

Relationships Between Functionality, Security, and Privacy

by

Rio LaVigne

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2020

© Massachusetts Institute of Technology 2020. All rights reserved.

Author
Department of Electrical Engineering and Computer Science
January 16, 2020

Certified by
Vinod Vaikuntanathan
Associate Professor of Electrical Engineering and Computer Science
at MIT
Thesis Supervisor

Accepted by
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Relationships Between Functionality, Security, and Privacy

by
Rio LaVigne

Submitted to the Department of Electrical Engineering and Computer Science
on January 16, 2020, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

One of the core goals of cryptography is to be able to offer security and privacy without sacrificing functionality. This thesis describes three different functionalities and notions of security and privacy and how they interplay. First, we delve into Topology-Hiding Computation, which has a very restrictive definition of privacy making it difficult to construct; nevertheless we provide a two constructions. Second, we invent a new cryptographic concept called Property Preserving Hashes, where we want to compress inputs while securely preserving a certain property between them, e.g. hamming distance. Finally, we explore Fine-Grained Cryptography, and develop a public key cryptosystem. In this notion of cryptography, security takes on a much less restrictive role (e.g. adversaries must run in $O(n^{100})$ time), but the protocols and security reductions must run in “fine-grained” time, e.g. less than $O(n^2)$.

Thesis Supervisor: Vinod Vaikuntanathan

Title: Associate Professor of Electrical Engineering and Computer Science at MIT

Acknowledgments

This is the acknowledgements section. You should replace this with your own acknowledgements. TODO.

I would first like to thank my advisors Vinod Vaikuntanathan and Virginia Vassilevska Williams.

Tal Moran

Elette Boyle

Andrea Lincoln

Other coauthors

Dan Boneh

Other members of the theory community.

Other members of the crypto community.

Family and Avery <3

Contents

1	Introduction	13
1.1	Results	15
1.2	Outline of Thesis	18
2	Preliminaries	19
3	Topology Hiding Computation	21
3.1	Introduction	21
3.1.1	Topology Hiding Computation	21
3.1.2	Contributions	22
3.1.3	Related Work	26
3.1.4	Outline	27
3.2	Preliminaries	28
3.2.1	Graphs and Random Walks	28
3.2.2	OR-Homomorphic PKCR Encryption Scheme	28
3.2.3	Random Walk Protocol from [1]	29
3.3	Model of Topology-Hiding Communication	31
3.3.1	Adversary	31
3.3.2	Communication Model	31
3.3.3	Security Model	33
3.4	Protocols	33
3.4.1	Protocol Leaking One Bit	34
3.4.2	Protocol Leaking a Fraction of a Bit	42
3.5	Getting Topology-Hiding Computation	45
3.5.1	All-to-all Multibit Broadcast	45
3.5.2	Sequential Execution Without Aggregated Leakage	46
3.5.3	Topology-Hiding Computation	46
3.6	Efficient Topology-Hiding Computation with FHE	47
3.6.1	Deeply Fully-Homomorphic Public-Key Encryption	48
3.6.2	Topology Hiding Computation from DFH-PKE.	48
3.7	Security Against Semi-malicious Adversaries	49
3.8	LWE based OR-Homomorphic PKCR Encryption	52
3.9	Protocols	53
3.9.1	Protocol Leaking a Fraction of a Bit	53
3.9.2	Soundness of the Protocol Leaking a Fraction of a Bit	55

3.10	Getting Topology-Hiding Computation	58
3.10.1	All-to-all Multibit Broadcast	58
3.10.2	Sequential Execution Without Aggregated Leakage	60
3.10.3	Topology-Hiding Computation	60
3.11	Deeply Fully-Homomorphic Public-Key Encryption	61
3.11.1	Instantiation of DFH-PKE from FHE	64
3.12	Topology Hiding Computation from DFH-PKE	66
3.13	Introduction	68
3.13.1	Contributions	69
3.13.2	Related Work	70
3.14	The Probabilistic Unknown Delay Model	71
3.14.1	Impossibility of Stronger Models	71
3.14.2	Adversary	71
3.14.3	Communication Network and Clocks	72
3.14.4	Additional Related Work	74
3.15	Protocols for Restricted Classes of Graphs	75
3.15.1	Synchronous THC from Random Walks	75
3.15.2	Protocol for Cycles	76
3.15.3	Protocol for Trees	79
3.16	Protocol for General Graphs	79
3.16.1	Preprocessing	80
3.16.2	Computation	81
3.16.3	Computing the Eulerian Cycle	83
3.17	Adversarially-Controlled Delays Leak Topology	84
3.17.1	Adversarially-Controlled Delay Indistinguishability-based Security Definition	85
3.17.2	Proof that Adversarially-Controlled Delays Leak Topology	87
3.18	PKCR* Encryption	89
3.18.1	Construction of PKCR* Based on DDH	90
3.19	Proof of Theorem 8	91
3.20	Details of the Protocol for Trees	96
3.21	The Function Executed by the Hardware Boxes	97
3.22	Proof of Theorem 10	101
4	Property Preserving Hashing	107
4.1	Introduction	107
4.1.1	Our Results and Techniques	109
5	Fine-Grained Cryptography	115
A	Tables	117
B	Figures	119

List of Figures

3-1	An example of the algorithm executed by the simulator \mathcal{S}_{OB} . The filled circles are the corrupted parties. The red line represents the random walk generated by \mathcal{S}_{OB} in Step 5, in this case of length $\ell = 3$. \mathcal{S}_{OB} simulates the Decrypt Stage by sending fresh encryptions of $(1, 1)$ at every round from every honest party to each of its corrupted neighbors, except in round $2T - 3$ from P_i to P_j . If no crash occurred up to that point, \mathcal{S}_{OB} sends encryption of $(b^{out}, 0)$. Otherwise, it queries the leakage oracle about the walk of length $T - 3$, starting at P_i	39
3-2	An example of a graph G (on the left) and the corresponding tree \mathcal{T} , computed by <code>EulerianCycle</code> (1, G) (on the right). The eulerian cycle (on the graph with doubled edges) is $(1, 2, 4, 1, 3, 1, 3, 5, 3, 4, 2, 1)$	85
3-3	Graphs used to prove the impossibility of THC with adversarial delays. P_S is the sender. The corrupted parties (black dots) are: P_L and P_R (they delay messages), and the detective P_D . The adversary determines whether P_D (and its two neighbors) are on the left or on the right. . .	86
B-1	Armadillo slaying lawyer.	119
B-2	Armadillo eradicating national debt.	120

List of Tables

- 3.1 Adversarial model and security assumptions of existing topology-hiding broadcast protocols. The table also shows the class of graphs for which the protocols have polynomial communication complexity in the security parameter and the number of parties. 26
- 3.2 The table provides more details on the communication and round complexity of the protocols in Table 3.1. We denote by κ the security parameter, n is the number of parties, D is a bound on the diameter of the graph and d is a bound on the maximum degree of the graph. . . 27
- A.1 Armadillos 117

Chapter 1

Introduction

One of the core goals of cryptography is to be able to offer security and privacy without sacrificing functionality. To do this, cryptographers define notions of both correctness and security. We can then build systems that we prove satisfy both of these definitions. Correctness states that the system provides the functionality we want, while security is essentially the notion that we can effectively hide information from an adversary. Depending on the notion of security, this adversary may be all powerful (information-theoretic or computationally unbounded), may run only in polynomial time (computationally bounded), or even bounded by a specific polynomial runtime like $O(n^2)$ (a fine-grained adversary). This thesis will focus on realizing three different functionalities, each one hiding different information, and three different notions of security.

In the first part of this thesis, we discuss results in the area of Topology Hiding Computation (THC). THC is a generalization of secure multiparty computation. In this model, parties are in an incomplete but connected communication network, each party with its own private inputs. The functionality these parties want to realize is evaluating some function (computation) over all of their inputs. On the security and privacy side, these parties want to hide both their private inputs and who their neighbors are. So, the goal is for these parties to run a protocol, communicating only with their neighbors, so that by the end of the protocol, they learn the output of the function on their inputs and *nothing else*, including information about what the communication network looks like other than their own neighborhood.

It turns out that THC is a difficult notion to realize, mired in impossibility results. We circumvent two impossibility results in THC in two different ways. First, there is an impossibility result for THC stating that it is impossible to design THC against adversaries that can “turn off” parties during the protocol [59]. In essence, an adversary that has this power can always learn something about the graph. So, instead of designing a protocol that would attempt to leak no information, we designed a protocol to limit the amount of information leaked as much as possible. We ran into similar impossibility results when considering a model more similar to the real-world: channels between parties now have unknown delays on them, and an adversary may be able to control the delay. We had to find a model that mirrored this real-world complication, and this was also not impossible to achieve.

In the second part of the thesis, we focus on Adversarially Robust Property Preserving Hashes (PPH). PPHs are a generalization of collision resistant hash functions (CRHFs). Recall that a CRHF is a family of hash functions $\mathcal{H} = \{h : \{0, 1\}^n \rightarrow \{0, 1\}^m\}$ that is compressing ($m < n$), and has the property that no Probabilistic Polynomial-Time (PPT) adversary given h can find two inputs $x_1 \neq x_2$ such that $h(x_1) = h(x_2)$ except with negligible probability. Looking at CRHFs from a slightly different perspective, their functionality is to preserve the equality predicate between two inputs while compressing them, preventing an adversary from coming up with inputs where the equality predicate is not preserved (even though these inputs exist, they are hard to find). A PPH is also a compressing function that preserves some other property. For example, we focus on the property of “gap-hamming” distance: if two inputs are close in hamming distance, then their hashes are also close, and if two inputs are far, their hashes are also far. Note that this example is of a promise predicate since we do not care about inputs that are neither close nor far.

Property preserving hashes were a new cryptographic primitive that we designed, and so it was important to understand what was possible and what was not in our new model. We found strong connections between One-Way Communication Complexity (OWC complexity) and our primitive. Fortunately for us, OWC is a rich, well-studied area of complexity. In many cases, we could not directly port OWC results to our setting, but we could use their proof techniques and get lower bounds. Once we had these lower bounds, we had a much better sense of what was and was not possible, and could construct PPHs more easily.

In the final chapter, for a different perspective, we design a public-key cryptography functionality against weak adversaries. This functionality allows for a party to publish a public key while they keep the secret key, and any other party to use that public key to encrypt a message that only the party with the secret key can decrypt, *hiding* that message from any eavesdroppers. Unlike in standard models for cryptography, however, our eavesdroppers much less powerful: their runtime is bounded by an explicit polynomial instead of “probabilistic polynomial-time”, e.g. can only run in time $O(n^{100})$. This, of course, is only interesting as a cryptographic notion if the adversary has the same or more time to run than the honest parties. So, we get a notion of cryptography we call *fine-grained*: for examples, honest parties might only need to run in $O(n^2)$ time, while any adversary running in time $O(n^{100})$ time still cannot glean any useful information with some probability. Motivating the study of this cryptography is the fact that it does not rely on any of the normal cryptographic assumptions, including $P \neq NP$, $BPP \neq NP$, or even the assumption that one-way functions exist.

