

Our KeyRoll Function. Our construction depends on the discrete logarithm assumption to instantiate $\mathsf{KeyRoll}(\mathsf{k}, x)$ as k^x. That is to say, let key space $\mathbb{K}$ be a prime-order group G in which the discrete log problem is hard, and let update space $\mathcal{X}$ be $\mathbb{Z}_{|\mathsf{G}|-1}$. This construction easily satisfies our commutativity requirement since $(\mathsf{k}^x)^{x'} = (\mathsf{k}^{x'})^x$. For appropriately chosen parameters, the construction is trivially unpredictable. If the discrete logarithm problem is hard in G, then $\mathsf{KeyRoll}$ is also one-way on its second input.

6.1.5 Key Lattice for Generic Key Management

The key lattice enables building a concurrent group messaging protocol from existing primitives such as pairwise channels. The following generic approach uses a key lattice to build concurrent group messaging with three building blocks: (1) an initial group key, (2) a secure pairwise channel between all parties in a group and (3) an AEAD scheme for sending payload messages.

- Given the initial group key k_0, the parties initialize their key lattice with $(\mathbf{0}, \mathsf{k}_0)$, and assign $\perp$ to the key at every other lattice point.

- If a party at index $d \in [n]$ wants to update the key for the jth time, it samples $x \xleftarrow{\$} \mathcal{X}$ and sends (d, j, x) using the secure pairwise channels.

- Upon receiving (d, j, x) the receiver adds key $\mathsf{k}' \leftarrow \mathsf{KeyRoll}(\mathsf{k}, x)$ to the lattice at point $\mathbf{i}'$ where k is the maximal key at point $\mathbf{i}$ in the lattice and $\mathbf{i}' \leftarrow \mathsf{increment}(\mathbf{i}, d)$.

- If a party at index $d \in [n]$ wants to send an application message, it encrypts the message using the maximal key k in its local key lattice and sends the ciphertext to the group members (without using the secure pairwise channels). The ciphertext is encrypted using AEAD where the associated data is the lattice index that corresponds to the key that was used to encrypt the message.

- Upon receiving the ciphertext encrypting a payload message, the receiver checks whether it has the key in the key lattice required to decrypt. If so, then the receiver decrypts it immediately. Otherwise, the receiver buffers the message until it receives sufficient information to decrypt.

- Of course, storing all the keys that are in the key lattice is expensive and trades off forward security. Every party also runs $\mathsf{Forget}(L, w)$ for its lattice L and the window parameter w every time the party processes an update message.

6.2 Group Key Agreement

To agree on the very first shared key we use an existing group key agreement (GKA) protocol. Many GKA protocols exist in the literature [32, 36, 37, 111]; for our purposes adapt the one from [37] as it captures strong-forward secrecy and a strong corruption model. In this section, we reproduce the definition with syntactic tweaks for the context of a group.[5]

[5]GKA protocols were originally formulated for 2-party messaging.

Definition 6.4 (Group Key Agreement)**.** *We use $G \subseteq \mathcal{P}$ to denote some group of players that participate in the protocol. Each party $U \in \mathcal{P}$ is assumed to already have a long term public/private key pair $(\mathtt{pk}_U, \mathtt{sk}_U)$. We assume a PKI exists and the public keys are available to all parties.*

The protocol consist of two stateful algorithms.

- $\{m_V\}_{V \in G} \leftarrow \mathsf{GKA.Init}(G)$*: Initialize an instance of the GKA protocol for a group represented by G and return a set of responses, one for every party in G.*
- $\{m_V\}_{V \in G} \leftarrow \mathsf{GKA.Recv}(M)$*: Process message M and return a set of responses.*

The GKA may output done *with a key k to notify to the party that the protocol is completed.*

6.2.1 Security of GKA

Defining security requires additional terminology. We use $\Pi^{\mathsf{gka}}_{U,i}$ to denote an oracle which models the i-th instance of party U engaging in the group key agreement protocol. Below, we will also use the notation (U, i) to refer to this oracle instance. We write $n = |\mathcal{P}|$ to define the total number of participants, and we assume that each participant U can engage in at at most n_S sessions, i.e. $i \in [1, \ldots, n_S]$.

An oracle $\Pi^{\mathsf{gka}}_{U,i}$ maintains a number of variables $(\delta_{U,i}, \kappa_{U,i}, \mathsf{gid}_{U,i}, \mathsf{sid}_{U,i}, \mathsf{k}_{U,i})$.

- The value $\delta_{U,i}$ denotes the current state of the oracle, which can be one of the following
  - pending: this is the initial state of each oracle. It signals that the oracle has not yet determined a key.
  - accept: this state indicates that the oracle has determined a key.

– abort: this indicates that for some reason the oracle has aborted.

– The value $\kappa_{U,i} \in \{\perp, \mathsf{corrupted}\}$ indicates the corruption state of the oracle. It is initially set to be $\perp$.

– The value $\mathsf{gid}_{U,i} \subseteq \mathcal{P}$ denotes the intended group with which the oracle intends to engage in a group discussion. For convenience we assume $U \in \mathsf{gid}_{U,i}$.

– The value $\mathsf{sid}_{U,i}$ is a session identifier. Note that the index i in $\Pi^{\mathsf{gka}}_{U,i}$ is not the same as $\mathsf{sid}_{U,i}$. The value i acts as an internal session identifier. $\mathsf{sid}_{U,i}$ is a global session identifier which the protocol needs to establish. Once established, all group members share the same sid.

– Finally $\mathsf{k}_{U,i}$ is a key for the group $\mathsf{gid}_{U,i}$, which is initially set to $\perp$.

Several functions can be called on the oracles $\Pi^{\mathsf{gka}}_{U,i}$, which allow us to model the protocol and respond to messages. The adversary has complete control of the network and so can decide what messages to send to parties, and when.

– $\Pi^{\mathsf{gka}}_{U,i}.\mathsf{Init}(G)$: Initialize an instance of the GKA protocol for the group members in G where $U \in G$. Sets $\mathsf{gid}_{U,i} \leftarrow G$ and return a set of messages $\{M_V\}_{V \in G}$, where M_V is a message intended to be passed to an oracle associated with party V.

– $\Pi^{\mathsf{gka}}_{U,i}.\mathsf{Recv}(M)$:

  – If $\delta_{U,i} = \mathsf{abort}$ then this call does nothing.

  – Otherwise, the oracle $\Pi^{\mathsf{gka}}_{U,i}$ responds with a set of messages $\{M_V\}_{V \in G}$, where M_V is a message intended to be passed to an oracle associated with party V.

  If $\delta_{U,i}$ is changed to accept by this call, then $\Pi^{\mathsf{gka}}_{U,i}$ outputs done.

– $\Pi^{\mathsf{gka}}_{U,i}.\mathsf{Corrupt}()$: This sets $\kappa_{U,i} = \mathsf{corrupted}$ for all i associated with identity U. This command will return $\mathtt{sk}_U$. Since the long-term keys are associated with the party U across all instances, calling $\Pi^{\mathsf{gka}}_{U,i}.\mathsf{Corrupt}$ is the same as $\Pi^{\mathsf{gka}}_{U,j}.\mathsf{Corrupt}$ where $i \neq j$.

– $\Pi^{\mathsf{gka}}_{U,i}.\mathsf{Reveal}()$: If $\delta_{U,i} \neq \mathsf{accept}$, do nothing. Otherwise, return the shared group key $\mathsf{k}_{U,i}$.

– $\Pi^{\mathsf{gka}}_{U,i}.\mathsf{StateReveal}()$: Return the internal state $\mathsf{state}_{U,i}$.

– $\Pi^{\mathsf{gka}}_{U,i}.\mathsf{Test}()$: If $\delta_{U,i} \neq \mathsf{accept}$, abort the protocol. Otherwise, sample either output the shared group key from the GKA protocol, or a random key, depending on the challenger's choice.

Using these definitions, we can give the security definition of a GKA protocol. As is usual the key definitions to define security for key agreement are *partnering* and *freshness*.

For our partnering definition we slightly deviate from the formalism of [37], in that we define partnering to be defined only for the whole group; whereas [37] does this in a pairwise manner. By transitivity of the pairwise partnering relation the two are essentially equivalent.

Definition 6.5 (Partnering of GKA). *Given a group $G \subseteq \mathcal{P}$ and a set of pairs $Q = (U, i_U)_{U \in G}$ (there is one pair per group member) defining associated oracles $\Pi^{\mathsf{gka}}_{U,i_U}$, we say the oracles corresponding to $Q' \subseteq Q$ are* partnered *if the following conditions hold:*

1. *For all $(U, i_U) \in Q'$ we have $\delta^{i_U}_U = \mathsf{accept}$.*

2. *For all $(U, i_U) \in Q'$ we have $\mathsf{gid}_{U,i_U} = G$.*

3. *There is a single value $\mathsf{sid}_{Q'}$ such that for all $(U, i_U) \in Q'$ we have $\mathsf{sid}^{i_U}_U = \mathsf{sid}_{Q'}$.*

4. *No oracles, apart from $\Pi^{\mathsf{gm}}_{U,i_U}$ for $(U, i_U) \in Q'$, accept with session identifier $\mathsf{sid}_{Q'}$.*

The freshness definition below describes the state where an oracle is unaffected by the adversary. It is a form of "adversary restriction" which stops the adversary from winning the game using trivial attacks, e.g., revealing the shared key and then immediately making a test query. Freshness also helps us define forward secrecy implicitly as we will see in the security definition in Definition 6.7. This freshness definition is different to one in Section 6.4 because GKA does not have the concept of a key lattice so we use the traditional freshness definition.

Definition 6.6 (Freshness of GKA)**.** *An oracle* $\Pi_{U,i}^{\mathsf{gka}}$ *is considered fresh if*

- *No* $(U, i) \in \mathsf{gid}_{U,i}$ *is asked for a* Corrupt *query prior to a* $\Pi_{V,j}^{\mathsf{gka}}.\mathsf{Recv}(M)$ *such that* $(V, j) \in \mathsf{gid}_{U,i}$ *before the partners of* $\Pi_{U,i}^{\mathsf{gka}}$ *are in the accept state.*
- *Neither* (U, i) *or its partners are asked for a* StateReveal *query before they are in the accept state.*
- *Neither* (U, i) *or its partners are asked for a* Reveal *query after having accepted.*

Definition 6.7 (Security of GKA)**.** *Security of Group Key Agreement is defined by the following sequence of steps:*

1. *All queries can be executed without restriction.*
2. *The adversary selects a* fresh *target* (U, i) *and calls* $\Pi_{U,i}^{\mathsf{gka}}.\mathsf{Test}()$*. The challenger samples a bit* $b \xleftarrow{\$} \{0, 1\}$ *and outputs the real shared group key* k *or a random key* r *sampled uniformly at random.*
3. *Continue interacting with the GKA oracles.*
4. *The adversary outputs a bit* b' *and terminates.*

Any time that a session is accepted, the sid *and the* gid *are passed to the adversary. The advantage of the adversary* $\mathcal{A}$ *in this game is*

$$\mathsf{Adv}^{\mathsf{gka}}_{\mathcal{A}} = 2 \cdot |\Pr[b = b'] - 1/2|.$$

The correctness definition is described below. It is similar to the definition in [37] except we modify it to use our group-based partnering definition.

Definition 6.8 (Correctness of GKA)**.** *A key agreement protocol is said to be* correct *if for a group* $G \subseteq \mathcal{P}$ *and a set of pairs* $Q = (U, i_U)_{U \in G}$ *(with one pair per group member) giving associated oracles* $\Pi^{\mathsf{gka}}_{U,i_U}$*, then the oracles being* partnered *implies that each oracle has the same shared group key* k_{U,i_U}*, i.e. for all* $(U, i_U), (V, i_V) \in Q$ *we have* $k_{U,i_U} = k_{V,i_V}$*.*

6.3 Group Randomness Messaging

We present the group randomness messaging (GRM) abstraction through which the parties communicate update messages. The main functionality is to broadcast some authenticated data and a ciphertext that encrypts a random message, which is used to update the key lattice, to all members in the group using a pairwise channel. We require the point-to-point channels to have FS & PCS properties. Below we give the definition of GRM and an instantiation using public-key AEAD (PKAEAD) (Definition A.9).

Definition 6.9 (Group Randomness Messaging (GRM))**.** *Consider the player executing the protocol is* U*, a GRM scheme consists of three stateful algorithms.*

– $\{c_{U,V}\}_{V \in G} \leftarrow \mathsf{GRM}_U.\mathsf{Init}(\mathsf{k}, w, G)$*: initialize the GRM instance using the initial key* k*, the window size* w*, and the group members* G*.*

This step initializes the internal state $\mathsf{state}_{U,i}$. *The output is a set of ciphertexts, one for every player in* G.

- $\{c_{U,V}\}_{V\in G} \leftarrow \mathsf{GRM}_U.\mathsf{Evolve}()$: *output a ciphertext* $c_{U,V}$ *for every* $V \in G$.

- $\sigma_{V,U} \leftarrow \mathsf{GRM}_U.\mathsf{Recv}(c_{V,U})$: *process the ciphertext* $c_{V,U}$, *update the internal state and return the plaintext* $\sigma_{V,U}$ *if the decryption is successful. If decryption is unsuccessful, return* $\perp$.

In the above definition, $\sigma_{V,U}$ is a triple (U, j, x) where U is the identity of the sender, j is a positive integer and $x \in \mathcal{X}$.

6.3.1 Security

Security for GRM is defined in Definition 6.10. Compared to the syntax in Definition 6.9, we add StateReveal and Test queries. Furthermore, Init, Evolve and Recv are modified as follows: Init does not take a key k because these would allow the adversary to trivially win the security game described later. Additionally, Init takes a set of oracles Q instead of a set of players G. This change is required because the challenger needs to make sure the adversary initializes the oracles that correspond to the same session, using the same key k. For the same reason, the Evolve oracle does not output the plaintext anymore. Finally, Recv takes an additional flag dec_flag which allows the adversary to see the plaintext messages of the updates it uses to evolve the oracle's states. In other words, Recv can be used as a decryption oracle.

Intuitively, our security definition aims to captures FS and PCS. Namely, the adversary is allowed to reveal the state either before or after the test query. Nevertheless, as long as the group member under attack had a chance to recover from the corruption or deleted its old state, the adversary should learn nothing about the plaintexts that the group member sends or sent. We assume out-of-order messages, including repetition, do not happen in the point-to-point channels. In practice, this kind of attack can be detected using sequence numbers or hash chains.

Definition 6.10 (GRM Security). *The security of a GRM scheme is defined by a game between the adversary and a challenger. A mapping* QtK *between a group of oracles* Q *and a key* k *is kept so that the challenger uses the same key for oracles in the same* Q *during initialization. This mapping is not revealed to the adversary. The adversary has access to oracles* $\Pi^{\mathsf{grm}}_{U,i}$*, each of which maintains internal state* $\mathsf{state}_{U,i}$ *and can be invoked as follows:*

– $\{c_{U,V}\}_{(V,\cdot)\in Q} \leftarrow \Pi^{\mathsf{grm}}_{U,i}.\mathsf{Init}(w, Q)$*: initializes GRM for the window size* w *and oracles* Q*. If* $\mathsf{QtK}[Q] =\perp$*, sample a symmetric key* k *and set* $\mathsf{QtK}[Q] \leftarrow k$*. Finally, return* $\mathsf{GRM.Init}(\mathsf{QtK}[Q], w, G)$.

– $\{c_{U,V}\}_{V\in G} \leftarrow \Pi^{\mathsf{grm}}_{U,i}.\mathsf{Evolve}()$*: return* $\mathsf{GRM.Evolve}()$.

– $\sigma \leftarrow \Pi^{\mathsf{grm}}_{U,i}.\mathsf{Recv}(c, \mathsf{dec\_flag})$*: process the ciphertext* c *using* $\mathsf{GRM.Recv}(c)$*. If* $\mathsf{dec\_flag} = 1$*, output the plaintext message* σ*, otherwise output* $\sigma =\perp$.

– $\mathsf{state}_{U,i} \leftarrow \Pi^{\mathsf{grm}}_{U,i}.\mathsf{StateReveal}()$*: returns* $\mathsf{state}_{U,i}$ *to the caller.*

– $(x_0, x_1) \leftarrow \Pi^{\mathsf{grm}}_{U,i}.\mathsf{Test}(c^*)$*: described in the game below.*

The security game is divided into phases, separated by the adversary's $\mathsf{Test}()$ *query, as follows:*

1. *All queries can be executed without restriction.*

2. *The adversary calls the test query.* $\Pi^{\mathsf{grm}}_{U,i}.\mathsf{Test}(c^*)$*. The challenger samples a bit* $b \xleftarrow{\$} \{0,1\}$*, samples a value* $x_0 \xleftarrow{\$} \mathcal{X}$ *and then computes* $(V, j, x_1) \leftarrow \mathsf{GRM.Recv}(c^*)$*. The adversary is given the output* $(V, j, x_b), (V, j, x_{1-b})$.

3. *All queries can be executed without restriction.*

4. *At any time the adversary can stop making queries and output a bit b' and win the game if $b' = b$.*

The game above is additionally constrained by the following restrictions, which prevent trivial attacks:

– *The adversary is not allowed to call* Recv *with* $\mathsf{dec\_flag} = 1$ *on* c^**, the ciphertext given to the test oracle.*

– *Let* c^* *be the test ciphertext. There may be no other* c' *such that* c^* *and* c' *were both output from the same call to* Evolve*, for which* c' *has already been an input to any oracle query* $\Pi^{\mathsf{grm}}_{V,i}.\mathsf{Recv}(c', \mathsf{dec\_flag} = 1)$*.*

– *Further, consider the call to* $\Pi^{\mathsf{grm}}_{V,i}.\mathsf{Test}(c^*)$*, where* c^* *is taken from a call to* $\Pi^{\mathsf{grm}}_{U,i}.\mathsf{Evolve}$*. We do not allow* $\Pi^{\mathsf{grm}}_{V,i}.\mathsf{StateReveal}$ *to be called until* $w + 1$ *calls have been made to* $\Pi^{\mathsf{grm}}_{V,i}.\mathsf{Evolve}$ *from the time that* $\Pi^{\mathsf{grm}}_{V,i}.\mathsf{Test}(c^*)$ *is called. This condition ensures that oracle* $\Pi^{\mathsf{grm}}_{V,i}$ *has refreshed its state.*

The advantage of the adversary $\mathcal{A}$ *in this game is*

$$\mathsf{Adv}^{\mathsf{grm}}_{\mathcal{A}} = 2 \cdot |\Pr[b = b'] - 1/2|.$$

The scheme is secure if, for any probabilistic polynomial-time adversary, the advantage is negligible in the security parameter.

6.3.2 Correctness

The correctness definition for GRM requires correct decryption of all key evolutions under two conditions which already appeared in the security definition (Definition 6.10).

1. Messages in the point-to-point channels are not reordered, i.e., these channels are modelled as FIFO queues.
2. Each point-to-point channel can buffer at most w messages, this is similar to the final restriction that prevents trivial attacks from Definition 6.10.

The constraints above can be viewed as a ω-well-ordered execution from **??** when $\omega = 0$.

Definition 6.11 (GRM Correctness)**.** *A GRM protocol is* correct *if in every infinite execution by every PPT adversary $\mathcal{A}$ who must deliver all messages, for all $U \in G$, for all $\{c_{U,V}\}_{V \in G} \leftarrow \mathsf{GRM}_U.\mathsf{Evolve}()$, there exists a σ and for all $V \in G$ there exists an oracle call $\sigma_{V,U} \leftarrow \mathsf{GRM}_V.\mathsf{Recv}(c_{U,V})$ such that $\sigma_{V,U} = \sigma$ and $\sigma \neq \perp$.*

Correctness (Definition 6.11) of the protocol described in Section 6.3.3 holds by construction. That is, the secret keys used for decryption are guaranteed to be available as long as there are no more than w messages in the FIFO queue.

6.3.3 Instantiation

We instantiate GRM using PKAEAD. In essence, every party keeps a queue of w public and secret key-pairs. This queue is updated every time the party calls Evolve by dropping the oldest keypair and adding a new one. Each party U also maintains a public key for every other party V which is updated whenever U receives the output of V's Evolve. U uses this public key in order to encrypt messages to V. U also maintains an integer j_V that tracks the index of the latest public key U has received from V.

This initial message sent by each party is a pair $(\mathtt{pk}_U^0, m)$, where $\mathtt{pk}_U^0$ is the party's initial ephemeral public key, m is a MAC on the public key using the key k provided as input to Init. Where k is the key output by a GKA execution, this effectively "ties" a GRM to the GM application that uses it, as the MAC links the output k of a GKA session with the GRM session that will be used to evolve the key.

On a high level, the protocol achieves PCS because public keys are cycled over time and FS because old keys are dropped. Our construction is detailed below. Let the set $\mathcal{X}$ to be domain from which updates are randomly sampled.

- $\mathsf{GRM}_U.\mathsf{Init}(\mathsf{k}, w, G)$: Generate an ephemeral key pair $(\mathtt{pk}_U^0, \mathtt{sk}_U^0)$. Initialize $\mathsf{state}_U.\mathtt{sk}s = \{\mathtt{sk}_U^0\}$ and $\mathsf{state}_U.\mathtt{pk}s = \emptyset$, and save w as the window parameter. Compute $m \leftarrow \mathsf{MAC}(\mathtt{pk}_U^0; \mathsf{k})$, where k is the input key, $\mathtt{pk}_U^0$ is the message and MAC is a cryptographic MAC scheme. Send the same message $(\mathtt{pk}_U^0, m)$ to every member in G.
- $\mathsf{GRM}_U.\mathsf{Evolve}()$:
  1. A new private key $\mathtt{sk}_U^{j+1}$ is generated, along with its public key $\mathtt{pk}_U^{j+1}$.
  2. Sample $x \xleftarrow{\$} \mathcal{X}$ and let $\sigma \leftarrow (U, j+1, x)$, where j is the index of the latest secret key in $\mathsf{state}_U.\mathtt{sk}s$.
  3. Repeat the steps below for every $V \in G$ (including U).
     - If the public key of the receiver V is not known, abort.
     - Call $(c, t) \leftarrow \mathsf{PKAEAD.Enc}(\mathtt{pk}_U^{j+1} \| \sigma, j_V; \mathtt{pk}_V^{j_V})$ and then set $c_{U,V} \leftarrow (c, t, j_V)$. Note that $\mathtt{pk}_V^{j_V}$ can be found in $\mathsf{state}_U.\mathtt{pk}s$ and j_V is the index of the public key associated with V.
  4. state_U is updated as follows.

    - Add $\mathtt{sk}_U^{j+1}$ to $\mathsf{state}_{U,i}.\mathtt{sk}s$
    - If $|\mathsf{state}_U.\mathtt{sk}s| > w$, remove the oldest one (i.e., $\mathtt{sk}_U^{j-w}$).

- $\mathsf{GRM}_U.\mathsf{Recv}(c_{V,U})$: There are two possible message formats. The message output by Init is an ephemeral public key $\mathtt{pk}_V^0$ with a Mac; if the message is this type, then verify the Mac using the key k provided to Init[6] and then set V's public key in $\mathsf{state}_U.\mathtt{pk}s$ to be $(0, \mathtt{pk}_V^0)$. All other messages are handled as follows.

    1. Parse the message $c_{V,U}$ as (c, t, j), where j is an index into the current user U's secret key.
    2. Find secret key $\mathtt{sk}_U^j$. Abort the protocol if it does not exist.
    3. $\mathtt{pk}_V^{j_V} \| \sigma_{V,U} \leftarrow \mathsf{PKAEAD.Dec}(c, t, j; \mathtt{sk}_U^j)$, abort if this step returns $\perp$.
    4. Add or update V's public key in $\mathsf{state}_U.\mathtt{pk}s$ to be $(j, \mathtt{pk}_V^{j_V})$.
    5. Let j_{min} be the smallest j in $\{(j, \mathtt{pk}_V^{\mathtt{i}_V}) : V \in G\}$.
    6. Delete all secret keys $\mathtt{sk}_U^j$ where $j < j_{\mathsf{min}}$.
    7. Return $\sigma_{V,U}$

Theorem 6.1. *Let $\mathcal{A}$ be an adversary against the GRM game, let $\mathcal{B}$ be an adversary against the PKAEAD game, and let $\mathcal{C}$ be an adversary against the MAC EUF-CMA game. Then*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{grm}} \leq n_S \cdot \mathsf{Adv}_{\mathcal{C}}^{\mathsf{mac}} + 2 \cdot |Q|_{\max} \cdot n_Q \cdot \mathsf{Adv}_{\mathcal{B}}^{\mathsf{pkaead}}.$$

where $|Q|_{\max}$ is the upperbound for the number of oracles in a group, n_Q is the upperbound of the number of queries to the encryption oracle that $\mathcal{B}$ makes on behalf of $\mathcal{A}$ for the instance under test, and $n_S = \mathsf{poly}(\lambda)$ is the maximum number of concurrent GRM sessions that $\mathcal{A}$ is allowed to invoke in its security game.

The proof of this therorem is in Appendix E.1.

[6]If verification fails due to trying the wrong key from multiple concurrent sessions, return $\perp$ and process the incoming message via the Recv function of a different session.

6.4 Group Messaging

In this section, we provide a GM protocol (Definition 3.13) built on GKA, GRM, and a key lattice, and prove its security. In our construction, parties who wish to participate in a GM instance begin by running a GKA protocol to obtain a shared symmetric key k. They use k to initialize their key lattice, and then use GRM to securely communicate update messages that can be applied to the key lattice to evolve the shared group key. When a party encrypts an application (payload) message, it always uses the latest key in its key lattice.

6.4.1 Group Messaging Security

The security of GM is modeled via an oracle game, directed by the adversary, in which it activates oracles corresponding to parties running a polynomial number of protocol executions. In the game, we explicitly use the key lattice to track the evolution of the group key(s) over time. The adversary invokes a semantic security challenge against a "fresh" key on one of the lattices. A key is "fresh" precisely if the adversary cannot derive that key from its view of the execution thus far; graphically, this means that the key is black in the corresponding graph akin to Figure 6.1b. The adversary wins the semantic security challenge if it can distinguish two ciphertexts encrypted under a fresh key.

The adversary invokes oracles $\Pi_{U,i}^{\mathsf{gm}}$ where U is a group member and $i \in [1, \ldots, n_S]$, where the subscript i denotes a specific instance of the oracle that belongs to party U. Different instances that belong to the same party may share long-term keys, e.g., identity keys.

Each oracle $\Pi_{U,i}^{\mathsf{gm}}$ maintains internal variables to track each party's view of the key lattice and the group messages that have been received by that party. They also collectively maintain global state that tracks which elements of the key lattice and which key updates have been explicitly revealed to the adversary. We denote by $L_{\mathsf{sid}}^{\mathsf{rev}}$ the key lattice describing all keys (points on the lattice) which are revealed to the adversary, and we denote by $E^{\mathsf{rev}}_{\mathsf{sid}}$ the set of key updates, modeled as edges in the graphical interpretation of the key lattice, which are revealed to the adversary. $S^{\mathsf{rev}}_{\mathsf{sid}} = (L^{\mathsf{rev}}_{\mathsf{sid}}, E^{\mathsf{rev}}_{\mathsf{sid}})$ denotes all of the key material that is revealed to the adversary in some session sid. The session ID sid is a unique identifier for the group members who have successfully completed the initial group key agreement and established a session (described in detail in Section 6.2 since it is a property inherited from GKA). Indeed, sid is not defined when a GKA session begins, but this is not an issue since the session's lattice is instantiated only after the session is established. The full information on the key lattice available to the adversary is given by $\mathsf{Computable}(L^{\mathsf{rev}}_{\mathsf{sid}}, E^{\mathsf{rev}}_{\mathsf{sid}})$. We remark that the session ID (sid) is not the same as the instance ID. The instance of an oracle, e.g., (U, i), is established when the oracles are initialized, but the session ID is only established some time later, after the oracles are ready to evolve keys.

Specifically, the oracles maintain the following state:

- $\delta_{U,i} \in \{\mathsf{pending}, \mathsf{accept}, \mathsf{abort}\}$ indicates whether the oracle is ready to start evolving keys.
- $L_{U,i}$ represents the key lattice maintained by oracle $\Pi^{\mathsf{gm}}_{U,i}$. We use the language from Section 6.1 to describe the key lattice.
- $\mathsf{state}_{U,i}$ is the remaining state that the implementation may keep.
- $S^{\mathsf{rev}}_{\mathsf{sid}} = (L^{\mathsf{rev}}_{\mathsf{sid}}, E^{\mathsf{rev}}_{\mathsf{sid}})$ represents the key lattice $L^{\mathsf{rev}}_{\mathsf{sid}}$ containing all the revealed keys by the adversary as well as the revealed updates $E^{\mathsf{rev}}_{\mathsf{sid}}$ in session sid.

The full details of the GM oracles are specified below.

- $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Init}(G, w)$: Initialize an instance of the GM protocol for the group members in G where $U \in G$ and w is the window size. Set $\delta_{U,i} = \mathsf{pending}$. The response is returned to the adversary.

- $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Corrupt}()$: Return the long-term secret to the adversary.
- $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Reveal}()$: If $\delta_{U,i} \neq \mathsf{accept}$ then return $\perp$. Otherwise, return the set of keys that are computable from $L_{U,i}$, and add these keys to $L^{\mathsf{rev}}_{\mathsf{sid}}$
- $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{StateReveal}()$: If $\delta_{U,i} \neq \mathsf{accept}$ then return $\perp$. Else, return the internal state $\mathsf{state}_{U,i}$ except the computable keys $L_{U,i}$. Add any revealed key updates to $E^{\mathsf{rev}}_{\mathsf{sid}}$.
- $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Evolve}()$: If $\delta_{U,i} = \mathsf{abort}$ then return $\perp$. Else, return a set of message $\{M_V\}_{V\in G}$.
- $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Recv}(M)$:
  - If $\delta_{U,i} = \mathsf{abort}$ then this call does nothing.
  - Otherwise process the message, optionally update the state $\mathsf{state}_{U,i}$ and the key lattice $L_{U,i}$. Return a set of messages $\{M_V\}_{V\in G}$. The input M should be from either the output of Recv or Evolve.
- $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Dec}(c)$: Use the available internal state to decrypt the ciphertext c and output the plaintext. If the oracle does not have enough information to decrypt the message, then it is buffered.
- $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Enc}(m)$: Encrypts the plaintext m using the maximal key in $L_{U,i}$ and returns a ciphertext.
- $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Test}(m_0, m_1)$: This is defined in the security game below.

By execution of $\mathsf{Corrupt}$, Reveal and $\mathsf{StateReveal}$ queries the adversary can learn the entire secret internal state of the oracle $\Pi^{\mathsf{gm}}_{U,i}$. Specifically, $\mathsf{Corrupt}$ gives the party's long-term public key and secret key, Reveal gives the party's current group keys, and $\mathsf{StateReveal}$ gives the party's internal state except for what is provided by the former two queries. Also note how the above gives the adversary a decryption oracle via Dec.

Modeling Pairwise Channels in the Oracle Game:

In the general oracle game, the adversary is permitted to invoke the oracles in any order, which models an asynchronous network. However, to describe the guarantees that the protocol achieves when windowing, we define a syntactic model to describe the messages sent "between parties" in the oracle game. Specifically, between every ordered pair of parties (U, V) the adversary maintains a special buffer $\mathcal{C}_{U,V}$ called a *channel* representing the pairwise connection between U and V. When an oracle query returns a message c to be sent from U to V, the adversary places (c, n) into $\mathcal{C}_{U,V}$, where n is an integer recording that c is the nth message placed into the channel.

In the above game description, each oracle provides three queries to generate messages to other parties. $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Enc}(m)$ encrypts a message using the oracle's latest key and returns a ciphertext which is forwarded to all other parties. Whenever a $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Enc}(m)$ query is made, the returned message c is simultaneously put into the channels $\mathcal{C}_{U,V}$ for all $V \in G$. $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Evolve}()$ generates a key evolution, but returns *a different message* for each other party in the execution. Similarly, $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Recv}(M)$ may output a different message for every other party in the execution, but it may also output no messages. Whenever a $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Evolve}()$ or $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Recv}(M)$ query is made, the oracle returns a list of ciphertexts c_V, one for each $V \in G$. Each of these messages is immediately placed into the corresponding channel $\mathcal{C}_{U,V}$ along with its index.

A message c generated by an Enc query is removed from its corresponding buffer only when it is input to a corresponding oracle $\Pi^{\mathsf{gm}}_{V,j}.\mathsf{Dec}(c)$. A message c generated by an Recv or Evolve query is removed from its corresponding buffer only when it is input to a corresponding oracle $\Pi^{\mathsf{gm}}_{V,j}.\mathsf{Recv}(c)$. Note that if an oracle receives a message that it cannot yet process due to reordering of messages over a pairwise channel, then the oracle is expected to buffer the message until it can process the message, and return the result once it can process the message.

The adversary may additionally invoke Recv or Dec oracles on messages that have not been placed in channels but instead were adversarially generated. These actions do not affect the channels.

Partnering:

For group messaging, *partnering* is analogous to the case for GKA. Intuitively, a group in a GKA protocol is partnered if the parties participate in the same session and agreed on the same group key. For group messaging, parties are partnered if they are running a protocol with each other to agree on a lattice of group keys.

Definition 6.12 (Partnering)**.** *Given a group $G \subseteq \mathcal{P}$ and a set of pairs $Q = (U, i_U)_{U \in G}$ defining associated oracles $\Pi^{\mathsf{gm}}_{U,i_U}$, we say the oracles are* partnered *if the underlying GKA oracles $\Pi^{\mathsf{gka}}_{U,i_U}$ (see Definition 6.5) are partnered.*

For some security parameter λ we define a security game for the adversary $\mathcal{A}$, this consists of the set of participants $\mathcal{P}$ where n (the number of participants) is a polynomial function of λ, as is the maximum number of sessions per participant n_S. Thus the number of oracles $\Pi^{\mathsf{gm}}_{U,i}$ is also a polynomial function of λ. The adversary $\mathcal{A}$ is given at the start of the game all the public keys $\mathtt{pk}_U$ for $\mathtt{pk} \in \mathcal{P}$ and it interacts with the oracles $\Pi^{\mathsf{gm}}_{U,i}$ via the sequence of oracle queries as above.

Freshness:

We now define freshness for our game. Intuitively, we say that a key is *fresh* if it has not been revealed to the adversary, either explicitly via Reveal queries, or implicitly, via a combination of Reveal and $\mathsf{StateReveal}$ queries. The global state $S_{\mathsf{sid}}^{\mathsf{rev}}$ tracks the keys computable by the adversary, and a key is fresh if and only if it is not computable from $S_{\mathsf{sid}}^{\mathsf{rev}}$.

Definition 6.13 (Freshness)**.** *In a session* sid*, a key* $\mathsf{k}_{\mathbf{i}^*}$ *with at index* $\mathbf{i}^*$ *is fresh if and only if it is not computable from* $S_{\mathsf{sid}}^{\mathsf{rev}}$ *using the* $\mathsf{Computable}$ *function, as defined in the group messaging definition (Definition 3.13).*

Security Game:

The security game tries to break the semantic security of a message sent between the parties. It runs in two phases, the division between the two phases is given by the point in which the adversary executes a Test query.

– **Phase 1:** All queries can be executed without restriction.

– **Test Query:** $\Pi_{U,i}^{\mathsf{gm}}.\mathsf{Test}(m_0, m_1)$: Given two equal length messages m_0 and m_1, if k_U is fresh, where k_U is the maximal key of instance(U, i), then the challenger selects a bit $b \in \{0, 1\}$ and applies $\Pi_{U,i}^{\mathsf{gm}}.\mathsf{Enc}(m_b)$, returning the output $\mathtt{ct}^*$ to the adversary. We denote the test oracle by $\Pi_{U^*,i^*}^{\mathsf{gm}}$. We call $\mathbf{i}^*$ the test index.

– **Phase 2:** All queries can be executed except for:

1. Any query that would add $\mathsf{k}_{\mathbf{i}^*}$ to the set of keys computable from $S_{\mathsf{sid}}^{\mathsf{rev}}$.
2. If the message $\mathtt{ct}^*$ is at any point processed by $\mathsf{Dec}(\mathtt{ct}^*)$, by the oracles, then the result is not returned to the adversary (however the game still continues).