However, due to its fine-grained nature many standard cryptographic reductions (like Goldreich-Levin hardcore bits [35]) no longer work. And so, in this thesis we demonstrate variations that can work in our setting, including a notion of fine-grained one-way functions, fine-grained hardcore-bits, a fine-grained key exchange, and finally fine-grained public-key cryptography.

1.1 Results

In this thesis, we explore these three different notions of hiding while preserving a functionality:

1. hide private inputs and network topology while preserving computation,
2. hide as much information as possible while preserving a property between inputs,
3. and hide messages from a weak adversary while preserving the communication and fine-grained runtime.

Topology-Hiding Computation To address the first perspective, we explore the realm of topology-hiding computation (THC). THC is a generalization of secure multiparty computation (MPC), where we hide not only each party’s input, but also the communication graph; parties know who their neighbors are, but learn nothing else about the structure of the graph. In 2017, we showed that THC is possible against a very weak adversary [2], but there were still many open questions surrounding the nature of THC.

- **Fail-stop Adversaries.** Fail-stop adversaries can turn off parties during the protocol. As shown by Moran, Orlav, and Richelson in 2015 [59], it is impossible to achieve perfect topology hiding when giving the adversary this power. The next question is how to get around such an impossibility while still providing meaningful privacy and security. Since the adversary was going to learn something in any protocol we could devise, we decided to use a model that would incorporate this leakage.

Our results here were a definition of security that included access to a “leakage oracle” and a protocol that used this oracle in the security proof. This oracle would spit out yes or no answers to specific queries, leaking one bit of information at a time. In the security reduction of our protocol, a simulator would only need to call this oracle at most once, leaking at most one bit of information to an adversary. Concretely, we could bound the probability that the simulator would have to call the leakage oracle, meaning that we would leak only a polynomial-fraction of a bit. Our protocol ran in $\tilde{O}(\kappa n^4/\rho)$ rounds, where κ is the security parameter, n is the number of parties, and ρ is the probability that we leak a bit.

These results are detailed in our paper at TCC 2018 [53].

- **Probabilistic Delay Model.** This model was the result of first trying to build THC in a purely asynchronous setting, only to prove that it was impossible to do so. So, to get around this without just reducing ourselves to the fail-stop case, we designed a new model that more closely resembled an asynchronous, real-world setting.

More formally, each edge between parties in the Probabilistic Delay Model has its own distribution over delays. Each time a party sends a message to another party, that delay is sampled *independently*. This simulates real world traffic having different delays as it gets sent from one server to another.

Our results also included two new constructions in this model. The first is a construction that works under standard cryptographic assumptions, but only on trees, cycles, or graphs with small circumference. The second is a construction that requires secure hardware boxes/tokens, but works for arbitrary communication graphs. Although this is a very powerful assumption, there are still many challenges involved since an adversarial party can query this hardware at any point with any input.

These results are currently in submission.

Adversarially Robust Property Preserving Hashes and One-Way Communication Lower Bounds Addressing the second perspective, we look at a new cryptographic primitive: adversarially robust Property Preserving Hashes (PPH). The inspiration here is two-fold. First, collision-resistant hash functions (CRHFs) are a staple in cryptography: no PPT adversary can produce an $x_1 \neq x_2$ such that $h(x_1) = h(x_2)$ except with negligible probability over h sampled from the CRHF family and coins of the adversary, and hence one can preserve the “equality” predicate on compressed inputs even against adversarially chosen inputs. We want to compress inputs while maintaining some property other than equality between them, even in the presence of adversarially chosen inputs. The properties we considered were those that already had non-robust constructions, in particular locality-sensitive hashing (LSH) [46]. An LSH family is a hash function family $\mathcal{H} = \{h : \{0, 1\}^n \rightarrow \{0, 1\}^m\}$ that compresses inputs ($m < n$) and is *mostly* correct. For example, say we were preserving a gap- ℓ_p norm for a gap between r and cr for some constant $c > 1$ (note that in our case, gap-hamming is equivalent to gap- ℓ_0 norm). Then, we have a threshold τ so that for any pair $x_1, x_2 \in \{0, 1\}^n$

- if $\|x_1 - x_2\|_p < r$, $\|h(x_1) - h(x_2)\|_p < \tau$, and
- if $\|x_1 - x_2\|_p > cr$ then $\|h(x_1) - h(x_2)\|_p \geq \tau$

with high probability over our choice of h sampled from \mathcal{H} .

We can use almost the same language to define PPHs for some property $P : \{0, 1\}^n \times \{0, 1\}^n \rightarrow \{0, 1\}$, except our probability will be negligible and taken over both our sampled hash function and the coins of a PPT adversary. We also generalize the predicate evaluation: instead of using the same predicate on hashed values, we can use an “evaluation function” called **Eval**. So, even in the presence of (PPT) adversarially chosen values of x_1 and x_2 ,

- if $P(x_1, x_2) = 0$, then $\text{Eval}(h(x_1), h(x_2)) = 0$, and
- if $P(x_1, x_2) = 1$, then $\text{Eval}(h(x_1), h(x_2)) = 1$

with all but negligible probability. In other words, for any PPT adversary given h sampled from \mathcal{H} , the probability that the adversary produces x_1 and x_2 such that $P(x_1, x_2) \neq \text{Eval}(h(x_1), h(x_2))$ is negligible.

The link between robust PPH and LSH was very clear, but less obvious was the link between PPH and one-way communication (OWC): in essence, compressing an input as much as possible for a OWC protocol is akin to compressing an input as much as possible while preserving some property of that input as we would want to do for a hash function. These connections led to the following results, which will be included in the thesis.

- OWC is a rich field with a lot of work detailing lower bounds on the complexity of certain predicates. These lower bounds *almost* directly translated to impossibility results for PPHs. However, we needed to be careful because OWC lower bounds dealt with constant error, where as cryptographers, we care about negligible error. We characterized these OWC lower bounds and how they mapped to PPH lower bounds. We showed that a class of predicates we call “reconstructing predicates” could not have PPHs. This class includes useful predicates like GreaterThan, Index, and ExactHamming.
- With these lower bounds and previous results from LSH in mind, we turned to constructing a PPH for Gap-Hamming: the promise predicate where we can distinguish between pairs of points that are $(1-\epsilon)d$ *close* in hamming distance or $(1+\epsilon)d$ *far*. For all pairs within the gap, we “don’t care” how our PPH behaves. We build two constructions for this predicate, each with different drawbacks and benefits. Our first construction relies only on collision-resistant hashing. Our second uses a new assumption related to the hardness of syndrome decoding [6].
- Rounding out this paper and its myriad of definitions, we demonstrate that even for very simple predicates, we require collision resistance, and even if we weaken our main definition of a robust PPH, we still need one-way functions.

These results were published in ITCS 2019 [14].

In unpublished follow-up research with Cyrus Rashtchian, we explored the intricacies of the CRHF method for constructing gap-hamming PPHs. We fleshed out the relationship between d , the “center” of our gap, and the size of the hash function output. We also discussed how to use this kind of construction to get a gap- ℓ_1 -norm PPH and found that with standard methods of transforming ℓ_1 into ℓ_0 , it could not work.

Fine-Grained Cryptography and Barriers Addressing the final perspective, this thesis will discuss our work in Fine-Grained cryptography. With my coauthors Lincoln and Vassilevska Williams, we built the first fine-grained key exchange built from fine-grained assumptions. The premise was to explore what cryptography could be like in “Pessiland,” a world in which there are no cryptographic one-way functions (which also implies no public key cryptography). We looked at this through the lens of

fine-grained complexity, using both assumptions and reduction techniques from that field.

While designing cryptography for this setting, we came across multiple barriers. The first was that some fine-grained problems would not lend themselves to build cryptography without refuting NSETH [18]. So, we would need to use a fine-grained problem that was not associated with that barrier. Another form of barrier we ran into was worst-case to average-case reductions for problems that would make for good cryptography. Even now, the only worst-case to average-case reductions for fine-grained problems are for counting versions of these problems. Unfortunately, counting-style problems do not lend themselves well to building cryptography, as there are no longer small enough witnesses. Finally, many standard cryptographic reductions no longer work in a fine-grained world, since they take a non-trivial polynomial amount of time, and so a challenge we faced was to ensure all of our reductions were fine-grained and build different types of primitives based off of these restricted reductions.

Despite these difficulties, we achieved the following results:

- Assuming that an average-case version of Zero- k -Clique requires $n^{k-o(1)}$ time, we constructed a non-interactive key exchange that required honest parties to run in $O(N)$ time and an adversary to run in time $\tilde{\Omega}(N^{1.5-\epsilon})$, where ϵ decreases with respect to k .¹
- Generalize the properties of Zero- k -Clique to construct this key exchange: plantable, list-hard, and splittable.
- Define and construct fine-grained hard-core bits.

This work was published in CRYPTO 2019 [52].

In unpublished follow-up work, we also show how to use another property of Zero- k -Clique to construct a key exchange where the adversary now needs $\Omega(N^{2-\epsilon})$ time. This N^2 gap is the best we are able to do since we base these constructions on Merkle puzzles [10].

1.2 Outline of Thesis

TODO

¹The tilde in $\tilde{\Omega}$ ignores any *subpolynomial* factors.

Chapter 2

Preliminaries

TODO

Chapter 3

Topology Hiding Computation

TODO

3.1 Introduction

3.1.1 Topology Hiding Computation

Secure communication over an insecure network is one of the fundamental goals of cryptography. The security goal can be to hide different aspects of the communication, ranging from the content (secrecy), the participants' identity (anonymity), the existence of communication (steganography), to hiding the topology of the underlying network in case it is not complete.

Incomplete networks arise in many contexts, such as the Internet of Things (IoT) or ad-hoc vehicular networks. Hiding the topology can, for example, be important because the position of a node within the network depends on the node's location. This could in turn leak information about the node's identity or other confidential parameters. The goal is that parties, and even colluding sets of parties, can not learn anything about the network, except their immediate neighbors.

Incomplete networks have been studied in the context of communication security, referred to as secure message transmission (see, e.g. [32]), where the goal is to enable communication between any pair of entities, despite an incomplete communication graph. Also, anonymous communication has been studied extensively (see, e.g. [22, 67, 69]). None of these approaches can be used to hide the network topology. In fact, secure message transmission protocols assume (for their execution) that the network graph is public knowledge.

The above problem of *topology-hiding communication* was introduced by Moran et al. [60]. The authors propose a broadcast protocol, which does not reveal any additional information about the network topology to an adversary who can access the internal state of any number of passively corrupted parties (that is, they consider the semi-honest setting). This allows to achieve topology-hiding MPC using standard techniques to transform broadcast channels into secure point-to-point channels. Since then, other, more efficient solutions have been proposed for the model with passive corruptions [42, 4, 1].

A natural next step is to extend these results to settings with more powerful adversaries. Unfortunately, even a protocol in the setting with fail-corruptions (in addition to passive corruptions) turns out to be difficult to achieve. In fact, as shown already in [60], some leakage in the fail-stop setting is inherent. It is therefore no surprise that all previous protocols (secure against passive corruptions) leak information about the network topology if the adversary can crash parties.

A first step in this direction was recently achieved in [9] where a protocol for topology-hiding communication secure against a fail-stop adversary is given. However, the presented solution requires a very strong set-up assumption, where parties have access to secure hardware modules which are initialized with correlated, pre-shared keys. In this paper we propose the first topology-hiding MPC protocol secure against fail-stop adversaries (with arbitrarily small leakage) that is based on standard assumptions. Our protocol does not require setup, and its security can be based on the DDH, the QR or the LWE assumptions.

The above topology-hiding MPC protocol is obtained by compiling a MPC protocol from a topology-hiding broadcast protocol. This incurs a multiplicative overhead in the round complexity. We then present a topology hiding protocol to evaluate any poly-time function using FHE whose round complexity will amount to that of a single broadcast execution.

We also explore another natural extension of semi-honest corruption, the so-called *semi-malicious* setting. As for passive corruption, the adversary selects a set of parties and gets access to their internal state. But in addition, the adversary can also set their randomness during the protocol execution. This models the setting where a party uses an untrusted source of randomness which could be under the control of the adversary. This scenario is of interest as tampered randomness sources have caused many security breaches in the past [40, 24]. In this paper, we propose a general compiler that enhances the security of protocols that tolerate passive corruption with crashes to semi-malicious corruption with crashes.

3.1.2 Contributions

In this work, we provide the first topology hiding broadcast protocol for any network topology which tolerates an adversary who does any number of static passive corruptions and can additionally crash adaptively any number of parties chosen at any time during the protocol. The protocol leaks at most an arbitrary small amount of information and achieves security with abort under standard cryptographic assumptions (either DDH, QR or LWE). If the adversary does not crash any party, then our protocol does not leak any information. We emphasize that in our setting leakage is necessary and security with abort is the best one can hope for, as shown in [60].

Theorem 1 (informal). *If DDH, QR or LWE is hard, then for any $p \in (0, 1]$, there exists a topology-hiding broadcast protocol for any network graph G leaking at most a fraction p of a bit, secure against an adversary that does any number of static passive corruptions and adaptive crashes. The round and communication complexity is polynomial in κ and $1/p$.*

Furthermore, we show how to compile our topology hiding broadcast protocol into a generic topology hiding MPC secure against the same type of adversary, and leaking at most a fraction p of a bit. We remark that this compiler is only straightforward when there is no leakage (as it is the case in previous works which only consider adversaries that only do passive corruptions [60, 42, 4, 1]). However, if we consider a broadcast secure in the fail-stop model that leaks at most 1 bit, composing t of these broadcasts could potentially lead to leaking t bits.

Theorem 2 (informal). *If DDH, QR or LWE is hard, then for any MPC functionality \mathcal{F} , there exists a topology-hiding protocol realizing \mathcal{F} for any network graph G leaking at most an arbitrarily small fraction p of a bit, which is secure against an adversary that does any number of static passive corruptions and adaptive crashes. The round and communication complexity is polynomial in κ and $1/p$.*

We observe that compiling MPC from broadcast is rather expensive, since we need a broadcast for every round. We then show that a FHE scheme with additive overhead can be used to construct a protocol which evaluates any poly-time function f in a topology-hiding manner, whose round complexity is equal to that of a single broadcast.

Theorem 3 (informal). *If FHE with additive overhead exists, then for any poly-time function f , there is a topology-hiding protocol evaluating f for any network graph G leaking an arbitrarily small fraction p of a bit, which is secure against an adversary that does any number of static passive corruptions and adaptive crashes. The round complexity is polynomial in κ and $1/p$.*

Finally, we show how to compile any protocol that tolerates an adversary that can do static passive corruptions and adaptive crashes into a protocol that tolerates in addition semi-malicious corruption. As a corollary, we obtain that any MPC functionality can be realized with arbitrarily small leakage and tolerating any number of static semi-malicious corruptions and adaptive crashes.

Theorem 4 (informal). *Let \mathcal{F} be an MPC functionality and let Π be a protocol that topology-hidingly realizes \mathcal{F} in the presence of static passive corruptions and adaptive crashes. Then, Π can be compiled into a protocol Π' that topology-hidingly realizes \mathcal{F} in the presence of static semi-malicious corruption and adaptive crashes.*

Corollary 1 (informal). *If DDH, QR or LWE is hard, then for any MPC functionality \mathcal{F} there is a topology-hiding protocol realizing \mathcal{F} leaking at most an arbitrarily small fraction p of a bit, which is secure against an adversary that does any number of static semi-malicious corruptions and adaptive crashes. The round and communication complexity is polynomial in κ and $1/p$.*

Overview. As stated above, the main result of this paper consists of four parts. In the first part, eventually proving Theorem 1, we show how to implement topology hiding communication in the presence of an adversary, who can statically passively

corrupt and adaptively crash any number of parties. That is, we show how to construct a topology hiding broadcast protocol, which leaks at most a fraction p of a bit. Note that in our setting broadcast channels can be used to achieve secure point to point connections by applying standard techniques, using public key encryption. In order to explain our protocol, we first present a simpler construction, which leaks at most one bit of information. This construction is later extended to leak at most a fraction of a bit, using a technique similar to the one introduced in [9]. We note, however, that implementing these techniques without trusted hardware requires some not straightforward ideas. Both of the constructions presented in this paper build upon the protocol from [1] for the setting. The paper [1] introduced the idea to use random walks on graphs. This idea is essential to understand our protocols. In Section 3.2.3 we give its informal description, which should be sufficient to understand the core ideas and can be read before the rest of the paper.

The second part of the main result, where we prove Theorem 2, concerns topology hiding computation. It is well known that without leakage any functionality can be implemented on top of secure communication. Perhaps surprisingly, this statement cannot be directly lifted to the setting with leakage. In essence, if a communication protocol is used multiple times, it leaks multiple bits. However, we show that our broadcast protocol, leaking at most a fraction p of a bit, can be executed sequentially and in parallel, such that the result leaks also at most the same fraction p . As a consequence, any protocol can be compiled into one that hides topology and known results on implementing any multiparty computation can be lifted to the topology hiding setting.

In the third part of the paper, we provide a protocol using FHE with additive overhead that evaluates any poly-time function f in a topology hiding manner. The round complexity and leakage amount to that of a single broadcast execution. To do that, we first define an enhanced encryption scheme, which we call DFH-PKE, with similar properties as the PKCR scheme presented in [4, 1] and provide an instantiation of DFH-PKE under FHE. We then show how to obtain a protocol using DFH-PKE to evaluate any poly-time function f in a topology hiding manner.

Finally, we consider the setting with semi-malicious corruptions. We present a compiler, which can transform any protocol secure in the setting with passive and fail-stop corruptions into one additionally secure against semi-malicious corruptions. The compiler preserves leakage (that is, the resulting protocol leaks exactly the same amount of information as the original one) and adds only additive polynomial overhead to round and communication complexity. The main idea behind the compiler is to generate all randomness needed by the protocol before it is started. This way semi-malicious corruptions cannot influence execution after randomness is generated. What is left is to show a topology hiding protocol secure against passive and fail-stop corruptions, which outputs, for each party, a secret, uniform random value, independent of the choice of randomness of parties that are semi-maliciously corrupted. This is done by flooding the network with random values, generated by the parties.

We now give a slightly more technical overview of our main protocols: the one leaking exactly one bit and its extension, leaking a fraction of a bit. Both protocols are based on the random walk protocol presented in [1]. We refer to Section 3.2.3 for

an informal overview of the random-walk idea.

To achieve broadcast, the protocols compute the OR. Every party has an input bit: the sender inputs the broadcast bit and all other parties use 0 as input bit.

Technical overview of the main protocols. We first describe the protocol leaking one bit. Following the approach of [9], we divide the protocol into n consecutive phases, where in each phase only one party gets an output. In each phase, the parties execute a modification of the random walk protocol from [1]. Each party P_i has two bits: the input bit b_i and an *unhappy* bit u_i . Initially, $u_i = 0$. At a very high level, each phase is divided into an Aggregate Stage and a Decryption Stage. In the Aggregate Stage, the parties send at each round an encrypted tuple $[b, u]_{\text{pk}}$ with the corresponding public key pk along the random walk, where b contains the OR of the input bits along the walk and the unhappy bits, and u contains the OR of the unhappy bits in the walk. This is done using a OR-homomorphic PKCR scheme, a scheme that allows a party P_i to add a layer of encryption and homomorphically OR two ciphertexts encrypted under the same public key. In the Decrypt Stage, each party deletes the layer of encryption added in the Aggregate Stage. Then, if a random walk traverses all nodes in the graph, the resulting encrypted bit will contain the OR of all the input bits of the parties. To ensure that only the party P_i gets the output in phase i , all parties except P_i start their random walks with encryptions of 1 instead of their input bits. This guarantees that every party other than P_i obtains the bit 1 at the end of the phase. On the other hand, the party P_i will start exactly one random walk with its bit b_i . As a consequence, P_i learns the broadcast bit at the end of the phase as long as no party crashes. Intuitively, the adversary learns a bit of information if he manages to break the random walk that P_i started with its input bit (all other random walks contain the tuple $[1, 1]$).

We then extend this protocol to leak at most any non-negligible fraction p of a bit. We achieve this by repeating each phase ρ times, where each repetition is modified in such a way that it can randomly fail with probability $1 - p$. Moreover, the failure becomes apparent to the adversary only at the end of the phase. To achieve this, instead of an encrypted tuple $[b, u]$, the parties send $\lfloor 1/p \rfloor + 1$ encrypted values $[b^1, \dots, b^{\lfloor 1/p \rfloor}, u]$ along the walk. At phase i , the party P_i starts exactly one walk with all but one of the $\lfloor 1/p \rfloor + 1$ bits b^k set to 1. The last bit contains the unhappy bit, and all other random walks start with the tuple $[1, \dots, 1]$. This means that only one of the bits b^k is not blinded. In the Aggregate Stage, each party P_j process the ciphertexts in the same way as in the first protocol, but before sending the encryptions along the walk, P_j shuffles the bits b^k randomly. This means that as long as the walk traverses a non-passively corrupted party, the adversary does not know which of the ciphertexts contain dummy values. At the end of the Aggregate Stage, the corresponding party selects one of the $\lfloor 1/p \rfloor$ ciphertexts and executes the Decrypt Stage as before with this ciphertext and the ciphertext containing the unhappy bit. Intuitively, the adversary can only learn information if he manages to break the random walk that P_i started with its input bit and the party at the end of the Aggregate Stage selected the correct ciphertext.