Queries to Reveal, $\mathsf{StateReveal}$ or $\mathsf{Corrupt}$ during Phase 1 are used for capturing PCS and queries to these oracles during Phase 2 captures FS. At the end of the game, the adversary $\mathcal{A}$ needs to output its guess b', and wins the game if $b = b'$. We define

$$\mathsf{Adv}_{\mathcal{A}}(\lambda) = 2 \cdot |\Pr[b = b'] - 1/2|.$$

Definition 6.14 (Security of Group Messaging)**.** *A GM scheme is secure if for any probabilistic polynomial time adversary $\mathcal{A}$ the advantage $\mathsf{Adv}_{\mathcal{A}}(\lambda)$ is negligible in the security parameter λ.*

Correctness

Intuitively, a GM protocol is correct if every message that is encrypted with the group key is correctly decrypted by every recipient. We write the formal definition with respect to the oracles defined for our security game. Our definition of correctness requires all encrypted messages must eventually be correctly decrypted under a property called "well-ordered execution" which we define as well.

Definition 6.15 (Correctness of Group Messaging)**.** *A GM protocol is* correct *if in every infinite execution by every PPT adversary $\mathcal{A}$ who is allowed to query the GM oracles except* $\mathsf{Corrupt}, \mathsf{StateReveal}, \mathsf{Reveal}$ *and* Test *and must deliver all messages, for all U, i, for all $c \leftarrow \Pi^{\mathsf{gm}}_{U,i}.\mathsf{Enc}(m)$, and for all $V \in G \setminus \{U\}$ there exists a j and an oracle call $m' \leftarrow \Pi^{\mathsf{gm}}_{V,j}.\mathsf{Dec}(c)$ such that (U, i) is partnered with (V, j) and $m' = m$.*

6.4.2 GM from GRM and GKA

We first present our construction of GM from GKA, GRM, and a CCA-secure AEAD scheme; we then prove security of GM based on the underlying primitives.

Protocol Overview

In our construction of a group messaging protocol, parties maintain local versions of a global key lattice in order to track the group key. They then encrypt and decrypt messages using keys from the lattice, and they update the group key by adding new keys on the key lattice.

Our protocol uses the above primitives to initialize their key lattices, encrypt and decrypt messages using the keys in the lattice, send updates to the group key, and remove keys from their lattices. Specifically, each party maintains a local key lattice $\mathcal{L}$, a local set of key updates $\mathcal{E}$, and a buffer B of unprocessed messages, which contains both GRM messages that it cannot yet process and encryptions that it is not yet able to decrypt. Every update $e \in \mathcal{E}$ has the form (d, i, x) where $d \in [n]$ corresponds to the dimension of the party that generates the update, i is an index and x is key transformation data. Parties also maintain a list of index vectors $\mathcal{I} \in (\mathbb{N}^n)^n$ that tracks each party's view of the current key of every other party, which is used to optimistically exclude keys from its state.

Message Headers and the Recv Subprotocol. We make the distinction between *protocol messages* and *application messages*. Protocol messages in GM are either GKA messages (to agree on an initial group key) or GRM messages (to evolve the group key). Application messages are encryptions under some group key.

Our construction uses a single Recv function to process every incoming protocol message, provided in Figure 6.7, which directs the incoming message to the appropriate subprotocol (either GKA or GRM). To help distinguish between GKA protocol messages and GRM protocol messages in the descriptions of the protocols and the proofs, we say that a message is a "GKA message" if it contains a prefix gka, and a message is a "GRM message" if it contains a prefix grm. In an implementation, these headers can be encoded as flags. Where the context is clear, we elide these prefixes from the exposition.

Initialization:

When a group of parties begin a GM protocol, they initialize the execution via $\mathsf{GM.Init}()$, which is described in Figure 6.5. Each party saves the set of other parties in the protocol and the window parameter. They also agree on a hash function H described below, which is a public parameter. The parties then run GKA in order to agree on an initial group key. Note that the key lattice and GRM is *not* initialized yet; they can only be initialized after the GKA outputs the initial key as shown in Figure 6.6.

Sending and Receiving Key Updates:

Our GM construction uses GRM as a transport for generating and communicating random key updates. In Figure 6.6 and Figure 6.7 we specify how parties generate new key updates and process updates form other parties, respectively.

Specifically, when a party wishes to evolve the group key, it invokes $\mathsf{GRM.Evolve}()$ to receive a random key update σ along with an encryptions of the update to send to each other party via pairwise channel. The calling party adds σ to its set of edges $\mathcal{E}$ and computes any possible new points in $\mathcal{L}$. When a party receives a key update, it calls $\mathsf{GRM.Recv}()$ on the update, and if a key update is returned then it adds the update as an edge in $\mathcal{E}$ and computes any possible new keys in $\mathcal{L}$. If it cannot yet decrypt the key update, it buffers the message.

Encrypting and Decrypting a Message:

Whenever a party wishes to encrypt a message m using the group key, it calls $\mathsf{GM.Enc}$ using the maximal key in its key store. Specifically, we require a hash function $\mathsf{H}\colon \mathbb{K} \to \mathsf{K}$, that maps from the keyspace of the key lattice to the keyspace for a CCA-secure AEAD encryption scheme.[7] When a party encrypts a message, it provides the hashed key corresponding to the maximal index $\mathbf{i}$ in its key lattice $\mathcal{L}$ as input to $\mathsf{AEAD.Enc}$, and it includes the index $\mathbf{i}$ as associated data. The encrypting party then forwards the encrypted message to every other

[7]This hash function's purpose is semantic to convert between types. We only require that if the adversary does not know k then it does not know $\mathsf{H}(k)$. We elide discussion of H in the proof.

party.

When a party seeks to decrypt a message, it looks up the corresponding key (the index of which is found in associated data), and supplies the hashed key to $\mathsf{AEAD.Dec}$. When a party receives an encrypted message, it checks whether the index of the key used to encrypt is in $\mathsf{Computable}(\mathcal{L}, \mathcal{E})$. If so, it uses the key at that index to decrypt the message. If not, it adds the message to the buffer B. The implementations of encryption and decryption in given in Figure 6.8 and Figure 6.9.

Pruning the Key Lattice:

Parties continuously attempt to prune elements from their local state, both in order to manage the size of the state they keep, and also because deleting old keys facilitates forward secrecy. When a party knows that it will no longer receive any messages encrypted with keys below a particular key index $\mathbf{i}$, it optimistically prunes all such keys from its lattice via $\mathsf{Forget}(\mathcal{L}, \mathbf{i})$. Additionally, if ever a key index exceeds the key window (keys whose index vector that are less than the threshold index vector $\mathbf{i}_w$) it purges the key (and relevant updates) from $\mathcal{L}$ (and $\mathcal{E}$).

Whenever a party receives an encryption from a party V, it updates its index vector $\mathcal{I}[\phi(V)]$ tracking the keys used by V. Recall that because our construction requires key updates to move toward higher lattice indices, the set of future indices is the union of the n-dimensional hyperplanes $\mathcal{H}^* = \bigcup_{\mathbf{i}_V \in \mathcal{I}} \mathcal{H}_{\geq \mathbf{i}_V}$. Any index *outside* this union represents an obsolete key, and the related keys are deleted via Forget in Figure 6.9.

In summary, keys and edges that fall outside the window parameter are deleted as specified in Figure 6.7. Keys and edges that will not be used in the future are deleted as specified in Figure 6.9. This is possible because parties also send their maximal lattice point along with their message (in Figure 6.8) so that the receiving party can compute the minimum view (lattice point) of all parties and delete keys and edges that are smaller than the minimum view.

$\mathsf{GM.Init}(G, w)$

On execution of $\mathsf{GM.Init}()$, run $\mathsf{GKA.Init}(G)$ and output the result. Note that U holds the long-term key pair $(\mathtt{pk}_U^{lt}, \mathtt{sk}_U^{lt})$.

Figure 6.5: Algorithm for $\mathsf{GM.Init}(G, w)$

$\mathsf{GM.Evolve}()$

U calls $\{c_{U,X}\}_{X \in G} \leftarrow \mathsf{GRM.Evolve}()$, and outputs $c_{U,X}$ to X for $X \in G$.

Figure 6.6: Algorithm for $\mathsf{GM.Evolve}()$

$\mathsf{GM.Recv}(M)$

– If M is a GKA message:
  – Compute $\{m_{U,V}\}_{V \in G} \leftarrow \mathsf{GKA.Recv}(M)$, and outputs $m_{U,V}$ to party V for $V \in G$.
  – If GKA outputs done with a key k:
    * Initialize $\mathcal{L}$ with the point $(\mathbf{0}, \mathsf{k})$.
    * Initialize a GRM execution via $\{c_{U,V}\}_{V \in G} \leftarrow \mathsf{GRM.Init}(\mathsf{k}, w, G)$ and send $c_{U,V}$ to V for $V \in G$.
    * Initialize an empty message buffer $\mathtt{B} \leftarrow \emptyset$.

– If M is a GRM message received from party V:
  1. Compute $\sigma \leftarrow \mathsf{GRM.Recv}(M)$. If $\sigma = \perp$, then add M to $\mathtt{B}$ and return. Otherwise, let $(d, j, x) \leftarrow \sigma$, add (d, j, x) to the set of edges $\mathcal{E}$ and then compute $\mathcal{L} \leftarrow \mathsf{Computable}(\mathcal{L}, \mathcal{E})$.[a]
  2. Delete deprecated keys using $\mathcal{L} \leftarrow \mathsf{Forget}(\mathcal{L}, w)$.
  3. Delete deprecated edges from $\mathcal{E}$ that precede the corresponding index in the threshold index vector (see Section 6.1.3). Specifically, suppose the threshold index vector is $\mathbf{i}_w = (i_1, \ldots, i_{n_S})$ and $E = \{(d_k, j_k, x_k)\}_k$, then remove all edges (d_k, j_k, x_k) where $j_k < i_{d_k}$.
  4. While $\mathtt{B}$ is not empty or $\mathtt{B}$ has not changed from the previous iteration:
     – For every message $M \in \mathtt{B}$, execute $\mathsf{GM.Recv}(M)$

[a]A sanity check would be that $d = \phi(V)$ and j should equal the dth element of the maximal index vector of $\mathcal{L}$.

Figure 6.7: Algorithm for $\mathsf{GM.Recv}(M)$

GM.Enc(M)

Player U finds the $\phi(U)$-maximal lattice point $\mathbf{i}$ in its local lattice $\mathcal{L}$, computes $(\mathtt{ct}, t) \leftarrow \mathsf{AEAD.Enc}(m, U\|\mathbf{i}, ; \mathsf{H}(k_{\mathbf{i}}))$, and then returns $(\mathtt{ct}, U\|\mathbf{i}, t)$.

Figure 6.8: Algorithm for $\mathsf{GM.Enc}(M)$

GM.Dec(M)

Parse M as $(\mathtt{ct}, V\|\mathbf{i}, t)$. If M is not of this form, return $\perp$. Then:

- If $\mathbf{i} < \mathbf{i}_w$, where $\mathbf{i}_w$ is the threshold index vector, or if $\mathbf{i} < \mathcal{I}[\phi(V)]$, then return $\perp$.
- Update $\mathcal{I}[\phi(V)] \leftarrow \mathbf{i}$, compute $\mathbf{i}_{\mathsf{min}}$ as the index vector of the element-wise minimum of all $\mathbf{i} \in \mathcal{I}$, and then execute $\mathcal{L} \leftarrow \mathsf{Forget}(\mathcal{L}, \mathbf{i}_{\mathsf{min}})$.
- Find the key at $\mathbf{i}$ in $\mathcal{L}$ using $\mathsf{Computable}(\mathcal{L}, \mathcal{E})$, if $\mathsf{k}_{\mathbf{i}} = \perp$, then add M to $\mathtt{B}$ and return $\perp$.
- If $\mathsf{k}_{\mathbf{i}} \neq \perp$, compute $m \leftarrow \mathsf{AEAD.Dec}(\mathtt{ct}, V\|\mathbf{i}, t; \mathsf{H}(k_{\mathbf{i}}))$. If $m = \perp$, abort the protocol. Otherwise, return m.

Figure 6.9: Algorithm for $\mathsf{GM.Dec}(M)$

6.4.3 Well-Ordering and Correctness

Recall that our oracle game tracks the order in which messages are returned from oracles to be sent to other parties via our abstraction of pairwise channels, and that the adversary may delay and reorder messages sent via the pairwise channels. A channel is *ω-well-ordered* if the nth message sent over C is removed from the channel before the $(n+\omega)$th message (via delivery to the correct oracle), for all $n \in \mathbb{N}$. An execution is *ω-well-ordered* if all pairwise channels are ω well-ordered.

We claim that when windowing with our protocol, for any ω-well-ordered execution, if the window parameter w is greater than or equal to ω, then the protocol is correct. The proof is trivial by construction of the protocol. When $w < \omega$, windowing may force some decryption keys to be purged before the corresponding message is delivered.

Remark 6.1 (Well Ordering and Network Synchrony)**.** *Well-ordering is a strict relaxation of network synchrony that depends on ordering messages rather than on time. In a synchronous* *network, a delay parameter of Δ implies Δ-well-ordered channels; therefore, setting $w = \Delta$ implies correctness. If the network is asynchronous, then w must be set to ∞ in order to guarantee correctness. However, this sacrifices forward secrecy, as parties may store old group keys indefinitely.*

6.4.4 Security Theorem

We now state our theorem for the security of our construction and provide an overview of the proof. The full proof is in Appendix E.2.

Theorem 6.2 (Security of Group Messaging)**.** *If $\mathcal{A}$ is an adversary against the GM game, then there exist adversaries $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$ such that $\mathsf{Adv}^{\mathsf{gm}}(\mathcal{A}) \leq 2n_S\mathsf{Adv}^{\mathsf{gka}}(\mathcal{B}) + 2n_S n\mathsf{Adv}^{\mathsf{grm}}(\mathcal{C}) + n_S\psi\mathsf{Adv}^{\mathsf{cca}}(\mathcal{D})$, where $n_S = \mathsf{poly}(\lambda)$ is the maximum the number of GM sessions $\mathcal{A}$ may invoke, and $\psi = \mathsf{poly}(\lambda)$ is the maximum number of keys that $\mathcal{A}$ may query in a session.*

Proof Sketch. The proof proceeds via three reductions. In the first, we present an adversary $\mathcal{B}$ for the GKA game. $\mathcal{B}$ attacks the initial key established by GKA for GM, and it uses $\mathcal{A}$ to distinguish between the output of a GKA scheme and a randomly sampled key. Intuitively, $\mathcal{B}$ simulates a GM execution for $\mathcal{A}$, and $\mathcal{B}$ itself generates all of the keys and edges in the lattice for $\mathcal{A}$ after the initial key is produced by GKA. In the test session (which $\mathcal{B}$ must guess), $\mathcal{B}$ targets the initial key by invoking the GRM oracle's $\mathsf{Test}()$ query to be provided either the key output by GKA or a random key. On all other sessions, $\mathcal{B}$ uses its own Reveal query to learn the GKA key. In either case, this GKA key is used as the initial key for the session's key lattice. $\mathcal{B}$ then internally simulates GRM for $\mathcal{A}$ and, because it knows all of the keys, can perform any requested encryption and decryption. The core idea is that $\mathcal{A}$ will attack a specific key k^* in the lattice. However, because $\mathcal{B}$ knows the transformations from the initial key to k^*, $\mathcal{B}$ can also perform the inverse computation to attack the initial GKA key (although $\mathcal{B}$ does not do this explicitly in the reduction).

In the second reduction, we present an adversary $\mathcal{C}$ for the GRM game. Similar to the GKA adversary $\mathcal{B}$, $\mathcal{C}$ simulates an execution of GM and tracks the keys and updates for $\mathcal{A}$ in the key lattice defined by the execution. $\mathcal{C}$ learns every edge except for one, which it uses to call its own $\mathsf{Test}()$ query. $\mathcal{C}$ guesses its test edge such that with some (polynomial) probability, the key that $\mathcal{A}$ tests will depend on the update represented by $\mathcal{C}$'s chosen edge. If $\mathcal{A}$ is able to distinguish between games in which this edge is faithful to the GRM protocol and a random edge, then $\mathcal{C}$ directly inherits the advantage to distinguishing whether the decryption output by its challenger was a faithful decryption of the update or random.

In the final reduction, we present an adversary $\mathcal{D}$ that attacks the CCA-security of an AEAD scheme. $\mathcal{D}$ again simulates GM for $\mathcal{A}$, and in this case $\mathcal{D}$ knows *every key and update except for the test key.* $\mathcal{D}$ therefore answers every encryption and decryption query that $\mathcal{A}$ asks using its knowledge of the key lattice; for queries on the test key, $\mathcal{D}$ forwards $\mathcal{A}$'s requests to its own challenger. The one difficulty of the proof is that $\mathcal{D}$ does not know which key $\mathcal{A}$ will attack in GM, and $\mathcal{A}$ is able to ask for encryptions under that key before it makes its Test query, which is $\mathcal{D}$'s only indicator of the test key. Therefore, $\mathcal{D}$ adaptively guesses which key will be the one tested. Observe that if $\mathcal{A}$ makes n evolutions, 2^n keys are defined. If $n = \mathsf{poly}(\lambda)$, then this is exponential in the security parameter. However, $\mathcal{A}$ can only run in polynomial time (in the security parameter), and therefore can only explore a polynomial number ψ of them. $\mathcal{D}$ adaptively guesses that the *next* key explored by $\mathcal{A}$ will be the test key with probability $\frac{1}{\psi}$. We show that by this strategy, $\mathcal{D}$ chooses the correct key with some probability greater than the inverse of a polynomial in the security parameter.

The theorem requires that $\mathsf{KeyRoll}$ and its inverse are unpredictable, and therefore the adversary cannot learn arbitrary vertices by revealing a single vertex in the lattice. The properties of $\mathsf{KeyRoll}$ do not appear in the reductions because they directly imply the definition of *freshness*, which rules on which Reveal and $\mathsf{StateReveal}$ queries that $\mathcal{A}$ may make in the game to enforce that $\mathcal{A}$ may never explicitly or implicitly reveal the test key. If KeyRoll is not unpredictable, then if the adversary reveals any vertex, the entire lattice is revealed. If KeyRoll is one-way, the the adversary is permitted to learn more information about vertices and edges around the test vertex than if KeyRoll is only unpredictable. In the following discussion, we explain how the properties of KeyRoll impact the information that the adversary learns from its corruption queries.

Discussion

The Key Lattice, Forward Secrecy, and Post-Compromise Secrecy. We frame our analyses of the above reductions in terms of the graph that represents the collective key lattice defined by the execution. Specifically underpinning our analyses, we require that GKA is forward secure and that GRM updates are both forward secure and post-compromise secure. Consider the conceptualization in which vertices and edges on the key lattice are black if they are not revealed to the adversary, and red if they are revealed to the adversary. Also color red any vertex that is discoverable by the adversary by starting at a revealed vertex and following a path of revealed edges. If GRM is both forward secure and post-compromise secure, then these are the only edges that are computable from those that are revealed, and therefore the adversary cannot learn new edges – and as a result, new keys – on the graph.

Our constructions – and our definition of freshness – explicitly rely on this paradigm. If some part of the key lattice L defined by some execution is revealed to the adversary, then the adversary should not be able to learn any more information on the lattice, except for what it can compute by following paths of red edges from red vertices. Our simulators explicitly traverse the key lattice in order to attack their underlying primitives. The GKA adversary $\mathcal{B}$ builds a path from the initial vertex to the key tested by the GM adversary; the distinction between its two environments is whether the GKA key it tests was sampled at random. Intuitively, if the GM adversary can distinguish between environments in its game, then the GKA adversary $\mathcal{B}$ can backtrack through the path of updates it constructed to the initial GKA key, and in doing so win its GKA challenge. The GRM adversary $\mathcal{C}$ similarly simulates an execution of GM, and it chooses an edge on the key lattice to attack that the GM adversary's key depends on. By forward and post-compromise secrecy, this edge should not be discoverable by any adversary given other revealed vertices and edges. The fact that the GM adversary distinguishes in its game implies that, given some information on the other keys and updates in the lattice, the GM adversary can learn information about this edge. $\mathcal{C}$ exploits this information in order to win its game. The CCA adversary $\mathcal{D}$ similarly requires that the GM adversary cannot learn the test key from the other vertices and edges in the graph which it reveals.

Properties of the Evolution Function. Implicit in the above reductions is that the GM adversary $\mathcal{A}$ cannot learn vertices or edges in the key lattice which it does not explicitly reveal. At a high level, this is enforced by the fact that key updates are sampled at random, and that the evolution function maintains that the evolved key is hard to guess without knowing the key update. In depth, to enforce the fact that the adversary cannot learn additional components of the key lattice (which preserve the definition of freshness), we depend on properties of the key evolution function $\mathsf{KeyRoll}$, described in Section 6.3. We now relate the properties of $\mathsf{KeyRoll}$ to the (in)ability of the adversary to learn additional components of the key lattice. In the following discussion, we refer to $\mathsf{KeyRoll}$ simply as F.

If F is *unpredictable* in its second input (Definition 6.2), then given only the key k corresponding to a vertex v, the adversary cannot learn learn the key k' at any successor of v for which the connecting edge is unrevealed. More granularly, given only k, $\mathcal{A}$ cannot learn $F(k, x)$ when x was sampled at random (as the protocol specifies). Similarly, given only $F(k, x)$, where k and x were sampled at random, the adversary cannot "traverse the graph backwards" to learn the preceding key k. This is the only property of $\mathsf{KeyRoll}$ which our proof requires. We next describe the impact of additional desirable properties of KeyRoll on the ability of the adversary to infer points on the key lattice.

If F is *one-way on the second input* (Definition 6.3), then given a pair of vertices (u, v), $\mathcal{A}$ cannot learn the transformation that describes the key evolution between them. Letting k be the key associated with the vertex u and $F(k, x)$ be the key associated with v, $\mathcal{A}$ therefore cannot learn x. Recall that in our construction, each key evolution corresponds to a group of edges in the key graph. Therefore, it may be the case that the adversary learns a number of vertices $(u_1, u_2, \ldots, u_\ell)$ and $(v_1, v_2, \ldots, v_\ell)$ (for $\ell < n$), where all u_i correspond to lattice points with the same index j in dimension d, and all v_i correspond to points with index $j+1$ in dimension d. Then the adversary may attempt to use multiple points in order to learn the transformation, which is guaranteed to be the same between each u_i and v_i. We would like to prevent $\mathcal{A}$ from learning some target key k^* which also has index $j+1$ in dimension d, even if $\mathcal{A}$ already knows the preceding key (the key with index j in dimension d but the same index in all other dimensions, which corresponds to the key before applying transformation x). We therefore call on an ℓ-point version of the one-wayness definition (Definition 6.16), which says that given the keys (k_{u_i}, k_{v_i}) corresponding to predecessor-successor vertices (u_i, v_i) for $i \in [1, \ell]$ and the key $k_{u_{\ell+1}}$ corresponding to the vertex u_{m+1}, the adversary still cannot learn the key $k_{v_{\ell+1}}$ corresponding to vertex $v_{\ell+1}$, even if all u_i follow the same transition (the same key evolution) to v_i.

Definition 6.16 (ℓ-Point One-Wayness (on the Second Input))**.** *$\mathcal{F} = \{F_\lambda\}_\lambda$ is ℓ-point one-way on its second input if there exists a negligible function* negl *such that for every probabilistic polynomial-time adversary $\mathcal{A}$ and every λ*

$$\Pr[x' = x \colon \vec{k} \leftarrow \mathbb{K}_\lambda^\ell, x \leftarrow \mathcal{X}_\lambda, x' \leftarrow \mathcal{A}(1^\lambda, \vec{k}, [F_\lambda(k_1, x), \ldots, F_\lambda(k_\ell, x)])] \leq \mathsf{negl}(\lambda).$$

where x and $\vec{k} = (k_1, \ldots, k_\ell)$ are sampled randomly from their respective domains.

One-Wayness on the First Input Although not used in our construction, it is still illustrative to explain the properties of the revealed lattice if F is *one-way on the first input* (Definition 6.3). Given given the update x on an edge (u, v) and the key corresponding to vertex v, the adversary still cannot learn the key corresponding to vertex u. For multiple points, the discussion is analogous to the previous discussion of ℓ-point one-wayness on the second input.

6.5 Extension to Dynamic Groups

We provide an extension of our framework that permits dynamic group membership "for free," and additionally handles simultaneous adds and removals with no additional effort, thus completely avoiding "splitting" [10] issues in synchronous protocols where multiply parties make competing simultaneous updates. The extension is a feature of the key lattice, and group messaging protocols that are defined with respect to a key lattice such as ours should be able to adapt the protocol to permit dynamic membership with little cost, save application-specific rules. We do not re-prove our theorems for security of our protocol with dynamic groups.

Recall that our general definition of the key lattice is an n-dimensional space, where n is the set of all players. Let $\mathcal{P}$ be the set of all identities. The function $\phi \colon \mathcal{P} \to \mathbb{N}$ assigns a canonical ordering to $\mathcal{P}$. Players who evolve the group key only do so in their respective dimensions corresponding to ϕ. When a new party joins the group, points become defined in a new dimension corresponding to that party. When a party leaves the group, its future group updates become invalid. Parties running the protocol maintain a lossless compression of the lattice by tracking only indices corresponding to participants in the protocol.

Treating dynamic membership in this way averts all of the problems of concurrency incurred by other works – including with respect to insider attacks – since groups including the new members are only defined in the lattice as successor points of the addition operation, and we incur no conflicts by maintaining multiple copies of the lattice that correspond to groups both with and without the new member.

Adding Members For a group G that starts an execution, all of the members of G are initialized at point 0 in their dimension. In other words, there exists a key at the lattice point $\mathbf{0}$ which is the initial group key. All other parties not in G are initialized at point $\perp$ (which is compressed by omitting the dimension from any index vectors). At any point, a member can add a new party to the group with a message $(\mathsf{AddMember}, U, \mathbf{i})$, where U is the identity of the newly added member and $\mathbf{i}$ is the maximal lattice point of the adding party (which includes the new dimension). The adding party also sends $\mathsf{k}_\mathbf{i}$ to party U. In response, all parties that receive the message will begin receiving from and sending messages to U.

When a new member is added to the group, that member is defined to belong to the group only for messages (and key updates) that *succeed* the $\mathsf{AddMember}$ message on the key lattice. (The newly added party is defined to *not be in the group* with respect to messages that are concurrent to the $\mathsf{AddMember}$ message.) When U is added to the group, the adding party must send the party a key with which to initialize its lattice (at $\mathbf{i}$). To prevent V from learning group messages from before it was a member, the adding party must associate the addition with a key update, which it sends along with the $\mathsf{AddMember}$ message to the other parties in the group. If two parties add the same group member concurrently, the new member is defined to be part of the group for all messages that succeed *either* of the corresponding $\mathsf{AddMember}$ messages. Where the two concurrent updates meet in the lattice, the two key updates provided by the adding parties compose naturally by the commutativity of $\mathsf{KeyRoll}$. Note that since we assume PKI, the long-term public key of U can be fetched by all the parties.

Removing Members Any member of a group can remove another member with a message $(\mathsf{RemoveMember}, U, \mathbf{i})$, where U is identity of the member to be deleted, and $\mathbf{i}$ is the lattice point for which U should no longer receive updates. (The deleting party sets $\mathbf{i}$ as its maximal lattice point at the time it sends the RemoveMember message.) A group member is defined to be removed from the group for all messages that *succeed* the RemoveMember message, meaning all lattice points that are greater than $\mathbf{i}$. For messages that are concurrent to RemoveMember in the key lattice, the party is still in the group.

If two parties attempt to concurrently remove the same member from the group (and that party is already in the group), then the party is removed for all messages that succeed either RemoveMember message. If two parties attempt to concurrently add and remove a member from the group who is not yet in the group, then the RemoveMember message must be invalid and only the AddMember is processed. If two parties attempt to concurrently add and remove a member who is already in the group, then the AddMember message must be invalid and only the RemoveMember message is processed.

Chapter 7

Composing Timed Cryptographic Primitives

The work on which this chapter is based [62] observes that the current understanding of time-release cryptography is based on idealized, non-falsifiable models. This chapter initiates a foundational study of time-release cryptography by introducing a falsifiable, formal notion of security with which to characterize the time-release. The foundations of time-lock puzzles must address composability in order to allow puzzles to serve a larger distributed computation setting, specifically multi-party computation in which time-lock puzzles are solved as part of the protocol. Section 7.5 presents a new depth-bounded model of secure multi-party computation that includes time-release cryptography. The composition theorems in Section 7.5.4 reveal degradation in security when composing timed primitives.

The model in Section 7.5 presents new constraints on the usual asynchronous model of Canetti [39] or malicious model of [75]. Specifically, both the adversary and the environment in our model are depth-bounded, and protocols that are run concurrently by the environment both count against the environment's depth. While it allows an adversary attacking a timed primitive to execute an arbitrary (polynomial) number of side-sessions in order to assist its attack, these sessions must be run in parallel (time) to the main protocol it is attacking. In other words, the depth of each side session counts against the adversary's time budget, even if the adversary is not using that time to attack its main protocol. This is in contrast to the referenced models by [39] and [75], which allow the environment to run protocols completely "asynchronously," meaning the environment can pause its target protocol and run a full side-session of another protocol (or polynomially many other protocols) without counting towards the depth used to attack the target protocol.

7.1 Subtleties and Inconsistencies in Random Oracle Analysis for Time-Lock Puzzles

To a large extent, timed cryptography still treats cryptographic puzzles as essentially a random oracle [13, 21, 22][1]. At a high level, an iterated algebraic computation is modeled such that each iteration gives a random result, and hence the intermediate computations are all random until the determined time when the puzzle is solved. Others similarly cast the solution of a puzzle into a generic group (or ring) model [81, 117, 124] in order to make analogous claims about the information available before the solution time (in generic models the intermediate results are again random and not related to each other). As a common theme, these works arrive at an "instantaneous" revelation model, where the solver only learns the solution in the final step.[2]

However, Mahmoody et al. [91] showed that time-lock puzzles with superpolynomial gap

[1]While the authors in [69] do not explicitly treat their base puzzles as random until solved, we argue that their analysis implicitly does so, as they treat the repeated squaring assumption as having "instantaneous" reveal.

[2]In some cases, this is a limitation of the definition of time-lock puzzles only quantifying secrecy until the time when an adversary can guess the solution with non-negligible probability. In generic group models, the solution is explicitly hidden until the last step.

cannot be constructed from just random oracles. Their analysis explicitly considers a timed primitive for which each intermediate step is random. Therefore, analyzing *any algebraic* timed primitive in this way is applying an analysis to the solver that has already been shown does not match reality (or even approximate it) if assuming that a puzzle can be generated (via a trapdoor) much faster than it can be solved. Simply, applying random-oracle analysis to the solver is inconsistent with an algebraic puzzle generator, as we know random-oracle analysis does not yield a puzzle with superpolynomial gap. This leaves an open question of modeling the leakage mode of computational puzzles without a random oracle, which is necessary to fully analyze a time-lock puzzle with superpolynomial gap. We expound on this mismatch below.

Mahmoody et al.'s Result and Popular Idealized Analysis. Mahmoody et al.[91] showed that time-lock puzzles based only on random oracles *cannot* provide super-polynomial gap between generation and solving time. Their analysis explicitly considers time-lock puzzles for which each step in the solving process produces a random result.

This analysis is mirrored by analyses of time-lock puzzle solving in the literature. For example, Baum et al. [21, 22] *explicitly* model an idealized time-lock functionality that provides a uniformly random result at each step of the solving process. This form of analysis provides the next step in the solve algorithm as a random element that reveals no more information than the element itself. In the strong algebraic group model of [81, 124] and the generic ring model of [117], each element is expressed as a function of factors or as an inverse of another element, which gives algebraic structure to the elements that have been seen so far, but leaks nothing more about the final solution.

Analytical Mismatch. Analyzing a trapdoor-based time-lock construction by modeling the solving process as if a random oracle leads to apparent contradiction. On the one hand, algebraic constructions are believed to realize super-polynomial gaps between generation and solving. On the other hand, Mahmoody's analysis in which each next step is random and independent has been shown to only yield polynomial gaps. The state of current analysis is that puzzles are generated using a trapdoor and then the solving process is treated as a random oracle. These analyses do not match the realization, which should account for the computational difficulty of the solving process.

Implicit Random Oracles. Other works do not explicitly model the solving process via a random oracle, but either the modeling implies a random oracle or it overlooks leakage as the puzzle is partially solved. For example, the base time-lock puzzle in the construction of Freitag et al.[69] defer to analysis by Pietrzak [109] that assumes the hardness of repeated squaring. But the formalization simply assumes it is infeasible to guess the solution of a repeated squaring until the final squaring; either it implicitly treats the process as if the probability of guessing the solution before the end is negligible, or it uses a game-based definition that implies the solution process is essentially a random oracle. Therefore, these techniques as well are not differentiated in any meaningful way from the analysis of Mahmoody et al. and incur the same analytical mismatch as above.

Modeling Leakage After the Lock Expires. In the examples above, the repeated squaring assumption fails to model the leakage as the solver approaches the final squaring. Moreover, time-lock definitions such as those by [27] or [69] only require that the puzzle remain hard to solve until "close" to the honest solution time fall implicitly into the same trap. For these definitions, the modeling is still incomplete. The difference between the time when the honest parties arrive at the solution (following the honest solve algorithm), and the hypothesized time when the "gap" expires and the adversary can guess the solution with noticeable probability, is important to modeling a time-lock puzzle utilized during an MPC protocol. In this time, the adversary *can use* the puzzle solution with a head-start over the honest parties, even when honest parties are fully synchronized. We therefore advocate for a complete analysis that *fully models* the leakage of computational puzzles.

7.2 A Framework for Computational Puzzles

Current computational puzzles follow the following blueprint:[3]

1. The puzzle generator uses a trapdoor to efficiently sample a puzzle.
2. The solver uses a sequential algorithm to solve the puzzle. The puzzle is parameterized such that the sequential algorithm is faster than any known method to recover the trapdoor.

In this blueprint, the solver must be able to recover the secret within time that is polynomial in the puzzle's security parameter. Therefore, the (leaky) iterative process occupies a regime of fine-grained polynomial complexity, where (too much) information must not be leaked to an adversary with some polynomial depth d, but all information must be leaked when surpassing a different polynomial depth $d' > d$.

The above guides our work into a model of cryptography with fine-grained polynomial depth which, as we explain below, brings new challenges in modeling and intricate formal definitions.

7.2.1 Residual Complexity

To formalize the above notion of fine-grained polynomial hardness – in which some problems are solvable while *related underlying problems* remain hard – we introduce our definition of

[3]Recall that by Mahmoody et al.[91], there is no time-lock puzzle based solely on random oracles with superpolynomial time-gap, and therefore in practice puzzles depend on trapdoors.

residual complexity. Intuitively, residual complexity quantifies the "remaining hardness" of a puzzle that has already been (partially) solved by an adversary of depth d.[4]

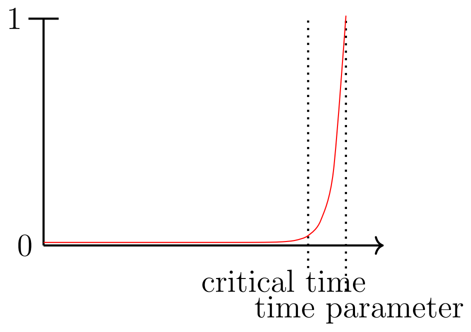
Definition 7.1 (Residual Complexity (Informal)). *A puzzle scheme has residual complexity r_d if no depth-d adversary can guess the solution of a randomly sampled puzzle with probability more than r_d.*

By this definition, 1-r_d is the "remaining hardness" of the puzzle after attempting to solve it in d depth. Our formalization (Definition 7.5) is a generalization of a technique in defining the depth-hardness of certain computational problems in [81] and others.

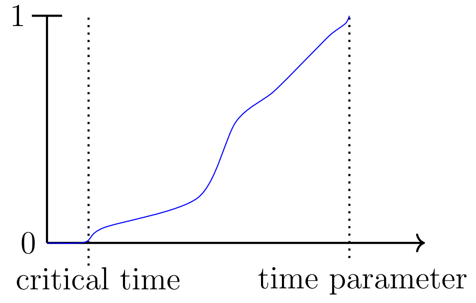
Residual complexity models the entire leakage profile of a puzzle by defining the "leakage" of a puzzle as the decrease in residual complexity of the puzzle between every two levels of depth of the best solving algorithm. We provide example curves of puzzle schemes in Figure 7.1. In the figure, the x axis represents time, and the y axis represents the best adversary's probability of guessing the solution. A point (x, y) on the curve represents that the best x-depth adversary guesses the solution with probability y. At the moment of the time-parameter, the puzzle is guaranteed to be solvable with probability 1 by the honest strategy.

As an illustration of the complexity degradation of a time-lock puzzle consider the RSW time-lock puzzle construction [114], which depends on the hardness of an iterative squaring mod RSA modulus N, while the underlying trapdoor problem of finding $\Phi(N)$ remains hard. Specifically, a solution χ is encoded via a puzzle (α, t, N) by setting $\chi = \alpha^{2^t} \bmod N$, where t is a hardness parameter that determines how many times the solver must repeatedly square $\alpha \bmod N$ in order to discover the solution, and $\log(N)$ is a function of the security parameter λ.

[4]Note that the remaining hardness measures *pseudo-entropy* rather than entropy, as the solution of a timed primitive is always committed at the moment it is generated. (Otherwise the solving algorithm could not be deterministic.)



(a) Example leakage profile for a time-lock puzzle.



(b) Example leakage profile for a puzzle scheme that briefly hides its solution.

Figure 7.1: Illustration of leakage profiles for two kinds of puzzles.

For RSW on small N, it is easy to see that revealing such intermediate steps leaks information about the nearby upcoming solutions. For larger moduli, even knowing $x^2 \bmod N$ leaks non-trivial information about $(x \pm \delta)^2 \bmod N$ for small δ (on the order of $\log(N)$).[5] By expanding the modulus to tune the hardness of the problem, the leakage diminishes but does not disappear because the algebraic structure remains.

In realistic computing scenarios and using clever algorithms[6], the solver learns a nontrivial distribution on the puzzle solution before it performs t squarings; we upper-bound the ability of an adversary to learn the solution by hypothesizing a leakage curve for the RSW scheme given a particular security parameter λ and time parameter t. We then quantify the difference in time between when the best adversary can guess a puzzle solution (with non-negligible probability) and the time that the honest parties learn the solution via the scheme's solve algorithm, and can name the moment when the adversary can guess with non-negligible probability as the *critical time*.[7]

[5]This is in contrast to a random oracle analysis, for which the next step of a puzzle is always independent of the current state.

[6]As another example, using parallel processors to compute forward mapping tables for quadratic residues modulo N.

[7]A negligible function is an asymptotic notion. For each security parameter, the protocol designer can choose a probability that is "unacceptable" for guessing the solution, and designate the depth for which the residual complexity meets this threshold as the critical time. This specifies the moment at which the time-lock is considered to expire.

7.2.2 Leakage and Temporary Privacy

The time-release of information is modeled by idealizing leaky functionalities which slowly provide information to the parties in a set of *phases* which take the place of time steps. A leaky functionality is parameterized by the leakage curve of the puzzle scheme it idealizes, which determines how much information (as a reduction in pseudo-entropy, as described above) the functionality provides to the adversary at each phase.

This treatment of idealized leakage captures applications where sensitive information is revealed during the computation, but must not be revealed before some specific point in time. For example, consider an accountable computing application, where parties time-lock their inputs and are held accountable to them at a later time. In traditional definitions of MPC, there is no way to quantify security of such a protocol. These inputs would be output by all parties, making any security reduction trivial. Because the simulator would receive all parties' inputs (as one party's output), the standard reduction for proving security would declare that no adversary could learn more information than a simulator *which already knows all of the parties' inputs.*

The formalization of temporary privacy requires that the simulator knows no more information at each phase than the adversary, which by definition learns information as the ideal functionality reveals it. Therefore, the simulator's input does not include the information which is revealed slowly throughout a computation; instead, it learns the information at the same rate as the adversary. It follows that in the security reduction, the proven statement is that the adversary can do no worse than a simulator *which knows the same amount of information at each step of the computation.* We can then claim that for time-release computations, privacy of some information holds for a specific amount of time, after which it is revealed and the adversary (respectively simulator) can use it.

7.2.3 Simulation Budgets and Depth-Secure MPC

Our treatment of time-based primitives and protocols requires a granular, depth-based definition of secure computation which departs from the standard cryptographic regime of "security up to arbitrary composition within complexity class $\mathbb{P}$," and must account for the depths of all involved machines – the adversary, the simulator, and the distinguisher.[8]

Specifically, security should hold with respect to an adversary with depth that is bounded by a fixed polynomial (in comparison to any polynomial in the security parameter). We bound the depth of a distinguisher (or environment) in tandem with the adversary. After surpassing these parameterized depths, it is alright for the information to be revealed.

Crucially, the simulator must also be bounded to some depth less than the puzzle requires to solve. Otherwise, the claim of privacy via reduction will completely fail: the reduction becomes a claim that an adversary can do no worse *than a simulator that could solve a puzzle outright and learn the solution*.[9] Therefore, the simulator has a granular "depth budget" and it must run in less time than privacy is required to hold. We give the formal definition in Section 7.5.3 and describe it informally as follows:

Definition 7.2 (Depth-Secure Multi-Party Computation (Informal)). *A protocol Π (d_a, d_s, d_e)-securely implements a functionality F if Π's simulator runs in no more than d_s depth, and the distribution of views produced by the simulator remains indistinguishable from the distribution of real executions for any d_a-bounded adversary and any d_e-depth bounded distinguisher (environment).*

Our observation on the simulator's depth budget leads to new questions about whether existing works apply to a realistic model such as ours. The simulator in the work by Baum

[8]To generalize between the works of [69] and [81], our definition states the depth of the environment, but the variable could be either polynomially bounded or unbounded. See [69] for discussion.

[9]For phased simulations, discussed in Section 7.5.5, this becomes that the simulator should run in less depth per phase than the adversary.

et al. [22] explicitly solves a time-lock puzzle during simulation, and is able to shortcut the solving process only because the simulator is not bound by the global clock functionality. [10] Freitag et al. [69] allow the simulator to explicitly solve puzzles, and artificially constrain the distinguisher by allowing it only to see a function of the solution of modified puzzles, which does not conform with meaningful definitions of a distinguisher *which could run the simulator on its own.* These are very delicate arguments, and while the corresponding constructions are elegant, they fall short of the nuances our fine-grained model brings to light: since the simulators are of a much different type than usual, nuances regarding the qualitative properties of the proofs using them follow.

7.2.4 Composition of Depth-Secure Protocols

We encounter additional challenges when composing secure timed primitives. For example, sequentially compose protocols Π and ρ, where Π is proven secure in the F-hybrid model against a d_a-depth adversary, and ρ securely implements F against a d'_a adversary. The composition Π^ρ is not trivially secure against a $d_a + d'_a$-depth adversary! An adversary against Π could use the time during ρ in order to continue attacking Π; similarly, an adversary against ρ could use the time *after* ρ *concludes* and Π resumes in order to continue attacking ρ. Similar issues occur in concurrent composition, although they are of the same ilk – in our model, the depth used by an environment to run a "side session" during an attack against Π counts towards its depth in the attack.

When composing protocols with timed primitives, the composed simulation must also be shorter than the time that privacy must hold. We also show that the black-box composition is secure only against the *smaller* of the two protocols' distinguishers, and against an adversary that is *smaller* adversary than the first protocol's adversary by the size of the second's

[10]This observation is more an indictment of bounding time with a global clock functionality than of the simulation technique, since the simulator is not constrained by the functionality and therefore not granularly constrained by depth/time.

simulator.

Theorem 7.1 (General Composition (Informal))**.** *Let* Π (d_a, d_s, d_e)*-securely implement* F *and let* ρ (d'_a, d'_s, d'_e)*-securely implement* G*. The composition of* Π *and* ρ *is* $(d_a - d'_s, d_s + d'_s, \mathsf{min}(d_e, d'_e))$*-secure.*

The term $d_a - d'_s$ comes from our simulation technique. Intuitively, if the composition is *not* secure against this depth of adversary, then there exists a d_a-depth adversary that simulates an execution of ρ in parallel to its attack on Π and uses the simulation to break Π.

The above theorem considers concurrent as well as sequential composition. We additionally prove another specialized, relaxed composition theorem, for protocols that cannot be proven concurrently composable but may be proven sequentially composable (e.g. if the simulator must be rewound).

Theorem 7.2 (Special Sequential Composition (Informal))**.** *Let* Π (d_a, d_s, d_e)*-securely implement* F *in the* G*-Hybrid model and let* ρ (d'_a, d'_s, d'_e)*-securely implement* G*. The composition* Π^ρ $(d_a - d'_s, d_s \cdot d'_s, \mathsf{min}(d_e, d'_e))$*-securely implements* F*.*

The multiplication in the middle term results from considering rewinding.

We present formal versions of these composition theorems for depth-bounded secure computation (Theorems 7.3 and 7.4) in Section 7.5.4. Note that our techniques are limited to composing depth-secure protocols in a black-box manner, and we do not prove tightness of degradation. The theorems also provide endpoints of a spectrum of budget depletion between the two extremes; when computing depletion for a specific composition in a non-black-box manner, it is possible to reason about the execution time of each simulator. There may be better techniques, including those with knowledge of the underlying protocols, that show tighter security preservation under composition.

7.3 Example Application: Simultaneous Multiple Round Auction

Our framework for depth-secure multiparty computation with timed cryptographic primitives enables us to reason about composition of timed primitives in a realistic setting. Consider that an application requires privacy of a primitive until time t[11]. The protocol designer parameterizes the scheme such that no one refutes (no experiment or mathematical analysis has shown otherwise to date) that the puzzle is unsolvable in t depth. We therefore conclude that the scheme is secure against a t-depth adversary. After depth t, we assume that the adversary can use the solution (earlier than honest parties learn the solution, who must wait for the time parameter).

As an illustration, consider the following variant of a Simultaneous Multiple Round Auction [97] with partial knowledge restriction and forced reveal.[12] As is standard in time-lock literature, time-locked bids guarantee that no party has information about other parties' bids within a round, while allowing for forced reveal.

Stage 1: Every party i submits a bid $b_{1,i}$ in an auction for the first round of bidding, to be opened not before time t_1.

Stage 2: After t_1, all bids for the first stage of the auction are opened. The parties who submitted the top c (constant) bids in Round 1 are chosen to submit stage-2 bids ($b_{2,i}$), to be opened not before t_2.

Winner: The winner of the auction is the party i that maximizes $B = b_{1,i} + b_{2,i}$ constrained such that i's bid in Stage 2 was opened. Party i pays \$$B$.

[11] t is the *critical time* described in Section 7.2.1

[12] This variant also has only a fixed number of rounds, but this can be trivially extended as per a true Simultaneous Multiple Round Auction.

Observe that this construction requires both *concurrent* and *sequential* composition of timed primitives. Within a stage of bidding, all of the time-lock puzzles require security *in concurrent composition* with each other. For each discrete round of bidding, the concurrently composed bids require sequential composition in order to be analyzed in the framework of a larger protocol.

Tuning for Timed Security We apply our composition theorems to estimate parameters to tune the security of concurrent primitives *within the first round.* Assume a protocol for submitting and solving an individual puzzle bid which can be set to be (d_a, d_s, d_e)-secure. If the concurrent composition of n puzzles must be secure against an adversary of depth t_1, then each individual puzzle must be tuned such that $d_a = t_1 + (n-1)d_s$. The depth of the composed simulation is at most nd_s. The concurrent composition for this round is then (at least) (t_1, nd_s, d_e)-secure.

To analyze the full example above, including sequential composition between rounds, we analogously apply the composition theorems again, tuning so that the first round is secure for t_1 depth, the second round is secure for $t_2 - t_1$ depth, and the full composition is secure for t_2 depth.

7.4 Defining Time-Lock Puzzles

The following definitions for time-lock puzzles are adopted from Bitansky et al. ([27] Definition 3.1).

Definition 7.3 (Puzzle). *A* puzzle *for solution domain $M = \{M_\lambda\}_\lambda$ is a pair of algorithms* $\mathsf{Puz} = (\mathsf{Puz.Gen}, \mathsf{Puz.Solve})$ *for which*

– $Z \leftarrow \mathsf{Puz.Gen}(t, \chi)$ *is a probabilistic algorithm over difficulty parameter* $t \in \mathbb{N}$ *and* *solution $\chi \in M_\lambda$, where λ is a security parameter, and outputs puzzle Z.*

- $\chi \leftarrow \mathsf{Puz.Solve}(Z)$ *is a deterministic algorithm that takes as input puzzle Z and outputs solution $\chi \in M_\lambda$.*

subject to the following constraints:

- ***Completeness:*** *For every security parameter λ, difficulty parameter t, solution $\chi \in M_\lambda$, and puzzle Z in the support of* $\mathsf{Puz.Gen}(t, \chi)$, $\mathsf{Puz.Solve}(Z)$ *outputs χ.*
- ***Efficiency:***
  - $Z \leftarrow \mathsf{Puz.Gen}(t, \chi)$ *can be computed in size* $\mathsf{poly}(\log t, \lambda)$.
  - $\mathsf{Puz.Solve}(Z)$ *can be computed in size* $t \cdot \mathsf{poly}(\lambda)$.

We continue by adapting the more constrained definition of a *time-lock* puzzle by Bitansky et al. ([27] Definition 3.2).

Definition 7.4 (Time-Lock Puzzles)**.** *A puzzle* $\mathsf{Puz} = (\mathsf{Puz.Gen}, \mathsf{Puz.Solve})$ *is a* time-lock *puzzle for solution domain $M = \{M_\lambda\}_\lambda$ with gap $\varepsilon < 1$ if there exists a polynomial $r(\cdot)$ such that for every polynomial $t(\cdot) \geq r(\cdot)$ and every polynomial size, t^ε-depth-bounded adversary $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda \in \mathbb{N}}$, there exists a negligible function* negl *such that for every $\lambda \in \mathbb{N}$, and every pair of solutions $\chi_0, \chi_1 \in M_\lambda$:*

$$\Pr[b \leftarrow \mathcal{A}_\lambda(Z) \colon b \leftarrow \{0,1\}, Z \leftarrow \mathsf{Puz.Gen}(t(\lambda), \chi_b)] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$$

7.4.1 Residual Complexity and Leakage

Residual Complexity

Residual complexity more granularly describes the remaining hardness of a partially-solved, randomly sampled puzzle than Definition 7.4 above. Specifically, it measures the pseudo-entropy [78, 123] of a puzzle solution from the perspective of a computationally bounded solver.

Definition 7.5 (Residual Complexity)**.** *For a function $r\colon \mathbb{N} \to [0,1]$, we say that a puzzle* Puz *with solution domain $M = \{M_\lambda\}_{\lambda\in\mathbb{N}}$ has (d, r) residual complexity if for every depth d-bounded adversary A_d, and every $\lambda \in \mathbb{N}$:*

$$\Pr[\chi \leftarrow A_d(Y)\colon \chi \leftarrow M_\lambda, Y \leftarrow \mathsf{Puz.Gen}(\lambda, \chi)] \leq r(\lambda)$$

When d is implied by context, we refer the residual complexity of a puzzle by the function r. When we consider the residual complexity of a puzzle at a particular depth d, we explicitly write r_d. One can consider $1 - r(\lambda)$ to be the remaining hardness of the puzzle.

Leakage

Residual complexity naturally induces a definition of a puzzle's leakage over time. We call the function describing the loss in pseudo-entropy at each level of depth a *leakage curve.* For depths d_1 and d_2 the quantity $r_{d_2} - r_{d_1}$ represent the loss in pseudo-entropy of a puzzle between d_2 and d_1.

For any puzzle scheme, there exists a family of leakage functions indexed by the security parameter and the time parameter, denoted $\mathcal{L} = \{\ell_{\lambda,\tau}\}_{\lambda,\tau}$, such that each function describes, based on the parameterization, the information a party can extract from the puzzle over time.

The time parameter τ describes the number of sequential operations required to "solve" the puzzle in the honest case. The security parameter λ tunes the computational difficulty of guessing the solution before τ has elapsed. Specifically, λ parameterizes the underlying computational problem which the iterative process solves; for the RSW puzzle, λ describes the size of the modulus used for repeated squaring. In this work, we always consider a specific leakage function ℓ and elide the subscripts from the notation because they are implied by context.

Intuition: The Distribution of Solutions Intuitively, we consider the *leakage* that a party obtains on a puzzle to inform the distribution that the party learns on the puzzle's solution. Any puzzle-solving strategy must imply a distribution on the strategy's "best guess" of a puzzle solution at any point in time. When a party receives a puzzle, the distribution of its best guess has very high pseudo-entropy. As the party learns leakage on the solution, the pseudo-entropy of the distribution of the solution decreases, and the distribution redistributes its mass until eventually all of the mass lies in a single point: the puzzle's solution. We can then understand the leakage of a puzzle to provide a distribution on the solution of the puzzle for every depth d. The residual complexity r_d gives an upper bound on the mass implied at the point of the puzzle solution.

7.5 Modeling Multi-Party Computation

This section discusses in detail the modeling issues that arise from composition of timed primitives with other cryptographic computations, including simulating leaky functionalities.

To provide a full treatment of depth-secure multi-party computation, we present two models:

1. A "general" model which adapts the Universal Composability (UC) framework [39]

such that all parties (including the environment, trusted third party, and adversary) are modeled as interactive circuits.

2. A “sequential” model, which is useful for proving security of sequential composition of protocols which cannot be proven secure in our more general model, but is otherwise similar, and adapts standard sequential models to our fine-grained treatment.

We then present our definitions for depth-secure computation and theorems – both general and sequential – for how depth-secure protocols compose.

7.5.1 General Execution Model

In our generalized, UC-like model, we consider an execution in the presence of an *environment* that provides inputs to parties and reads their outputs. The environment is an interactive Turing machine which directs the execution, analogously to the environment in the UC [39]. It delivers inputs to parties as well as messages that have been sent to them by the adversary. The environment is also responsible for delivering messages between parties.

The adversary informs the environment which parties it will (adaptively) corrupt, and the environment passes the adversary all of the corrupt parties’ inputs, the queries they make, and the responses they receive (the latter two are analogous to the messages they send and receive, adapted for our model). The adversary may also inform the environment before the execution which parties it will corrupt from the start; in this case, the environment passes the adversary those parties’ inputs and the adversary may choose to replace their inputs by responding to the environment. Only after this exchange, the environment provides inputs to all honest parties. This models that an adversary may select inputs in order to affect a computation, which is within the application scenario of accountable computation.

For a full treatment of the execution model, refer to Appendix F.1.

The Ideal/Real Paradigm in the General Model

We next describe our general ideal/real paradigm for granular-depth secure multi-party computation (MPC).

Execution in the Real Model. In the real model, participants execute a protocol Π to compute the desired functionality $\mathcal{F}$ without a trusted party. At the end of the execution, honest parties output their protocol outputs. The corrupt parties output nothing. The adversary outputs an arbitrary function of its inputs and the messages that corrupt parties have received. The environment learns every output. The random variable $\mathsf{REAL}_{\Pi,\mathcal{A}(z),\mathcal{Z}}(\overline{x})$ denotes the output of the environment in a real execution of Π with honest inputs $\overline{x}$, auxiliary input z to $\mathcal{A}$, with environment $\mathcal{Z}$.

Execution in the Ideal Model. In an ideal execution, the parties interact with a trusted party by submitting all of their inputs to the trusted party in the beginning of the execution. The trusted party for a leaky functionality responds to the parties by dividing an execution into *phases* such that at the end of each phase, the parties receive some output.

At the end of an execution, honest parties output whatever they have received from the trusted party. Corrupt parties output nothing, and the adversary outputs an arbitrary function of its input and the messages that corrupt parties have received from the trusted party. The environment learns every output. The random variable $\mathsf{IDEAL}_{\mathcal{F},\mathcal{A}(z),\mathcal{Z}}(\overline{x})$ denotes the output of the environment in an *ideal execution* of functionality $\mathcal{F}$ on honest inputs $\overline{x}$, auxiliary input z to $\mathcal{A}$, with environment $\mathcal{Z}$.

7.5.2 Sequential Model

Our sequential model is like the general model, except that each protocol execution is considered in isolation, and instead of being directed by the environment, it is directed by the adversary itself. The adversary that controls message deliveries and may adaptively corrupt parties throughout an execution. The adversary activates all parties within each round, and within each round all parties execute a computation of the same depth. The adversary can additionally adaptively corrupt parties and inject messages, analogously to the exposition in Appendix F.1.

The Real/Ideal Paradigm in the Sequential Model

Execution in the Real Model In the real model, the parties execute a protocol Π in the presence of an adversary $\mathcal{A}$. The random variable $\mathsf{REAL}_{\Pi,\mathcal{A}(z)}(\overline{x})$ denotes the execution transcript on a real execution of Π with honest inputs $\overline{x}$ and auxiliary input z to adversary $\mathcal{A}$. The execution transcript includes all of the honest parties' inputs, the honest parties' outputs, and the adversary's output.

Execution in the Ideal Model As in the general model, in the ideal experiment the honest parties send their inputs to a trusted third party, and the third party delivers the results. The simulator generates an execution transcript by interacting with the third party and with the adversary on behalf of the honest parties. The random variable $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S}(z)}(\overline{x})$ denotes an execution transcript generated by an adversary $\mathcal{S}$ in an idealized execution of functionality $\mathcal{F}$ on honest inputs $\overline{x}$ and auxiliary input z to $\mathcal{S}$.

7.5.3 Depth-Bounded Secure Multi-Party Computation

Depth Constraints For a meaningful definition of secure multi-party computation (MPC) with timed primitives, the computational power of the simulator must be constrained in a manner similar to the adversary's. Otherwise, if the depth of the simulator is substantially more than the adversary, then the simulator could (for example) solve a time-lock puzzle, and use the solution in the simulation. It would be meaningless to argue privacy by claiming that any information the adversary can learn about the honest parties' inputs in a real execution could also be learned by a simulator *which explicitly solves* a time-lock puzzle in order to learn secret information (such as honest parties' inputs).

Our definitions below therefore constrain the depths of both the simulator and the adversary. We also depth-constrain the distinguisher, intuitively because for timed primitives we need only to show security *for some amount of time.*

Definition 7.6 (Depth-Bounded Secure Computation: General)**.** *Let $d_a = d_a(\lambda)$, $d_s = d_s(\lambda)$, and $d_e = d_e(\lambda)$. Protocol Π (d_a, d_s, d_e)-depth securely computes $\mathcal{F}$ if there exists a d_s-depth-bounded $\mathcal{S}$ such that for every real-world d_a-depth-bounded adversary $\mathcal{A}$ and every d_e-depth-bounded environment $\mathcal{Z}$, the following two ensembles are d_e-depth indistinguishable:*

$$\{\mathsf{REAL}_{\Pi,\mathcal{A}(z),\mathcal{Z}}(\overline{x})\}_{\overline{x}\in(\{0,1\}^*)^n, z\in\{0,1\}^*}$$

$$\{\mathsf{IDEAL}_{\mathcal{F},\mathcal{S}(z),\mathcal{Z}}(\overline{x})\}_{\overline{x}\in(\{0,1\}^*)^n, z\in\{0,1\}^*}$$

Remark 7.1. *Ours definitions for composition say that a protocol (d_a, d_s, d_e)-securely computes some functionality if there is a d_s-depth bounded universal simulator $\mathcal{S}$ such that for every d_a-depth-bounded adversary, $\mathcal{S}$ produces a distribution of views that is d_e-depth indistinguishable from a real execution. Although we reverse the order of quantifiers for the simulator and adversary in the definition from the standard ordering, most proofs are written by providing a universal simulator that works for any adversary.*

The depth of the distinguisher. The constraint on a distinguisher's depth (in this case, the environment; below, the distinguisher) is a significant weakening of the definition compared to those by Goldreich or Lindell's [75, 89], as neither constrains the depth of the distinguisher by a granular polynomial. However, this weakening is sufficient for our setting, since in practice, if a time-locked output will eventually be revealed anyway, we require indistinguishability of the simulation only for the duration of the experiment.

Depth-Secure Computation: Sequential In the sequential model, as explained above, the execution is directed by the adversary, and the real and ideal experiments should be indistinguishable to a depth-bounded distinguisher who receives a transcript of the execution.

Definition 7.7 (Depth-Bounded Secure Computation: Sequential). *Let $d_a = d_a(\lambda)$, $d_s = d_s(\lambda)$, and $d_e = d_e(\lambda)$. Protocol Π (d_a, d_s, d_e)-depth securely computes $\mathcal{F}$ if there exists a d_s-depth-bounded $\mathcal{S}$ such that for every d_a-depth-bounded real-world adversary $\mathcal{A}$, the following two ensembles are d_e-depth indistinguishable:*

$$\{\mathsf{REAL}_{\Pi,\mathcal{A}(z)}(\overline{x})\}_{\overline{x}\in(\{0,1\}^*)^n, z\in\{0,1\}^*}$$

$$\{\mathsf{IDEAL}_{\mathcal{F},\mathcal{S}(z)}(\overline{x})\}_{\overline{x}\in(\{0,1\}^*)^n, z\in\{0,1\}^*}$$

7.5.4 Composition

We now treat the composition of depth-secure protocols. In the following, we use the notation Π^ρ to denote that protocol Π calls ρ as a subroutine, as per the convention by Canetti [39]. We use the notation that $\Gamma^{\Pi,\rho}$ denotes the concurrent composition of Π and ρ.

General/Concurrent Composition

We now state our general composition theorem, which includes concurrent composition.

Theorem 7.3 (Composition of Two Depth-Secure Protocols)**.** *Let* Π (d_a, d_s, d_e)*-depth-securely compute functionality* F *and let* ρ (d'_a, d'_s, d'_e)*-depth-securely compute functionality* G*. Then* $\Gamma^{\Pi,\rho}$ *is* $(d_a - d'_s, d_s + d'_s, \mathsf{min}(d_e, d'_e))$*-secure.*

Sketch. We define a simulator $\mathcal{S}$ for Π^ρ that simply composes the simulators $\mathcal{S}^\Pi$ and $\mathcal{S}^\rho$ which exist by assumption. We then perform a reduction that shows if there is an attack against Π^ρ, we can isolate an attack against Π in the G-hybrid model. The reduction is straightforward, although it must carefully consider the depths of all simulators and adversaries. Given an adversary $\mathcal{A}$ which attacks Π^ρ, we define an adversary $\mathcal{B}$ such that $\mathcal{B}$ runs $\mathcal{A}$ as a black box, and $\mathcal{B}$ forwards messages sent by $\mathcal{A}$ to their recipients. The only exception is that $\mathcal{B}$ must simulate an execution of ρ for $\mathcal{A}$ when $\mathcal{A}$ expects ρ to be called. The full proof is deferred to Section 7.7. □

Remark 7.2 (The Depths d_a and d'_e)**.** *For all composition theorems, we require that* $d_a < d'_e$*. This is a natural choice; in particular if* $d_a \geq d'_e$ *then the theorem is not meaningful. Specifically, if* $d_a \geq d'_e$*, then the adversary for the first protocol is deep enough to distinguish an execution of the protocol* ρ *which is called by it from the callee's simulation; the composition therefore does not have meaningful real-world consequences, since a realistic adversary against the composition implies an adversary for the callee protocol. For all following theorems, we elide the statement of this requirement.*

We see from the composition theorem that when composing two depth-secure protocols in order to achieve security against any d^*_a-depth adversary, the composed protocols must be parameterized so that they are secure against stronger adversaries, due to the loss in security that results from composition. Moreover, the composition remains secure only against the smaller of the two distinguishing environments.

Discussion: Non-malleability. One might think that Theorem 7.3 implies that any depth-secure puzzle is non-malleable because we have shown that the protocols are naively composable. This is not quite true; our result says that a secure protocol remains secure (and non-malleable) only for the *min* granular depth of the concurrent runtimes, and against a smaller adversary. In comparison, nonmalleability definition as in the definition of [69] are secure for *arbitrary polynomial* runtime of the adversary; however, they prove only bounded nonmalleability (is possible), and require tuning parameters based on the number of composed primitives (as we do).

Sequential Composition

In some cases, a protocol cannot be proven concurrently composable, if the simulator needs to be rewound. We therefore provide a "weaker" theorem for the sequential composition of protocols that cannot be proven secure with respect to the general theorem.

Theorem 7.4 (Sequential Composition of Two Depth-Secure Protocols)**.** *Let* Π (d_a, d_s, d_e)*-depth-securely compute* F *in the* G*-hybrid model, and let* ρ (d'_a, d'_s, d'_e)*-depth-securely compute* G*.* Π^ρ $(d_a - d'_s, d_s \cdot d'_s, \mathsf{min}(d_e, d'_e))$*-depth-securely computes* F*.*

Observe that the decrease in simulation budget for the concurrent composition theorem appears to be "better" than the "weaker" sequential theorem because the simulation budget does not deteriorate as much; however, this is attributable to the fact that the simulator for a concurrently composable protocol must already be more efficient than the simulator for the sequential theorem above, as rewinding is not permitted (as in the UC[39]).

We note that the $(d_s \cdot d'_s)$ term in the $(\cdot, d_s \cdot d'_s, \cdot)$-depth security of the composed protocols is too pessimistic in some cases. In the case that the simulator for the calling protocol never needs to rewind over the invocation of the subroutine protocol, we can prove stronger security for the composition. This is in fact a direct fallback to Theorem 7.3.

Corollary 7.1 (Optimistic Sequential Composition of Depth-Secure Protocols)**.** *Let* Π (d_a, d_s, d_e)*-depth-securely compute* F *in the* G*-hybrid model, and let* ρ (d'_a, d'_s, d'_e)*-depth-securely compute* G*. If the simulator for* Π *in the* G*-hybrid model never rewinds over the point at which* G *is invoked, then* Π^ρ $(d_a - d'_s, d_s + d'_s, \mathsf{min}(d_e, d'_e))$*-depth-securely computes* F*.*

Sketch. This follows immediately from Theorem 7.3, and in fact when the simulator does not need to be rewound, the protocol is also concurrently composable. □

Discussion. Theorem 7.4 and Corollary 7.1 give the bounds on the spectrum of "simulation budget depletion" that may occur when composing depth-secure protocols. Specifically, in order to make a meaningful statement about security, the middle term d_s must remain smaller than both of the outer terms d_a and d_e. For a particular composition, the protocol designer may compute the actual security statement by computing the runtime of the composed simulator.

7.5.5 Simulation for Leaky Functionalities

In the standard definition of secure multi-party computation (MPC) [75, 89], the simulator is given – as input – the adversary's ideal-world outputs, and then it must produce a view for the adversary that is indistinguishable from its view in a real execution. For functionalities where honest parties' inputs are not revealed, privacy is implied by this definition because the simulator must produce such an execution *without access to the honest parties' inputs or outputs*. However, in some leaky applications the adversary *may* learn the honest parties' inputs, but crucially, the honest parties' inputs are hidden for some period of time.[13] For such applications, the standard MPC definition does not imply privacy of honest parties' inputs up to the point in time that they are revealed because the simulator receives the honest parties' inputs in the beginning.

When we require privacy for some amount of time, we decompose the simulation into a series of phases such that in each phase the simulator learns only the information which is permitted to be revealed at the end of that phase.[14] Specifically, an ideal functionality is parameterized with the leakage function ℓ of the puzzle it emulates. In each phase, the simulator receives from the functionality only the information that the functionality is defined to release during that phase. The leakage is specified by the difference in residual complexity between phases, as described in Section 7.4.1, which is given by the parameterized leakage function ℓ. This means that in contrast to standard definitions of secure computation, the simulator does not receive all of the adversary's outputs as an input to the computation. This enforces that the simulator does not learn any information before it is supposed to, which implies privacy of the inputs until they are revealed.

7.6 Residual Complexity of a Time-Lock Puzzle

The qualification of a *time-lock* puzzle tells us that for any circuit $\mathcal{A}_d$ attempting to solve a puzzle for which d is much less than the depth required by $\mathsf{Puz.Solve}$, the probability of guessing the solution should be no better than random guessing plus negligible advantage. However, a circuit $\mathcal{A}_d$ whose depth d exceeds t^ε (as enforced in the definition) may have non-negligible advantage in guessing the solution. Therefore, a time-lock puzzle constrains the residual complexity function r of the puzzle to remain small for as long as the time-lock

[13]For example, in accountable computing scenarios.

[14]This is morally similar to the simulation technique of Baum[21], where the simulator forwards next-step queries to the ideal functionality and returns the result. Our simulator automatically learns the leakage for a phase (which may be longer than one step) at the beginning of the phase so that it can simulate the rest.

endures. We now formally prove this intuition.

Theorem 7.5 (Time-Lock Puzzle Implies Small Residual Complexity). *Let* $\mathsf{Puz} = (\mathsf{Puz.Gen}, \mathsf{Puz.Solve})$ *be a time-lock puzzle for solution domain* $M = \{\chi_\lambda\}_\lambda$ *with gap* $\varepsilon < 1$ *for which* $|M_\lambda|$ *is super-polynomial in* λ. *Then there exists a polynomial* $r(\cdot)$ *for which for every polynomial* $t(\cdot) > r(\cdot)$ *and* t^ε*-depth-bounded* $\mathcal{A}_t$, *there exists a negligible function* negl *such that for every* t^ε*-depth-bounded* $\mathcal{B}$, *and every* $\lambda \in \mathbb{N}$

$$\Pr[\chi \leftarrow \mathcal{B}(Y) \colon \chi \leftarrow M_\lambda, Y \leftarrow \mathsf{Puz.Gen}(\lambda, \chi)] \leq \mathsf{negl}$$

Proof. We prove the lemma by showing that if there exists an adversary $\mathcal{B}_t$ for which $\Pr[\chi \leftarrow \mathcal{B}(Y, z)] > \mathsf{negl}$, then there exists an adversary $\mathcal{A}_t$, infinitely many λ and corresponding solutions $\chi_0, \chi_1 \in M_\lambda$ such that $\mathcal{A}_\lambda$ can win the time-lock challenge with probability more than $\frac{1}{2} + \mathsf{negl}$. For the sake of the proof, let $r > \mathsf{negl}$ be the probability with which $\mathcal{B}$ outputs χ in the above challenge game.

Actually, we show a result corresponding to a stronger statement. If there exists an adversary $\mathcal{B}$ that wins the above challenge game with non-negligible advantage, then there exists an adversary $\mathcal{A}$ such that for every χ_0 there are many solutions χ_1 such that $\mathcal{A}_\lambda$ can win the time-lock challenge with probability more than $\frac{1}{2} + \mathsf{negl}$. Our time-lock game is slightly weaker than the definition of a time-lock puzzle (and therefore if our time-lock game is broken, the puzzle is not a time-lock puzzle). Rather than quantifying over all χ_0 and χ_1, we allow the adversary $\mathcal{A}$ to choose any $\chi_0, \chi_1 \in \chi_\lambda$ and provide them to a challenger, who samples b and provides $\mathcal{A}$ with a puzzle. $\mathcal{A}$ must guess b' and wins if $b' = b$.

We now explain how $\mathcal{A}$ uses $\mathcal{B}$. Recall that $\mathcal{B}$ is given a randomly sampled puzzle and outputs a guess χ' of the solution. In the time-lock game, $\mathcal{A}$ samples χ_0 and χ_1 at random and must determine which one has been encoded in a challenge puzzle Z. $\mathcal{A}$ forwards Z to $\mathcal{B}$. At the end, $\mathcal{A}$ inspects the guess χ' that $\mathcal{B}$ makes. If χ' is equal to either χ_0 or χ_1, then

$\mathcal{A}$ guesses the b for which $\chi_b = \chi'$. If neither χ_0 nor χ_1 is guessed by $\mathcal{B}$, then $\mathcal{A}$ samples b' uniformly at random and outputs b'. Note that the depth of $\mathcal{A}$ is the same as $\mathcal{B}$.

Recall that $\mathcal{B}$ wins its game with probability r, and that by assumption r is non-negligble. We now analyze the probability with which $\mathcal{A}$ wins its game.

Claim 7.1. $\Pr[\chi' \in \{\chi_0, \chi_1\}] \geq \Pr[\mathcal{B}\ \textit{wins}] > \mathsf{negl}$

Proof. Follows immediately from the definition that $\mathcal{B}$ wins when it guesses the solution, and by assumption that $\mathcal{B}$ wins with non-negligible probability. □

Claim 7.2. $\Pr[\chi' = \chi_{1-b}] = \mathsf{negl}$

Proof. Consider that χ_{1-b} is selected at random by $\mathcal{A}$, and $\mathcal{B}$ has no information about χ_{1-b}. Recall that conditioned on the fact that $\mathcal{B}$ guesses some possible solution with non-negligible probability (the true solution), and let X be the part of the solution space for which $\mathcal{B}$ outputs solutions in X with non-negligible probability. Let Y be the part of the solution space for which $\mathcal{B}$ guesses solutions with negligible probability. We claim that X composes a negligible proportion of the solution space, and that therefore χ_{1-b} is in Y except for negligible probability. The proof proceeds by counting. For all of the points in X, $\mathcal{B}$ must guess each point with probability at least the inverse of some polynomial. It follows that there may only be a polynomial number of points in X. However, there are a super-polynomial number of points in the solution space. Therefore, the probability that χ_{1-b} is in Y is overwhelming. And by definition of Y, the probability that $\mathcal{B}$ guesses χ_{1-b} is negligible. □

It follows from the previous claim that conditioned on $\mathcal{B}$ outputting χ_0 or χ_1, $\mathcal{A}$ wins with probability $1 - \mathsf{negl}$.