We remark that the above schemes rely on instantiations of a OR-Homomorphic

privately key-commutative and rerandomizable encryption (PKCR). While an instantiation of a OR-Homomorphic PKCR scheme is known under the DDH assumption [42, 4, 1] and under the QR assumption [3], we provide a new instantiation under the LWE assumption.

3.1.3 Related Work

Hiding the network topology is more of a concern in the literature on *anonymous communication* [22, 67, 69]. Here, the goal is to hide the identity of the sender and receiver in a message transmission. A classical technique to achieve anonymity is the so-called mix-net technique, introduced by Chaum [22]. Here, *mix* servers are used as proxies which shuffle messages sent between peers to disable an eavesdropper from following a message’s path. The onion routing technique [69, 67] is perhaps the most known instantiation of the mix-technique. Another anonymity technique known as *Dining Cryptographers networks*, in short DC-nets, was introduced in [23] (see also [13, 37]).

Topology Hiding Communication. A comparison of our results to previous works in topology-hiding communication is found in Tables 3.1 and 3.2.

Table 3.1: Adversarial model and security assumptions of existing topology-hiding broadcast protocols. The table also shows the class of graphs for which the protocols have polynomial communication complexity in the security parameter and the number of parties.

Adversary	Graph	Hardness Asm.	Model	Reference
semi-honest	log diam.	Trapdoor Perm.	Standard	[60]
	log diam.	DDH	Standard	[42]
	cycles, trees, log circum.	DDH	Standard	[4]
	arbitrary	DDH or QR	Standard	[1]
fail-stop	arbitrary	OWF	Trusted Hardware	[9]
semi-malicious & fail-stop	arbitrary	DDH or QR or LWE	Standard	[This work]

Existing constructions to achieve *topology-hiding communication* over an incomplete network focus mainly on the cryptographic setting for passive corruptions. The first protocol was given in [60]. Here, the authors provide a feasibility result for a broadcast protocol secure against static, passive corruptions. At a very high level, [60] uses a series of nested multi-party computations, in which each node is emulated by a secure computation of its neighbor. This emulation then extends to the entire graph recursively. In [42], the authors improve this result and provide a construction that makes only black-box use of encryption and where the security is based on the DDH assumption. However, both results are feasible only for graphs with logarithmic

diameter. Topology hiding communication for certain classes of graphs with large diameter was described in [4]. This result was finally extended to allow for arbitrary (connected) graphs in [1].

The fail-stop setting was first considered in [60] where the authors give a construction for a fail-stop adversary with a very limited corruption pattern: the adversary is not allowed to corrupt any complete neighborhood of a party and is also not allowed an abort pattern that disconnects the graph. In this work, the authors also prove that topology-hiding communication without leakage is impossible if the adversary disconnects the graph. A more recent work, [9], provides a construction for a fail-stop adversary with one bit/non-negligible leakage but requires that parties have access to secure hardware modules which are initialized with correlated, pre-shared keys.

In the information-theoretic setting, the main result is negative [41]: any MPC protocol in the information-theoretic setting inherently leaks information about the network graph. They also show that if the routing table is leaked, one can construct an MPC protocol which leaks no additional information.

Table 3.2: The table provides more details on the communication and round complexity of the protocols in Table 3.1. We denote by κ the security parameter, n is the number of parties, D is a bound on the diameter of the graph and d is a bound on the maximum degree of the graph.

Adversary	Communication	Rounds	Reference
semi-honest	$O(\text{poly}(d)^D n \kappa)$	$O(\text{poly}(d)^D)$	[60]
	$O((d+1)^D n \kappa)$	$5 \cdot D$	[42]
	$O(n^2 \kappa)$	$O(n)$	[4]
	$O(n^5(\kappa + \log(n)))$	$O(n^3(\kappa + \log(n)))$	[3]
fail-stop	$O(nD(d + \kappa))$	$O(nD)$	[9]
semi-malicious & fail-stop	$O(n^6(\kappa + \log(n)))$	$O(n^4(\kappa + \log(n)))$	[This work]

3.1.4 Outline

The remainder of this paper is organized as follows. In Section 3.2 we provide some notation and building blocks for our protocols. In Section 3.14 we discuss the communication and security model for our topology-hiding protocols. In Section 3.4 we present a topology-hiding broadcast protocol leaking arbitrarily low leakage, secure against an adversary that can statically passively corrupt and adaptively crash any number of parties. In Section 3.5 we show how to execute sequentially and in parallel broadcasts with leakage to get topology-hiding MPC with arbitrarily low leakage. In Section 3.6 we present a more efficient protocol for topology-hiding MPC based on FHE. In Section 3.7 we show how to enhance the security of topology-hiding protocols to semi-malicious adversaries. In Section 3.8 we give an implementation of a PKCR scheme based on LWE.

3.2 Preliminaries

3.2.1 Graphs and Random Walks

In an undirected graph $G = (V, E)$ we denote by $\mathbf{N}_G(P_i)$ the neighborhood of $P_i \in V$. The k -neighborhood of a party $P_i \in V$ is the set of all parties in V within distance k to P_i .

In our work we use the following lemma from [1]. It states that in an undirected connected graph G , the probability that a random walk of length $8|V|^3\tau$ covers G is at least $1 - \frac{1}{2^\tau}$.

Lemma 1 ([1]). *Let $G = (V, E)$ be an undirected connected graph. Further let $\mathcal{W}(u, \tau)$ be a random variable whose value is the set of nodes covered by a random walk starting from u and taking $8|V|^3\tau$ steps. We have*

$$\Pr_{\mathcal{W}}[\mathcal{W}(u, \tau) = V] \geq 1 - \frac{1}{2^\tau}.$$

3.2.2 OR-Homomorphic PKCR Encryption Scheme

As in [1], our protocols require a public key encryption scheme with additional properties, called *Privately Key Commutative and Rerandomizable encryption*. We assume that the message space is bits. Then, a PKCR encryption scheme should be: (1) privately key commutative and (2) homomorphic with respect to the OR operation. We formally define these properties below.¹

Let \mathcal{PK} , \mathcal{SK} and \mathcal{C} denote the public key, secret key and ciphertext spaces. As any public key encryption scheme, a PKCR scheme contains the algorithms **KeyGen** : $\{0, 1\}^* \rightarrow \mathcal{PK} \times \mathcal{SK}$, **Encrypt** : $\{0, 1\} \times \mathcal{PK} \rightarrow \mathcal{C}$ and **Decrypt** : $\mathcal{C} \times \mathcal{SK} \rightarrow \{0, 1\}$ for key generation, encryption and decryption respectively (where **KeyGen** takes as input the security parameter).

Notation

For a public-key \mathbf{pk} and a message m , we denote the encryption of m under \mathbf{pk} by $[m]_{\mathbf{pk}}$. Furthermore, for k messages m_1, \dots, m_k , we denote by $[m_1, \dots, m_k]_{\mathbf{pk}}$ a vector, containing the k encryptions of messages m_i under the same key \mathbf{pk} .

For an algorithm $A(\cdot)$, we write $A(\cdot; U^*)$ whenever the randomness used in $A(\cdot)$ should be made explicit and comes from a uniform distribution. By \approx_c we denote that two distribution ensembles are computationally indistinguishable.

Privately Key-Commutative

We require \mathcal{PK} to form a commutative group under the operation \otimes . So, given any $\mathbf{pk}_1, \mathbf{pk}_2 \in \mathcal{PK}$, we can efficiently compute $\mathbf{pk}_3 = \mathbf{pk}_1 \otimes \mathbf{pk}_2 \in \mathcal{PK}$ and for every \mathbf{pk} ,

¹PKCR encryption was introduced in [4, 1], where it had three additional properties: key commutativity, homomorphism and rerandomization, hence, it was called Privately Key Commutative and *Rerandomizable* encryption. However, rerandomization is actually implied by the strengthened notion of homomorphism. Therefore, we decided to not include the property, but keep the name.

there exists an inverse denoted \mathbf{pk}^{-1} . This \mathbf{pk}^{-1} must be efficiently computable given the secret key corresponding to \mathbf{pk} .

This group must interact well with ciphertexts; there exists a pair of efficiently computable algorithms $\text{AddLayer} : \mathcal{C} \times \mathcal{SK} \rightarrow \mathcal{C}$ and $\text{DelLayer} : \mathcal{C} \times \mathcal{SK} \rightarrow \mathcal{C}$ such that

- For every public key pair $\mathbf{pk}_1, \mathbf{pk}_2 \in \mathcal{PK}$ with corresponding secret keys \mathbf{sk}_1 and \mathbf{sk}_2 , message $m \in \mathcal{M}$, and ciphertext $c = [m]_{\mathbf{pk}_1}$,

$$\text{AddLayer}(c, \mathbf{sk}_2) = [m]_{\mathbf{pk}_1 \otimes \mathbf{pk}_2}.$$

- For every public key pair $\mathbf{pk}_1, \mathbf{pk}_2 \in \mathcal{PK}$ with corresponding secret keys \mathbf{sk}_1 and \mathbf{sk}_2 , message $m \in \mathcal{M}$, and ciphertext $c = [m]_{\mathbf{pk}_1}$,

$$\text{DelLayer}(c, \mathbf{sk}_2) = [m]_{\mathbf{pk}_1 \otimes \mathbf{pk}_2^{-1}}.$$

Notice that we need the secret key to perform these operations, hence the property is called *privately* key-commutative.

OR-Homomorphic

We also require the encryption scheme to be OR-homomorphic, but in such a way that parties cannot tell how many 1's or 0's were OR'd (or who OR'd them). We need an efficiently-evaluatable homomorphic-OR algorithm, $\text{HomOR} : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$, to satisfy the following: for every two messages $m, m' \in \{0, 1\}$ and every two ciphertexts $c, c' \in \mathcal{C}$ such that $\text{Decrypt}(c, \mathbf{sk}) = m$ and $\text{Decrypt}(c', \mathbf{sk}) = m'$,

$$\begin{aligned} & \{(m, m', c, c', \mathbf{pk}, \text{Encrypt}(m \vee m', \mathbf{pk}; U^*))\} \\ & \quad \approx_c \\ & \{(m, m', c, c', \mathbf{pk}, \text{HomOR}(c, c', \mathbf{pk}; U^*))\} \end{aligned}$$

Note that this is a stronger definition for homomorphism than usual; usually we only require correctness, not computational indistinguishability.

In [42], [4] and [1], the authors discuss how to get this kind of homomorphic OR under the DDH assumption, and later [3] show how to get it with the QR assumption. For more details on other kinds of homomorphic cryptosystems that can be compiled into OR-homomorphic cryptosystems, see [3].

In this paper we show how to instantiate a PKCR encryption scheme under the LWE assumption (for details, see Section 3.8).

3.2.3 Random Walk Protocol from [1]

Our protocols are based on the random walk protocol from [1]. We therefore give a high level overview of that protocol and explain why it is correct and sound. We recall that the random walk protocol achieves security against static passive corruptions. To achieve broadcast, the protocol actually computes an OR. Every party has an

input bit: the sender inputs the broadcast bit and all other parties use 0 as input bit. Computing the OR of all those bits is thus equivalent to broadcasting the sender's message.