Claim 7.3. $\Pr[\mathcal{A}\ \textit{wins} \mid \chi' \in \{\chi_0, \chi_1\}] = 1 - \mathsf{negl}$

Proof. The probability that $\mathcal{A}$ wins given that one of the solutions output by $\mathcal{B}$ is divided into cases:

1. $\chi' = \chi_{1-b}$. $\mathcal{A}$ loses
2. $\chi' = \chi_b$. $\mathcal{A}$ wins

By the previous claim, the probability of the first event is negl. In the remaining case, $\mathcal{A}$ wins. Note that because the second case with non-negligible probability, this case dominates, as the other composes a negligible proportion of the event space. It follows that $\mathcal{A}$ wins with probability $1 - \mathsf{negl}$ given $\mathcal{B}$ outputs either χ_0 or χ_1. □

We can now conclude the proof:

$$\begin{aligned}
\Pr[\mathcal{A} \text{ wins}] &= \Pr[\mathcal{A} \text{ wins } \mid \chi' \in \{\chi_0, \chi_1\}] \Pr[\chi' \in \{\chi_0, \chi_1\}] \\
&\quad + \Pr[\mathcal{A} \text{ wins} \mid \chi' \notin \{\chi_0, \chi_1\}] \Pr[\chi' \notin \{\chi_0, \chi_1\}] \\
&= (1 - \mathsf{negl}) \Pr[\chi' \in \{\chi_0, \chi_1\}] + \frac{1}{2} \Pr[\chi' \notin \{\chi_0, \chi_1\}]
\end{aligned}$$

Recall that the two events $\chi' \in \{\chi_0, \chi_1\}$ and $\chi' \notin \{\chi_0, \chi_1\}$ are complements. Therefore, if $\Pr[\chi' \in \{\chi_0, \chi_1\}] > \mathsf{negl}$, then $\Pr[\mathcal{A} \text{ wins}] > \frac{1}{2} + \mathsf{negl}$. The proof concludes by the first claim, which states that $\Pr[\chi' \in \{\chi_0, \chi_1\}] \geq \Pr[\mathcal{B} \text{ wins}] > \mathsf{negl}$. □

7.7 Concurrent Composition of Depth-Secure Protocols: Proof of Theorem 7.3

In this section, we provide the full proof of Theorem 7.3, which we restate below for convenience. Recall the notation that $\Gamma^{\Pi,\rho}$ denotes the concurrent composition of Π and ρ.

Similarly, $\zeta^{F,G}$ is a functionality that concurrently provides functionalities F and G. We also let $\mathsf{VIEW}_{\mathcal{A}}(\cdot)$ denote the view of $\mathcal{A}$ during the enclosed experiment.

Theorem 7.3 (Composition of Two Depth-Secure Protocols)**.** *Let Π (d_a, d_s, d_e)-depth-securely compute functionality F and let ρ (d'_a, d'_s, d'_e)-depth-securely compute functionality G. Then $\Gamma^{\Pi,\rho}$ is $(d_a - d'_s, d_s + d'_s, \mathsf{min}(d_e, d'_e))$-secure.*

Proof. First we create a simulator $\mathcal{S}$ for the composition. $\mathcal{S}$ works by invoking the simulators $\mathcal{S}^{\Pi}$ and $\mathcal{S}^{\rho}$ (for Π and ρ, respectively) in parallel. Note that its depth is at most $d_s + d'_s$.

For the sake of the following lemma, we use the notation $\overline{x}$ to denote the honest parties' inputs and z to denote an auxiliary input. Because we consider two separate protocols in concurrent composition, we let $\overline{x} = (\overline{x}_1, \overline{x}_2)$ where $\overline{x}_1$ are for Π and $\overline{x}_2$ are for ρ, and similarly we let $z = (z_1, z_2)$ with analogous association.

We now state our main lemma, from which the proof follows.

Lemma 7.1. *Let $f' = \mathsf{min}(d_e, d'_e)$. For every $d_a - d'_s$-depth adversary $\mathcal{A}$, every $\mathsf{min}(d_e, d'_e)$-depth environment $\mathcal{Z}$, and every $\overline{x} \in (\{0,1\}^{\mathsf{poly}})^n$ and $z \in \{0,1\}^{\mathsf{poly}}$:*

$$\mathsf{REAL}_{\Gamma^{\Pi,\rho},\mathcal{A}(z),\mathcal{Z}}(\overline{x}) \overset{f'}{\approx} \mathsf{IDEAL}_{\zeta^{F,G},\mathcal{S}(z),\mathcal{Z}}(\overline{x})$$

Proof. Assume towards contradiction that the above is not true. Then there exist a $(d_a - d'_s)$-depth adversary $\mathcal{A}$, a $\mathsf{min}(d_e, d'_e)$-depth environment $\mathcal{Z}$, and inputs $\overline{x}, z$ for which $(\mathcal{A}, \mathcal{Z})$ distinguishes the two distributions (for any simulator $\mathcal{S}$).

We build an adversary $\mathcal{B}$ and environment $\mathcal{E}$ that distinguish the execution of Π from its simulation on honest inputs $\overline{x}$ and advice string z. $\mathcal{E}$ will run $\mathcal{Z}$ as a black box, forwarding messages to $\mathcal{Z}$, sending whatever messages $\mathcal{Z}$ sends, and outputting whatever $\mathcal{Z}$ outputs. $\mathcal{B}$ will use $\mathcal{A}$ and $\mathcal{Z}$ to attack its real-world execution of Π, but $\mathcal{B}$ will simulate the concurrent execution of ρ for $\mathcal{A}$ (and $\mathcal{Z}$) *in parallel* to the execution of Π. By the assumption that ρ

is secure, this will imply that $\mathcal{B}$ and $\mathcal{E}$ use $\mathcal{A}$ and $\mathcal{Z}$ to distinguish Π from its simulation, reaching contradiction.

We first introduce notation for an experiment which $\mathcal{B}$ uses to attack Π. In this experiment, $\mathcal{B}$ and $\mathcal{E}$ will attack a real execution of Π by running $\mathcal{A}$ and $\mathcal{Z}$ as black boxes; when they expect messages from the run of ρ, $\mathcal{B}$ simulates a concurrent execution of ρ using $\mathcal{S}^{\rho}$. We denote the experiment by $\mathsf{REAL}^{\mathcal{B}}_{\Gamma^{\Pi,G},\mathcal{A}(z),\mathcal{Z}}(\overline{x})$. We argue that by the security of ρ, $\mathcal{A}$'s view of this distribution must be indistinguishable from its view of $\mathsf{REAL}_{\Gamma^{\Pi,\rho},\mathcal{A}(z),\mathcal{Z}}(\overline{x})$.

Claim 7.4. *Let $f' = \mathsf{min}(d_e, d'_e)$. For any f'-depth $\mathcal{Z}$, for all $\overline{x} \in (\{0,1\}^{\mathsf{poly}})^n$ and $z \in \{0,1\}^{\mathsf{poly}}$*

$$\mathsf{VIEW}_{\mathcal{A}}(\mathsf{REAL}_{\Gamma^{\Pi,\rho},\mathcal{A}(z),\mathcal{Z}}(\overline{x})) \overset{f'}{\approx} \mathsf{VIEW}_{\mathcal{A}}(\mathsf{REAL}^{\mathcal{B}}_{\Gamma^{\Pi,G},\mathcal{A}(z),\mathcal{Z}}(\overline{x}))$$

Proof. The difference between the two distributions is that on the right, $\mathcal{B}$ simulates an execution of ρ using the simulator $\mathcal{S}^{\rho}$ and provides those messages to $\mathcal{A}$ (and $\mathcal{Z}$), and then continues to call $\mathcal{A}$ after the call to $\mathcal{S}^{\rho}$ using messages from its real execution. By assumption, $\mathcal{A}$ is $(d_a - d'_s)$-depth-bounded and $d_a < d'_e$. Therefore, $\mathcal{A}$ must not be able to distinguish the messages in the real execution of ρ on the left from the simulation on the right. By a similar argument, neither can (any) $\mathcal{Z}$. The claim follows from the additional fact that all other messages in $\mathcal{A}$'s view are distributed indistinguishably in both experiments, since they are both from a real execution of Π. $\square$

We make another claim that is analogous to the previous, but for the ideal experiment. We claim that $\mathcal{A}$ cannot distinguish between an idealized execution of $\zeta^{F,G}$ in which $\mathcal{S}$ generates $\mathcal{A}$'s view of the execution, and an idealized execution of F in which $\mathcal{B}$ forwards messages generated for it by $\mathcal{S}^{\Pi}$, and in place of the ideal functionality call to G, $\mathcal{B}$ generates a view of the call to ρ (realizing G) by simulating $\mathcal{S}^{\rho}$, and forwards these messages to $\mathcal{A}$. (The right-hand distribution denoted $\mathsf{IDEAL}^{\mathcal{B}}_{\zeta^{F,G},\mathcal{S}^{\Pi}(z),\mathcal{Z}}(\overline{x})$ represents the ideal world execution of

$\mathcal{B}$'s attack on Π, in which $\mathcal{B}$ must still simulate the functionality G for $\mathcal{A}$.)

Claim 7.5. *For all* $\overline{x} \in (\{0,1\}^{\mathsf{poly}})^n$ *and* $z \in \{0,1\}^{\mathsf{poly}}$

$$\mathsf{VIEW}_{\mathcal{A}}(\mathsf{IDEAL}_{\zeta^{F,G},\mathcal{S}(z),\mathcal{Z}}(\overline{x})) \equiv \mathsf{VIEW}_{\mathcal{A}}(\mathsf{IDEAL}^{\mathcal{B}}_{\zeta^{F,G},\mathcal{S}^{\Pi}(z),\mathcal{Z}}(\overline{x}))$$

Proof. The proof is analogous to the previous. However, in this case, $\mathcal{B}$ perfectly simulates the execution of ρ in comparison to $\mathcal{A}$'s view in the ideal execution of Π^{ρ}, since $\mathcal{B}$ does exactly the same thing that $\mathcal{S}$ does: both run $\mathcal{S}^{\rho}$. In light of this observation, the claim is mostly notational, since on the left $\mathcal{A}$ receives messages from $\mathcal{S}$, and on the right it receives the same messages, simply forwarded by $\mathcal{B}$ (and generated by $\mathcal{B}$ for the call to G). □

Note that $\mathcal{B}$ runs $\mathcal{A}$ and $\mathcal{S}^{\rho}$ as black boxes, so its depth is $d_a - d'_s + d'_s = d_a$. $\mathcal{E}$'s depth is at most $\mathsf{min}(d_e, d'_e)$ because it is identical to $\mathcal{Z}$.

If there exist $\overline{x}$, z for which $\mathcal{A}$,$\mathcal{Z}$ distinguish $\mathsf{REAL}_{\Gamma^{\Pi,\rho},\mathcal{A}(z),\mathcal{Z}}(\overline{x})$ and $\mathsf{IDEAL}_{\zeta^{F,G},\mathcal{S}(z),\mathcal{Z}}(\overline{x})$, then by Claims 7.4 and 7.5, $\mathcal{B}$ and $\mathcal{E}$ distinguish $\mathsf{REAL}^{\mathcal{B}}_{\Gamma^{\Pi,G},\mathcal{A}(z),\mathcal{Z}}(\overline{x})$ and $\mathsf{IDEAL}^{\mathcal{B}}_{\zeta^{F,G},\mathcal{S}^{\Pi}(z),\mathcal{Z}}(\overline{x})$. The latter two are exactly $\mathcal{B},\mathcal{E}$'s game against Π, except that we specified a strategy by which $\mathcal{B}$ simulates a concurrent execution of ρ which it feeds to $\mathcal{A}$ when it runs $\mathcal{A}$. Therefore, we have a contradiction to the security of Π, because $(\mathcal{B},\mathcal{E})$ are a (d_a, d_e) adversary and distinguisher for Π.

□

□

Chapter 8

Future Work

This dissertation provides a number of unanswered questions and opportunities to build on results that have been presented. Chapter 5 introduces a model that can be used to analyze a variety of chain structures used across a growing field of PoX protocols. It also provides a framework that can be used to model consensus protocols that do not use graphs, but still use PoX-type objects. Chapter 6 can be extended to reduce the number of encryptions per group update from $O(n)$ to $O(\log n)$, which is the goal of the MLS standard. Chapter 7 introduces a new falsifiable model for timed cryptographic primitives and opens new directions for multi-party computation in timed models, including accountable computing scenarios.

Bibliography

[1] I. Abraham, T. H. Chan, D. Dolev, K. Nayak, R. Pass, L. Ren, and E. Shi. Communication complexity of byzantine agreement, revisited. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019.

[2] I. Abraham, S. Devadas, D. Dolev, K. Nayak, and L. Ren. Synchronous byzantine agreement with expected o(1) rounds, expected o(n{ˆ2}) communication, and optimal resilience. In *International Conference on Financial Cryptography and Data Security*, 2019.

[3] I. Abraham, D. Dolev, I. Eyal, and J. Y. Halpern. Colordag: An incentive-compatible blockchain. Cryptology ePrint Archive, Paper 2022/308, 2022.

[4] I. Abraham, D. Dolev, A. Kagan, and G. Stern. Authenticated consensus in synchronous systems with mixed faults. Cryptology ePrint Archive, Paper 2022/805, 2022. https://eprint.iacr.org/2022/805.

[5] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin. Sync hotstuff: Simple and practical synchronous state machine replication. Cryptology ePrint Archive, Report 2019/270, 2019. https://eprint.iacr.org/2019/270.

[6] B. Altmann, M. Fitzi, and U. M. Maurer. Byzantine agreement secure against general adversaries in the dual failure model. In *DISC*, 1999.

[7] J. Alwen, B. Auerbach, M. C. Noval, K. Klein, G. Pascual-Perez, and K. Pietrzak. Decaf: Decentralizable continuous group key agreement with fast healing. Cryptology ePrint Archive, Paper 2022/559, 2022. https://eprint.iacr.org/2022/559.

[8] J. Alwen, B. Auerbach, M. C. Noval, K. Klein, G. Pascual-Perez, K. Pietrzak, and M. Walter. CoCoA: Concurrent continuous group key agreement. Cryptology ePrint Archive, Report 2022/251, 2022. https://eprint.iacr.org/2022/251.

[9] J. Alwen, S. Coretti, Y. Dodis, and Y. Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In D. Micciancio and T. Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 248–277. Springer, Heidelberg, Aug. 2020.

[10] J. Alwen, S. Coretti, D. Jost, and M. Mularczyk. Continuous group key agreement with active security. Cryptology ePrint Archive, Report 2020/752, 2020. https://eprint.iacr.org/2020/752.

[11] J. Alwen, D. Hartmann, E. Kiltz, and M. Mularczyk. Server-aided continuous group key agreement. In *CCS*. ACM, 2022.

[12] J. Alwen, D. Jost, and M. Mularczyk. On the insider security of MLS. *IACR Cryptol. ePrint Arch.*, 2020.

[13] M. Arapinis, N. Lamprou, and T. Zacharias. Astrolabous: A universally composable time-lock encryption scheme. Cryptology ePrint Archive, Report 2021/1246, 2021. https://ia.cr/2021/1246.

[14] C. Attiya, D. Dolev, and J. Gil. Asynchronous byzantine consensus. In *PODC*. ACM, 1984.

[15] M. Backes and C. Cachin. Reliable broadcast in a computational hybrid model with byzantine faults, crashes, and recoveries. In *DSN*. IEEE Computer Society, 2003.

[16] C. Badertscher, P. Gazi, A. Kiayias, A. Russell, and V. Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In *ACM Conference on Computer and Communications Security*, 2018.

[17] C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas. Bitcoin as a transaction ledger: A composable treatment. In *CRYPTO*. Springer, 2017.

[18] M. Ball, A. Rosen, M. Sabin, and P. N. Vasudevan. Average-case fine-grained hardness. In *STOC*, 2017.

[19] M. Ball, A. Rosen, M. Sabin, and P. N. Vasudevan. Proofs of work from worst-case assumptions. In *CRYPTO (1)*, 2018.

[20] R. Barnes, B. Beurdouche, J. Millican, E. Omara, K. Cohn-Gordon, and R. Robert. The Messaging Layer Security (MLS) Protocol. Internet-Draft draft-ietf-mls-protocol-11, Internet Engineering Task Force, Dec. 2020. Work in Progress.

[21] C. Baum, B. David, R. Dowsley, J. B. Nielsen, and S. Oechsner. Craft: Composable randomness beacons and output-independent abort mpc from time. Cryptology ePrint Archive, Report 2020/784, 2020. https://eprint.iacr.org/2020/784.

[22] C. Baum, B. David, R. Dowsley, J. B. Nielsen, and S. Oechsner. TARDIS: A foundation of time-lock puzzles in UC. In *EUROCRYPT (3)*, 2021.

[23] M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, 1983.

[24] I. Bentov, P. Hubácek, T. Moran, and A. Nadler. Tortoise and hares consensus: the meshcash framework for incentive-compatible, scalable cryptocurrencies. *IACR Cryptology ePrint Archive*, 2017.

[25] I. Bentov, R. Pass, and E. Shi. The sleepy model of consensus. *IACR Cryptology ePrint Archive*, 2016.

[26] A. Bienstock, Y. Dodis, and P. Rösler. On the price of concurrency in group ratcheting protocols. In R. Pass and K. Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 198–228. Springer, Heidelberg, Nov. 2020.

[27] N. Bitansky, S. Goldwasser, A. Jain, O. Paneth, V. Vaikuntanathan, and B. Waters. Time-lock puzzles from randomized encodings. In *ITCS-2016*, 2016.

[28] L. Blum, M. Blum, and M. Shub. Comparison of two pseudo-random number generators. In *Crypto82*, 1982.

[29] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. Verifiable delay functions. In *CRYPTO*, 2018.

[30] D. Boneh, B. Bünz, and B. Fisch. A survey of two verifiable delay functions. Cryptology ePrint Archive, Report 2018/712, 2018. https://eprint.iacr.org/2018/712.

[31] D. Boneh and M. Naor. Timed commitments. In *Crypto'00*, 2000.

[32] C. Boyd, A. Mathuria, and D. Stebila. *Protocols for Authentication and Key Establishment*. Information Security and Cryptography. Springer Berlin Heidelberg, 2020.

[33] E. Bresson, O. Chevassut, and D. Pointcheval. Provably authenticated group Diffie-Hellman key exchange – the dynamic case. In C. Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 290–309. Springer, Heidelberg, Dec. 2001.

[34] E. Bresson, O. Chevassut, and D. Pointcheval. Dynamic group Diffie-Hellman key exchange under standard assumptions. In L. R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 321–336. Springer, Heidelberg, Apr. / May 2002.

[35] E. Bresson, O. Chevassut, D. Pointcheval, and J.-J. Quisquater. Provably authenticated group Diffie-Hellman key exchange. In M. K. Reiter and P. Samarati, editors, *ACM CCS 2001*, pages 255–264. ACM Press, Nov. 2001.

[36] E. Bresson and M. Manulis. Securing group key exchange against strong corruptions. In M. Abe and V. Gligor, editors, *ASIACCS 08*, pages 249–260. ACM Press, Mar. 2008.

[37] E. Bresson, M. Manulis, and J. Schwenk. On security models and compilers for group key exchange protocols. In A. Miyaji, H. Kikuchi, and K. Rannenberg, editors, *IWSEC 07*, volume 4752 of *LNCS*, pages 292–307. Springer, Heidelberg, Oct. 2007.

[38] C. Brzuska. *On the foundations of key exchange*. PhD thesis, Darmstadt University of Technology, Germany, 2013.

[39] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000.

[40] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, 1999.

[41] T. H. Chan, R. Pass, and E. Shi. Round complexity of byzantine agreement, revisited. *IACR Cryptology ePrint Archive*, 2019.

[42] L. Chen, L. Xu, N. Shah, Z. Gao, Y. Lu, and W. Shi. On security analysis of proof-of-elapsed-time. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, 2017.

[43] P. Chvojka, T. Jager, D. Slamanig, and C. Striecks. Versatile and sustainable timed-release encryption and sequential time-lock puzzles. Cryptology ePrint Archive, Report 2020/739, 2020. https://ia.cr/2020/739.

[44] R. Cohen, J. A. Garay, and V. Zikas. Completeness theorems for adaptively secure broadcast. In *CRYPTO (1)*, volume 14081 of *Lecture Notes in Computer Science*, pages 3–38. Springer, 2023.

[45] K. Cohn-Gordon. *On secure messaging*. PhD thesis, University of Oxford, UK, 2018.

[46] K. Cohn-Gordon, C. Cremers, L. Garratt, J. Millican, and K. Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *ACM CCS 2018*, pages 1802–1819. ACM Press, Oct. 2018.

[47] K. Cong, K. Eldefrawy, N. P. Smart, and B. Terner. The key lattice framework for concurrent group messaging. *IACR Cryptol. ePrint Arch.*, 2022.

[48] R. Cramer and V. Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, 2003.

[49] C. Cremers, B. Hale, and K. Kohbrok. The complexities of healing in secure group messaging: Why cross-group effects matter. In M. Bailey and R. Greenstadt, editors, *USENIX Security 2021*, pages 1847–1864. USENIX Association, Aug. 2021.

[50] P. Daian, R. Pass, and E. Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In *Financial Cryptography*, 2019.

[51] B. David, P. Gazi, A. Kiayias, and A. Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *EUROCRYPT (2)*, 2018.

[52] C. Decker and R. Wattenhofer. Information propagation in the bitcoin network. In *P2P*. IEEE, 2013.

[53] A. Degwekar, V. Vaikuntanathan, and P. N. Vasudevan. Fine-grained cryptography. In *CRYPTO (3)*, 2016.

[54] D. Dolev, C. Dwork, and L. J. Stockmeyer. On the minimal synchronism needed for distributed consensus. In *FOCS*. IEEE Computer Society, 1983.

[55] D. Dolev and H. R. Strong. Authenticated algorithms for byzantine agreement. 1983.

[56] J. R. Douceur. The sybil attack. In *IPTPS*, 2002.

[57] B. Dowling, F. Günther, and A. Poirrier. Continuous authentication in secure messaging. In V. Atluri, R. Di Pietro, C. D. Jensen, and W. Meng, editors, *ESORICS 2022 , Part II*, volume 13555 of *LNCS*, pages 361–381. Springer, Heidelberg, Sept. 2022.

[58] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 1988.

[59] L. Eckey, S. Faust, and J. Loss. Efficient algorithms for broadcast and consensus based on proofs of work. *IACR Cryptology ePrint Archive*, 2017.

[60] W. Eddy. Transmission Control Protocol (TCP). RFC 9293, Aug. 2022.

[61] S. Egashira, Y. Wang, and K. Tanaka. Fine-grained cryptography revisited. In *ASIACRYPT (3)*, 2019.

[62] K. Eldefrawy, S. Jakkamsetti, B. Terner, and M. Yung. Standard model time-lock puzzles: Defining security and constructing via composition. Cryptology ePrint Archive, Paper 2023/439, 2023. https://eprint.iacr.org/2023/439.

[63] K. Eldefrawy, J. Loss, and B. Terner. How byzantine is a send corruption? In *ACNS*, 2022.

[64] etherchain.org. The ethereum blockchain explorer, 2020.

[65] L. Fan, J. Katz, P. Thai, and H.-S. Zhou. A permissionless proof-of-stake blockchain with best-possible unpredictability. Cryptology ePrint Archive, Report 2021/660, 2021. https://eprint.iacr.org/2021/660.

[66] P. Feldman and S. Micali. An optimal probabilistic protocol for synchronous byzantine agreement. *SIAM J. Comput.*, 1997.

[67] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 1985.

[68] M. Fitzi. *Generalized Communication and Security Models in Byzantine Agreement*. PhD thesis, ETH Zurich, 3 2003. Reprint as vol. 4 of ETH Series in Information Security and Cryptography.

[69] C. Freitag, I. Komargodski, R. Pass, and N. Sirkin. Non-malleable time-lock puzzles and applications. Cryptology ePrint Archive, Report 2020/779, 2020. https://eprint.iacr.org/2020/779.

[70] J. A. Garay, J. Katz, C. Koo, and R. Ostrovsky. Round complexity of authenticated broadcast with a dishonest majority. In *FOCS*. IEEE Computer Society, 2007.

[71] J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT*, 2015.

[72] J. A. Garay and K. J. Perry. A continuum of failure models for distributed computing. In *WDAG*, Lecture Notes in Computer Science, 1992.

[73] C. Gentry, S. Halevi, H. Krawczyk, B. Magri, J. B. Nielsen, T. Rabin, and S. Yakoubov. YOSO: you only speak once - secure MPC with stateless ephemeral roles. In *CRYPTO*, 2021.

[74] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *SOSP*, pages 51–68. ACM, 2017.

[75] O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. 2009.

[76] Y. Guo, R. Pass, and E. Shi. Synchronous, with a chance of partition tolerance. Cryptology ePrint Archive, Report 2019/179, 2019. https://eprint.iacr.org/2019/179.

[77] K. Hashimoto, S. Katsumata, E. Postlethwaite, T. Prest, and B. Westerbaan. A concrete treatment of efficient continuous group key agreement via multi-recipient PKEs. In G. Vigna and E. Shi, editors, *ACM CCS 2021*, pages 1441–1462. ACM Press, Nov. 2021.

[78] J. Håstad, R. Impagliazzo, L. A. Levin, and M. Luby. A pseudorandom generator from any one-way function. *SIAM J. Comput.*, 1999.

[79] I. Ingemarsson, D. T. Tang, and C. K. Wong. A conference key distribution system. *IEEE Trans. Inf. Theory*, 1982.

[80] J. Katz and C. Koo. On expected constant-round protocols for byzantine agreement. In *CRYPTO*, 2006.

[81] J. Katz, J. Loss, and J. Xu. On the security of time-lock puzzles and timed commitments. In *TCC (3)*, volume 12552 of *LNCS*, pages 390–413. Springer, 2020.

[82] J. Katz and M. Yung. Scalable protocols for authenticated group key exchange. In D. Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 110–125. Springer, Heidelberg, Aug. 2003.

[83] Y. Kim, A. Perrig, and G. Tsudik. Group key agreement efficient in communication. *IEEE Transactions on Computers*, 2004.

[84] K. Kursawe. Distributed protocols on general hybrid adversary structures. 2004.

[85] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 1978.

[86] R. LaVigne, A. Lincoln, and V. V. Williams. Public-key cryptography in the fine-grained setting. In *CRYPTO*, 2019.

[87] A. Lewis-Pye and T. Roughgarden. A general framework for the security analysis of blockchain protocols. *CoRR*, 2020.

[88] B. Libert, M. Joye, and M. Yung. Born and raised distributively: fully distributed non-interactive adaptively-secure threshold signatures with short shares. In *PODC*. ACM, 2014.

[89] Y. Lindell. How to simulate it - A tutorial on the simulation proof technique. In *Tutorials on the Foundations of Cryptography*. 2017.

[90] J. Loss and G. Stern. Zombies and ghosts: Optimal byzantine agreement in the presence of omission faults. Cryptology ePrint Archive, Paper 2023/954, 2023. https://eprint.iacr.org/2023/954.

[91] M. Mahmoody, T. Moran, and S. P. Vadhan. Time-lock puzzles in the random oracle model. In *CRYPTO*, 2011.

[92] G. Malavolta and S. A. K. Thyagarajan. Homomorphic time-lock puzzles and applications. In *CRYPTO (1)*, 2019.

[93] D. Malkhi, K. Nayak, and L. Ren. Flexible byzantine fault tolerance. *CoRR*, 2019.

[94] S. Micali. Byzantine agreement , made trivial. 2017.

[95] S. Micali, M. O. Rabin, and S. P. Vadhan. Verifiable random functions. In *FOCS*. IEEE Computer Society, 1999.

[96] S. Micali, S. Vadhan, and M. Rabin. Verifiable random functions. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS 99. IEEE Computer Society, 1999.

[97] P. Milgrom. Putting auction theory to work: The simultaneous ascending auction. *Journal of political economy*, 2000.

[98] A. Miller, A. Juels, E. Shi, B. Parno, and J. Katz. Permacoin: Repurposing bitcoin work for data preservation. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.

[99] A. Miller, A. Kosba, J. Katz, and E. Shi. Nonoutsourceable scratch-off puzzles to discourage bitcoin mining coalitions. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[100] A. Miller and J. J. LaViola Jr. Anonymous byzantine consensus from moderately-hard puzzles: A model for bitcoin. *Available on line: http://nakamotoinstitute.org/research/anonymous-byzantine-consensus*, 2014.

[101] T. Moran and I. Orlov. Simple proofs of space-time and rational proofs of storage. Cryptology ePrint Archive, Report 2016/035, 2016. https://eprint.iacr.org/2016/035.

[102] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[103] P. R. Parvédy and M. Raynal. Uniform agreement despite process omission failures. In *IPDPS*, page 212. IEEE Computer Society, 2003.

[104] R. Pass, L. Seeman, and a. shelat. Analysis of the blockchain protocol in asynchronous networks. *IACR Cryptology ePrint Archive*, 2016.

[105] R. Pass and E. Shi. Rethinking large-scale consensus. In *Computer Security Foundations Symposium (CSF), 2017 IEEE 30th*, 2017.

[106] R. Pass and E. Shi. The sleepy model of consensus. In *ASIACRYPT (2)*, 2017.

[107] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 1980.

[108] K. J. Perry and S. Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Trans. Softw. Eng.*, 12(3):477–482, mar 1986.

[109] K. Pietrzak. Simple verifiable delay functions. In *ITCS*, 2019.

[110] J. Pijnenburg and B. Poettering. On secure ratcheting with immediate decryption. In *ASIACRYPT*, volume 13793 of *Lecture Notes in Computer Science*, pages 89–118. Springer, 2022.

[111] B. Poettering, P. Rösler, J. Schwenk, and D. Stebila. SoK: Game-based security models for group key exchange. In K. G. Paterson, editor, *CT-RSA 2021*, volume 12704 of *LNCS*, pages 148–176. Springer, Heidelberg, May 2021.

[112] M. Raynal. Consensus in synchronous systems: A concise guided tour. In *2002 Pacific Rim International Symposium on Dependable Computing, 2002. Proceedings.*, pages 221–228. IEEE, 2002.

[113] E. Rescorla. Subject: [MLS] TreeKEM: An alternative to ART. MLS Mailing List, 2019. https://mailarchive.ietf.org/arch/msg/mls/e3ZKNzPC7Gxrm3Wf0q96dsLZoD8/, Accessed 2022-01-19.

[114] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, 1996.

[115] T. Rocket, M. Yin, K. Sekniqi, R. van Renesse, and E. G. Sirer. Scalable and probabilistic leaderless BFT consensus through metastability. *CoRR*, 2019.

[116] P. Rösler, C. Mainka, and J. Schwenk. More is less: On the end-to-end security of group chats in signal, whatsapp, and threema. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018.* IEEE, 2018.

[117] L. Rotem and G. Segev. Generically speeding-up repeated squaring is equivalent to factoring: Sharp thresholds for all generic-ring delay functions. In *CRYPTO (3)*, 2020.

[118] sawtooth.hyperledger.org. Hyperledger sawtooth poet 1.0 specification, 2020.

[119] Y. Sompolinsky, Y. Lewenberg, and A. Zohar. Spectre: A fast and scalable cryptocurrency protocol. *IACR Cryptology ePrint Archive*, 2016.

[120] Y. Sompolinsky and A. Zohar. PHANTOM: A scalable blockdag protocol. *IACR Cryptology ePrint Archive*, 2018.

[121] M. Steiner, G. Tsudik, and M. Waidner. Diffie-Hellman key distribution extended to group communication. In L. Gong and J. Stern, editors, *ACM CCS 96*, pages 31–37. ACM Press, Mar. 1996.

[122] B. Terner. Permissionless consensus in the resource model. In *Financial Cryptography*, 2022.

[123] S. P. Vadhan and C. J. Zheng. Characterizing pseudoentropy and simplifying pseudorandom generator constructions. In *STOC*. ACM, 2012.

[124] A. van Baarsen and M. Stevens. On time-lock cryptographic assumptions in abelian hidden-order groups. In *ASIACRYPT (2)*, Lecture Notes in Computer Science, 2021.

[125] J. Wan, H. Xiao, S. Devadas, and E. Shi. Round-efficient byzantine broadcast under strongly adaptive and majority corruptions. In *TCC*, 2020.

[126] J. Wan, H. Xiao, E. Shi, and S. Devadas. Expected constant round byzantine broadcast under dishonest majority. In *TCC*, 2020.

[127] M. Weidner, M. Kleppmann, D. Hugenroth, and A. R. Beresford. Key agreement for decentralized secure group messaging with strong security guarantees. In G. Vigna and E. Shi, editors, *ACM CCS 2021*, pages 2024–2045. ACM Press, Nov. 2021.

[128] M. A. Weidner. Group messaging for secure asynchronous collaboration. M.phil thesis, University of Cambridge, 6 2019. https://mattweidner.com/acs-dissertation.pdf.

[129] B. Wesolowski. Efficient verifiable delay functions. In *EUROCRYPT*, 2019.

[130] WhatsApp Inc. Whatsapp encryption overview. Online, Sep 2021. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf, Accessed 2022-01-19.

[131] V. Zikas, S. Hauser, and U. M. Maurer. Realistic failures in secure multi-party computation. In *TCC*, 2009.

Appendix A

Encryption Definitions

A.1 CCA Secure Encryption Scheme

Definition A.1 (Symmetric Key Encryption Scheme). *A symmetric key encryption scheme consists of three algorithms:*

- $\mathsf{KeyGen}(1^\lambda)$*: Output a symmetric key with security parameter* λ*.*
- $\mathsf{Enc}(m; k)$*: On plaintext input* m*, output a ciphertext* c *encrypted under the symmetric key* k*.*
- $\mathsf{Dec}(c; k)$*: Decrypt the ciphertext input* c *using* k *and output the plaintext* m *if successful, otherwise output* $\perp$*.*

Definition A.2 (Symmetric Encryption Scheme IND-CCA Security). *The security of an IND-CCA symmetric encryption scheme is defined by a game between a challenger and an adversary* $\mathcal{A}$ *as follows:*

1. *Challenger samples a symmetric key* $k \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$*.*

2. The adversary $\mathcal{A}$ outputs two messages m_0, m_1.

3. The challenger selects $b \overset{\$}{\leftarrow} \{0,1\}$ and computes $c^* \leftarrow \mathsf{Enc}(m_b; k)$.

4. The challenger sends c^* to $\mathcal{A}$.

5. $\mathcal{A}$ outputs b'.

The adversary has access to an encryption oracle and a decryption oracle. On input x, the former outputs $\mathsf{Enc}(x;k)$ and the latter outputs $\mathsf{Dec}(x;k)$. After the adversary learns c^*, it is not allowed to query the decryption oracle on c^*. The advantage of the adversary is

$$2 \cdot |\Pr[b = b'] - 1/2|.$$

We say an encryption scheme described in Definition A.1 is secure if, for any polynomial-time adversary $\mathcal{A}$, the advantage of the game above is negligible in the security parameter λ.

A.1.1 Message Authentication Code (MAC)

Definition A.3 (MAC)**.** *A MAC consists of three algorithms*

- $k \leftarrow \mathsf{MAC.KeyGen}(1^\lambda)$,
- $t \leftarrow \mathsf{MAC}(m;k)$, *and*
- $b \leftarrow \mathsf{MAC.Verify}(m,t;k)$.

For correctness we require for every λ, every key k and every $m \in \{0,1\}^*$ it holds that $\mathsf{MAC.Verify}(m, \mathsf{MAC}(m;k);k) = 1$.