First, we will explain a simplified version of the protocol that is unfortunately not sound, but this gets the principal across. Each node will take its bit, encrypt it under a public key and forward it to a random neighbor. The neighbor OR's its own bit, adds a fresh public key layer, and it randomly chooses the next step in the walk that the message takes, choosing a random neighbor to forward the bit. Eventually, after about $O(\kappa n^3)$ steps, the random walk of every message will visit every node in the graph, and therefore, every message will contain the OR of all bits in the network. Now we start the backwards phase, reversing the walk and peeling off layers of encryption.

This scheme is not sound because seeing where the random walks are coming from reveals information about the graph! So, we need to disguise that information. We will do so by using correlated random walks, and will have a walk running down each direction of each edge at each step (the number of walks is then $2 \times$ number of edges). The walks are correlated, but still random. This way, at each step, each node just sees encrypted messages all under new and different keys from each of its neighbors. So, intuitively, there is no way for a node to tell anything about where a walk came from.

In more detail, and to demonstrate security, consider a single node v with d neighbors. During the forward phase at step t , v gets d incoming messages — one from each of its neighbors — homomorphically OR's its bit to each, computes d fresh public keys, adding a layer to each, and finally computes a random permutation π_t on its neighbors, forwarding the message it got from neighbor i to neighbor $\pi_t(i)$ and so on. During the backwards phase, node v removes the public key layers it added during the corresponding forward round, and then reverses the permutation, sending the message it got from neighbor j to neighbor $\pi_t^{-1}(j)$. Because all messages are encrypted under semantically secure encryption, v cannot tell whether it has received a 0 or 1 from any of its neighbors, and because all of its neighbors are layering their own fresh public keys onto the messages, there is no way for v to tell where that message came from or if it had seen it before. Intuitively, this gives us soundness (see [1] for details).

Now, this protocol is also correct: every walk will, with all but negligible probability, visit every node in the network, and therefore every message will, with all but negligible probability, contain an encryption of the OR of all bits in the graph by the end of the forward phase. The backward phase then takes that message at the end of the walk, and reverses the walk exactly, popping off the public key layer that was added at each step. By the end of the backward phase, the node that started the walk gets the decryption of the message: the OR of all bits in the graph. Because all of the walks succeed, and every node started a walk, every node gets the correct output bit as desired.

3.3 Model of Topology-Hiding Communication

3.3.1 Adversary

In this work, we consider an adversary, who can statically select a set of parties to passively or even semi-maliciously corrupt. Additionally, we allow the adversary to adaptively crash any number of parties.

Most of our results concern an adversary, who can *passively corrupt* an arbitrary set (i.e., $t < n$) of parties \mathcal{Z}^p before the protocol execution. Passively corrupted parties follow the protocol instructions (this includes the generation of randomness), but the adversary can access their internal state during the protocol execution.

A *semi-malicious* corruption (see, e.g., [7]) is a stronger variant of a passive corruption. Again, we assume that the adversary selects the set of semi-malicious parties \mathcal{Z}^s before the protocol execution. These parties follow the protocol instructions, but the adversary can access their internal state and can additionally choose their randomness.

In all of our results we assume a *fail-stop* adversary that can crash adaptively arbitrary parties. After being crashed, a party stops sending messages. Note that crashed parties are not necessarily corrupted. In particular, the adversary has no access to the internal state of a crashed party unless it is in the set of corrupted parties. This type of fail-stop adversary is stronger and more general than the one used in [9], where only passively corrupted parties can be crashed. In particular, in our model the adversary does not necessarily learn the neighbors of crashed parties, whereas in [9] they are revealed to it by definition.

3.3.2 Communication Model

We state our results in the UC framework. Following the approach in [60], to model the restricted communication network we define the \mathcal{F}_{NET} -hybrid model. The \mathcal{F}_{NET} functionality takes as input a description of the graph network from a special “graph party” P_{setting} and then returns to each party P_i a description of its neighborhood. After that, the functionality acts as an “ideal channel” that allows parties to communicate with their neighbors according to the graph network.

Similarly to [9], we change the \mathcal{F}_{NET} functionality from [60] to deal with a fail-stop adversary. However, our version differs from the one in [9] in two aspects. First, we do not strengthen the functionality \mathcal{F}_{NET} in the real world to notify the neighbors of a crashed party, even though we deal with adversaries that are beyond semi-honest. Secondly, as mentioned in Section 3.3.1, our model does not assume that only passively corrupted parties can be crashed.

Functionality \mathcal{F}_{NET}

The functionality keeps the following variables: the set of crashed parties \mathcal{C} and the graph G .

Initially, $\mathcal{C} = \emptyset$ and $G = (\emptyset, \emptyset)$.

Initialization Step:

1. The party P_{setting} sends graph G' to \mathcal{F}_{NET} . \mathcal{F}_{NET} sets $G = G'$.
2. \mathcal{F}_{NET} sends to each party P_i its neighborhood $\mathbf{N}_G(P_i)$.

Communication Step:

1. If the adversary crashes party P_i , then \mathcal{F}_{NET} sets $\mathcal{C} = \mathcal{C} \cup \{P_i\}$.
2. If a party P_i sends the command (SEND, j, m) , where $P_j \in \mathbf{N}_G(P_i)$ and m is the message to P_j , to \mathcal{F}_{NET} and $P_i \notin \mathcal{C}$, then \mathcal{F}_{NET} outputs (i, m) to party P_j .

Observe that since \mathcal{F}_{NET} gives local information about the network graph to all corrupted parties, any ideal-world adversary should also have access to this information. For this reason, [60] introduces the functionality $\mathcal{F}_{\text{INFO}}$, which contains only the Initialization Step of \mathcal{F}_{NET} . $\mathcal{F}_{\text{INFO}}$ is then made available to the simulator.

As shown in [60], in the fail-stop model and if the graph can potentially be disconnected, leaking some information about the network topology is inherent. We follow the approach in [9] and model the leakage by allowing the adversary access to an oracle, which, once during the protocol execution, evaluates a (possibly probabilistic) function \mathcal{L} . The inputs to \mathcal{L} include the network graph, the set of crashed parties and arbitrary input from the adversary.

We will say that a protocol leaks one bit of information if the leakage function \mathcal{L} outputs one bit. We also consider the notion of leaking a fraction p of a bit. This is modeled by having \mathcal{L} output the bit only with probability p (otherwise, \mathcal{L} outputs a special symbol \perp). Here our model differs from the one in [9], where in case of the fractional leakage, \mathcal{L} always gives the output, but the simulator is restricted to query its oracle with probability p over its randomness. As noted in [9], the formulation we use is stronger.

The leakage function \mathcal{L} is a parameter of the functionality $\mathcal{F}_{\text{INFO}}$ (we denote our information functionality by $\mathcal{F}_{\text{INFO}}$). $\mathcal{F}_{\text{INFO}}$ consists of two phases: an initialization phase as in \mathcal{F}_{NET} and a leakage phase, where the adversary can query the leakage oracle \mathcal{L} .

Functionality $\mathcal{F}_{\text{INFO}}$

The functionality keeps the following variables: the set of crashed parties \mathcal{C} and the graph G . Initially, $\mathcal{C} = \emptyset$ and $G = (\emptyset, \emptyset)$.

Initialization Step:

1. The party P_{setting} sends graph $G' = (V, E)$ to $\mathcal{F}_{\text{INFO}}$. $\mathcal{F}_{\text{INFO}}$ sets $G = G'$.
2. $\mathcal{F}_{\text{INFO}}$ sends to each party P_i its neighborhood $\mathbf{N}_G(P_i)$.

Leakage Step:

1. If the adversary crashes party P_i , then $\mathcal{F}_{\text{INFO}}$ sets $\mathcal{C} = \mathcal{C} \cup \{P_i\}$.
2. If the adversary sends the command (LEAK, q) to $\mathcal{F}_{\text{INFO}}$ for the first time, then $\mathcal{F}_{\text{INFO}}$ outputs $\mathcal{L}(q, \mathcal{C}, G)$ to the adversary.

3.3.3 Security Model

Our protocols provide security with abort. In particular, the adversary can choose some parties, who do not receive the output (while the others still do). That is, no guaranteed output delivery and no fairness is provided. Moreover, the adversary sees the output before the honest parties and can later decide which of them should receive it.

Technically, we model such ability in the UC framework in a way very similar to [9]. First, the ideal world adversary receives from the ideal functionality the outputs of the corrupted parties. Then, it inputs to the functionality an *abort vector* containing a list of parties who do not receive the output.

Definition 1. *We say that a protocol Π topology-hidingly realizes a functionality \mathcal{F} with \mathcal{L} -leakage, in the presence of an adversary who can statically passive corrupt and adaptively crash any number of parties, if it UC-realizes $(\mathcal{F}_{\text{INFO}} \parallel \mathcal{F})$ in the \mathcal{F}_{NET} -hybrid model.*

3.4 Protocols

In this section, we provide a protocol that securely realizes the broadcast functionality \mathcal{F}_{BC} (with abort) in the \mathcal{F}_{NET} -hybrid world with at most one bit leakage. The protocol is secure in the presence of an adversary that corrupts statically passively any number of nodes and can also crash adaptively any number of nodes. It works under the DDH, the QR or the LWE assumption. We then extend this protocol to leak at most any non-negligible fraction of a bit.

If no crashes occur, both protocols do not leak any information.

Functionality \mathcal{F}_{BC}

When a party P_i sends a bit $b \in \{0, 1\}$ to the functionality \mathcal{F}_{BC} , then \mathcal{F}_{BC} sends b to each party $P_j \in \mathcal{P}$.

As in [1], our protocols actually compute the OR of the input bits of the parties. To achieve broadcast, every party inputs the bit 0, except for the sender, who inputs the broadcast bit.

3.4.1 Protocol Leaking One Bit

We now present a topology-hiding broadcast protocol that is secure (with abort) against an adversary that can passively corrupt and adaptively crash parties, and leaks at most one bit of information about the network topology. If the adversary only does passive corruptions, no leakage occurs.

Our broadcast protocol **BC-OB** builds upon the random-walk idea from the work [1]. To prevent the adversary from learning too much information (by crashing a party), we do not use a single random walk phase. Instead, we use n consecutive random-walk phases, where in each phase only one party gets an output. We note that the same idea is used in the recent work [9].

In our protocol every party P_i holds an *unhappy-bit* u_i . Initially, every party P_i is happy, i.e., $u_i = 0$. If a neighbor of P_i crashes, then in the next phase P_i becomes unhappy and sets $u_i = 1$.

As said above, the protocol consists of n phases. In each phase, the parties execute the random-walk protocol from [1] with the following modifications.