Definition A.4 (MAC EUF-CMA Security)**.** *The security of a MAC is modelled using the existentially unforgeable under an adaptive chosen-message attack (EUF-CMA) game between challenger $\mathcal{C}$ and adversary $\mathcal{A}$.*

- *$\mathcal{C}$ generates $k \leftarrow \mathsf{MAC.KeyGen}(1^\lambda)$.*
- *$\mathcal{A}$ is allowed to query the MAC oracle (i.e., $\mathsf{MAC}(m;k)$) on for any message m of his choice. All the queried messages are stored in a table T. Additionally, $\mathcal{A}$ is also allowed the verification oracle $\mathsf{MAC.Verify}(m,t;k)$ on his input (m,t).*
- *Eventually, $\mathcal{A}$ outputs (m^*,t^*) to $\mathcal{C}$.*
- *$\mathcal{A}$ wins the game if $\mathsf{MAC.Verify}(m^*,t^*;k) = 1$ and $m^* \notin T$.*

The advantage of the adversary is given as

$$\mathsf{Adv}^{\mathsf{mac}}_{\mathcal{A}} = \Pr[\mathcal{A} \textit{ wins EUF-CMA}].$$

The scheme is secure if, for any polynomial-time adversary $\mathcal{A}$, the advantage is negligible in the security parameter λ.

A.2 Key Encapsulation Mechanism (KEM)

Definition A.5 (KEM)**.** *A KEM consists of three algorithms*

- $(\mathtt{pk}, \mathtt{sk}) \leftarrow \mathsf{KEM.KeyGen}(1^\lambda)$*,*
- $(c,k) \leftarrow \mathsf{KEM.Encap}(\mathtt{pk})$*, and*
- $k \leftarrow \mathsf{KEM.Decap}(c;\mathtt{sk})$*.*

For correctness we require if $(c, k) \leftarrow \mathsf{KEM.Encap}(\mathtt{pk})$ *then* $k = \mathsf{KEM.Decap}(c; \mathtt{sk})$ *for all* $(\mathtt{pk}, \mathtt{sk}) \leftarrow \mathsf{KEM.KeyGen}(1^\lambda)$.

Definition A.6 (KEM IND-CCA Security)**.** *The security of KEM is modelled using a game between challenger* $\mathcal{C}$ *and adversary* $\mathcal{A}$.

- $\mathcal{C}$ *generates* $(\mathtt{pk}, \mathtt{sk}) \leftarrow \mathsf{KEM.KeyGen}(1^\lambda)$,
- $\mathcal{C}$ *generates a random key* k_0 *from the symmetric key space.*
- $\mathcal{C}$ *runs encapsulation algorithm* $(c^*, k_1) \leftarrow \mathsf{KEM.Encap}(\mathtt{pk})$
- $\mathcal{C}$ *samples* $b \xleftarrow{\$} \{0, 1\}$ *and outputs* (c^*, k_b) *to* $\mathcal{A}$.
- *Finally* $\mathcal{A}$ *outputs a bit* b'.

During the game, $\mathcal{A}$ *is allowed to query the decapsulation oracle* $\mathsf{KEM.Decap}(c; \mathtt{sk})$ *on any* c *that is not* c^*. *The advantage of the adversary is given as*

$$\mathsf{Adv}^{\mathsf{kem}}_{\mathcal{A}} = 2 \cdot |\Pr[b = b'] - 1/2|.$$

The scheme is secure if, for any polynomial-time adversary $\mathcal{A}$, *the advantage is negligible in the security parameter* λ.

A.3 Authenticated Encryption with Associated Data (AEAD)

Definition A.7 (AEAD)**.** *An AEAD scheme consists of three algorithms:*

- $k \leftarrow \mathsf{AEAD.KeyGen}(1^\lambda)$,

– $(c, t) \leftarrow \mathsf{AEAD.Enc}(m, d; k)$, *and*

– $\{m, \perp\} \leftarrow \mathsf{AEAD.Dec}(c, d, t; k)$.

For correctness, we require that if $(c, t) \leftarrow \mathsf{AEAD.Enc}(m, d; k)$ *then we will obtain* $m = \mathsf{AEAD.Dec}(c,\ d, t; k)$ *for all* $(\mathtt{pk}, \mathtt{sk}) \leftarrow \mathsf{AEAD.KeyGen}(1^\lambda)$*,* $m \leftarrow \{0,1\}^*$ *and* $d \leftarrow \{0,1\}^*$ *where* m *is the message and* d *is the associated data.*

Definition A.8 (AEAD IND-CCA Security)**.** *The security of AEAD is described using a game between a challenger* $\mathcal{C}$ *and an adversary* $\mathcal{A}$*.*

1. $\mathcal{C}$ *generates* $k \leftarrow \mathsf{AEAD.KeyGen}(1^\lambda)$.
2. $\mathcal{C}$ *samples* $b \overset{\$}{\leftarrow} \{0,1\}$.
3. $\mathcal{A}$ *calls the test query* $\mathsf{Test}((m_0, d_0), (m_1, d_1))$.
4. *The challenger returns* $(c^*, t^*) \leftarrow \mathsf{AEAD.Enc}(m_b, d_b; k)$.
5. $\mathcal{A}$ *outputs a bit* b'.

$\mathcal{A}$ *is allowed to query the encryption oracle* $\mathsf{AEAD.Enc}(m, d; k)$ *for any* (m, d) *and the decryption oracle* $\mathsf{AEAD.Dec}(c, d, t; k)$ *for any* c, d, t *except when* $c = c^*$ *or* $t = t^*$*. The advantage of the adversary is given as*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{aead}} = 2 \cdot |\Pr[b = b'] - 1/2|.$$

The scheme is secure if, for any polynomial-time adversary $\mathcal{A}$*, the advantage is negligible in the security parameter* λ*.*

A.4 Public Key Authenticated Encryption with Associated Data (PKAEAD)

Definition A.9 (PKAEAD)**.** *A PKAEAD scheme consists of the following algorithms:*

- $(\texttt{pk}, \texttt{sk}) \leftarrow \mathsf{PKAEAD.KeyGen}(1^\lambda)$: *generate the key pair.*
- $(c, t) \leftarrow \mathsf{PKAEAD.Enc}(m, d; \texttt{pk})$: *encrypt the plaintext m and authenticate the associated data d under the public key* $\texttt{pk}$*. The ciphertext c and the authentication tag t is returned.*
- $\{m, \perp\} \leftarrow \mathsf{PKAEAD.Dec}(c, d, t; \texttt{sk})$: *decrypt the ciphertext c using* $\texttt{sk}$ *and then return the plaintext m. This procedure fails if t is not a valid authentication tag for c or d.*

For correctness we require that if $(c, t) \leftarrow \mathsf{PKAEAD.Enc}(m, d; \texttt{pk})$ *then, for all* $m \leftarrow \{0,1\}^*$, $d \leftarrow \{0,1\}^*$, *and* $(\texttt{pk}, \texttt{sk}) \leftarrow \mathsf{PKAEAD.KeyGen}(1^\lambda)$, *we will obtain* $m = \mathsf{PKAEAD.Dec}(c, d, t; \texttt{sk})$.

Definition A.10 (PKAEAD IND-CCA Security)**.** *Here we describe a typical IND-CCA security adapted to PKAEAD.*

1. *The challenger generates* $(\texttt{pk}, \texttt{sk}) \leftarrow \mathsf{PKAEAD.KeyGen}(1^\lambda)$ *and sends* $\texttt{pk}$ *to the adversary.*
2. *The challenger samples* $b \xleftarrow{\$} \{0, 1\}$.
3. $\mathcal{A}$ *calls the test query* $\mathsf{Test}((m_0, d_0), (m_1, d_1))$.
4. *The challenger returns* $\mathsf{PKAEAD.Enc}(m_b, d_b; \texttt{pk})$.
5. $\mathcal{A}$ *outputs a bit* b'.

The adversary is allowed to query the the decryption oracle $\mathsf{PKAEAD.Dec}(c, d, t; \mathtt{sk})$ *for any* c, d, t *except when* $(c = c^*$ *or* $t = t^*)$*. The advantage of the adversary is given as*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{pkaead}} = 2 \cdot |\Pr[b = b'] - 1/2|.$$

The scheme is secure if, for any polynomial-time adversary $\mathcal{A}$*, the advantage is negligible in the security parameter* λ*.*

Below we instantiate PKAEAD scheme using a KEM and a symmetric key AEAD.

- $\mathsf{PKAEAD.KeyGen}(1^\lambda)$: return $\mathsf{KEM.KeyGen}(1^\lambda)$.
- $\mathsf{PKAEAD.Enc}(m, d; \mathtt{pk})$: $(k, e) \leftarrow \mathsf{KEM.Encap}(\mathtt{pk})$, $(c, t) \leftarrow \mathsf{AEAD.Enc}(m, d; k)$, return $((e, c), t)$.
- $\mathsf{PKAEAD.Dec}((e, c), d, t; \mathtt{sk})$: $k \leftarrow \mathsf{KEM.Decap}(\mathtt{sk}; e)$, return the plaintext via $\mathsf{AEAD.Dec}(c, d, t; k)$.

The security of this construction can be proven in a similar way as hybrid ciphers [48]. We sketch the proof below. In G_0, $\mathcal{A}$ plays the standard PKAEAD IND-CCA game. In G_1, we modify the test query to use a different symmetric key to encrypt and authenticate the message in the AEAD component. This allows us to build an adversary $\mathcal{A}_1$ for the KEM game by asking its challenger for a key which might or might not be the key that corresponds to the encapsulation. Additionally, we can build an adversary $\mathcal{A}_2$ that plays the AEAD game. Namely, $\mathcal{A}_2$ forwards $((m_0, d_0), (m_1, d_1))$ to the AEAD challenger and creates a "fake" KEM encapsulation for $\mathcal{A}$. The security of PKAEAD holds from the security of AEAD and KEM.

Appendix B

Dolev and Strong's Impossibility

In this section we provide an exposition of Dolev and Strong's lower-bound on the round complexity of a deterministic broadcast protocol. *We highlight the fact that the proof requires only send-corruptions and not fully byzantine corruptions, and therefore the impossibility result holds for only send-corruptions.*

Recall that the result by Dolev and Strong proves there is no deterministic broadcast protocol tolerating b byzantine parties that terminates in at most b rounds. The proof proceeds by assuming such a protocol and considering a "good" execution in which all messages of the protocol are delivered in every round. It then proceeds to define a series of "neighboring" executions such that every two neighboring executions are *identical* except that in one of the two executions, there is one exactly one round in which a message sent by one party is dropped. The transition between neighboring executions maintains two invariants:

1. In every pair of neighboring executions A and B, there is some honest party q such that the view of q is identical in A and B. (This means that q receives exactly the same messages in the two neighboring executions.)

2. In no execution are more than b parties corrupted.

Because of invariant 1, we require that in every execution in the sequence, all honest parties output the same value. This follows from the fact for every pair of neighboring executions, the party q whose view is identical in the two executions must output the same value in both, and all other parties must output q's value in both executions by consistency. Contradiction follows by arriving at an execution in which the dealer sends no messages; therefore, the output of every party must be independent of the dealer's value.

In order to define a series of executions that satisfy the above properties, the proof defines a "communication graph" that describes all messages sent in an execution. A communication graph is a directed acyclic graph (DAG) which is divided into "levels" such that in each level, every party is represented by a distinct vertex. If in some execution, party A sends a message to party B in round r that B receives round $r+1$, then there is an edge from A's vertex in level r to B's vertex in level $r+1$. (We assume synchronous communication in which all messages sent in round r are always delivered in round $r+1$.)

The proof begins with the full execution graph and defines a recursive procedure by which all messages sent by a party in round r and greater can be removed from the graph, while maintaining the properties required above. Let R be the final round in an execution, and consider two graphs such that the only difference is that a message from party A to party B sent in round $R-1$ and received in R is removed from one of the two graphs. Clearly, all parties except for B have views that are identical between the two executions. Next, consider two graphs such that the only difference between the two is that a message from party A to party B sent in round $r < R-1$ and received in $r+1$ is removed from one of the two graphs, but for which B sends no messages in any round $r' \geq r+1$. Again, all parties except for B have the same view of the execution in the two graphs. Similarly, the reverse operations also preserve invariant 1. Namely, an edge can be "restored" from A to

B in either of the two above scenarios.

The proof shows how to remove all edges from the sender while maintaining the above invariants. In order to remove all messages from a party P starting at round r: For every party Q that receive a message that P sends in round r and are delivered in round $r+1$, remove all future edges sent by Q starting in round $r+1$. Then remove the edge from P to Q sent in round r and delivered in round $r+1$. Then restore all edges sent by Q starting in round $r+1$. The proof guarantees that in any execution graph, at most one party is corrupted *per round of the protocol in which messages are dropped*, which maintains that if the protocol requires R rounds, then at most R parties need to be corrupted. The contradiction follows for any protocol requiring fewer than $t_{\mathsf{byz}}+1$ rounds – or in our case, $t_{\mathsf{snd}}+1$ rounds.

For a full treatment, we refer the reader to the very thorough explanation by [41], complete with diagrams.

Appendix C

Proof of Graph Consensus Protocol Π^G

We present the proof of Theorem 5.3, which proves graph consistency and liveness for our graph consensus protocol Π^G.

Theorem 5.3. *For all N, all ρ, and all ε, and for all $\alpha > \rho(1-\alpha)((3-\alpha)\rho+\frac{\varepsilon}{\alpha}+\frac{\varepsilon}{\rho}+\varepsilon+1)$ every (α,ε)-honest, ρ-rate-limited, admissible execution of $\Pi^G(\alpha,\varepsilon,\rho)$ satisfies graph consistency and f,h-liveness for $f(N) = h(N) = \alpha N - \varepsilon - \rho(\ell^* + 1)$, where ℓ^* is a derived constant defined as in the protocol.*

Most of our effort towards proving Theorem 5.3 is focused on the proof of Proposition 5.1, which we restate here.

Proposition 5.1. *Let $c = (3-\alpha)\rho + \frac{\varepsilon}{\alpha} + \frac{\varepsilon}{\rho} + \varepsilon + 1$ (as in Protocol Π^G). If $\alpha > \rho\beta c$, then for all k, times t and t', and honest participants p and q active at t and t', respectively, if $\mathsf{D}(G_p^{(t)}) > k + \ell^*$ and $\mathsf{D}(G_q^{(t')}) > k + \ell^*$, then $\mathsf{extract}(G_p^{(t)})|_k = \mathsf{extract}(G_q^{(t')})|_k$.*

Consistency will follow directly, and liveness will follow easily from the techniques we use for Proposition 5.1. We begin to prove Proposition 5.1 by showing consistency of the honest vertices that honest participants output. Recall for the duration of the proof that we require $\alpha > \rho\beta c$, where β and c are defined as in the protocol specification; particularly, $\beta = 1 - \alpha$ is the long-term proportion of corrupt resources and c is a derived constant.

Recall that in Section 5.5.4, we presented an overview of the proof of Proposition 1. We reproduce these lemmas here and then present their proofs.

Lemma 5.1 proves that the honest vertices in honest participants' extracted graphs are consistent.

Lemma 5.1 (Honest Vertex Extraction). *For every time t, honest participant p active at t, and honest vertex $v \in \mathbb{G}^{(t)}$: $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) > \ell_1 \implies v \in \mathsf{extract}(G_p^{(t)})$.*

We prove it by decomposition into Lemmas 5.2 and 5.3

Lemma 5.2 (Depth-Based Indicator for Honest Vertices). *For all t, honest p active at t, and honest vertex $v \in \mathbb{G}^{(t)}$: $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) > \ell_1 \implies v \in G_p^{(t)}$.*

Lemma 5.3 (Extracting All Honest Vertices in a Local Graph). *For every time t, honest participant p active at t, and honest vertex $v \in \mathbb{G}^{(t)}$: $v \in G_p^{(t)} \implies v \in \mathsf{extract}(G_p^{(t)})$.*

We then prove consistency of corrupt vertices by proving Lemma 5.4.

Lemma 5.4 (Honest Reachability Requirement for Extraction). *For all t, participant p active at t, and vertex $v \in \mathsf{extract}(G_p^{(t)})$: $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) > \ell_2$ implies there exists an honest vertex u reachable from v such that $\mathsf{D}(u) - \mathsf{D}(v) \leq \ell_2$.*

In section C.2 we prove Lemma 5.2. In section C.3 we prove Lemma 5.3 and complete the proof of Lemma 5.1. In Section C.4 we prove Lemma 5.4. Finally, in Section C.5 we conclude the proofs of Proposition 5.1 and Theorem 5.3.

C.1 Properties of an Execution

Before presenting the proof, we first observe a number of useful properties of an execution.

Constrained Vertex Generation A participant can generate a vertex *only* when it is allocated a resource. It immediately follows that the set of all vertices that have been generated in an execution at some point in time is the set of resources that have been allocated in the execution up to that point in time. Furthermore, constraints on the rate at which vertices are generated and the proportion of honest vertices in an execution inherit directly from the respective constraints on resource allocation. Specifically,

- the rate at which vertices are generated in an execution is also upperbounded by ρ vertices per Δ time (Definition 5.4), and
- the proportion of vertices generated by honest participants is the proportion of honest resources allocated in an α, ε-honest execution (Definition 5.6)

Consistency Properties of Local Graphs We say that a graph G is *completely described* if for every vertex v in G, every one of v's predecessors is also in G. The protocol specification enforces the invariant that every honest participant's local graph is always completely described. Recall that each participant's graph is initialized to a graph with only the root vertex, and that participants' local graphs grow when they generate vertices and when they receive messages. No participant's local graph can become incompletely described when it generates a vertex, and any message that might cause a graph to be incompletely described is discarded. Therefore, if v is in an honest participant's local graph, then all of v's predecessors must be in the graph as well.

Recall that each vertex that is generated is uniquely committed to its predecessors. Because of this and the fact that every honest participant's local graph is always completely described, it follows that if v is in both $G_p^{(t)}$ and $G_q^{(t')}$, then v's predecessor graph is the same in both graphs. Moreover, it is immediate that $\mathsf{D}_{G_p^{(t)}}(v) = \mathsf{D}_{G_q^{(t')}}(v)$.

Temporal Ordering Consider that because participants cannot "make up" resources (Definition 5.2), at the moment when the inbound edges for a vertex v are chosen, v cannot have an inbound edge from any vertex u which has not yet been generated. Because each vertex is uniquely committed to its predecessors, it follows that the predecessor-successor relations among vertices in a participant's local graph obey the temporal order in which the vertices are generated. Specifically, for all vertices v and u in any participant's graph, if v is generated before u in the execution, then u cannot be a predecessor of v.

C.2 Consistency of Views for Honest Participants

Towards proving Lemma 5.2, we begin our technical lemmas with a foundational statement that lowerbounds the growth rate of the depth of $\mathbb{G}$ in an execution as a function of the number of vertices that are generated.

Intuitively, the depth of $\mathbb{G}$ is driven up by honest participants which add vertices that increase the depths of their local graphs. As a tool to understand what vertices *must* be in a participant's local graph at any point in time, we define a virtual graph $G_{\mathcal{H}}^{(t)}$, which for time t answers "what is the *smallest* graph of an honest participant at time t?" One may consider that $G_{\mathcal{H}}^{(t)}$ is guaranteed to contain *at least* all of the honest vertices that are generated before $t - \Delta$, since each honest vertex is immediately multicast when it is generated, and at most Δ time may elapse before the multicast message is guaranteed to be delivered.

The following lemma lowerbounds the growth of $G_{\mathcal{H}}$ between any t and $t' > t$ as a function of the number of resources allocated between t and t'. The growth of $G_{\mathcal{H}}$ is lowerbounded by the number of resources that are allocated to honest participants and by how many honest resources can be allocated concurrently. $G_{\mathcal{H}}$ must grow by at least 1 depth for every ρ honest vertices that are generated. This is because at most ρ honest participants can concurrently generate vertices with the same depth before one of their vertices is guaranteed to be delivered, and increases the depth of all honest graphs that have not yet reached that depth.

Lemma C.1 (Lowerbound Honest Growth)**.** *Define $G_{\mathcal{H}}^{(t)}$ as:*

$$G_{\mathcal{H}}^{(t)} = \bigcap_{t' \geq t, p \text{ active at } t'} G_p^{(t')}$$

For all t and $t' > t$: $\mathsf{D}(G_{\mathcal{H}}^{(t')}) \geq \mathsf{D}(G_{\mathcal{H}}^{(t)}) + \frac{\alpha|\Psi^{(t,t')}| - \varepsilon - \rho}{\rho}$.

Proof. Assume towards contradiction that for some t and t' in an execution, $\mathsf{D}(G_{\mathcal{H}}^{(t')}) - \mathsf{D}(G_{\mathcal{H}}^{(t)}) < \frac{\alpha|\Psi^{(t,t')}| - \varepsilon - \rho}{\rho}$. The lemma follows from the following three claims.

Claim C.1. *Between times t and t' in any execution, at least $\alpha|\Psi^{(t,t')}| - \varepsilon - \rho$ honest vertices generated between t and t' are in $G_{\mathcal{H}}^{(t')}$.*

Proof. Consider an execution between times t and t'. By α, ε-honest execution (Definition 5.6), at least $\alpha|\Psi^{(t,t')}| - \varepsilon$ resources are allocated to honest participants between t and t'. Recall that by the protocol specification, whenever an honest participant receives a resource, it immediately generates and multicasts a new vertex. The only reason why an honest vertex may not be in $G_p^{(t')}$ for any participant p active at t' is if the vertex is delayed over the network; therefore, only vertices generated after $t' - \Delta$ may not be in $G_{\mathcal{H}}^{(t')}$. By ρ-rate-limiting (Definition 5.4), at most ρ resources may be allocated between $t' - \Delta$ and t'. Therefore, at least $\alpha|\Psi^{(t,t')}| - \varepsilon - \rho$ honest vertices generated between t and t' are in $G_{\mathcal{H}}^{(t')}$. □

Claim C.2. *For any time t in an execution, every honest vertex generated after t has depth greater than $\mathsf{D}(G_{\mathcal{H}}^{(t)})$.*

Proof. Recall by the protocol specification, whenever an honest participant p generates a vertex v at time $s > t$, v is the unique deepest vertex in $G_p^{(s)}$. Thus $\mathsf{D}(v) = \mathsf{D}(G_p^{(s)})$. Additionally, observe that v cannot be in $G_{\mathcal{H}}^{(t)}$ since it cannot be delivered to all honest participants by t if it is generated at $s > t$. Because $G_{\mathcal{H}}^{(t)} \subseteq G_p^{(s)}$ by definition, v is the unique deepest vertex in $G_p^{(s)}$, and $v \notin G_{\mathcal{H}}^{(t)}$, it follows immediately that $\mathsf{D}(G_p^{(s)}) \geq \mathsf{D}(G_{\mathcal{H}}^{(t)}) + 1$, and therefore $\mathsf{D}(v) > \mathsf{D}(G_{\mathcal{H}}^{(t)})$. □

Claim C.3. *For any time t in an execution, there may be at most ρ honest vertices in $\mathbb{G}^{(t)}$ with the same depth.*

Proof. Recall by the protocol specification, whenever an honest participant p generates a vertex, the generated vertex is the unique deepest vertex in p's graph. Therefore, if an honest participant generates a vertex of depth d, then before it generated the vertex, its graph had depth $d-1$. It follows that if more than ρ honest vertices with depth d are generated, then there must be more than ρ honest participants which, when allocated a resource, have graphs of depth $d-1$. Let $p_1, \ldots, p_{\rho+1}$, be the first participants, in order, which generate vertices when their local graphs have depth $d-1$. Let $v_1, \ldots, v_{\rho+1}$ be the vertices that they generate, and let the vertices be generated at $t_{v_1}, \ldots, t_{v_{\rho+1}}$, respectively.

It must be that $t_{v_{\rho+1}} > t_{v_1} + \Delta$ because of ρ-rate limiting (Definition 5.4). But this implies that v_1 must be in $G_{p_{\rho+1}}^{(t_{v_{\rho+1}})}$, and therefore $\mathsf{D}(G_{p_{\rho+1}}^{(t_{v_{\rho+1}})}) \geq d$. This is a contradiction. □

We now conclude the proof of the lemma. By Claim C.1, at least $\alpha|\Psi^{(t,t')}| - \varepsilon - \rho$ of the vertices which are allocated between t and t' are in $G_{\mathcal{H}}^{(t')}$. By Claim C.2, all such vertices have depth greater than $\mathsf{D}(G_{\mathcal{H}}^{(t)})$. By the contradiction hypothesis, $\mathsf{D}(G_{\mathcal{H}}^{(t')}) - \mathsf{D}(G_{\mathcal{H}}^{(t)}) < \frac{\alpha|\Psi^{(t,t')}| - \varepsilon - \rho}{\rho}$.

Therefore, there must be some depth $d > \mathsf{D}(G_{\mathcal{H}}^{(t)})$ such that more than ρ honest vertices in $\mathbb{G}^{(t')}$ have depth d. This is a contradiction with Claim C.3. □

Next, we present a lemma that bounds the difference between the depth of $\mathbb{G}^{(t)}$ and the depth of $G_p^{(t)}$ for any honest participant p active at any time t. Intuitively, this bounds how far behind $\mathbb{G}$ that an honest participant's view can lag at any point in time.

Lemma C.2 (Bounding $\mathsf{D}(G_p^{(t)})$ relative to $\mathsf{D}(\mathbb{G}^{(t)})$)**.** *Let $\gamma = (1+\beta)\rho + \varepsilon + \frac{\varepsilon}{\rho} + 1$. If $\frac{\alpha}{\rho} > \beta$, then for all t and honest participant p active at t, $\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(G_p^{(t)}) \leq \gamma$.*

Sketch. The proof technique is to select an honest vertex v_c in reference to which the growth of both $\mathbb{G}^{(t)}$ and $G_p^{(t)}$ can be measured. We then upperbound the difference $\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(G_p^{(t)})$ by upperbounding $\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(v_c)$ and lowerbounding $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v_c)$. The crux of the proof is to show that there must exist a vertex v_c with respect to which the growth of each graph can be measured. Given the existence of v_c, we can bound the differences $\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(v_c)$ and $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v_c)$ in terms of the number of vertices that have been generated between the time when v_c is generated and t. We then use these bounds to show the desired statement.

Proof. Assume for the sake of reaching a contradiction that in some execution at time t and for some participant p active at t

$$\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(G_p^{(t)}) > \gamma \tag{C.1}$$

Let v_d be the vertex in $\mathsf{D}(\mathbb{G}^{(t)})$ with the greatest depth. (If there are multiple such vertices, choose any one as v_d.) Choose any *longest path* in $\mathbb{G}^{(t)}$ from root to v_d, which is defined to be a path $\mathsf{root} \to v_d$ such that the depth spanned by every edge is 1. Note that such a path *must* exist, since the depth of each vertex is defined to be one more than its deepest predecessor, and it is therefore always possible to walk backwards from v_d to root via a path in which each edge spans depth 1. Let v_c be the honest vertex with the maximum depth on this path subject to the constraint that v_c is in $G_{\mathcal{H}}^{(t)}$; in the worst case (if no honest vertices on the path are in $G_{\mathcal{H}}^{(t)}$), $v_c = \mathsf{root}$. We let t_{v_c} denote the time at which v_c is generated. (If $v_c = \mathsf{root}$, then let $t_{v_c} = 0$.)

We upperbound $\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(G_p^{(t)})$, using v_c as a reference point, by first decomposing it into parts

$$\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(G_p^{(t)}) = [\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(v_c)] - [\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v_c)] \tag{C.2}$$

It will suffice to upperbound $\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(v_c)$ and to lowerbound $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v_c)$. We begin with an upperbound for $\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(v_c)$:

Claim C.4.

$$\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(v_c) \leq \beta|\Psi^{(t_{v_c},t)}| + \varepsilon + \rho \tag{C.3}$$

Proof. Recall that because v_d is the deepest vertex in $\mathbb{G}^{(t)}$, $\mathsf{D}(v_d) = \mathsf{D}(\mathbb{G}^{(t)})$ by definition. We upperbound $\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(v_c)$ by upperbounding $\mathsf{D}(v_d) - \mathsf{D}(v_c)$. Recall also that there must be a path $v_c \to v_d$ on which each edge of the path spans 1 depth. To upperbound $\mathsf{D}(v_d) - \mathsf{D}(v_c)$, it therefore suffices to upperbound the number of vertices on the path $v_c \to v_d$. We divide the analysis into two parts: first we upperbound the number of honest vertices on the path, and then we upperbound the number of corrupt vertices on the path.

We claim that there may be at most ρ honest vertices on the path $v_c \to v_d$. Recall that v_c is defined to be the deepest vertex on a longest path from root to v_d which is also in $G_{\mathcal{H}}^{(t)}$. All of the honest vertices on the path (which are successors of v_c) must not be in $G_{\mathcal{H}}^{(t)}$. In order for an honest vertex v to not be in $G_{\mathcal{H}}^{(t)}$, there must be some honest participant q activated at some $t' \geq t$ for which v is not in $G_q^{(t')}$. This is only possible if v is delayed over the network to q at t'; therefore, any honest vertex which is on the path $v_c \rightarrow v_d$ but is not in $G_{\mathcal{H}}^{(t)}$ must have been generated after $t - \Delta$ (by Definition 3.2). By the limit on the rate of resource allocations per Δ time (Definition 5.4), there may be at most ρ such honest vertices. Therefore, there may be at most ρ honest vertices on the path $v_c \rightarrow v_d$.

We now upperbound the number of corrupt vertices on the path $v_c \rightarrow v_d$. Consider that all corrupt vertices on the path $v_c \rightarrow v_d$ must be generated after t_{v_c} because each is a successor of v_c. By the definition of a β, ε-corrupt execution (Definition 5.6), at most $\beta|\Psi^{(t_{v_c},t)}|+\varepsilon$ corrupt resources may be generated between t_{v_c} and t. It follows that there are at most $\beta|\Psi^{(t_{v_c},t)}|+\varepsilon$ corrupt vertices on the path $v_c \rightarrow v_d$.

Summing the upperbounds for honest and corrupt vertices on the path $v_c \rightarrow v_d$, it follows that

$$\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(v_c) \leq \beta|\Psi^{(t_{v_c},t)}|+\varepsilon + \rho$$

as claimed. □

We next lowerbound $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v_c)$:

Claim C.5.

$$\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v_c) \geq \frac{\alpha|\Psi^{(t_{v_c}+\Delta,t)}|-\varepsilon - \rho}{\rho} \tag{C.4}$$

Proof. Lowerbounding the difference $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v_c)$ is challenging because we do not have enough information about v_c to directly upperbound its depth. However, we do know that $\mathsf{D}(G_{\mathcal{H}}^{(t_{v_c}+\Delta)}) \geq \mathsf{D}(v)$, since v must be in the view of every honest participant activated at or after $t_{v_c} + \Delta$, and by definition it must therefore be in $G_{\mathcal{H}}^{(t_{v_c}+\Delta)}$.

Given the upperbound of $\mathsf{D}(v_c)$ in terms of $G_{\mathcal{H}}^{(t_{v_c}+\Delta)}$, we complete the desired lowerbound of

$\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v_c)$ by lowerbounding $G_p^{(t)}$ in terms of $G_{\mathcal{H}}^{(t)}$ and directly invoking Lemma C.1. This is trivial, since we know $G_{\mathcal{H}}^{(t)} \subseteq G_p^{(t)}$ by definition, and therefore $\mathsf{D}(G_p^{(t)}) \geq \mathsf{D}(G_{\mathcal{H}}^{(t)})$.

We conclude:

$$\begin{aligned}
\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v_c) &\geq \mathsf{D}(G_p^{(t)}) - \mathsf{D}(G_{\mathcal{H}}^{(t_{v_c}+\Delta)}) \\
&\geq \mathsf{D}(G_{\mathcal{H}}^{(t)}) - \mathsf{D}(G_{\mathcal{H}}^{(t_{v_c}+\Delta)}) \\
&\geq \frac{\alpha|\Psi^{(t_{v_c}+\Delta,t)}| - \varepsilon - \rho}{\rho}
\end{aligned}$$

□

We now use the upperbound on $\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(v_c)$ and the lowerbound on $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v_c)$ to conclude the lemma. Recalling (in order) Inequality C.1, Equation C.2, Inequality C.3 and Inequality C.4, we conclude:

$$\begin{aligned}
\gamma &< \mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(G_p^{(t)}) \\
&= [\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(v_c)] - [\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v_c)] \\
&\leq [\beta|\Psi^{(t_{v_c},t)}| + \varepsilon + \rho] - [\frac{\alpha|\Psi^{(t_{v_c}+\Delta,t)}| - \varepsilon - \rho}{\rho}] \\
&= \beta|\Psi^{(t_{v_c},t_{v_c}+\Delta)}| + \beta|\Psi^{(t_{v_c}+\Delta,t)}| + \varepsilon + \rho + \frac{\varepsilon}{\rho} + 1 - \frac{\alpha|\Psi^{(t_{v_c}+\Delta,t)}|}{\rho} \\
&\leq (\beta - \frac{\alpha}{\rho})|\Psi^{(t_{v_c}+\Delta,t)}| + (1+\beta)\rho + \varepsilon + \frac{\varepsilon}{\rho} + 1
\end{aligned}$$

where the last inequality follows because $\beta|\Psi^{(t_{v_c},t_{v_c}+\Delta)}| \leq \beta\rho$, since at most ρ resources may be allocated between t_{v_c} and $t_{v_c} + \Delta$ by the rate limit on resource allocations (Definition 5.4).

Therefore, it must be the case that

$$(\beta - \frac{\alpha}{\rho})|\Psi^{(t_{v_c}+\Delta,t)}| + (1+\beta)\rho + \varepsilon + \frac{\varepsilon}{\rho} + 1 > \gamma \tag{C.5}$$

but when $\frac{\alpha}{\rho} > \beta$, this is true only when $|\Psi^{(t_{v_c}+\Delta,t)}|$ is negative. This is a contradiction because there cannot be negative resource allocations. □

We now complete the proof of Lemma 5.2, which we restate here. Intuitively, the lemma shows that if for some honest participant p active at time t, and some honest vertex v, $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) > \ell_1 = \rho + \gamma$, then more than Δ time has elapsed since v was generated and multicast. It will follow that $v \in G_p^{(t)}$.

Lemma 5.2 (Depth-Based Indicator for Honest Vertices)**.** *For all t, honest p active at t, and honest vertex $v \in \mathbb{G}^{(t)}$: $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) > \ell_1 \implies v \in G_p^{(t)}$.*

Proof. Assume that there is a vertex v generated by an honest participant q at time t_v, and there is another honest participant p active at time $t > t_v$ such that $v \notin G_p^{(t)}$. We will show that it must be the case that $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) \leq \ell_1$.

Consider that because $G_p^{(t)} \subseteq \mathbb{G}^{(t)}$, the difference $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v)$ is trivially upperbounded by $\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(v)$. The difference $\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(v)$ can be decomposed into the sum of two parts: $\mathsf{D}(\mathbb{G}^{(t_v)}) - \mathsf{D}(v)$, the difference in depth between v and $\mathbb{G}$ at the moment when v is generated, and $\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(\mathbb{G}^{(t_v)})$, or the amount that $\mathbb{G}$ has grown since v was generated.

First, we observe that $\mathsf{D}(\mathbb{G}^{(t_v)}) - \mathsf{D}(v) = \mathsf{D}(\mathbb{G}^{(t_v)}) - \mathsf{D}(G_q^{(t_v)})$, since v is the deepest vertex in $G_q^{(t_v)}$. We directly apply Lemma C.2 to bound $\mathsf{D}(\mathbb{G}^{(t_v)}) - \mathsf{D}(G_q^{(t_v)}) \leq \gamma$.

Second, we upperbound $\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(\mathbb{G}^{(t_v)})$ as follows. Recall that when an honest participant generates a vertex, it immediately multicasts the vertex. If v is not in $G_p^{(t)}$, then it must be delayed over the network; therefore, it must be that $t < t_v + \Delta$. We use the rate limit on resource allocations (Definition 5.4) to conclude that $|\Psi^{(t_v,t)}| \leq \rho$. Because $\mathbb{G}$ can increase in depth between t_v and t by at most the number of vertices which are generated between t_v and t, it follows that $\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(\mathbb{G}^{(t_v)}) \leq |\Psi^{(t_v,t)}| \leq \rho$.

Therefore we conclude,

$$\begin{aligned}
\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) &\leq \mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(v) \\
&= \mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(\mathbb{G}^{(t_v)}) + \mathsf{D}(\mathbb{G}^{(t_v)}) - \mathsf{D}(G_q^{(t_v)}) \\
&\leq |\Psi^{(t_v,t)}| + \gamma \\
&\leq \rho + \gamma \\
&= \ell_1
\end{aligned}$$

□

C.3 Outputting Consistent Honest Vertices

We now re-state and prove Lemma 5.3, which states that an honest participant always extracts every honest vertex in its local graph.