Happiness Indicator: Instead of a single encrypted bit, parties send an encrypted tuple $[b, u]$ along the random walk. Bit u is the OR of the unhappy-bits of parties in the walk, while b is the OR of their input bits and their unhappy-bits. In other words, a party P_i on the walk homomorphically ORs $b_i \vee u_i$ to b and u_i to u . If all parties on the walk were happy at the time of adding their bits, the bit b will actually contain the proper OR of their input bits. On the other hand, if a party was unhappy, the bits u and b will actually be set to 1.

Single Output Party: In phase i all parties except P_i start their random walks with encryptions of 1 instead of their input bits. This ensures that the outputs they get from the random walk will be 1. Hence, they do not learn any information in this phase. Party P_i , on the other hand, will start exactly one random walk with its actual input bit. This ensures (in case no party crashes) that P_i actually learns the broadcast bit.

More formally, parties execute, in each phase, protocol **RandomWalkPhase**. This protocol takes as global inputs the length T of the random walk and the P_o which should get output. Additionally, each party P_i has input (d_i, b_i, u_i) where d_i is its number of neighbors, u_i is its unhappy-bit, and b_i is its input bit.

Protocol RandomWalkPhase($T, P_o, (d_i, b_i, u_i)_{P_i \in \mathcal{P}}$)

Initialization Stage:

- 1: Each party P_i generates $T \cdot d_i$ keypairs $(\text{pk}_{i \rightarrow j}^{(r)}, \text{sk}_{i \rightarrow j}^{(r)}) \leftarrow \text{KeyGen}(1^\kappa)$ where $r \in \{1, \dots, T\}$ and $j \in \{1, \dots, d_i\}$.
- 2: Each party P_i generates $T-1$ random permutations on d_i elements $\{\pi_i^{(2)}, \dots, \pi_i^{(T)}\}$.
- 3: For each party P_i , if any of P_i 's neighbors crashed in any phase before the current one, then P_i becomes unhappy, i.e., sets $u_i = 1$.

Aggregate Stage: Each party P_i does the following:

```

1: // AddLayer and HomOR applied component-wise
2: // Send first ciphertexts
3: if  $P_i$  is the recipient  $P_0$  then
4:   Party  $P_i$  sends to the first neighbor the ciphertext  $[b_i \vee u_i, u_i]_{\text{pk}_{i \rightarrow 1}^{(1)}}$  and the public
   key  $\text{pk}_{i \rightarrow 1}^{(1)}$ .
5:   Party  $P_i$  sends to any other neighbor  $P_j$  ciphertext  $[1, 1]_{\text{pk}_{i \rightarrow j}^{(1)}}$  and the public
   key  $\text{pk}_{i \rightarrow j}^{(1)}$ .
6: else
7:   Party  $P_i$  sends to each neighbor  $P_j$  ciphertext  $[1, 1]_{\text{pk}_{i \rightarrow j}^{(1)}}$  and the public key
    $\text{pk}_{i \rightarrow j}^{(1)}$ .
8: // Add layer while ORing own input bit
9: for any round  $r$  from 2 to  $T$  do
10:  For each neighbor  $P_j$  of  $P_i$ , do the following:
11:  if  $P_i$  did not receive a message from  $P_j$  then
12:    Let  $k = \pi_i^{(r)}(j)$ .
13:    Party  $P_i$  sends ciphertext  $[1, 1]_{\text{pk}_{i \rightarrow k}^{(r)}}$  and public key  $\text{pk}_{i \rightarrow k}^{(r)}$  to neighbor  $P_k$ .
14:  else
15:    Let  $k = \pi_i^{(r)}(j)$ . Let  $\text{c}_{j \rightarrow i}^{(r-1)}$  and  $\overline{\text{pk}}_{j \rightarrow i}^{(r-1)}$  be the ciphertext and the public key
     $P_i$  received from  $P_j$ .
16:    Party  $P_i$  computes  $\overline{\text{pk}}_{i \rightarrow k}^{(r)} = \overline{\text{pk}}_{j \rightarrow i}^{(r-1)} \circledast \text{pk}_{i \rightarrow k}^{(r)}$  and  $\hat{\text{c}}_{i \rightarrow k}^{(r)} \leftarrow$ 
     $\text{AddLayer}(\text{c}_{j \rightarrow i}^{(r-1)}, \text{sk}_{i \rightarrow k}^{(r)})$ .
17:    Party  $P_i$  computes  $[b_i \vee u_i, u_i]_{\overline{\text{pk}}_{i \rightarrow k}^{(r)}}$  and  $\text{c}_{i \rightarrow k}^{(r)} =$ 
     $\text{HomOR}\left([b_i \vee u_i, u_i]_{\overline{\text{pk}}_{i \rightarrow k}^{(r)}}, \hat{\text{c}}_{i \rightarrow k}^{(r)}, \overline{\text{pk}}_{i \rightarrow k}^{(r)}\right)$ .
18:    Party  $P_i$  sends ciphertext  $\text{c}_{i \rightarrow k}^{(r)}$  and public key  $\overline{\text{pk}}_{i \rightarrow k}^{(r)}$  to neighbor  $P_k$ .

```

Decrypt Stage: Each party P_i does the following:

```

1: // DelLayer and HomOR applied component-wise
2: // Return ciphertexts
3: For each neighbor  $P_j$  of  $P_i$ :
4: if  $P_i$  did not receive a message from  $P_j$  at round  $T$  of the Aggregate Stage then
5:   Party  $P_i$  sends ciphertext  $\text{e}_{i \rightarrow j}^{(T)} = [1, 1]_{\overline{\text{pk}}_{j \rightarrow i}^{(T)}}$  to  $P_j$ .
6: else
7:   Party  $P_i$  computes and sends  $\text{e}_{i \rightarrow j}^{(T)} = \text{HomOR}\left([b_i \vee u_i, u_i]_{\overline{\text{pk}}_{j \rightarrow i}^{(T)}}, \text{c}_{j \rightarrow i}^{(T)}, \overline{\text{pk}}_{j \rightarrow i}^{(T)}\right)$  to
    $P_j$ .
8: // Remove layers
9: for any round  $r$  from  $T$  to 2 do
10:  For each neighbor  $P_k$  of  $P_i$ :
11:  if  $P_i$  did not receive a message from  $P_k$  then
12:    Party  $P_i$  sends  $\text{e}_{i \rightarrow j}^{(r-1)} = [1, 1]_{\overline{\text{pk}}_{j \rightarrow i}^{(r-1)}}$  to neighbor  $P_j$ , where  $k = \pi_i^{(r)}(j)$ .
13:  else

```

```

14: Denote by  $\mathbf{e}_{k \rightarrow i}^{(r)}$  the ciphertext  $P_i$  received from  $P_k$ , where  $k = \pi_i^{(r)}(j)$ .
15: Party  $P_i$  sends  $\mathbf{e}_{i \rightarrow j}^{(r-1)} = \text{DelLayer}(\mathbf{e}_{k \rightarrow i}^{(r)}, \mathbf{sk}_{i \rightarrow k}^{(r)})$  to neighbor  $P_j$ .
16: // Only the recipient has a proper output
17: if  $P_i$  is the recipient  $P_o$  and happy then
18:   Party  $P_i$  computes  $(b, u) = \text{Decrypt}(\mathbf{e}_{1 \rightarrow i}^{(1)}, \mathbf{sk}_{i \rightarrow 1}^{(1)})$ .
19:   Party  $P_i$  outputs  $(b, u, u_i)$ .
20: else
21:   Party  $P_i$  outputs  $(1, 0, u_i)$ .

```

The actual protocol BC-OB consists of n consecutive runs of the random walk phase RandomWalkPhase.

Protocol BC-OB($T, (d_i, b_i)_{P_i \in \mathcal{P}}$)

```

Each party  $P_i$  keeps bits  $b_i^{out}, u_i^{out}$  and  $u_i$ , and sets  $u_i = 0$ .
for  $o$  from 1 to  $n$  do
  Parties jointly execute  $((b_i^{tmp}, v_i^{tmp}, u_i^{tmp})_{P_i \in \mathcal{P}}) =$ 
  RandomWalkPhase( $T, P_o, (d_i, b_i, u_i)_{P_i \in \mathcal{P}}$ ).
  Each party  $P_i$  sets  $u_i = u_i^{tmp}$ .
  Party  $P_o$  sets  $b_o^{out} = b_o^{tmp}, u_o^{out} = v_o^{tmp}$ .
For each party  $P_i$ , if  $u_i^{out} = 0$  then party  $P_i$  outputs  $b_i^{out}$ .

```

The protocol BC-OB leaks information about the topology of the graph during the execution of the protocol RandomWalkPhase, in which the first crash occurs. (Every execution before the first crash proceeds almost exactly as the protocol in [1] and in every execution afterwards all values are blinded by the unhappy-bit u .) We model the leaked information by a query to the leakage function \mathcal{L}_{OB} . The function outputs only one bit and, since the functionality $\mathcal{F}_{\text{INFO}}$ allows for only one query to the leakage function, the protocol leaks overall one bit of information.

The inputs passed to \mathcal{L}_{OB} are: the graph G and the set \mathcal{C} of crashed parties, passed to the function by $\mathcal{F}_{\text{INFO}}$, and a triple (F, P_s, T') , passed by the simulator. The idea is that the simulator needs to know whether the walk carrying the output succeeded or not, and this depends on the graph G . More precisely, the set F contains a list of pairs (P_f, r) , where r is the number of round in the execution of RandomWalkPhase, at which P_f crashed. \mathcal{L}_{OB} tells the simulator whether any of the crashes in F disconnected a freshly generated random walk of length T' , starting at given party P_s .

Function \mathcal{L}_{OB} ($(F, P_s, T'), \mathcal{C}, G$)

```

if for any  $(P_f, r) \in F, P_f \notin \mathcal{C}$  then Return 0.
else
  Generate in  $G$  a random walk of length  $T'$  starting at  $P_s$ .
  Return 1 if for any  $(P_f, r) \in F$  removing party  $P_f$  after  $r$  rounds disconnects the
  walk and 0 otherwise.

```

Theorem 5. *Let κ be the security parameter. For $T = 8n^3(\log(n) + \kappa)$ the protocol $BC\text{-}OB(T, (d_i, b_i)_{P_i \in \mathcal{P}})$ topology-hidingly realizes $\mathcal{F}_{\text{INFO}}^{OB} || \mathcal{F}_{\text{BC}}$ (with abort) in the \mathcal{F}_{NET} hybrid-world, where the leakage function \mathcal{L}_{OB} is the one defined as above. If no crashes occur, then there is no abort and there is no leakage.*

Proof. Completeness. We first show that the protocol is complete. To this end, we need to ensure that the probability that all parties get the correct output is overwhelming in κ . That is, the probability that all non-dummy random walks (of length $T = 8n^3(\log(n) + \kappa)$) reach all nodes is overwhelming.