Lemma 5.3 (Extracting All Honest Vertices in a Local Graph)**.** *For every time t, honest participant p active at t, and honest vertex $v \in \mathbb{G}^{(t)}$: $v \in G_p^{(t)} \implies v \in \mathsf{extract}(G_p^{(t)})$.*

We show that every vertex in an honest participant's local graph must be either a starting vertex in the graph or a predecessor of a starting vertex in the graph. We prove this in two steps. First we show that every vertex v that is generated by an honest participant is guaranteed to gain an honest successor in $\mathbb{G}$ which is at most c deeper than v. Second, we show that if an honest vertex v is more than $c + \rho$ depth from the end of an honest participant's graph, then its guaranteed honest successor must also be in the graph. Recall that the starting vertices in a participant's graph are defined to be those with depth within $c + \rho$ of the graph itself. It follows that from every honest vertex which is not a starting vertex in a participant's graph, there must be a path to an honest starting vertex in the graph.

Before we proceed, we first introduce a useful property of an execution that bounds how many consecutive corrupt vertices may be generated in a span of time in which no honest vertices are generated.

Fact C.1. *For all t, t' in an α, ε-honest execution: if $|\Psi_{\mathsf{hon}}^{(t,t')}| = 0$ then $|\Psi^{(t,t')}| \leq \frac{\varepsilon}{\alpha}$.*

Proof. Direct from Definition 5.6. $|\Psi_{\mathsf{hon}}^{(t,t')}|$ is lower bounded by $\alpha|\Psi^{(t,t')}| - \varepsilon$, which is greater than 0 for all t, t' for which $|\Psi^{(t,t')}| > \frac{\varepsilon}{\alpha}$. □

Next we show that each honest vertex v is guaranteed to gain at least one honest vertex as a successor in $\mathbb{G}$ before $\mathbb{G}$ grows too far away from v. Specifically, we show that maximum the difference in depth between v and its guaranteed honest successor is c, and that at any time t after v is generated, if $\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(v) > c$ then v's honest successor is guaranteed to already exist in $\mathbb{G}$.

Lemma C.3. *Let $c = \gamma + \rho + \frac{\varepsilon}{\alpha}$. For every time t and honest vertex $v \in \mathbb{G}^{(t)}$, $\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(v) > c$ implies there exists an honest vertex u in $\mathbb{G}^{(t)}$ such that $\mathsf{D}(u) - \mathsf{D}(v) \leq c$ and u is reachable from v.*

Sketch. Let t_v be the time when v is generated. First, we show that if $\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(v) > c$, then there must be some honest vertex generated after $t_v + \Delta$. Let v_1 be the first honest vertex generated after $t_v + \Delta$. Second, we show that $\mathsf{D}(v_1) - \mathsf{D}(v) \leq c$, and that v_1 is reachable from v.

Proof. For a vertex u, use the notation that t_u is the time at which u is generated. The proof follows from the following two claims.

Claim C.6. *If $\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(v) > c$, then there must be an honest vertex generated after $t_v + \Delta$.*

Proof. Assume that $\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(v) > c$ but there is no honest vertex generated after $t_v + \Delta$.

Consider that

$$\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(v) = \mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(\mathbb{G}^{(t_v)}) + \mathsf{D}(\mathbb{G}^{(t_v)}) - \mathsf{D}(v)$$

Lemma C.2 immediately bounds $\mathsf{D}(\mathbb{G}^{(t_v)}) - \mathsf{D}(v) \leq \gamma$. If $\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(v) > c$ and $\mathsf{D}(\mathbb{G}^{(t_v)}) - \mathsf{D}(v) \leq \gamma$, then it must be the case that $\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(\mathbb{G}^{(t_v)}) > \rho + \frac{\varepsilon}{\alpha}$. This immediately implies that $|\Psi^{(t_v,t)}| > \rho + \frac{\varepsilon}{\alpha}$, because $\mathbb{G}$ cannot grow in depth between t_v and t more than the number of vertices which are generated in that time.

Let $v_1, \ldots, v_{\rho+\frac{\varepsilon}{\alpha}+1}$, be in chronological order the first $\rho + \frac{\varepsilon}{\alpha} + 1$ vertices generated between t_v and t. Consider that between t_v and $t_v + \Delta$, as most ρ vertices may have been generated because of the rate limit on vertex generation (Definition 5.4). It follows that $v_{\rho+1}, \ldots, v_{\rho+\frac{\varepsilon}{\alpha}+1}$ must all be generated after $t_v + \Delta$. Moreover, by the contradiction hypothesis, they are all corrupt. But this means that more than $\frac{\varepsilon}{\alpha}$ consecutive corrupt vertices are generated, which is a contradiction to Fact C.1. $\square$

Next, using many of the same techniques, we bound the difference in depth between v and this honest vertex generated after $t_v + \Delta$, and show that it is reachable from v.

Claim C.7. *In any execution, consider any honest vertex v for which some honest vertex is generated after $t_v+\Delta$, and let v_1 be the first vertex generated after $t_v+\Delta$. Then $\mathsf{D}(v_1)-\mathsf{D}(v) \leq c$ and v_1 is reachable from v.*

Proof. First we show that $\mathsf{D}(v_1) - \mathsf{D}(v) \leq c$. Assume that it $\mathsf{D}(v_1) - \mathsf{D}(v) > c$.

As in the previous claim, we observe that $\mathsf{D}(v_1) - \mathsf{D}(v)$ is upperbounded by the difference in depth between v and $\mathbb{G}$ at the moment that v is generated, plus the amount that $\mathbb{G}$ grows between t_v and t_{v_1}. Specifically,

$$\begin{aligned}\mathsf{D}(v_1) - \mathsf{D}(v) &= \mathsf{D}(v_1) - \mathsf{D}(\mathbb{G}^{(t_v)}) + \mathsf{D}(\mathbb{G}^{(t_v)}) - \mathsf{D}(v)\\ &\leq \mathsf{D}(\mathbb{G}^{(t_{v_1})}) - \mathsf{D}(\mathbb{G}^{(t_v)}) + \mathsf{D}(\mathbb{G}^{(t_v)}) - \mathsf{D}(v)\end{aligned}$$

First, by an immediate application of Lemma C.2, $\mathsf{D}(\mathbb{G}^{(t_v)}) - \mathsf{D}(v) \leq \gamma$. Second we bound how much $\mathbb{G}$ can grow between t_v and t_{v_1}. Clearly, $\mathsf{D}(\mathbb{G}^{(t_{v_1})}) - \mathsf{D}(\mathbb{G}^{(t)}) \leq |\Psi^{(t,t_{v_1})}|$, since $\mathbb{G}$ cannot grow by more vertices than the number of resources allocated in this span of time.

As in the previous claim, it must be the case that $|\Psi^{(t,t_{v_1})}| > \rho + \frac{\varepsilon}{\alpha}$. By an analogous argument to the previous claim, since only ρ vertices may be generated between t and $t + \Delta$, this implies that $|\Psi^{(t_v+\Delta,t_{v_1})}| > \frac{\varepsilon}{\alpha}$. But because v_1 is the *first* honest vertex generated after $t_v + \Delta$, this leads to the conclusion that more than $\frac{\varepsilon}{\alpha}$ consecutive corrupt vertices are generated between $t_v + \Delta$ and t_{v_1}, which is a contradiction with Fact C.1. We therefore conclude that $\mathsf{D}(v_1) - \mathsf{D}(v) \leq c$.

Next we show that v_1 is reachable from v. Let r be the participant that generates v_1. We claim that v must be in $G_r^{(t_{v_1})}$. Recall that v is generated by an honest participant and immediately multicast at t_v. Therefore, v must be in the local graph of every honest participant activated after $t_v + \Delta$. Because $t_{v_1} > t_v + \Delta$, it is immediate that v is in $G_r^{(t_{v_1})}$.

Consider that if v has outdegree 0 in r's local graph before adding t_{v_1}, then because $\mathsf{D}(v_1) - \mathsf{D}(v) \leq c$, the protocol specification requires that r add an edge from v to v_1. If $\mathsf{outdegree}(v) > 0$ in r's local graph before adding t_{v_1}, then there must be some vertex u in r's local graph with an edge from v. If $\mathsf{outdegree}(u) > 0$ in r's local graph, then recursively follow u's successors until reaching a vertex w that has outdegree 0 in r's view when

r generates v_1. Notice that because w is a successor of v, $\mathsf{D}(w) > \mathsf{D}(v)$, and it follows that $\mathsf{D}(w) - \mathsf{D}(v_1) < c$. Therefore, by the protocol specification, r must add an edge from w to v_1, and there is a path from v to v_1. □

The lemma follows immediately by composing the two claims. By Claim C.6 $\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(v) > c$ implies that there is an honest vertex generated after $t_v + \Delta$. By Claim C.7, the first honest vertex v_1 generated after $t_v + \Delta$ must be reachable from v and $\mathsf{D}(v_1) - \mathsf{D}(v) \leq c$. □

The previous lemma showed that each honest vertex v is guaranteed to gain an honest successor in $\mathbb{G}$ before $\mathbb{G}$ grows to be much deeper than v. However, although v's honest successor is guaranteed to exist in $\mathbb{G}^{(t)}$ if $\mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(v) > c$, it is not necessarily true that the honest successor is in $G_p^{(t)}$ if $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) > c$. In the following lemma we show that instead, we can guarantee that if $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) > c + \rho$, then v is guaranteed to have at least one honest successor in $G_p^{(t)}$. Intuitively, the extra ρ required to show the statement for honest participants' local graphs allows enough time for v's honest successor v_1 to be delivered over the network to every honest participant.

Lemma C.4. *For every time t, honest participant p active at t, and honest vertex $v \in G_p^{(t)}$: $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) > c + \rho$ implies there exists an honest vertex $u \in G_p^{(t)}$ which is reachable from which v.*

Proof. Assume that $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) > c + \rho$ but there is no honest vertex $u \in G_p^{(t)}$ which is reachable from v. Let t_v be the time at which v is generated, and let it be generated by q.

By Lemma C.3, there must be a vertex u in $\mathbb{G}^{(t)}$ which is reachable from v such that $\mathsf{D}(u) \leq \mathsf{D}(\mathbb{G}^{(t)}) - c$. It must therefore be the case that u is not in $G_p^{(t)}$.

Consider that when u is generated by an honest participant at time t_u, it is immediately multicast. The only way that u is not in $G_p^{(t)}$ is if it is delayed over the network. Therefore, it must be the case that $t \leq t_u + \Delta$.

This implies:

$$\begin{aligned}
\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) &\leq \mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(v) \\
&= \mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(G_q^{(t_v)}) \\
&= \mathsf{D}(\mathbb{G}^{(t)}) - \mathsf{D}(\mathbb{G}^{(t_u)}) + \mathsf{D}(\mathbb{G}^{(t_u)}) - \mathsf{D}(\mathbb{G}^{(t_v)}) \\
&\quad + \mathsf{D}(\mathbb{G}^{(t_v)}) - \mathsf{D}(G_q^{(t_v)}) \\
&\leq |\Psi^{(t_u,t)}| + |\Psi^{(t_v,t_u)}| + \gamma \\
&\leq 2\rho + \frac{\varepsilon}{\alpha} + \gamma \\
&= c + \rho
\end{aligned}$$

where $\mathsf{D}(\mathbb{G}^{(t_v)}) - \mathsf{D}(G_q^{(t_v)}) \leq \gamma$ by Lemma C.2, $|\Psi^{(t_u,t)}| \leq \rho$ because $t \leq t_u + \Delta$ and by Definition 5.4, and $|\Psi^{(t_v,t_u)}| \leq \rho + \frac{\varepsilon}{\alpha}$ by an argument used in Lemma C.3.

This is a contradiction with the premise of the lemma. □

Armed with Lemma C.4, Lemma 5.3 is straightforward. We re-state it and prove it.

Lemma 5.3 (Extracting All Honest Vertices in a Local Graph)**.** *For every time t, honest participant p active at t, and honest vertex $v \in \mathbb{G}^{(t)}$: $v \in G_p^{(t)} \implies v \in \mathsf{extract}(G_p^{(t)})$.*

Proof. If $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) \leq c + \rho$, this is trivial because v is a starting vertex. If $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) > c + \rho$ then it follows from Lemma C.4 that v is reachable from a starting vertex as follows. Consider the honest vertex $u \in G_p^{(t)}$ which is reachable from v by Lemma C.4. If u is a starting vertex, then we are done. If not, then recursively apply Lemma C.4 to u until a starting vertex is reached. The depth of the recursion is bounded by the fact that if u is reachable from v, then $\mathsf{D}(u) > \mathsf{D}(v)$. □

We now also restate and conclude the proof of Lemma 5.1, as it is immediate by composing Lemmas 5.2 and 5.3.

Lemma 5.1 (Honest Vertex Extraction). *For every time t, honest participant p active at t, and honest vertex $v \in \mathbb{G}^{(t)}$: $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) > \ell_1 \implies v \in \mathsf{extract}(G_p^{(t)})$.*

Proof. This is by Lemmas 5.2 and 5.3. By Lemma 5.2, if $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) > \ell_1$ then $v \in G_p^{(t)}$. By Lemma 5.3, if v is in $G_p^{(t)}$ then $v \in \mathsf{extract}(G_p^{(t)})$. □

C.4 Extracting Consistent Corrupt Vertices

Thus far we have shown that for any two honest participants p and q, active at t and t' respectively, for which $\mathsf{D}(G_p^{(t)}) > k + \ell_1$ and $\mathsf{D}(G_q^{(t')}) > k + \ell_1$, p and q extract the same honest vertices from their graphs up to depth k.

To complete the proof of Proposition 5.1, we now show an analogous consistency property of the corrupt vertices extracted by honest participants. We show that every corrupt vertex that is extracted from an honest participant's graph and is sufficiently far from the deepest vertices in the graph *must* be a predecessor of some honest vertex in the graph. We will show that consistency of extracted corrupt vertices will follow from the consistency of their honest successors. We proceed by re-stating and proving Lemma 5.4.

Lemma 5.4 (Honest Reachability Requirement for Extraction). *For all t, participant p active at t, and vertex $v \in \mathsf{extract}(G_p^{(t)})$: $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) > \ell_2$ implies there exists an honest vertex u reachable from v such that $\mathsf{D}(u) - \mathsf{D}(v) \leq \ell_2$.*

Sketch. We show that if a vertex v is both extracted from $G_p^{(t)}$ and is sufficiently far from the deepest vertices in $G_p^{(t)}$, then v must have an honest successor u whose depth is at most ℓ_2 more than $\mathsf{D}(v)$. Consider that if v is in $\mathsf{extract}(G_p^{(t)})$, then there must be some starting vertex z in $G_p^{(t)}$ which is reachable from v. If there is no honest vertex u reachable from v whose depth is within ℓ_2 of v, then z must be reachable from v via a long sequence of corrupt vertices which starts with v and extends either all the way to z or to some honest vertex u between v and z. Let w be the deepest corrupt vertex on this corrupt-only sequence. Intuitively, if w has an outbound edge to an honest vertex, or if w is a starting vertex in the view of any honest participant after it is generated, then the depth of w must be "close" to the depth of $\mathbb{G}^{(t_w)}$.

The proof shows that contrary to the above intuition, w is actually far from the depth of $\mathbb{G}^{(t_w)}$. We show this as follows. Because w is quite far from v (by contradiction hypothesis), there are many corrupt vertices on the path between v and w. However, if many corrupt vertices are on the path between v and w, then between t_v and t_w, many more honest vertices than corrupt vertices are generated. Those honest vertices must extend the depth of $\mathbb{G}$ so much that at t_w, w is very far (measured in depth) from the deepest vertices in $\mathbb{G}^{(t_w)}$. In fact, w is so far away from the deepest vertices in $\mathbb{G}^{(t_w)}$ that it could never be a starting vertex in the view of an honest participant, and it is not close enough to the deepest vertices in $G_{\mathcal{H}}^{(t_w)}$ (which lowerbounds the depths of honest participants at t_w) to ever gain an outbound edge to an honest vertex.

Proof. Assume for the sake of contradiction that there is a vertex v in $\mathsf{extract}(G_p^{(t)})$ such that $\mathsf{D}(G_p^{(t)})-\mathsf{D}(v) > \ell_2$ but there is no honest vertex v' reachable from v such that $\mathsf{D}(v')-\mathsf{D}(v) \leq \ell_2$. Because v is in $\mathsf{extract}(G_p^{(t)})$ and $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) \gg c+\rho$, there must be a starting vertex z in $G_p^{(t)}$ which is reachable from v. Moreover, there must be a path $v \to z$ in $G_p^{(t)}$.

Let w be the deepest corrupt vertex on the path $v \to z$ which is reachable from v via a path consisting of only corrupt vertices, and let t_w be the time at which w is generated. We now show that w must be quite far from v, measured in depth.

There are two cases. If there is no honest vertex on the path $v \to z$, then w is a starting vertex in $G_p^{(t)}$. Otherwise, there is an honest vertex u with an edge from w on the path $v \to z$.

(A) In the case that w is a starting vertex, it must be the case that

$$\mathsf{D}(G_p^{(t)}) - \mathsf{D}(w) \leq c + \rho \tag{C.6}$$

And we know by the premise of the lemma that $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(v) > \ell_2$. We can therefore conclude that $\mathsf{D}(w) - \mathsf{D}(v) > \ell_2 - (c + \rho)$.

(B) In the case that there is an honest vertex u with an edge from w, there must be an honest participant q that generates u at some time $t_u > t_w$ such that

$$\mathsf{D}(G_q^{(t_u)}) - \mathsf{D}(w) \leq c \tag{C.7}$$

and in particular that $\mathsf{D}(u) - \mathsf{D}(w) \leq c$.

Moreover, we know by the contradiction hypothesis that $\mathsf{D}(u) - \mathsf{D}(v) > \ell_2$. We can therefore conclude that $\mathsf{D}(w) - \mathsf{D}(v) > \ell_2 - c$.

In either case, it must be true that

$$\mathsf{D}(w) - \mathsf{D}(v) > \ell_2 - (c + \rho) \tag{C.8}$$

Henceforth we use this relationship between w and v.

We have shown that w is quite far from v. We next we lowerbound the total number of vertices that have been generated between t_v and t_w. Then we show that during any span of time in which this many corrupt vertices have been generated, so many more honest vertices must have been generated that $\mathbb{G}$ must have grown to be much deeper than w.

Claim C.8.

$$|\Psi^{(t_v,t_w)}| > \frac{\frac{\ell_2 - (c+\rho)}{c} - \varepsilon}{\beta} \tag{C.9}$$

Proof. We show the claim in two steps. We first use the distance between w and v to lowerbound the number of corrupt resources that are allocated between t_v and t_w. Then we use the number of corrupt vertices in order to lowerbound the total number of vertices which must have been generated between t_v and t_w.

Recall that w is reachable from v via a path consisting of only corrupt vertices. $\mathsf{D}(w) - \mathsf{D}(v)$ is therefore upperbounded by c times the number of vertices on the path $v \to w$, since by the protocol specification, no edge on the path may span more than c depth. Therefore,

$$c(\beta|\Psi^{(t_v,t_w)}|+\varepsilon) \geq \mathsf{D}(w) - \mathsf{D}(v) \tag{C.10}$$

The lowerbound on $|\Psi^{(t_v,t_w)}|$ follows from applying Inequality C.8 and Inequality C.10 to show

$$c(\beta|\Psi^{(t_v,t_w)}|+\varepsilon) \geq \mathsf{D}(w) - \mathsf{D}(v) > \ell_2 - (c+\rho)$$

and with algebra we arrive at Inequality C.9, completing our claim. □

What remains is to show that between t_v and t_w, the depth of $\mathbb{G}$ has grown so much that for any honest participant r active at any $t' \geq t_w$, $G_r^{(t')}$ must be too deep for w to be a starting vertex in $G_r^{(t')}$ and too deep for r to add a vertex with an edge to w.

Claim C.9. *For every time $t' \geq t_w$ and any honest participant r active at t', $\mathsf{D}(G_r^{(t')}) - \mathsf{D}(w) > c + \rho$.*

Proof. First, we lowerbound the difference $\mathsf{D}(G_r^{(t')}) - \mathsf{D}(w)$ in terms of $|\Psi^{(t_v,t_w)}|$. We then invoke the lowerbound on $|\Psi^{(t_v,t_w)}|$ from Claim C.8 to give a concrete bound.

We start by lowerbounding $\mathsf{D}(G_r^{(t')})$ in terms of $|\Psi^{(t_v,t_w)}|$ and of $\mathsf{D}(v)$.

$$\begin{aligned}
\mathsf{D}(G_r^{(t')}) &\geq \mathsf{D}(G_{\mathcal{H}}^{(t')}) \\
&\geq \mathsf{D}(G_{\mathcal{H}}^{(t_w)}) \\
&= \mathsf{D}(G_{\mathcal{H}}^{(t_w)}) - \mathsf{D}(G_{\mathcal{H}}^{(t_v)}) + \mathsf{D}(G_{\mathcal{H}}^{(t_v)}) \\
&\geq \frac{\alpha|\Psi^{(t_v,t_w)}| - \varepsilon - \rho}{\rho} + \mathsf{D}(\mathbb{G}^{(t_v)}) - \gamma \\
&\geq \frac{\alpha|\Psi^{(t_v,t_w)}| - \varepsilon - \rho}{\rho} + \mathsf{D}(v) - \gamma
\end{aligned}$$

where $\mathsf{D}(G_{\mathcal{H}}^{(t_v)}) \geq \mathsf{D}(\mathbb{G}^{(t_v)}) - \gamma$ by a direct application of Lemma C.2, and $\mathsf{D}(\mathbb{G}^{(t_v)}) \geq \mathsf{D}(v)$ trivially because $v \in \mathbb{G}^{(t_v)}$.

Recall that by Inequality C.10, $\mathsf{D}(w) \leq c(\beta|\Psi^{(t_v,t_w)}| + \varepsilon) + \mathsf{D}(v)$. We can therefore lowerbound $\mathsf{D}(G_r^{(t')}) - \mathsf{D}(w)$ as a function of $|\Psi^{(t_v,t_w)}|$

$$\begin{aligned}
\mathsf{D}(G_r^{(t')}) - \mathsf{D}(w) &\geq \frac{\alpha|\Psi^{(t_v,t_w)}| - \varepsilon - \rho}{\rho} + \mathsf{D}(v) - \gamma - (c(\beta|\Psi^{(t_v,t_w)}| + \varepsilon) + \mathsf{D}(v)) \\
&= (\frac{\alpha}{\rho} - c\beta)|\Psi^{(t_v,t_w)}| - \gamma - c\varepsilon - \varepsilon - \frac{\varepsilon}{\rho} - 1
\end{aligned}$$

When plugging in our lowerbound for $|\Psi^{(t_v,t_w)}|$ from Inequality C.9, we find that $\mathsf{D}(G_r^{(t')}) - \mathsf{D}(w) > c + \rho$ as claimed. □

This claim presents a contradiction with both cases above. In case (A), in which w is a starting vertex in $G_p^{(t)}$, this is a contradiction to Inequality C.6. In case (B), in which w has an edge from some honest vertex u, this is a contradiction to Inequality C.7. □

C.5 Consistency and Liveness of Π^G

We can now complete the proofs of Proposition 5.1 and Theorem 5.3.

Proposition 5.1. *Let* $c = (3-\alpha)\rho + \frac{\varepsilon}{\alpha} + \frac{\varepsilon}{\rho} + \varepsilon + 1$ *(as in Protocol* Π^G*). If* $\alpha > \rho\beta c$*, then for all* k*, times* t *and* t'*, and honest participants* p *and* q *active at* t *and* t'*, respectively, if* $\mathsf{D}(G_p^{(t)}) > k + \ell^*$ *and* $\mathsf{D}(G_q^{(t')}) > k + \ell^*$*, then* $\mathsf{extract}(G_p^{(t)})|_k = \mathsf{extract}(G_q^{(t')})|_k$*.*

Proof. Assume without loss of generality that there is a vertex v that is in $\mathsf{extract}(G_p^{(t)})|_k$ but not in $\mathsf{extract}(G_q^{(t')})|_k$. It is trivial that $\mathsf{D}(v) \leq k$ if $v \in \mathsf{extract}(G_p^{(t)})|_k$.

Assume that v is an honest vertex. By the protocol specification, v must be output by q at t' if v is extracted from $G_q^{(t')}$ because $\mathsf{D}(G_q^{(t')}) \geq \mathsf{D}(v) + \ell^*$. Therefore, v must not be extracted from $G_q^{(t')}$. But this is a contradiction with Lemma 5.1, which says that v must be extracted from $G_q^{(t')}$ since $\mathsf{D}(G_q^{(t')}) - \mathsf{D}(v) > \ell_1$.

Therefore, v must be a corrupt vertex. By Lemma 5.4, if v is in $\mathsf{extract}(G_p^{(t)})|_k$, then there must be an honest vertex u such that $\mathsf{D}(u) \leq \mathsf{D}(v) + \ell_2$ such that u is reachable from v. By Lemma 5.1, u must be in $\mathsf{extract}(G_q^{(t')})$ because $\mathsf{D}(G_q^{(t')}) - \mathsf{D}(u) \geq k + \ell^* - (k + \ell_2) > \ell_1$.

Because u is reachable from v, v must be in $\mathsf{extract}(G_q^{(t')})$ by the protocol specification. And because $\mathsf{D}(v) < k$ by assumption, v must be in $\mathsf{extract}(G_q^{(t')})|_k$. This is a contradiction. □

Corollary C.1 (Graph Consistency)**.** *Protocol* Π^G *achieves graph consistency.*

Proof. In any execution, consider any two times t, t' and p, q active at t and t', respectively. Without loss of generality, assume that $\mathsf{D}(G_p^{(t)}) \geq \mathsf{D}(G_q^{(t')})$. By Proposition 5.1, it must be that $\mathsf{extract}(G_p^{(t)})|_{\mathsf{D}(G_q^{(t')})-\ell^*} = \mathsf{extract}(G_q^{(t')})|_{\mathsf{D}(G_q^{(t')})-\ell^*}$, and therefore $\mathsf{extract}(G_q^{(t')})|_{\mathsf{D}(G_q^{(t')})-\ell^*} \subseteq \mathsf{extract}(G_g^{(t)})|_{\mathsf{D}(G_g^{(t)})-\ell^*}$. □

Liveness follows from the fact that an honest participant outputs every honest vertex in its local graph with depth more than ℓ^* from the end of its graph.

Lemma C.5 (h-Liveness)**.** *Protocol* Π^G *achieves* h*-liveness, for* $h(N, \alpha, \varepsilon, \rho) = \alpha N - \varepsilon - \rho(\ell^* + 1)$*.*

Proof. Recall that in order to compute its output, an honest participant extracts vertices from its view using the $\mathsf{extract}()$ function and then outputs the extracted vertices which are more than ℓ^* depth from the end of its graph. Recall that Lemma 5.3 show an honest participant always extracts every honest vertex in its local graph. We lowerbound the number of honest vertices that an honest participant outputs at any point in time by lowerbounding how many of the vertices in its view must be honest, and then upperbounding how many honest extracted vertices may have depth too high to be output.

First we lowerbound the number of vertices in an honest participant's graph which must be honest. By Claim C.1, we know that at least $\alpha|\Psi^{(0,t)}|-\varepsilon-\rho$ honest vertices which have been generated from the beginning of the execution until t must be in $G_p^{(t)}$. Consider also that the total number of vertices that have been generated up to any point in time upperbounds the number of vertices in a participant's view, or $|\Psi^{(0,t)}|\geq |G_p^{(t)}.V|$. It follows that

$$|\mathsf{hon}(G_p^{(t)}.V)|\geq \alpha|\Psi^{(0,t)}|-\varepsilon-\rho\geq \alpha|G_p^{(t)}.V|-\varepsilon-\rho$$

What remains is to upperbound the number of honest vertices in a participant's graph at any point in time which are not output. Recall that an honest participant outputs all of the vertices which it extracts from its local graph up to ℓ^* depth from the end of its graph. By Claim C.3, there may be at most ρ honest vertices in $\mathbb{G}^{(t)}$ with the same depth, which implies that at each depth in $G_p^{(t)}$, there may be at most ρ honest vertices. Therefore, there may be at most $\rho\ell^*$ honest vertices in G_p with depth more than $\mathsf{D}(G_p)-\ell^*$, which are therefore not output.

We conclude that $|\mathsf{extract}(G_p^{(t)})|_{\mathsf{D}(G_p^{(t)})-\ell^*}|\geq \alpha|G_p^{(t)}.V|-\varepsilon-\rho-\rho\ell^*$. The lemma follows. $\square$

Corollary C.2 (f-Liveness)**.** *Protocol Π^G achieves f-liveness, for $f(N,\alpha,\varepsilon,\rho)=\alpha N-\varepsilon-\rho(\ell^*+1)$.*

Proof. Immediate from Lemma C.5. f-liveness is lowerbounded by h-liveness. The lowerbound is tight because an honest participant could extract no corrupt vertices from its local graph. □

Appendix D

Proof of Permissionless One-Bit Consensus Protocol Π^{bit}

We restate Theorem 5.5.

Theorem 5.5. *For all ρ and all ε, and for all $\alpha > \rho(1-\alpha)((3-\alpha)\rho + \frac{\varepsilon}{\alpha} + \frac{\varepsilon}{\rho} + \varepsilon + 1)$ every every (α, ε)-honest, ρ-rate-limited admissible execution of $\Pi^{\mathsf{bit}}(\alpha, \varepsilon, \rho)$ satisfies termination, consistency, and validity.*

The proof of Theorem 5.5 follows the outline in Section 5.6.2. Consistency and termination are trivial, and we provide three lemmas to show validity. First, we show that for every depth k in an execution, there is a time after which no (corrupt) vertex of depth k can be added to $\mathbb{G}$ which will ever be extracted by any honest participant. Second, we show the maximum number of corrupt vertices ω that can be generated from the time that $\mathbb{G}$ reaches depth k to the time when no (corrupt) vertex of depth k can ever be added and subsequently extracted by an honest participant. Finally, we show that by the time an honest participant's graph reaches depth k^*, there are more than ω honest vertices in its graph up to k^* than corrupt vertices.

Lemma D.1. *Let $x = c\varepsilon + c + \rho + \frac{\varepsilon}{\rho} + 1$. For every vertex v generated at t_v: if $\mathsf{D}(G_{\mathcal{H}}^{(t_v)}) > \mathsf{D}(v) + x$, then there is no time $t \geq t_v$ and honest participant p active at t for which $v \in \mathsf{extract}(G_p^{(t)})$.*

Sketch. Recall that in order for v to be extracted from $G_p^{(t)}$, it must be either a starting vertex in $G_p^{(t)}$ or a predecessor of a starting vertex in $G_p^{(t)}$. We show that $\mathsf{D}(G_{\mathcal{H}}^{(t_v)})$ is already so much deeper than v that no honest participant which is activated in the future would ever have v as a starting vertex, and no honest participant which generates a vertex in the future would ever generate a vertex with an inbound edge from v or from any (corrupt) vertex which is reachable from v. To show this, we lowerbound the difference in depth between $G_{\mathcal{H}}$ and the deepest vertex reachable from v at any point in time $t > t_v$, as a function of the number of vertices that are generated between t and t_v. We show that the difference is always greater than $c+\rho$. Because $G_{\mathcal{H}}$ lowerbounds the view of an honest participant, we can therefore conclude that v will never be a starting vertex in any honest participant's graph and no honest participant could ever add a vertex with an inbound edge from a (corrupt) successor of v.

Proof. Assume that at the time t_v when v is generated, $\mathsf{D}(G_{\mathcal{H}}^{(t_v)}) \geq \mathsf{D}(v) + x$ and that there exists a time $t > t_v$ and honest participant p for which $v \in \mathsf{extract}(G_p^{(t)})$.

First, we claim that v cannot be a starting vertex for any honest participant's $\mathsf{extract}()$ function at any time $t' \geq t_v$. For every time $t' \geq t_v$ and every honest participant q active at t',

$$\mathsf{D}(G_q^{(t')}) \geq \mathsf{D}(G_{\mathcal{H}}^{(t')}) \geq \mathsf{D}(G_{\mathcal{H}}^{(t_v)}) > \mathsf{D}(v) + x > \mathsf{D}(v) + \rho + c$$

Therefore, because $v \in \mathsf{extract}(G_p^{(t)})$ and v is not a starting vertex in $G_p^{(t)}$, there must be a starting vertex $z \in G_p^{(t)}$ reachable from v. Specifically, if z is a starting vertex, then by definition

$$\mathsf{D}(G_p^{(t)}) - \mathsf{D}(z) < \rho + c \tag{D.1}$$

We now separately consider the following two cases regarding the path $v \to z$. First, we consider the case that there are no honest vertices on the path $v \to z$. Second we consider the case that there is at least one honest vertex on the path $v \to z$.

Consider the case that there are no honest vertices on the path $v \to z$. Towards contradiction with Inequality D.1, we lowerbound the difference $\mathsf{D}(z) - \mathsf{D}(v)$ by upperbounding $\mathsf{D}(z)$ with respect to $\mathsf{D}(v)$ and $|\Psi^{(t_v,t)}|$, and lowerbounding $\mathsf{D}(G_p^{(t)})$ with respect to $\mathsf{D}(v)$ and $|\Psi^{(t_v,t)}|$.

First, we upperbound $\mathsf{D}(z)$. We claim that

$$\mathsf{D}(z) \leq \mathsf{D}(v) + c(\beta|\Psi^{(t_v,t)}|+\varepsilon) \tag{D.2}$$

By assumption, there are only corrupt vertices on the path $v \to z$. Recall from the definition of a β, ε-corrupt execution (Definition 5.6) that at most $\beta|\Psi^{(t_v,t)}|+\varepsilon$ corrupt vertices may be generated between t_v and t. By the protocol specification, if $v \to z$ is in an honest participant's local graph, then each edge on the path may span no more than c depth. It follows that $\mathsf{D}(z)$ is no more than $\mathsf{D}(v)$ plus c depth for every corrupt vertex generated between t_v and t.

Second, we lowerbound $\mathsf{D}(G_p^{(t)})$ using the premise of this lemma and a direct application of

Lemma C.1:

$$\begin{aligned}
\mathsf{D}(G_p^{(t)}) &\geq \mathsf{D}(G_{\mathcal{H}}^{(t)}) \\
&= \mathsf{D}(G_{\mathcal{H}}^{(t)}) - \mathsf{D}(G_{\mathcal{H}}^{(t_v)}) + \mathsf{D}(G_{\mathcal{H}}^{(t_v)}) \\
&\geq \frac{\alpha|\Psi^{(t_v,t)}| - \varepsilon - \rho}{\rho} + \mathsf{D}(v) + x
\end{aligned}$$

We can immediately lowerbound the difference $\mathsf{D}(G_p^{(t)}) - \mathsf{D}(z)$ using the upperbound and lowerbound just computed

$$\mathsf{D}(G_p^{(t)}) - \mathsf{D}(z) \geq (\frac{\alpha}{\rho} - c\beta)|\Psi^{(t_v,t)}| + x - c\varepsilon - \frac{\varepsilon}{\rho} - 1 \tag{D.3}$$

but when $\frac{\alpha}{\rho} > c\beta$ this is a contradiction with Inequality D.1 because $|\Psi^{(t_v,t)}|$ must be non-negative.

Therefore, there must be an honest vertex on the path $v \to z$. Let w be the deepest corrupt vertex on the path $v \to z$ such that there are no honest vertices on the subpath $v \to w$. Then there must be an honest vertex u with an inbound edge from w. Let q be the participant that generates u, and let t_u be the time at which q generates u.

Now, because there are no honest vertices on the path $v \to w$, we can invoke the same argument that we used for the above case in which there are no honest vertices on the path $v \to z$, replacing z with w, and replacing t with the time t_u at which q generates u.

The only difference in the proof is that the difference $G_q^{(t_u)} - \mathsf{D}(w)$ is *less* than the difference $G_p^{(t)} - \mathsf{D}(z)$ above. Specifically, it must be the case that

$$\mathsf{D}(G_q^{(t_u)}) - \mathsf{D}(w) \leq c \tag{D.4}$$

The rest of the proof follows analogously. $\square$

Lemma D.2. *Let $\omega = \frac{\beta\rho}{\alpha}(x+\gamma+\frac{\varepsilon}{\rho}+1)+\varepsilon$, and let the notation t_k denote the earliest time for which $\mathsf{D}(\mathbb{G}^{(t_k)}) = k$. For every time t, honest participant p active at t, and depth k: at most ω corrupt vertices in $\mathsf{extract}(G_p^{(t)})|_k$ were generated after t_k.*

Sketch We show that if more than ω corrupt vertices in $\mathsf{extract}(G_p^{(t)})|_k$ were generated after t_k, then there must be some corrupt vertex u in $\mathsf{extract}(G_p^{(t)})|_k$ which was generated when $G_{\mathcal{H}}$ was already more than x depth deeper than u. This is a contradiction to Lemma D.1.

Proof. Assume that there are more than ω corrupt vertices generated between t_k and t that are in $\mathsf{extract}(G_p^{(t)})|_k$. Let u be the last such corrupt vertex that is generated, and let t_u be the time at which it is generated. Trivially, it must be the case that $t > t_u$ (otherwise u could not be in $G_p^{(t)}$).

We lowerbound $|\Psi^{(t_k,t_u)}|$ as follows. By the contradiction hypothesis, more than ω corrupt vertices have been generated between t_k and t_u. We can therefore lowerbound $|\Psi^{(t_k,t_u)}|$ using Definition 5.6 and the number of corrupt vertices which have been generated between t_k and t. Specifically, recall that the number of corrupt vertices that are generated between t_k and t_u is upperbounded by $\beta|\Psi^{(t_k,t_u)}|+\varepsilon$. By assumption, we have that $\beta|\Psi^{(t_k,t_u)}|+\varepsilon > \omega$, which implies that $|\Psi^{(t_k,t_u)}| > \frac{\omega-\varepsilon}{\beta}$.

Towards contradiction, we now lowerbound $\mathsf{D}(G_{\mathcal{H}}^{(t_u)})$. We do so by invoking Lemma C.1 to lowerbound how much $G_{\mathcal{H}}$ must grow as honest participants add vertices to $\mathbb{G}$ between t_k

and t_u. Specifically,

$$\begin{aligned}
\mathsf{D}(G_{\mathcal{H}}^{(t_u)}) &= \mathsf{D}(G_{\mathcal{H}}^{(t_u)}) - \mathsf{D}(G_{\mathcal{H}}^{(t_k)}) + \mathsf{D}(G_{\mathcal{H}}^{(t_k)}) \\
&\geq \frac{\alpha|\Psi^{(t_k,t_u)}|-\varepsilon-\rho}{\rho} + \mathsf{D}(\mathbb{G}^{(t_k)}) - \gamma \\
&> \frac{\alpha\frac{\omega-\varepsilon}{\beta} - \varepsilon - \rho}{\rho} + \mathsf{D}(\mathbb{G}^{(t_k)}) - \gamma \\
&\geq \frac{\alpha}{\beta\rho}(\omega-\varepsilon) - \gamma - \frac{\varepsilon}{\rho} - 1 + k \\
&\geq k + x
\end{aligned}$$

where it follows our definition of t_k that $\mathsf{D}(\mathbb{G}^{(t_k)}) = k$. Additionally, it follows from Lemma C.2 and the definition of $G_{\mathcal{H}}^{(t_k)} = \bigcap_{t' \geq t_k, q \text{ active at } t'} G_q^{(t')}$ that $\mathsf{D}(G_{\mathcal{H}}^{(t_k)}) \geq \mathsf{D}(\mathbb{G}^{(t_k)}) - \gamma$.

This is a contradiction with Lemma D.1. Recall that $\mathsf{D}(u) \leq k$ by assumption, and therefore $\mathsf{D}(G_{\mathcal{H}}^{(t_u)}) > \mathsf{D}(u) + x$. Lemma D.1 says that if at the time t_u when u is generated, $\mathsf{D}(G_{\mathcal{H}}^{(t_u)}) > \mathsf{D}(u) + x$, then u may never be in $\mathsf{extract}(G_p^{(t)})$ for any p active at $t > t_u$. □

Claim D.1. *For every time t and all k:* $\mathsf{D}(\mathbb{G}^{(t)}) \geq k \implies |\Psi_{\mathsf{hon}}^{(0,t)}| - |\Psi_{\mathsf{cor}}^{(0,t)}| \geq (\alpha-\beta)k - 2\varepsilon$

Proof. As a direct consequence of Definition 5.6, between any t and $t' > t$, $|\Psi_{\mathsf{hon}}^{(t,t')}| - |\Psi_{\mathsf{cor}}^{(t,t')}| \geq (\alpha-\beta)|\Psi^{(t,t')}| - 2\varepsilon$.

Using this fact and the fact that $|\Psi^{(0,t)}| \geq \mathsf{D}(\mathbb{G}^{(t)})$ (because $\mathbb{G}^{(t)}$ can be no deeper than the number of vertices in $\mathbb{G}^{(t)}$):

$$\begin{aligned}
|\Psi_{\mathsf{hon}}^{(0,t)}| - |\Psi_{\mathsf{cor}}^{(0,t)}| &\geq (\alpha-\beta)|\Psi^{(0,t)}| - 2\varepsilon \\
&\geq (\alpha-\beta)\mathsf{D}(\mathbb{G}^{(t)}) - 2\varepsilon \\
&\geq (\alpha-\beta)k - 2\varepsilon
\end{aligned}$$

□

Lemma D.3. *Let $k^* = \frac{\omega+2\varepsilon}{\alpha-\beta}$. For every time t and honest participant p active at t, if $\mathsf{D}(G_p^{(t)}) \geq k^* + \ell_1$, then the majority of vertices in $\mathsf{extract}(G_p^{(t)})|_{k^*}$ are honest.*

Proof. Let t^* be the earliest time at which $\mathsf{D}(\mathbb{G}^{(t^*)}) = k^*$. We show that there are more honest vertices with depth less than k^* generated between the beginning of the execution and t^* than the sum of (a) the number of corrupt vertices generated between the beginning of the execution and t^* and (b) the total number of corrupt vertices with depth less than or equal to k^* which can be generated after t^* and still extracted from any honest participant's graph after t^*. Because all honest vertices that have been generated with depth up to k^* are guaranteed to be extracted from an honest graph with depth $k^* + \ell_1$, it follows that there must be more extracted honest vertices up to depth k^* than extracted corrupt vertices up to depth k^*.

The formal argument follows. By Claim D.1, there are at least ω more honest vertices in $\mathbb{G}^{(t^*)}$ than corrupt vertices. By Lemma 5.2, if $\mathsf{D}(G_p^{(t)}) > k^* + \ell_1$, then all of the honest vertices generated before t^* are in $G_p^{(t)}$. By Lemma D.2, there are at most ω corrupt vertices in $\mathsf{extract}(G_p^{(t)})|_{k^*}$ which were generated after t^*. Therefore, if $\mathsf{D}(G_p^{(t)}) > k^* + \ell_1$, a majority of vertices in $\mathsf{extract}(G_p^{(t)})|_{k^*}$ are honest. □

Lemma D.4 (Validity of Π^{bit})**.** *If all honest participants have input $b \in \{0, 1\}$, then all honest participants that do not fail output b.*

Proof. Lemma D.3 shows that for all t and participants p active at t, $\mathsf{D}(G_p^{(t)}) > k^* + \ell_1$ implies a majority of the vertices in $\mathsf{extract}(G_p^{(t)})|_{k^*}$ are honest. Therefore, if all honest participants have input b, then for all t and p active at t such that $\mathsf{D}(G_p^{(t)}) > k^* + \ell^*$, a majority of the vertices in $\mathsf{extract}(G_p^{(t)})|_{k^*}$ have label b. It is immediate that every honest participant outputs b. □

Appendix E

Full Proofs for Group Messaging

E.1 Proof of Theorem 6.1

For the proof of Theorem 6.1 we briefly define a restricted variant of the EUF-CMA game for MAC forgery. The challenger samples a signing key k, which it does not provide to the adversary. It provides the adversary with oracle access for MACs on randomly sampled messages (by the challenger) which are guaranteed to verify under the key k. (Note the difference between this game and the EUF-CMA game, that the adversary does not sample the messages that it asks of the oracle.) The adversary wins the game if it can construct a message m^* that verifies using k. We require that in this game, the challenger samples messages from the domain of public keys for a public-key AEAD scheme.

Proof. In the zeroth game G_0, we handle all the queries normally, as prescribed by the GRM protocol. Note that MAC forgery is possible in this game. This happens when $\mathcal{A}$ outputs $(\mathtt{pk}_V, c_V) \leftarrow \Pi^{\mathsf{grm}}_{U,i}.\mathsf{Recv}(c, 0)$, where $\mathtt{pk}_V$ is a public key that did was not output by any oracle call to $\mathsf{Init}()$ but c_V is a valid MAC on $\mathtt{pk}_V$ verified using the key k used to initialize some oracle.

The game G_1 is the same as G_0 except we add a forgery check as follows.

- On $\Pi_{U,i}^{\mathsf{grm}}.\mathsf{Init}(w, Q)$: Generate $\{(\mathtt{pk}_V, \mathtt{sk}_V)\}_{V \in Q}$ and compute the MAC for $c_V \leftarrow \mathsf{MAC}(\mathtt{pk}_V)$ for every $V \in Q$. Output $\{(\mathtt{pk}_V, c_V) : V \in Q\}$.
- On $\Pi_{U,i}^{\mathsf{grm}}.\mathsf{Recv}(c, \mathsf{dec\_flag})$: If c has the format $(\mathtt{pk}_V, c_V)$ and $\mathtt{pk}_V$ is not one of the public keys generated then we consider two cases.
  1. If c_V is an invalid MAC on $\mathtt{pk}_V$, abort the game as prescribed.
  2. Otherwise, also abort the game and denote this event as F. This is the forgery event.

Observe that only difference between G_0 and G_1 is event F, the forgery event.

Claim E.1. *There exists and adversary $\mathcal{C}$ against the restricted EUM-CMA game described above that uses the adversary $\mathcal{A}$ for GRM such that* $\mathsf{Adv}^{\mathsf{mac}}(\mathcal{C}) \geq \Pr[\mathcal{A}$ *forges a MAC*$]$.

Proof. We first build an adversary $\mathcal{C}$ for the modified EUF-CMA game above as follows. In the security game for the GRM game, $\mathcal{A}$ invokes oracles via their $\mathsf{Init}()$ query, but the key k used to initialized a set of oracles Q is not exposed to the adversary. When an oracle U within a group Q is called with Init by the adversary, the game initializes that oracle and outputs a key $\mathtt{pk}_U$ along with a MAC c_U on $\mathtt{pk}_U$ which is guaranteed to verify with k.

$\mathcal{C}$ first guesses which session i^* of a maximum of n_S GRM sessions (where n_S is allowed to be polynomial in λ). When the adversary $\mathcal{A}$ for the GRM game calls Init on an oracle in a group Q in the i^*th session, the $\mathcal{C}$ forwards the query to its challenger as a request for a signed message. $\mathcal{C}$ stores the response $(\mathtt{pk}, c)$ and forwards it to $\mathcal{A}$. (Note that the $\mathcal{A}$ will make only up to $|Q|$ such queries because it can only call Init once per oracle in the group per session.) For every other session $\mathcal{C}$ samples a MAC and simulates the GRM session for $\mathcal{A}$ perfectly as if it were $\mathcal{A}$'s challenger.

In the i^*th session, whenever $\mathcal{A}$ calls Recv using a message $(\mathtt{pk}, c)$ that $\mathcal{C}$ has already forwarded to $\mathcal{A}$ in the response to an Init query, $\mathcal{C}$ continues the game as normal. If $\mathcal{A}$ never calls Recv using a message $(\mathtt{pk}, c)$ that $\mathcal{C}$ has not forwarded to $\mathcal{A}$, then $\mathcal{C}$ returns a random message and MAC to its challenger. When $\mathcal{A}$ calls Recv using a message $(\mathtt{pk}, c)$ that $\mathcal{C}$ has not forwarded to $\mathcal{A}$, $\mathcal{C}$ forwards the message to its challenger. $\mathcal{C}$ wins its game with at least the probability that $\mathcal{A}$ constructs a valid forgery, conditioned on guessing i^* correctly.

□ □

We bound $\Pr[\mathcal{A} \text{ wins } G_1]$ by designing adversary $\mathcal{B}$ against the PKAEAD game. At the start, $\mathcal{B}$ guesses the instance that $\mathcal{A}$ will query. This guess is correct with probability $1/|Q|_{\max}$. Additionally, $\mathcal{B}$ guesses the index of the ciphertext that $\mathcal{A}$ will use in Test. Specifically, there will be a critical query to $\mathcal{B}$'s $\mathsf{PKAEAD.Enc}$ oracle and the output of this query is used by $\mathcal{A}$ in the test query. The guess will be correct with probability $1/n_Q$, where n_Q is the upperbound of the number of queries to the encryption oracle that $\mathcal{B}$ makes on behalf of $\mathcal{A}$ for the instance under test. $\mathcal{B}$ sets $c^* \leftarrow \perp$.

Since forgery does not occur in G_1, we can replace the MACs by ideal MACs. That is, $\mathcal{B}$ stores a lookup table $\mathcal{M}$ that maps messages to MACs. Every time a MAC needs to be generated for message m, either $\mathcal{M}[m]$ is returned if it exsits, otherwise a MAC v sampled uniformly at random is returned and $\mathcal{M}[m] \leftarrow v$ is stored.

Concretely, the queries are handled as follows.

- $\Pi^{\mathsf{grm}}_{U,i}.\mathsf{Init}(w, Q)$: Follow the steps in Definition 6.10, i.e., call the function $\mathsf{GRM}_U.\mathsf{Init}(\mathsf{k}, w, G)$ using the same k for the same Q.
- $\Pi^{\mathsf{grm}}_{U,i}.\mathsf{Evolve}()$: Perform the steps described in the protocol instantiation except substitute $\mathsf{PKAEAD.Enc}$ to calls to $\mathcal{B}$'s oracle. If one of the calls to $\mathsf{PKAEAD.Enc}$ is the critical query, then instead of calling $\mathsf{PKAEAD.Enc}$, $\mathcal{B}$ samples $(r_0^*, r_1^*) \xleftarrow{\$} R^2$ and then

calls

$$c^* \leftarrow \mathsf{PKAEAD.Test}(m_0^*, m_1^*),$$

where $m_0^* = \mathtt{pk}_{U,i}^{j+1} \| r_0^*$ and $m_1^* = \mathtt{pk}_{U,i}^{j+1} \| r_1^*$.

– $\Pi_{U,i}^{\mathsf{grm}}.\mathsf{Recv}(c, \mathsf{dec\_flag})$: If $c = c^*$, abort. Otherwise, perform the steps described in the instantiation except $\mathcal{B}$ substitutes PKAEAD.Dec to calls to $\mathcal{B}$'s oracle.

– $\Pi_{U,i}^{\mathsf{grm}}.\mathsf{StateReveal}()$: return $\mathsf{state}_{U,i}$.

– $\Pi_{U,i}^{\mathsf{grm}}.\mathsf{Test}(c)$: If $c = c^*$, return (m_0^*, m_1^*). Otherwise, abort.

Finally, $\mathcal{A}$ will output a bit b' to $\mathcal{B}$ and $\mathcal{B}$ outputs the same bit b' to the challenger.

Observe that although $\mathcal{A}$ is allowed to call StateReveal, it is not allowed to call it if one of the secret keys can be used to decrypt c^* or any ciphertext from the same Evolve call due to our security definition. So $\mathcal{A}$ gains no advantage from having access to StateReveal. Specifically, consider $\Pi_{U,i}^{\mathsf{grm}}.\mathsf{Test}(c^*)$ where $\mathtt{pk}_{U,i}^*$ is used to encrypt the plaintext m^* such that $(c^*, t^*) \leftarrow \mathsf{PKAEAD.Enc}(m^*, d^*; \mathtt{pk}_{U,i}^*)$, then there are four cases.

1. $\Pi_{U,i}^{\mathsf{grm}}.\mathsf{StateReveal}$ is called before $\Pi_{U,i}^{\mathsf{grm}}.\mathsf{Test}$: the time when StateReveal is called would not have $\mathtt{sk}_{U,i}^*$ to decrypt the key.

2. $\Pi_{U,i}^{\mathsf{grm}}.\mathsf{StateReveal}$ is called after $\Pi_{U,i}^{\mathsf{grm}}.\mathsf{Test}$: the key $\mathtt{sk}_{U,i}^*$ would have been deleted when StateReveal is called.

3. $\Pi_{U,i}^{\mathsf{grm}}.\mathsf{StateReveal}$ is called before $\Pi_{V,j}^{\mathsf{grm}}.\mathsf{Test}$ where $(U, i) \neq (V, j)$: only the oracle $\Pi_{V,j}^{\mathsf{grm}}$ has the private key to decrypt c^*, so revealing the state of (U, i) does not give $\mathcal{A}$ an advantage.

4. $\Pi^{\mathsf{grm}}_{U,i}.\mathsf{StateReveal}$ is called after $\Pi^{\mathsf{grm}}_{V,j}.\mathsf{Test}$ where $(U,i) \neq (V,j)$: only the oracle $\Pi^{\mathsf{grm}}_{V,j}$ has the private key to decrypt c^*, so revealing the state of (U,i) does not give $\mathcal{A}$ an advantage.

Additionally, due to our security definition, $\mathcal{A}$ is not allowed to use the decryption oracle on c^* or any ciphertext c' that came from the same call to Evolve as c^*. So $\mathcal{A}$ gains no advantage in distinguishing the two plaintexts on top of its existing advantage from PKAEAD.

Our simulator perfectly simulates the view of $\mathcal{A}$ with probability $\frac{1}{|Q|_{\max} \cdot n_Q}$. Thus we have

$$\mathsf{Adv}^{\mathsf{grm}'}_{\mathcal{A}} \leq |Q|_{\max} \cdot n_Q \cdot \mathsf{Adv}^{\mathsf{pkaead}}_{\mathcal{B}},$$

where $\mathsf{Adv}^{\mathsf{grm}'}_{\mathcal{A}} = 2|\Pr[\mathcal{A} \text{ wins } G_1] - 1/2|$.

Recall that the adversaries of Game 0 and Game 1 attack disjoints events of the probability space of $\mathcal{A}$'s security game. Summing the inequalities of the adversaries' advantages, we obtain

$$\mathsf{Adv}^{\mathsf{grm}}_{\mathcal{A}} \leq n_S \cdot \mathsf{Adv}^{\mathsf{mac}}_{\mathcal{C}} + 2 \cdot |Q|_{\max} \cdot n_Q \cdot \mathsf{Adv}^{\mathsf{pkaead}}_{\mathcal{B}}.$$

□ □

E.2 Full Proof of GM Construction

Theorem 6.2 (Security of Group Messaging). *If $\mathcal{A}$ is an adversary against the GM game, then there exist adversaries $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$ such that $\mathsf{Adv}^{\mathsf{gm}}(\mathcal{A}) \leq 2n_S\mathsf{Adv}^{\mathsf{gka}}(\mathcal{B}) + 2n_S n\mathsf{Adv}^{\mathsf{grm}}(\mathcal{C}) + n_S\psi\mathsf{Adv}^{\mathsf{cca}}(\mathcal{D})$, where $n_S = \mathsf{poly}(\lambda)$ is the maximum the number of GM sessions $\mathcal{A}$ may invoke, and $\psi = \mathsf{poly}(\lambda)$ is the maximum number of keys that $\mathcal{A}$ may query* *in a session.*

Proof. We begin by defining three games which will allow us to complete the reduction. The first game $\mathsf{Game0}$ is the group messaging game. The second game, $\mathsf{Game1}$, is like the group messaging game, except that in the beginning of the game, the challenger selects a random initial group key. The third game, $\mathsf{Game2}$, is like $\mathsf{Game1}$, except that the challenger switches the update σ corresponding to some party's key evolution to a random update. We will show that if $\mathcal{A}$ is an adversary against the GM game, then we use these games to construct adversaries $\mathcal{B}$ for the GKA game, $\mathcal{C}$ for the GRM game, and $\mathcal{D}$ for the AEAD-CCA game that use $\mathcal{A}$ to win their respective games, with the advantages given in the theorem statement.

Lemma E.1. *There exists an adversary $\mathcal{B}$ for GKA such that* $\mathsf{Adv}^{\mathsf{gka}}(B) \geq \frac{1}{n_S}|\Pr[A \textit{ wins } \mathsf{Game0}] - \Pr[A \textit{ wins } \mathsf{Game1}]|$.

Proof. We show how to construct $\mathcal{B}$ such that it uses $\mathcal{A}$ to win the GKA game. Specifically, the difference between the two games is that $\mathsf{Game0}$ is the GM game, and in $\mathsf{Game1}$, the GKA key on the oracle under test is switched for a random key. $\mathcal{B}$ uses $\mathcal{A}$'s ability to distinguish between these two games in order to win its key indistinguishability game, which is exactly whether, on the Test instance, $\mathcal{B}$ is given the correct output of GKA or a random key.

$\mathcal{B}$'s challenger samples long-term keypairs $(\mathtt{pk}_U, \mathtt{sk}_U)$ for every parties U in the game. It provides $\mathcal{B}$ with the parties' public keys. It also samples a bit $b_{\mathsf{gka}} \leftarrow \{0,1\}$ which serves as the challenge bit.

We now describe how $\mathcal{B}$ emulates the environment for $\mathcal{A}$. $\mathcal{B}$ begins by guessing the instance $\hat{\mathsf{sid}}$ which $\mathcal{A}$ will test. Whenever $\mathcal{A}$ makes an oracle query that corresponds to a gka query, $\mathcal{B}$ forwards the query to its own oracle and returns the response directly to $\mathcal{A}$. For every instance $\mathsf{sid} \neq \hat{\mathsf{sid}}$, as a first step after any gka subprotocol outputs a key, $\mathcal{B}$ queries its oracle $\Pi^{\mathsf{gka}}_{U,j}.\mathsf{Reveal}()$ (for any instance (U,j) which has the initial group key for sid in its state) to learn the group key. For instance $\hat{\mathsf{sid}}$, $\mathcal{B}$ queries $\Pi^{\mathsf{gka}}_{U,i}.\mathsf{Test}()$ immediately after the gka session outputs a key (for appropriate (U, i) mapped to that instance), and receives either the group key or a random key. $\mathcal{B}$ denotes its initial group key in session sid by $k^{(0)}_{\mathsf{sid}}$.

After GKA has been executed for a session, $\mathcal{B}$ emulates the key evolutions and encrypted messaging of GM for $\mathcal{A}$. Whenever $\mathcal{A}$ makes an oracle query that corresponds to a GRM query, $\mathcal{B}$ emulates the GRM oracle internally. Specifically, once it has an initial group key $k^{(0)}_{\mathsf{sid}}$ for session sid, $\mathcal{B}$ begins to simulate a GRM instance and tracks every key update σ generated by every party. It applies these updates to $k^{(0)}_{\mathsf{sid}}$ as necessary in order to fulfill $\mathcal{A}$'s requests. Whenever $\mathcal{A}$ makes an encryption query $\Pi^{\mathsf{gm}}_{U,j}.\mathsf{Enc}()$, $\mathcal{B}$ evolves $k^{(0)}_{\mathsf{sid}}$ (for the appropriate corresponding sid) to the appropriate key (which is exactly the key corresponding to the maximal index in U's lattice in the instance mapped to sid). $\mathcal{B}$ similarly responds to decryption queries, first by looking up the key index referenced by the decryption message and then evolving the initial key using the appropriate updates.

We now provide the full details of the reduction. $\mathcal{B}$ receives all parties' long-term public keys as input in its game. Throughout the game, it maintains state for each party in the GM execution in order to emulate GM internally. It maintains the variables $\delta_{U,i} \in \{\mathsf{pending}, \mathsf{accept}, \mathsf{abort}\}$ to denote party U's pairing status in instance (U, i) (which maps to some session sid in which other (V, j) are present; $\mathcal{B}$ internally computes this mapping and we elide the details), and $\kappa_{U,i} \in \{\mathsf{corrupted}, \perp\}$ to denote U's corruption status.

In addition to the local state of all parties, $\mathcal{B}$ maintains additional state to track the simulation. L^{sid} is a key lattice of all the keys and updates defined in session sid. The vertices and edges that are colored red in L^{sid} correspond to $L^{\mathsf{rev}}_{\mathsf{sid}}$. We note that in the description of the simulator, we do not have $\mathcal{B}$ track the GM buffer B_U for each party U. The adversary $\mathcal{A}$ learns every message M that would be put in the buffer, and can invoke its oracles in order to mimic the behavior of the GM protocol, if it chooses.

$\mathcal{B}$ responds to oracle queries by $\mathcal{A}$ as follows:

- $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Init}(G)$: $\mathcal{B}$ initializes local states for party $U \in G$ for instance i by setting $\delta_{U,i} \leftarrow$ $\mathsf{pending}$ and $\rho_{U,i} \leftarrow \perp$. If U has not been initialized before, then $\mathcal{B}$ sets $\kappa_U \leftarrow \perp$. $\mathcal{B}$ initializes the GKA protocol for party i by forwarding $\mathcal{A}$'s query to its own oracle $\Pi^{\mathsf{gka}}_{U,i}.\mathsf{Init}(G)$ and returns the output to $\mathcal{A}$.
- $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Corrupt}()$:
  - $\mathcal{B}$ sets $\kappa_U \leftarrow \mathsf{corrupted}$.
  - $\mathcal{B}$ forwards the corruption query to $\Pi^{\mathsf{gka}}_{U,i}.\mathsf{Corrupt}()$ and returns to $\mathcal{A}$ the output.
  - Every future GKA message received by $\Pi^{\mathsf{gm}}_{U,j}$ *for any* j is passed to $\mathcal{A}$, and corresponding vertices and edges that it learns are colored red in the respective L^{sid} (for the sid mapped by (U, j)).
- $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Evolve}()$: $\mathcal{B}$ internally executes $\{(c_{U,V}, x_{U,V})\}_{V \in G} \leftarrow \Pi^{\mathsf{grm}}_{U,i}.\mathsf{Evolve}()$ to generate a message x (where each $x_{U,V}$ is equal to x) and a set of ciphertexts $\mathbf{c}$ encoding x. $\mathcal{B}$ updates L^{sid} (for sid mapped by (U, i)) by labeling the edges corresponding to the key evolution with x, and returns $\mathbf{c}$ to $\mathcal{A}$.
- $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Recv}(M)$:
  - If $\delta_{U,i} = \mathsf{abort}$, $\mathcal{B}$ does nothing.
  - Else if M is a GKA message (contains a header gka) and $\delta_{U,i} = \mathsf{pending}$, $\mathcal{B}$ queries $\Pi^{\mathsf{gka}}_{U,i}.\mathsf{Recv}(M)$ and returns the output to $\mathcal{A}$. If $\Pi^{\mathsf{gka}}_{U,i}$ outputs done, $\mathcal{B}$ sets $\delta_{U,i} = \mathsf{accept}$. If U is the first party in the session for which $\delta_{U,i} = \mathsf{accept}$ (meaning $\Pi^{\mathsf{gka}}_{U,i}$ is not yet partnered with any other oracles), then:
    * if the sid mapped by (U, i) is the test session sid^*, $\mathcal{B}$ queries $\Pi^{\mathsf{gka}}_{U,i}.\mathsf{Test}()$ and assigns the output to $k^{(0)}_{\mathsf{sid}}$, and updates L^{sid} by assigning $k^{(0)}_{\mathsf{sid}}$ to the vertex at $\mathbf{0}$.

    - if the sid mapped by (U, i) is not the test session sid^*, $\mathcal{B}$ queries $\Pi^{\mathsf{gka}}_{U,i}.\mathsf{Reveal}()$, assigns the output to $k^{(0)}_{\mathsf{sid}}$, and updates L^{sid} by assigning $k^{(0)}_{\mathsf{sid}}$ to the vertex at $\mathbf{0}$.

  - Else if M is a GRM message (contains a header grm) and $\delta_{U,i} = \mathsf{accept}$, $\mathcal{B}$ internally runs $\Pi^{\mathsf{grm}}_{U,i}.\mathsf{Recv}(M)$ and forwards to $\mathcal{A}$ anything that is returned

- $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Dec}(M)$:

  - If M is not of the form $(V \| \mathbf{i} \| \mathtt{ct})$ or $\delta_{U,i} \neq \mathsf{accept}$, then $\mathcal{B}$ sets $\delta_{U,i} = \mathsf{abort}$ and returns $\perp$ to $\mathcal{A}$.

  - Otherwise, $\mathcal{B}$ if there is no key at index $\mathbf{i}$ in U's lattice (for instance i), $\mathcal{B}$ returns $\perp$ to $\mathcal{A}$. Otherwise, $\mathcal{B}$ returns $m \leftarrow \mathsf{Dec}(k, \mathtt{ct})$ to $\mathcal{A}$, where k is the key at index $\mathbf{i}$ in U's lattice.

- $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Enc}(M)$:

  - If $\delta_{U,i} \neq \mathsf{accept}$ then set $\delta_{U,i} \leftarrow \mathsf{abort}$ and return $\perp$.

  - Otherwise, $\mathcal{B}$ computes $\mathbf{i}_U$ as the maximal index in U's local lattice and k as the key corresponding to $\mathbf{i}_U$ in U's lattice. $\mathcal{B}$ computes $\mathtt{ct} \leftarrow \mathsf{Enc}(\mathsf{k}, M)$, and returns $(U \| \mathbf{i}_U \| \mathtt{ct})$ to $\mathcal{A}$.

- $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Reveal}()$: If $\delta_{U,i} \neq \mathsf{accept}$ then $\mathcal{B}$ does nothing. If this query is called after $\mathsf{Test}()$ on the test session sid^* and the key with index $\mathbf{i}^*$ (defined in Test) is computable from U's local state (in instance (U, i)), then $\mathcal{B}$ does nothing. Otherwise $\mathcal{B}$ computes K as the set of all keys computable from (U, i)'s local state, and marks every vertex in K as red in $L^{\mathsf{rev}}_{\mathsf{sid}}$ (for the sid mapped by (U, i)). Finally, $\mathcal{B}$ computes $s \leftarrow \Pi^{\mathsf{gka}}_{U,i}.\mathsf{Reveal}()$ and returns the values (s, K) to $\mathcal{A}$.

- $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{StateReveal}()$: $\mathcal{B}$ computes $s \leftarrow \Pi^{\mathsf{gka}}_{U,i}.\mathsf{StateReveal}()$ and returns $(s, \mathsf{state}_{U,i})$ to $\mathcal{A}$, where $\mathsf{state}_{U,i}$ is the state that $\mathcal{B}$ maintains for (U, i) in its GRM instance. $\mathcal{B}$ also marks all edges in $L^{\mathsf{rev}}_{\mathsf{sid}}$ as red that are revealed by $\mathsf{state}_{U,i}$ or in $\mathcal{E}_{U,i}$.

– $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Test}(m_0, m_1)$: $\mathcal{B}$ computes $\mathbf{i}_U$ as the maximal index in U's local lattice and k_U as the key corresponding to $\mathbf{i}_U$ in U's lattice. If k_U is computable from $L^{\mathsf{rev}}_{\mathsf{sid}}$ (in the sid mapped by (U, i), then $\mathcal{B}$ ignores the request, as this key is not fresh. Otherwise, $\mathcal{B}$ sets $\mathbf{i}^* \leftarrow \mathbf{i}_U$, samples $b \leftarrow \{0, 1\}$ and returns $c \leftarrow \mathsf{Enc}(\mathsf{k}_U, m_b)$ to $\mathcal{A}$.

Note that $\mathcal{B}$ ignores $\mathcal{A}$'s Test query only if $\mathcal{A}$ is testing a key which has already been revealed to it; in this case, $\mathcal{A}$'s query is disallowed by the game.

The advantage of $\mathcal{B}$ in the GKA game follows directly from the advantage of $\mathcal{A}$ in its own game. Assume that $\mathcal{B}$ correctly guesses the instance sid^* which $\mathcal{A}$ tests (meaning $\hat{\mathsf{sid}} = \mathsf{sid}^*$); this occurs with probability at least $\frac{1}{n_S}$. When $\mathcal{B}$'s challenger's bit $b_{\mathsf{gka}} = 0$, $\mathcal{A}$'s environment is exactly $\mathsf{Game0}$. When $\mathcal{B}$'s challenger's bit $b_{\mathsf{gka}} = 1$, $\mathcal{A}$'s environment is exactly $\mathsf{Game1}$. When $\mathcal{A}$ outputs $b' = b$, $\mathcal{B}$ outputs 1. When $\mathcal{A}$ outputs $b' \neq b$, $\mathcal{B}$ outputs 0. It follows that $\mathcal{B}$'s advantage is lowerbounded by the probability that $\mathcal{B}$ guesses the instance correctly times the advantage of $\mathcal{A}$ in distinguishing $\mathsf{Game0}$ and $\mathsf{Game1}$.

$$\mathsf{Adv}^{\mathsf{grm}}(\mathcal{B}) \geq \frac{1}{n_S} |\Pr[\mathcal{A} \text{ wins } \mathsf{Game1}] - \Pr[\mathcal{A} \text{ wins } \mathsf{Game0}]| \tag{E.1}$$

□

We next proceed to the difference between $\mathsf{Game1}$ and $\mathsf{Game2}$.

Lemma E.2. *There exists an adversary $\mathcal{C}$ for GRM such that* $\mathsf{Adv}^{\mathsf{grm}}(\mathcal{C}) \geq \frac{1}{n_S n} |\Pr[A \textit{ wins } \mathsf{Game2}] - \Pr[A \textit{ wins } \mathsf{Game1}]|$.

Proof. Recall that $\mathsf{Game1}$ is the GM game, except that the initial group key is replaced with a random key on the test instance. Moreover, $\mathsf{Game2}$ is just like $\mathsf{Game1}$, except that some key update for the key under Test is swapped for a random update.

The GRM adversary $\mathcal{C}$ simulates the GM oracle queries for $\mathcal{A}$ by (a) internally simulating a GKA execution and (b) forwarding all GRM queries to its own challenger. Note that because the GRM adversary $\mathcal{C}$ now internally simulates all GKA instances for $\mathcal{A}$, it therefore knows the random initial group key for GM. (This is because $\mathcal{C}$ samples the long term keys for $\mathcal{A}$'s GM game, and therefore knows them for the execution of GKA.) Starting with the initial random key $k_{\mathsf{sid}}^{(0)}$ for each session sid, $\mathcal{C}$ initializes its GRM oracles and plays the GRM game. $\mathcal{C}$ responds to $\mathcal{A}$'s encryption queries by tracking all of the keys and key updates requested by $\mathcal{A}$, and encrypting messages under the appropriate keys. When $\mathcal{A}$ makes a query $\vec{c} \leftarrow \Pi_{U,i}^{\mathsf{gm}}.\mathsf{Evolve}()$, $\mathcal{C}$ forwards the query to $\Pi_{U,i}^{\mathsf{grm}}.\mathsf{Evolve}()$. When any ciphertext $c_V \in \vec{c}$ is submitted to $\Pi_{V,i}^{\mathsf{gm}}.\mathsf{Recv}(c_V)$, $\mathcal{C}$ invokes $x \leftarrow \Pi_{U,i}^{\mathsf{grm}}.\mathsf{Recv}(c, \mathsf{dec\_flag} = 1)$, and $\mathcal{C}$ then uses this value of x as the decryption of every $c' \in \vec{c}$. Because $\mathcal{C}$ knows the initial group key $k_{\mathsf{sid}}^{(0)}$ for session sid and every x, it can "fill out" the entire key lattice $L_{\mathsf{sid}}^{\mathsf{rev}}$ defined by the execution, and therefore it can derive every encryption key used in the session.

$\mathcal{C}$ adapts this strategy slightly in order to tie its $\mathsf{Test}()$ query to $\mathcal{A}$'s, and therefore derives an advantage in its game from $\mathcal{A}$'s advantage. Recall that $\mathcal{A}$ sends a pair of messages (m_0, m_1) to its challenger, receives the encryption of m_b under some party U^*'s latest key (where b is the bit sampled by the challenger), and must guess whether $b = 0$ or $b = 1$. Let $\mathbf{i}^*$ be the maximal index of a defined key in U^*'s state; this is the key k^* for U^*'s challenge. Let sid^* be the session in which $\mathcal{A}$ queries its $\mathsf{Test}()$ oracle. To respond to $\mathcal{A}$'s query, $\mathcal{C}$ must encrypt one of $\mathcal{A}$'s two messages under k^*. However, instead of faithfully encrypting $\mathcal{A}$'s query with the appropriate key $\mathcal{C}$ chooses an edge e along a path from $\mathbf{0}$ to $\mathbf{i}^*$, and calls $(x_0, x_1) \leftarrow \Pi_{U^*,\mathsf{sid}^*}^{\mathsf{grm}}.\mathsf{Test}(c)$, where c corresponds to the ciphertext encrypting U^*'s true update along the edge e. $\mathcal{C}$ samples x_β for a random $\beta \in \{0, 1\}$, and replaces the true update along e with x_β. If $\mathcal{C}$ guesses the correct update (β is the same as its challenger's test bit), then $\mathcal{A}$'s environment is exactly $\mathsf{Game1}$. If $\mathcal{C}$ guesses the random update, then $\mathcal{A}$'s environment is exactly $\mathsf{Game2}$.