By Lemma 1, a walk of length $8n^3\tau$ does not reach all nodes with probability at most $\frac{1}{2^\tau}$. Then, using the union bound, we obtain that the probability that there is a party whose walk does not reach all nodes is at most $\frac{n}{2^\tau}$. Hence, all n walks (one for each party) reach all nodes with probability at least $1 - \frac{n}{2^\tau}$. If we set this value to be overwhelming, e.g. $1 - \frac{1}{2^\kappa}$, we can set $\tau := \kappa + \log(n)$.

Soundness. We now need to show that no environment can distinguish between the real world and the simulated world, when given access to the adversarially-corrupted parties.

Simulator. In essence, the task of \mathcal{S}_{OB} is to simulate the messages sent by honest parties to passively corrupted parties. Below, we present the pseudocode of the simulator. The essential part of it is the algorithm **PhaseSimulation**, which is also illustrated in Figure 3-1.

Simulator \mathcal{S}_{OB}

1. \mathcal{S}_{OB} corrupts passively \mathcal{Z}^p .
2. \mathcal{S}_{OB} sends inputs for all parties in \mathcal{Z}^p to \mathcal{F}_{BC} and receives the output bit b^{out} .
3. For each $P_i \in \mathcal{Z}^p$, \mathcal{S}_{OB} receives $\mathbf{N}_G(P_i)$ from $\mathcal{F}_{\text{INFO}}$.
4. Throughout the simulation, if \mathcal{A} crashes a party P_f , so does \mathcal{S}_{OB} .
5. Now \mathcal{S}_{OB} has to simulate the view of all parties in \mathcal{Z}^p .

In every phase in which P_o should get the output, first of all the Initialization Stage is executed among the parties in \mathcal{Z}^p and the T key pairs are generated for every $P_i \in \mathcal{Z}^p$. Moreover, for every $P_i \in \mathcal{Z}^p$ the permutations $\pi_i^{(r)}$ are generated, defining those parts of all random walks, which pass through parties in \mathcal{Z}^p .

The messages sent by parties in \mathcal{Z}^p are generated by executing the protocol **RandomWalkPhase**. The messages sent by correct parties $P_i \notin \mathcal{Z}^p$ are generated by executing **PhaseSimulation**(P_o, P_i), described below.

6. \mathcal{S}_{OB} sends to \mathcal{F}_{BC} the abort vector (in particular, the vector contains all parties P_o who should receive their outputs in phases following the first crash and, depending on the output of \mathcal{L}_{OB} , the party who should receive its output in the phase with first crash).

Algorithm PhaseSimulation(P_o, P_i)

If $P_o \in \mathcal{Z}^p$, let w denote the random walk generated in the Initialization Stage (at the beginning of the simulation of this phase), which starts at P_o and carries the output bit. Let ℓ denote the number of parties in \mathcal{Z}^p on w before the first correct party. If $P_o \notin \mathcal{Z}^p$, w and ℓ are not defined.

For every $P_j \in \mathcal{Z}^p \cap \mathbf{N}_G(P_i)$, let $\mathbf{pk}_{j \rightarrow i}^{(r)}$ denote the public key generated in the Initialization Stage by P_j for P_i and for round r .

Initialization Stage

- 1: For every neighbor $P_j \in \mathcal{Z}^p$ of the correct P_i , \mathcal{S}_{OB} generates \mathbf{T} key pairs $(\mathbf{pk}_{i \rightarrow j}^{(1)}, \mathbf{sk}_{i \rightarrow j}^{(1)}), \dots, (\mathbf{pk}_{i \rightarrow j}^{(\mathbf{T})}, \mathbf{sk}_{i \rightarrow j}^{(\mathbf{T})})$.

Aggregate Stage

- 1: In round r , for every neighbor $P_j \in \mathbf{N}_G(P_i) \cap \mathcal{Z}^p$, \mathcal{S}_{OB} sends $([1, 1]_{\mathbf{pk}_{i \rightarrow j}^{(r)}}, \mathbf{pk}_{i \rightarrow j}^{(r)})$ to P_j .

Decrypt Stage

- 1: **if** \mathcal{A} crashed any party in any phase before the current one or $P_o \notin \mathcal{Z}^p$ **then**
- 2: In every round r and for every neighbor $P_j \in \mathbf{N}_G(P_i) \cap \mathcal{Z}^p$, \mathcal{S}_{OB} sends $[1, 1]_{\mathbf{pk}_{j \rightarrow i}^{(r)}}$ to P_j .
- 3: **else**
- 4: In every round r and for every neighbor $P_j \in \mathbf{N}_G(P_i) \cap \mathcal{Z}^p$, \mathcal{S}_{OB} sends $[1, 1]_{\mathbf{pk}_{j \rightarrow i}^{(r)}}$ to P_j unless the following three conditions hold: (a) P_i is the first party not in \mathcal{Z}^p on w , (b) P_j is the last party in \mathcal{Z}^p on w , and (c) $r = 2\mathbf{T} - \ell$.
- 5: If the three conditions hold (in particular $r = 2\mathbf{T} - \ell$), \mathcal{S}_{OB} does the following. If \mathcal{A} did not crash any party in a previous round, \mathcal{S}_{OB} sends $[b^{out}, 0]_{\mathbf{pk}_{j \rightarrow i}^{(r)}}$ to P_j .
- 6: Otherwise, let F denote the set of pairs $(P_f, s - \ell + 1)$ such that \mathcal{A} crashed P_f in round s . \mathcal{S}_{OB} queries $\mathcal{F}_{\text{INFO}}^{OB}$ for the leakage on input $(F, P_i, \mathbf{T} - \ell)$. If the returned value is 1, it sends $[1, 1]_{\mathbf{pk}_{j \rightarrow i}^{(r)}}$ to P_j . Otherwise it sends $[b^{out}, 0]_{\mathbf{pk}_{j \rightarrow i}^{(r)}}$ to party P_j .

Now we prove that no environment can tell whether it is interacting with \mathcal{F}_{NET} and the adversary in the real world or with $\mathcal{F}_{\text{INFO}}$ and the simulator in the ideal world.

We first argue why this simulator simulates correctly the real world. Consider a corrupted party P_c and its honest neighbor P_h . The messages sent from P_h to P_c during the Aggregate Stage are ciphertexts, to which P_h added a layer, and corresponding public keys. Since P_h is honest, the adversary does not know the secret keys corresponding to the sent public keys. Hence, \mathcal{S}_{OB} can simply replace them with encryptions of a pair $(1, 1)$ under a freshly generated public key. The group structure of keys in PKCR guarantees that a fresh key has the same distribution as the composed key (after executing **AddLayer**). Semantic security implies that the encrypted message can be replaced by $(1, 1)$.

Consider now the Decrypt Stage at round r . Let $\mathbf{pk}_{c \rightarrow h}^{(r)}$ be the public key sent by P_c to P_h in the Aggregate Stage (note that this is not the key discussed above; there we argued about keys sent in the opposite direction). \mathcal{S}_{OB} will send to P_c a fresh

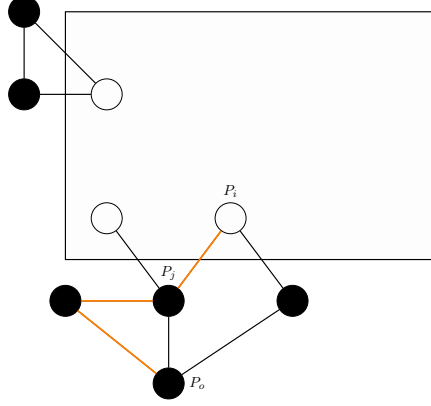


Figure 3-1: An example of the algorithm executed by the simulator \mathcal{S}_{OB} . The filled circles are the corrupted parties. The red line represents the random walk generated by \mathcal{S}_{OB} in Step 5, in this case of length $\ell = 3$. \mathcal{S}_{OB} simulates the Decrypt Stage by sending fresh encryptions of $(1, 1)$ at every round from every honest party to each of its corrupted neighbors, except in round $2T - 3$ from P_i to P_j . If no crash occurred up to that point, \mathcal{S}_{OB} sends encryption of $(b^{out}, 0)$. Otherwise, it queries the leakage oracle about the walk of length $T - 3$, starting at P_i .

encryption under $\mathbf{pk}_{c \rightarrow h}^{(r)}$. We now specify what it encrypts.

Note that the only interesting case is when the party P_o receiving output is corrupted and when we are in the round r in which the (only one) random walk carrying the output enters an area of corrupted parties, containing P_o (that is, when the walk with output contains from P_h all the way to P_o only corrupted parties). In this one message in round r the adversary learns the output of P_o . All other messages are simply encryptions of $(1, 1)$.

For this one meaningful message, we consider three cases. If any party crashed in a phase preceding the current one, \mathcal{S}_{OB} sends an encryption of $(1, 1)$ (as in the real world the walk is made dummy by an unhappy party). If no crashes occurred up to this point (round r in given phase), \mathcal{S}_{OB} encrypts the output received from \mathcal{F}_{BC} . If a crash happened in the current phase, \mathcal{S}_{OB} queries the leakage oracle \mathcal{L}_{OB} , which executes the protocol and tells whether the output or $(1, 1)$ should be sent, essentially telling the simulator if the relevant random walk was disrupted by the crash or not.

Hybrids and security proof. We present a description of the hybrids and security proof.

Hybrid 1. \mathcal{S}_1 simulates the real world exactly. This means, \mathcal{S} has information on the entire topology of the graph, each party's input, and can simulate identically the real world.

Hybrid 2. \mathcal{S}_2 replaces the real keys with the simulated public keys, but still knows everything about the graph as in the first hybrid.

More formally, in each random walk phase and for each party $P_i \in \mathcal{P} \setminus \mathcal{Z}^p$ where $\mathbf{N}_G(P_i) \cap \mathcal{Z}^p \neq \emptyset$, \mathcal{S}_2 generates T key pairs $(\mathbf{pk}_{i \rightarrow j}^{(1)}, \mathbf{sk}_{i \rightarrow j}^{(1)}), \dots, (\mathbf{pk}_{i \rightarrow j}^{(T)}, \mathbf{sk}_{i \rightarrow j}^{(T)})$

for every neighbor $P_j \in \mathbf{N}_G(P_i) \cap \mathcal{Z}^p$. In each round r of the corresponding Aggregate Stage and for every neighbor $P_j \in \mathbf{N}_G(P_i) \cap \mathcal{Z}^p$, \mathcal{S}_2 does the following. P_i receives ciphertext $[b, u]_{\mathbf{pk}_{* \rightarrow i}^{(r)}}$ and the public key $\mathbf{pk}_{* \rightarrow i}^{(r)}$ destined for P_j . Instead of adding a layer and homomorphically OR'ing the bit b_i , \mathcal{S}_2 computes $(b', u') = (b \vee b_i \vee u_i, u \vee u_i)$, and sends $[b', u']_{\mathbf{pk}_{i \rightarrow j}^{(r)}}$ to P_j . In other words, it sends the same message as \mathcal{S}_1 but encrypted with a fresh public key. In the corresponding Decrypt Stage, P_i will get back a ciphertext from P_j encrypted under this exact fresh public key.