$\mathcal{C}$'s full strategy therefore deviates from learning every update as follows. At the beginning of the game, $\mathcal{C}$ uniformly at random chooses some $\hat{U} \in \mathcal{P}$ and some $\hat{\mathsf{sid}} \in n_S$. When $\mathcal{A}$ makes its first call to $\Pi^{\mathsf{gm}}_{\hat{U},\hat{\mathsf{sid}}}.\mathsf{Evolve}()$, $\mathcal{C}$ queries $\vec{c} \leftarrow \Pi^{\mathsf{grm}}_{\hat{U},\hat{\mathsf{sid}}}.\mathsf{Evolve}()$. $\mathcal{C}$ then immediately makes its $\mathsf{Test}()$ query by choosing a $c \in \vec{c}$ and invoking $(x_0, x_1) \leftarrow \Pi^{\mathsf{grm}}_{\hat{U},\hat{\mathsf{sid}}}.\mathsf{Test}(c)$. $\mathcal{C}$ then randomly samples $\beta \in \{0,1\}$ and sets x_β as the value of the update corresponding to U's evolution along that edge in L^i for the duration of the game. If $\mathcal{C}$'s Test query corresponds to an update on the path from $\mathbf{0}$ to $\mathbf{i}^*$, then when $\mathcal{A}$ outputs a bit $b \in \{0,1\}$ for its game, $\mathcal{C}$ responds with the same bit. If $\mathcal{C}$'s Test query does not correspond to an update on the path to $\mathbf{i}^*$ or $\mathcal{C}$ guesses the wrong instance $\hat{\mathsf{sid}}$ ($\hat{\mathsf{sid}} \neq \mathsf{sid}^*$), then $\mathcal{C}$ outputs a uniformly random bit.

We remark here that for technical reasons, because $\mathsf{Game1}$ requires that the $\mathsf{Test}()$ instance have a random $k^{(0)}$, $\mathcal{C}$ must guess the session sid^* on which $\mathcal{A}$ will call its $\mathsf{Test}()$ query at the beginning of the simulation. $\mathcal{C}$ samples a random session $\hat{\mathsf{sid}}$, and samples a random key $k^{(0)}_{\hat{\mathsf{sid}}}$ independent of the execution of GKA for that session. ($\mathcal{C}$ is correct if $\hat{\mathsf{sid}} = \mathsf{sid}^*$.)

We now provide a full description of $\mathcal{C}$. $\mathcal{C}$ uniformly at random chooses some $\hat{U} \in \mathcal{P}$ and some $\hat{\mathsf{sid}} \in n_S$. It then proceeds as follows:

- $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Init}(G, w)$:
    - $\mathcal{C}$ sets $\delta_{U,i} \leftarrow \mathsf{pending}$, $\rho_{U,i} \leftarrow \perp$ and $\kappa_{U,i} \leftarrow \perp$.
    - $\mathcal{C}$ simulates $\Pi^{\mathsf{gka}}_{U,i}.\mathsf{Init}(G)$ and forwards to $\mathcal{A}$ anything that is returned
- $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Corrupt}()$:
    - $\mathcal{C}$ sets $\kappa_U \leftarrow \mathsf{corrupted}$.
    - $\mathcal{C}$ returns $\mathtt{sk}_U$ to $\mathcal{A}$, and mark all vertices and edges that are computable from messages that have been delivered to U as red in $L^{\mathsf{rev}}_{\mathsf{sid}}$. If no key has yet been derived (it must be the case that this party has not yet completed GKA in session i), then once this party derives the initial group key, mark it as red in $L^{\mathsf{rev}}_{\mathsf{sid}}$. Every

future GKA key learned by U in another session is revealed in the corresponding $L^{\mathsf{rev}}_{\mathsf{sid}}$.

– $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Evolve}()$:

  – $\mathcal{C}$ calls $\vec{c} \leftarrow \Pi^{\mathsf{grm}}_{U,i}.\mathsf{Send}()$

  – if $U = \hat{U}$ and (U, i) maps to $\hat{\mathsf{sid}}$, then $\mathcal{C}$ chooses an appropriate $c \in \vec{c}$ (corresponding to the ciphertext intended for any $V \neq U$) and computes $(x_0, x_1) \leftarrow \Pi^{\mathsf{grm}}_{V,i}.\mathsf{Test}(c)$. $\mathcal{C}$ samples $\beta \leftarrow \{0, 1\}$ uniformly at random, and applies x_β to the lattice $L^{\mathsf{rev}}_{\mathsf{sid}}$ on the edge corresponding to U's update. (This edge is still black in $L^{\mathsf{rev}}_{\mathsf{sid}}$.)

  – Otherwise, U chooses the appropriate $c \in \vec{c}$ (corresponding to the encryption of the update for U), and computes $x \leftarrow \Pi^{\mathsf{grm}}_{U,i}.\mathsf{Recv}(c, \mathsf{dec\_flag} = 1)$. $\mathcal{C}$ then applies x to the lattice $L^{\mathsf{rev}}_{\mathsf{sid}}$ (for sid mapped by (U, i)) on the edge corresponding to U's update.

– $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Reveal}()$:

  – If $\delta_{U,i} = \mathsf{abort}$, then $\mathcal{C}$ returns $\perp$.

  – If $\delta_{U,i} = \mathsf{pending}$ then $\mathcal{C}$ emulates the call $\Pi^{\mathsf{gka}}_{U,i}.\mathsf{Reveal}()$ for its simulation of U's view in GKA instance (U, i) and returns the response to $\mathcal{A}$.

  – If $\delta_{U,i} = \mathsf{accept}$, then $\mathcal{C}$ computes the set of pairs $(\mathbf{i}, \mathsf{k})$ from $\mathcal{L}_U$ (for sid mapped by (U, i)) corresponding to the vertices and keys in (U, i)'s local state. Let this set be R. If $\mathsf{Test}()$ has already been called and $(\mathbf{i}^*, \mathsf{k}^*)$ is included in R, then $\mathcal{C}$ ignores the query. Otherwise, $\mathcal{C}$ returns R to $\mathcal{A}$ and colors all of the vertices in R as red in $L^{\mathsf{rev}}_{\mathsf{sid}}$.

– $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{StateReveal}()$:

  – If $\delta_{U,i} = \mathsf{abort}$, then $\mathcal{C}$ returns $\perp$.

– If $\delta_{U,i} = \textsf{pending}$ then $\mathcal{C}$ emulates the call $\Pi^{\textsf{gka}}_{U,i}.\textsf{StateReveal}()$ for its simulation of U's view in GKA instance (U, i) and returns the response to $\mathcal{A}$.

– If $\delta_{U,i} = \textsf{accept}$, then $\mathcal{C}$ computes the set of edges $\mathcal{E}_U$ which are defined in (U, i)'s current state. If $\textsf{Test}()$ has already been called and coloring all of the edges of $\mathcal{E}_U$ red in $L^{\textsf{rev}}_{\textsf{sid}}$ would also color the vertex at $\mathbf{i}^*$ red, then $\mathcal{C}$ ignores the call. Otherwise, $\mathcal{C}$ returns E to $\mathcal{A}$ and colors all of the edges in E red in $L^{\textsf{rev}}_{\textsf{sid}}$ (for the session $\textsf{sid}$ mapped by (U, i)).

– $\Pi^{\textsf{gm}}_{U,i}.\textsf{Enc}(M)$: If $\delta^i_U \neq \textsf{accept}$ then set $\delta^i_U = \textsf{abort}$ and return. Otherwise, let $\mathbf{i}$ the maximal index in U's local lattice (in instance (U, i)), and let $\textsf{k}_{\mathbf{i}}$ be the key defined at that index. $\mathcal{C}$ computes $(\texttt{ct}, t) \leftarrow \textsf{AEAD.Enc}(m, U \| \mathbf{i}, ; k_{\mathbf{i}})$, and returns $(\texttt{ct}, U \| \mathbf{i}, t)$ to $\mathcal{A}$.

– $\Pi^{\textsf{gm}}_{U,i}.\textsf{Dec}(M)$: If $\delta^i_U = \textsf{abort}$, then $\mathcal{C}$ ignores the query. $\mathcal{C}$ parses M as $(\texttt{ct}, V \| \mathbf{i}, t)$. If M is not of this form, $\mathcal{C}$ returns $\perp$. Let $\textsf{k}$ be the key at index $\mathbf{i}$ in U's local state. If $\textsf{k}$ is not computable, then buffer M. If $\textsf{k}$ is computable from U's state, then $\mathcal{C}$ computes $m \leftarrow \textsf{AEAD.Dec}(\texttt{ct}, V \| \mathbf{i}, t; k_{\mathbf{i}})$. If $m =\perp$, $\mathcal{C}$ sets $\delta^i_U = \textsf{abort}$. Otherwise, $\mathcal{C}$ returns m to $\mathcal{A}$.

– $\Pi^{\textsf{gm}}_{U,i}.\textsf{Recv}(M)$:

– If $\delta_{U,i} = \textsf{abort}$, $\mathcal{B}$ does nothing.

– Else if M is a GKA message (contains a header $\textsf{gka}$) and $\delta_{U,i} = \textsf{pending}$, $\mathcal{C}$ simulates the execution of $\Pi^{\textsf{gka}}_{U,i}.\textsf{Recv}(M)$. If $\Pi^{\textsf{gka}}_{U,i}.\textsf{Recv}(M)$ does not output $\textsf{done}$, then $\mathcal{C}$ returns to $\mathcal{A}$ anything that is returned. If $\textsf{done}$ is returned, then $\mathcal{C}$ derives $k^{(0)}_{\textsf{sid}}$ (for the session $\textsf{sid}$ mapped by (U, i). Because $\mathcal{C}$ simulated this execution, it knows $k^{(0)}_{\textsf{sid}}$; recall as well that if $\textsf{sid} = \hat{\textsf{sid}}$, then $\mathcal{C}$ samples a uniformly random key $k^{(0)}_{\textsf{sid}}$ independent of the simulated GKA execution. $\mathcal{C}$ then calls $\Pi^{\textsf{grm}}_{U,i}.\textsf{Init}(k^{(0)}_{\textsf{sid}}, w)$, and returns to $\mathcal{A}$ any messages that were returned, except for the initial group key $k^{(0)}_{\textsf{sid}}$.

- Else if M is a GRM message (contains a header grm) and $\delta_{U,i} = \mathsf{accept}$:
  * if $\mathsf{Test}()$ has been called, and M is a ciphertext c' which was output in the same set of ciphertexts as the ciphertext on which $\mathsf{Test}()$ was called, then $\mathcal{C}$ updates the state of U (in instance (U, i)) as if calling $\mathsf{GM.Recv}$ where x_β is returned from $\mathsf{GRM.Recv}$, where x_β was the update chosen by $\mathcal{C}$ after receiving its challenge.
  * Otherwise, $\mathcal{C}$ compute $x \leftarrow \Pi^{\mathsf{grm}}_{U,i}.\mathsf{Recv}(M)$. If $x =\perp$, then do nothing. Otherwise, update U's local state (in instance (U, i)) as in the description of $\mathsf{GM.Recv}$ letting x be the decrypted update by adding x to U's set of edges E, propagating the changes by computing all additional keys in L, and forgetting keys as described in $\mathsf{GM.Recv}$.

- $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Test}(m_0, m_1)$:
  - if $\delta_{U,i} \neq \mathsf{accept}$ then return $\perp$
  - Let $\mathbf{i}^* = \mathbf{i}_U$ such that $\mathbf{i}_U$ is the maximal index in U's local state. If $\mathbf{i}^*$ is red in L^{sid} (for sid mapped by (U, i)) then ignore the query. Otherwise, $\mathcal{C}$ computes $\mathsf{k}^* = \mathsf{k}_U$ such that k_U is the key corresponding to $\mathbf{i}_U$.
  - $\mathcal{C}$ samples $b \xleftarrow{\$} \{0, 1\}$, computes $(\mathtt{ct}, t) \leftarrow \mathsf{AEAD.Enc}(m_b, U \| \mathbf{i}^*, ; \mathsf{k}^*)$ and returns $(\mathtt{ct}, U | \mathbf{i}^*, t)$ to $\mathcal{A}$.

We now analyze the advantage of $\mathcal{C}$ in winning the GRM game. $\mathcal{C}$ derives an advantage from $\mathcal{A}$'s ability to distinguish between $\mathsf{Game}1$ and $\mathsf{Game}2$ precisely when the edge that $\mathcal{C}$ chooses for its $\mathsf{Test}()$ query corresponds to an update used to compute k^*. We note that it may be the case that $\mathcal{A}$ never evolves the GM key, and always invokes $\mathsf{Test}()$ on the initial group key. This case is irrelevant to the lemma, as $\mathcal{A}$'s advantage can be shown to break either GKA (the first game hop) or CCA (the final game hop). Assume that $\mathcal{A}$ chooses to evolve the key at least once. Then there must be at least one party U on whose oracle $\mathcal{A}$

calls $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Evolve}()$. $\mathcal{C}$ derives an advantage from $\mathcal{A}$ if $\mathcal{C}$ correctly guesses this party, which it does with probability at least $\frac{1}{n}$. Additionally, $\mathcal{C}$ must correctly guesses the session on which $\mathcal{A}$ will execute its $\mathsf{Test}()$ query; this occurs with probability at least $\frac{1}{n_S}$. It follows that

$$\mathsf{Adv}^{\mathsf{grm}}(\mathcal{C}) \geq \frac{1}{n \cdot n_S} |\Pr[\mathcal{A} \text{ wins } \mathsf{Game2}] - \Pr[\mathcal{A} \text{ wins } \mathsf{Game1}]| \tag{E.2}$$

□

Lemma E.3. *There exists an adversary $\mathcal{D}$ for CCA such that* $\mathsf{Adv}^{\mathsf{gka}}(\mathcal{D}) = \frac{1}{n_S \psi} |\Pr[A \textit{ wins } \mathsf{Game2}] - \frac{1}{2}|$.

Proof. $\mathcal{D}$ uses $\mathcal{A}$'s advantage in the GM game in order to break the CCA security of an encryption scheme as follows. $\mathcal{D}$ emulates the entire execution of GM for $\mathcal{A}$, and forwards only encryptions and decryptions on its own challenge key to $\mathcal{A}$. Specifically, $\mathcal{D}$ samples long term public and private keys for all parties, emulates each GKA execution for $\mathcal{A}$, and learns the initial key $k^{(0)}$ for every group of oracles. $\mathcal{D}$ then simulates the execution of GRM internally, and learns every key evolution output by GRM. $\mathcal{D}$ uses this information to construct a key lattice L for each execution and label every vertex and edge on the lattice with the appropriate key and update, respectively. When $\mathcal{A}$ requests an encryption or decryption, $\mathcal{D}$ uses its knowledge of the key lattice to compute the encryption; specifically, computes the lattice key at the index that $\mathcal{A}$ queries, and evaluates H in order to compute the encryption key corresponding to that point. Similarly, when $\mathcal{A}$ requests a decryption of a message, $\mathcal{D}$ uses its knowledge of the key to decrypt $\mathcal{A}$'s message. For technical reasons as in the previous lemma, $\mathcal{D}$ also chooses some update output by $\mathsf{Evolve}()$ to be replaced with a random update; this is chosen as described below such that the test key depends on the random update.

$\mathcal{D}$ only does not answer encryption queries on its own when $\mathcal{A}$ issues queries to Enc,Dec, or Test on $\mathbf{i}^*$, where $\mathbf{i}^*$ is defined as the current lattice index of the oracle on which $\mathcal{A}$ issues $\Pi^{\mathsf{gm}}_{U,i}.\mathsf{Test}()$. When $\mathcal{A}$ makes encryption or decryption requests for $\mathsf{k}_{\mathbf{i}^*}$, $\mathcal{D}$ forwards the requests to its own challenger. Note that because the key at $\mathsf{k}_{\mathbf{i}^*}$ is randomly distributed from the view of $\mathcal{A}$, and because $\mathcal{D}$'s challenger selects the encryption key at random, the responses of $\mathcal{D}$'s challenger are distributed just as $\mathcal{A}$'s challenger in its game. Specifically, when $\mathcal{A}$ makes its Test query, $\mathcal{D}$ forwards the messages (m_0, m_1) to its own challenger, and it returns the resulting encryption to $\mathcal{A}$; when $\mathcal{A}$ outputs a bit $b' \in \{0, 1\}$, $\mathcal{D}$ outputs the same bit, and wins with the same probability that $\mathcal{A}$ wins.

However, $\mathcal{D}$ does not know the key on which $\mathcal{A}$ will call $\mathsf{Test}()$, and moreover it cannot wait for $\mathcal{A}$ to call $\mathsf{Test}()$ to begin forwarding queries to its own adversary, because $\mathcal{A}$ may request encryptions under some key before it calls $\mathsf{Test}()$ on that key. Therefore, $\mathcal{D}$ must guess the key on which $\mathcal{A}$ will call $\mathsf{Test}()$. First, $\mathcal{D}$ uniformly at random guesses the session on which $\mathcal{A}$ will call $\mathsf{Test}()$, and guesses correctly with probability $\frac{1}{n_S}$. Second, $\mathcal{D}$ guesses the key within that session on which $\mathcal{A}$ will guess $\mathsf{Test}()$. Observe that when $\mathcal{A}$ makes n $\mathsf{Evolve}()$ queries to define new keys, in fact there are 2^n keys defined. This is an exponential number of keys. However, $\mathcal{A}$ must be a polynomial-time adversary and therefore can only explore ψ keys by making queries to them. $\mathcal{D}$ therefore guesses that *this* key will be the one tested by $\mathcal{A}$ with probability $\frac{1}{\psi - l}$ each time that $\mathcal{C}$ issues a query to a new key, where l is a counter that tracks how many times $\mathcal{A}$ has challenged a key. This scheme adaptively guesses the next key uniformly at random with total probability $\frac{1}{\psi}$ for each key.

It follows that

$$\mathsf{Adv}^{\mathsf{cca}}(\mathcal{D}) \geq \frac{1}{n_S \cdot \psi} |\Pr[\mathcal{A} \text{ wins } \mathsf{Game2}] - \frac{1}{2}| \tag{E.3}$$

□

Using the above lemmas we complete the proof by computing the advantage of the adversaries

$\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$ with respect to $\mathcal{A}$.

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{gm}}(\mathcal{A}) &= 2 \cdot |\Pr[\mathcal{A} \text{ wins } G_0] - \frac{1}{2}| \\
&= 2 \cdot |\Pr[\mathcal{A} \text{ wins } G_0] - \frac{1}{2} + \Pr[\mathcal{A} \text{ wins } G_1] - \Pr[\mathcal{A} \text{ wins } G_1] \\
&\qquad + \Pr[\mathcal{A} \text{ wins } G_2] - \Pr[\mathcal{A} \text{ wins } G_2]| \\
&\leq 2 \cdot |\Pr[\mathcal{A} \text{ wins } G_0] - \Pr[\mathcal{A} \text{ wins } G_1]| \\
&\qquad + 2 \cdot |\Pr[\mathcal{A} \text{ wins } G_1] - \Pr[\mathcal{A} \text{ wins } G_2]| \\
&\qquad + 2 \cdot |\Pr[\mathcal{A} \text{ wins } G_2] - \frac{1}{2}| \\
&\leq 2 \cdot n_S \cdot \mathsf{Adv}^{\mathsf{gka}}(\mathcal{B}) + 2 \cdot n_S \cdot n \cdot \mathsf{Adv}^{\mathsf{grm}}(\mathcal{C}) + 2 \cdot n_S \cdot \psi \cdot \mathsf{Adv}^{\mathsf{cca}}(\mathcal{D})
\end{aligned}$$

□

Appendix F

Full Model and Proofs for Depth-Secure Computation

F.1 Model for Depth-Secure MPC

This appendix is an extension of Section 7.5. Here, we discuss in more detail the execution of the ideal model.

F.1.1 Execution Model

Our execution model is based on a simpler version of the Universal Composability (UC) framework, modified for our application scenario and depth-bounded computation. In our execution model, all parties (including the environment, trusted third party, and adversary) are modeled as interactive circuits.

The Environment: As in the UC framework, we consider an execution in the presence of an *environment* that provides inputs to parties and reads their outputs. The environment directs the execution by proceeding in rounds. It delivers inputs to parties, activates each party in every round, and delivers messages between parties. The environment controls the time elapse of an execution via the number of protocol rounds it has directed. In every round, each party is permitted to perform computation of the same depth.

When the adversary is activated, it learns the corrupt parties' inputs, the queries they send, and the responses they receive. In the beginning of the execution, the adversary informs the environment of the identities of the parties it wishes to corrupt. The environment responds with the corrupt parties' inputs, and the adversary may choose new inputs for the corrupt parties based on the provided inputs and its auxiliary information. (This models the fact that inputs for corrupted parties may be adversarially selected, which is in the application scenario of accountable computation.)

The environment activates the adversary after activating all other parties in each round, informing the adversary of the messages the corrupt parties send and the responses they receive. The adversary can respond to the environment, including by corrupting additional parties.

Defining a View: The *view* of any party is defined to be the ordered list of inputs and events it receives from the environment, along with the ordered list of messages it receives from other parties. Formally, we denote the view of party i in an execution of protocol Π on inputs $\vec{x}$ and security parameter λ as $\mathsf{View}_i^{\Pi}(\vec{x}, \lambda) = (x_i; r; \vec{m})$, where x_i is party i's input, r is the party's randomness, and $\vec{m}$ is the set of messages that party i receives from other parties and the environment.

F.1.2 The Ideal/Real Paradigm

Execution in the Ideal Model.

We define an ideal model in which parties interact with a trusted third party in an execution that is secure by definition.

Interaction with the Trusted Party In an ideal execution, the parties interact with a trusted party as follows:

1. **Initialization:** The adversary $\mathcal{A}$ receives an auxiliary input z, and may choose to corrupt some parties. It informs $\mathcal{Z}$ of the corruptions.

2. **Inputs:** The environment sends the corrupt parties' inputs to $\mathcal{A}$, which choose new inputs for the corrupted parties based on its auxiliary information and the inputs provided by the environment. It then forwards the new inputs to the environment. All parties then receive inputs from the environment.

3. **Send Inputs to Trusted Party:** Each party sends its input x_i to the trusted party.

4. **Computing Functionalities:** After receiving all inputs, the trusted third party computes the functionality outputs over the provided inputs and saves the outputs.

5. **Phased Output Release:** An execution is divided into *phases* such that at the end of each phase, the parties learn some information from the trusted party. The moment that the trusted party provides the protocol participants with their ith message denotes the end of the ith phase and the beginning of the $i+1$st phase.

6. **Protocol Outputs:** At the end of an execution, honest parties output whatever they have received from the trusted party. Corrupt parties output nothing, and the

adversary outputs an arbitrary function of its input, the messages it has received from the environment, and the messages that corrupt parties have received from the trusted party. The environment learns every output.

The random variable $\mathsf{IDEAL}_{\mathcal{F},\mathcal{A}(z),\mathcal{Z}}(\overline{x})$ denotes the output of the environment in an *ideal execution* of functionality $\mathcal{F}$ on honest inputs $\overline{x}$, auxiliary input z to $\mathcal{A}$, with environment $\mathcal{Z}$.

Execution in the Real Model.

In the real model, participants execute a protocol Π to compute the desired functionality $\mathcal{F}$ without a trusted party. At the end of the execution, honest parties output their protocol outputs. The corrupt parties output nothing. The adversary outputs an arbitrary function of its inputs and the messages that corrupt parties have received.

The random variable $\mathsf{REAL}_{\Pi,\mathcal{A}(z),\mathcal{Z}}(\overline{x})$ denotes the output of the environment in a real execution of Π with honest inputs $\overline{x}$, auxiliary input z to $\mathcal{A}$, with environment $\mathcal{Z}$. The environment learns every output.

F.2 Sequential Composition of Depth-Secure Protocols: Proof of Theorem 7.4

In this section, we provide the full proof of Theorem 7.4, which we restate below for convenience.

Theorem 7.4 (Sequential Composition of Two Depth-Secure Protocols). *Let Π (d_a, d_s, d_e)-depth-securely compute F in the G-hybrid model, and let ρ (d'_a, d'_s, d'_e)-depth-securely compute G. Π^ρ $(d_a - d'_s, d_s \cdot d'_s, \mathsf{min}(d_e, d'_e))$-depth-securely computes F.*

Notation. For the proof of Theorem 7.4, we require notation to describe the distribution of executions in the ideal world for a fixed simulator, fixed distinguisher, and fixed inputs, making explicit the adversary. Let $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S}(z)}(\overline{x})$ denote the distribution of executions of the naive protocol in the ideal world that calls functionality $\mathcal{F}$, with simulator $\mathcal{S}$ and advice string z, on honest inputs $\overline{x}$. (In this experiment, the parties forward their inputs the ideal functionality, and the simulator generates a view for $\mathcal{A}$ that is indistinguishable from the real experiment.)

Proof. The proof will use the simulators $\mathcal{S}^{\Pi}$ for Π and $\mathcal{S}^{\rho}$ for ρ to construct a new simulator $\mathcal{S}$ for Π^{ρ} such that $\mathcal{S}$ is $(d_s \cdot d'_s)$-depth bounded, and for every $(d_a - d'_s)$-depth $\mathcal{A}$, and every $\mathsf{min}(d_e, d'_e)$-depth $\mathcal{Z}$, the distributions $\mathsf{REAL}_{\Pi^{\rho},\mathcal{A}(z)}(\overline{x})$ and $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S}(z)}(\overline{x})$ are $\mathsf{min}(d_e, d'_e)$-depth indistinguishable.

The simulator $\mathcal{S}$ works by composing the simulators $\mathcal{S}^{\Pi}$ and $\mathcal{S}^{\rho}$. Specifically, to simulate an execution of Π^{ρ} up to the point that ρ is called, $\mathcal{S}$ runs $\mathcal{S}^{\Pi}$. When ρ is called, $\mathcal{S}$ invokes $\mathcal{S}^{\rho}$. After ρ terminates, $\mathcal{S}$ resumes $\mathcal{S}^{\Pi}$.

Claim F.1. *$\mathcal{S}$'s depth is bounded by $d_s \cdot d'_s$.*

Proof. The claim follows from the observation that every time $\mathcal{S}^{\Pi}$ is rewound, $\mathcal{S}^{\rho}$ must also be rewound the maximum number of times. If $\mathcal{S}^{\Pi}$'s running time is at most d_s, then for each rewinding of $\mathcal{S}^{\Pi}$, $\mathcal{S}^{\rho}$ must be rewound at most d'_s times. The total run-time of $\mathcal{S}$ is thus $d_s \cdot d'_s$. □

We proceed with our main lemma, which completes the proof:

Lemma F.1. *For every $(d_a - d'_s)$-depth adversary $\mathcal{A}$, and every $\overline{x} \in (\{0,1\}^{\mathsf{poly}})^n$ and $z \in \{0,1\}^{\mathsf{poly}}$ the distributions $\mathsf{REAL}_{\Pi^{\rho},\mathcal{A}(z)}(\overline{x})$ and $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S}(z)}(\overline{x})$ are $\mathsf{min}(d_e, d'_e)$-depth indistinguishable.*

Proof Sketch: If there is an adversary $\mathcal{A}$ and a distinguisher $\mathcal{D}$ that distinguishes the above two distributions, then we create another adversary $\mathcal{B}$ and distinguisher E that isolates an attack against the caller protocol Π in the G-hybrid model. $\mathcal{B}$ runs $\mathcal{A}$ as a black box, and when Π must call ρ, $\mathcal{B}$ simply simulates an execution of ρ (using $\mathcal{S}^\rho$), feeding messages to $\mathcal{A}$ so that $\mathcal{A}$ believes it is running a full execution of Π^ρ. Similarly, E is provided with the execution transcript generated by $\mathcal{B}$, with the call to ρ in the transcript replaced by the simulated output generated by $\mathcal{B}$. Because the transcript of the simulation of ρ is indistinguishable from a real execution by assumption, this attack must distinguish an execution of Π in the real model from its simulation, contradicting the security of Π.

Proof. Assume to the contrary that the lemma statement is false. Then there exists a $(d_a - d'_s)$-depth adversary $\mathcal{A}$, a $\mathsf{min}(d_e, d'_e)$-depth distinguisher $\mathcal{D}$, and inputs $\overline{x}, z$ such that the distributions $\mathsf{REAL}_{\Pi^\rho,\mathcal{A}(z)}(\overline{x})$ and $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S}(z)}(\overline{x})$ are $\mathsf{min}(d_e, d'_e)$-depth distinguishable (for any $(d_s \cdot d'_s)$-depth $\mathcal{S}$).

We will show how to use $\mathcal{A}$ for Π^ρ in order to build an adversary $\mathcal{B}$ to contradict the (d_a, d_s, d_e)-security of Π in the G-hybrid model.

In an execution of Π in the G-hybrid model, $\mathcal{B}$ works as follows:

1. Until the point at which G is invoked, $\mathcal{B}$ runs $\mathcal{A}$ as a black box, forwarding any messages output by $\mathcal{A}$
2. When G is invoked, $\mathcal{B}$ submits its input y to G and receives some output w. $\mathcal{B}$ runs the simulator $\mathcal{S}^\rho(y, w)$ for ρ, forwarding messages provided by the simulator to $\mathcal{A}$, and forwarding the replies by $\mathcal{A}$ to $\mathcal{S}^\rho$ to continue the simulation.
3. After $\mathcal{S}^\rho$ terminates, $\mathcal{B}$ resumes calling $\mathcal{A}$ as a black box given messages from its execution of Π. $\mathcal{B}$ outputs whatever $\mathcal{A}$ outputs.

Claim F.2. *$\mathcal{B}$ runs in depth at most d_a.*

Proof. $\mathcal{B}$ runs the adversary $\mathcal{A}$ as a black box, which requires depth at most $d_a - d'_s$. $\mathcal{B}$ also runs the simulator $\mathcal{S}^\rho$, which requires depth at most d'_s. (Recall that we have already counted the depth of rewinding $\mathcal{A}$ during this step towards the depth d'_s.) The sum of the two run-times is $d_a - d'_s + d'_s = d_a$ which concludes the claim. □

We proceed to compare the views of the adversary $\mathcal{A}$ when it is running in its own execution, or being called by $\mathcal{B}$. Let $\mathsf{VIEW}_{\mathcal{A}}(\mathsf{REAL}_{\Pi^\rho,\mathcal{A}(z)}(\overline{x}))$ denote the view of $\mathcal{A}$ in a real execution of Π^ρ, and let $\mathsf{VIEW}_{\mathcal{A}}(\mathsf{REAL}_{\Pi^G,\mathcal{A}(z)}(\overline{x}))$ denote the view of $\mathcal{A}$ in a real execution of Π in the G-hybrid model, in which $\mathcal{B}$ calls $\mathcal{A}$. Similarly, we denote by $\mathsf{VIEW}_{\mathcal{A}}(\mathsf{IDEAL}^{\mathcal{B}}_{\mathcal{F},\mathcal{S}(z)}(\overline{x}))$ the view of $\mathcal{A}$ in support of the ideal experiment in which $\mathcal{B}$ calls $\mathcal{A}$, and $\mathcal{S}$ runs both the simulators for Π and for ρ; and we denote by $\mathsf{VIEW}_{\mathcal{A}}(\mathsf{IDEAL}^{\mathcal{B}}_{\mathcal{F},\mathcal{S}^\Pi(z)}(\overline{x}))$ the view of $\mathcal{A}$ in support of the ideal experiment in which $\mathcal{B}$ must call the simulator for ρ.

Claim F.3. *Let $f' = \mathsf{min}(d_e, d'_e)$. For all $\overline{x} \in (\{0,1\}^{\mathsf{poly}})^n$ and $z \in \{0,1\}^{\mathsf{poly}}$*

$$\mathsf{VIEW}_{\mathcal{A}}(\mathsf{REAL}_{\Pi^\rho,\mathcal{A}(z)}(\overline{x})) \overset{f'}{\approx} \mathsf{VIEW}_{\mathcal{A}}(\mathsf{REAL}_{\Pi^G,\mathcal{A}(z)}(\overline{x})\})$$

Proof. The difference between the two distributions is that on the right, $\mathcal{B}$ simulates an execution of ρ using the simulator $\mathcal{S}^\rho$ and provides those messages to $\mathcal{A}$, and then continues to call $\mathcal{A}$ after the call to $\mathcal{S}^\rho$ using messages from its real execution. By assumption, $\mathcal{A}$ is $(d_a - d'_s)$-depth-bounded and $d_a < d'_e$. Therefore, $\mathcal{A}$ must not be able to distinguish the messages in the real execution of ρ on the left from the simulation on the right. The claim follows from the additional fact that all other messages in $\mathcal{A}$'s view are distributed identically in both experiments, since they are from the real execution of Π. □

We make another claim that $\mathcal{A}$ cannot distinguish between an idealized execution of $\mathcal{F}$ in which $\mathcal{S}$ generates its view of the execution and an idealized execution of $\mathcal{F}$ in which $\mathcal{B}$ interacts with $\mathcal{S}^\Pi$ in the G-hybrid model, forwarding its messages to $\mathcal{A}$ and when $\mathcal{B}$'s execution of in the G-hybrid model invokes G, $\mathcal{B}$ runs $\mathcal{S}^\rho$ to generate a view for $\mathcal{A}$.

Claim F.4. *For all $\overline{x} \in (\{0,1\}^{\mathsf{poly}})^n$ and $z \in \{0,1\}^{\mathsf{poly}}$*

$$\mathsf{VIEW}_{\mathcal{A}}(\mathsf{IDEAL}^{\mathcal{B}}_{\mathcal{F},\mathcal{S}(z)}(\overline{x})) \equiv \mathsf{VIEW}_{\mathcal{A}}(\mathsf{IDEAL}^{\mathcal{B}}_{\mathcal{F},\mathcal{S}^{\Pi}(z)}(\overline{x}))$$

Proof. The proof is analogous to the previous. However, in this case, $\mathcal{B}$ perfectly simulates the execution of ρ in comparison to $\mathcal{A}$'s view in the ideal execution of Π^ρ, since $\mathcal{B}$ does exactly the same thing that $\mathcal{S}$ does: both run $\mathcal{S}^\rho$. □

To complete the proof, we describe how the distinguisher E is built from D. E simply runs D as a black box and outputs whatever D outputs.

Next we claim that D's view in support of $\mathsf{REAL}_{\Pi^\rho,\mathcal{A}(z)}(\overline{x})$ is $\mathsf{min}(d_e, d'_e)$-depth indistinguishable from its view in support of $\mathsf{REAL}_{\Pi,\mathcal{B}(z)}(\overline{x})$ (as forwarded by E). This follows from Claim F.3, due to the fact that $\mathcal{A}$'s views in support of the two distributions are $\mathsf{min}(d_e, d'_e)$-depth indistinguishable, and $\mathcal{D}$ sees the transcript of $\mathcal{A}$'s interaction with the real protocol, and $\mathcal{A}$'s outputs must not be distinguishable by the claim.

Similarly, $\mathcal{D}$'s view in support of $\mathsf{IDEAL}^{\mathcal{A}}_{\mathcal{F},\mathcal{S}(z),D}(\overline{x})$ is $\mathsf{min}(d_e, d'_e)$-depth indistinguishable from its view in support of $\mathsf{IDEAL}^{\mathcal{B}}_{\mathcal{F},\mathcal{S}^{\Pi}(z),E}(\overline{x})$ (as forwarded by E). This follows from Claim F.4, via the same argument as above.

It follows that if D distinguishes $\mathsf{REAL}_{\Pi^\rho,\mathcal{A}(z),D}(\overline{x})$ and $\mathsf{IDEAL}^{\mathcal{A}}_{\mathcal{F},\mathcal{S}(z),D}(\overline{x})$, then E distinguishes $\mathsf{REAL}_{\Pi,\mathcal{B}(z),E}(\overline{x})$ and $\mathsf{IDEAL}^{\mathcal{B}}_{\mathcal{F},\mathcal{S}^{\Pi}(z),E}(\overline{x})$. Notice that because $\mathcal{B}$'s depth is bounded by d_a (by Claim F.2), and because E's depth is bounded by $\mathsf{min}(d_e, d'_e)$ (by assumption toward contradiction, since E's depth is exactly D's depth), this contradicts the (d_a, d_s, d_e)-depth security of Π in the G-hybrid model. □

□