Hybrid 3. \mathcal{S}_3 now simulates the ideal functionality during the Aggregate Stage. It does so by sending encryptions of $(1, 1)$ instead of the actual messages and unhappy bits. More formally, in each round r of the Aggregate Stage and for all parties $P_i \in \mathcal{P} \setminus \mathcal{Z}^p$ and $P_j \in \mathbf{N}_G(P_i) \cap \mathcal{Z}^p$, \mathcal{S}_3 sends $[1, 1]_{\mathbf{pk}_{i \rightarrow j}^{(r)}}$ instead of the ciphertext $[b, u]_{\mathbf{pk}_{i \rightarrow j}^{(r)}}$ sent by \mathcal{S}_2 .

Hybrid 4. \mathcal{S}_4 does the same as \mathcal{S}_{OB} during the Decrypt Stage for all steps except for round $2T - \ell$ of the first random walk phase in which the adversary crashes a party. This corresponds to the original description of the simulator except for the 'Otherwise' condition of Step 6 in the Decrypt Stage.

Hybrid 5. \mathcal{S}_5 is the actual simulator \mathcal{S}_{OB} .

In order to prove that no environment can distinguish between the real world and the ideal world, we prove that no environment can distinguish between any two consecutive hybrids when given access to the adversarially-corrupted nodes.

Claim 1. *No efficient distinguisher D can distinguish between Hybrid 1 and Hybrid 2.*

Proof. The two hybrids only differ in the computation of the public keys that are used to encrypt messages in the Aggregate Stage from any honest party $P_i \in \mathcal{P} \setminus \mathcal{Z}^p$ to any dishonest neighbor $P_j \in \mathbf{N}_G(P_i) \cap \mathcal{Z}^p$.

In Hybrid 1, party P_i sends to P_j an encryption under a fresh public key in the first round. In the following rounds, the encryption is sent either under a product key $\overline{\mathbf{pk}}_{i \rightarrow j}^{(r)} = \overline{\mathbf{pk}}_{k \rightarrow i}^{(r-1)} \circledast \mathbf{pk}_{i \rightarrow j}^{(r)}$ or under a fresh public key (if P_i is unhappy). Note that $\overline{\mathbf{pk}}_{k \rightarrow i}^{(r-1)}$ is the key P_i received from a neighbor P_k in the previous round.

In Hybrid 2, party P_i sends to P_j an encryption under a fresh public key $\mathbf{pk}_{i \rightarrow j}^{(r)}$ in every round.

The distribution of the product key used in Hybrid 1 is the same as the distribution of a freshly generated public-key. This is due to the (fresh) $\mathbf{pk}_{i \rightarrow j}^{(r)}$ key which randomizes the product key. Therefore, no distinguisher can distinguish between Hybrid 1 and Hybrid 2. \blacksquare

Claim 2. *No efficient distinguisher D can distinguish between Hybrid 2 and Hybrid 3.*

Proof. The two hybrids differ only in the content of the encrypted messages that are sent in the Aggregate Stage from any honest party $P_i \in \mathcal{P} \setminus \mathcal{Z}^p$ to any dishonest neighbor $P_j \in \mathbf{N}_G(P_i) \cap \mathcal{Z}^p$.

In Hybrid 2, party P_i sends to P_j in the first round an encryption of $(b_i \vee u_i, u_i)$. In the following rounds, P_i sends to P_j either an encryption of $(b \vee b_i \vee u_i, u \vee u_i)$, if message (b, u) is received from neighbor $\pi_i^{-1}(j)$, or an encryption of $(1, 1)$ if no message is received.

In Hybrid 3, all encryptions that are sent from party P_i to party P_j are replaced by encryptions of $(1, 1)$.

Since the simulator chooses a key independent of any key chosen by parties in \mathcal{Z}^p in each round, the key is unknown to the adversary. Hence, the semantic security of the encryption scheme guarantees that the distinguisher cannot distinguish between both encryptions. \blacksquare

Claim 3. *No efficient distinguisher D can distinguish between Hybrid 3 and Hybrid 4.*

Proof. The only difference between the two hybrids is in the Decrypt Stage. We differentiate two cases:

- A phase where the adversary did not crash any party in this or any previous phase. In this case, the simulator \mathcal{S}_3 sends an encryption of (b_W, u_W) , where $b_W = \bigvee_{P_j \in W} b_j$ is the OR of all input bits in the walk and $u_W = 0$, since no crash occurred. \mathcal{S}_4 sends an encryption of $(b^{out}, 0)$, where $b^{out} = \bigvee_{P_i \in \mathcal{P}} b_i$. Since the graph is connected, $b^{out} = b_W$ with overwhelming probability, as proven in Lemma 1. Also, the encryption in Hybrid 4 is done with a fresh public key which is indistinguishable with the encryption done in Hybrid 3 by OR'ing many times in the graph, as shown in Claim 2.1 in [1].
- A phase where the adversary crashed a party in a previous phase or any round different than $2T - \ell$ of the first phase where the adversary crashes a party. In Hybrid 4 the parties send an encryption of $(1, 1)$. This is also the case in Hybrid 3, because even if a crashed party disconnected the graph, each connected component contains a neighbor of a crashed party. Moreover, in Hybrid 4, the messages are encrypted with a fresh public key, and in Hybrid 3, the encryptions are obtained by the homomorphic OR operation. Both encryptions are indistinguishable, as shown in Claim 2.1 in [1].

\blacksquare

Claim 4. *No efficient distinguisher D can distinguish between Hybrid 4 and Hybrid 5.*

Proof. The only difference between the two hybrids is in the Decrypt Stage, at round $2T - \ell$ of the first phase where the adversary crashes.

Let F be the set of pairs (P_f, r) such that \mathcal{A} crashed P_f at round r of the phase. In Hybrid 4, a walk W of length T is generated from party P_o . Let W_1 be the region

of W from P_o to the first not passively corrupted party and let W_2 be the rest of the walk. Then, the adversary's view at this step is the encryption of $(1, 1)$ if one of the crashed parties breaks W_2 , and otherwise an encryption of $(b_W, 0)$. In both cases, the message is encrypted under a public key for which the adversary knows the secret key.

In Hybrid 5, a walk W'_1 is generated from P_o of length $\ell \leq T$ ending at the first not passively corrupted party P_i . Then, the simulator queries the leakage function on input $(F, P_i, T - \ell)$, which generates a walk W'_2 of length $T - \ell$ from P_i , and checks whether W'_2 is broken by any party in F . If W'_2 is broken, P_i sends an encryption of $(1, 1)$, and otherwise an encryption of $(b_W, 0)$. Since the walk W' defined as W'_1 followed by W'_2 follows the same distribution as W , $b_W = b^{out}$ with overwhelming probability, and the encryption with a fresh public key which is indistinguishable with the encryption done by OR'ing many times in the graph, then it is impossible to distinguish between Hybrid 4 and Hybrid 5. ■

This concludes the proof of soundness. □

3.4.2 Protocol Leaking a Fraction of a Bit

In this section we present a broadcast protocol BC-FB_p , which leaks only a fraction of a bit of information about the network topology. If no crashes occur during the execution of BC-FB_p , then no information about the topology is leaked and all parties receive the correct output with overwhelming probability.

Formally, fractional leakage is modeled by an oracle, which is very similar to the one presented in Section 3.4.1 for the protocol BC-OB , however, it answers its query only with probability p . That is, with probability $1 - p$ the response is \perp (in this case no information leaks) and with probability p it is either 0 or 1, according to the leaked information.

Function $\mathcal{L}_{\text{FB}_p}((F, P_s, T'), \mathcal{C}, G)$

Let $p' = 1/\lceil 1/p \rceil$. With probability p' , return $\mathcal{L}_{\text{OB}}((F, P_s, T'), \mathcal{C}, G)$ and with probability $1 - p'$ return \perp .

The probability p is a parameter of our protocol. In other words, we can tolerate arbitrarily low leakage. However, the communication and round complexity of our protocol are proportional to $1/p$. As a consequence, $1/p$ must be polynomial and the leakage probability p cannot be negligible.

The Protocol BC-FB_p . The idea is to leverage the fact that the adversary can gain information from crashing fail-stop corrupted parties only in one execution of RandomWalkPhase . (This is because a phase with a crash causes an abort, that is, all subsequent phases produce no outputs and reveal no information.) Imagine that RandomWalkPhase can randomly fail with probability $1 - p$. A failure means that an execution does not give any outputs nor any information (similarly to the behavior

in case of an abort). Moreover, assume that a failure becomes apparent (both to the honest parties and to the adversary) only at the last moment, when the output is computed. Of course, we now have to repeat the phase a number of times, in order to preserve correctness. However, the adversary has only one chance to choose the execution of **RandomWalkPhase**, in which it crashes the first party, at the same time losing the ability to learn anything from the subsequent phases. Moreover, it must choose it before it knows whether the execution failed. Since the chance of failure is $1 - p$, the adversary learns any information only with probability p .

More formally, the protocol **BC-FB_p**, similarly to protocol **BC-OB**, consists of n random-walk phases. However, each of the n phases is modified so that with probability at least $1 - p$ it fails, and then repeated ρ times (for sufficiently large ρ). We now describe the modification, which makes a random-walk phase fail. The change concerns only the Aggregate Stage. The idea is that instead of an encrypted tuple $[b, u]$, $\lfloor 1/p \rfloor + 1$ encrypted values $[b^1, \dots, b^{\lfloor 1/p \rfloor}, u]$ are sent along the walk. The bit u is the OR of the unhappy-bits of the parties on the walk. Every bit b^k is equivalent to the bit b in the protocol **RandomWalkPhase**. That is, b^k is the OR of the input bits and the unhappy-bits of the parties. However, all but one of the $\lfloor 1/p \rfloor$ bits b^k are initially set to 1 by the party who starts the walk. This means that only one of the bits b^k is not dummy, but without knowing the secret key it is impossible to tell which one. Throughout the Aggregate Stage, the parties process every ciphertext corresponding to a bit b^k the same way they processed the encryption of b in **RandomWalkPhase**. The encryption of the unhappy-bit u is also processed the same way. However, before sending the ciphertexts to the next party on the walk, the encryptions of the bits b^k are randomly shuffled. This way we make sure that the adversary does not know which of the ciphertexts contain dummy values (the adversary cannot tell which values are dummy as soon as they have been permuted by at least one non-passively corrupted party). At the end of the Aggregate Stage (after T rounds), the last party chooses uniformly at random one of the $\lfloor 1/p \rfloor$ ciphertexts and uses it, together with the encryption of the unhappy-bit, to execute the Decrypt Stage as in **RandomWalkPhase**.

After ρ repetitions of the phase, a party P_o does the following in order to get the output. If in any repetition the unhappy-bit was set to 1, P_o cannot compute the output (that is, an abort was detected). Otherwise, P_o outputs the AND of all bits output by individual repetitions.

Overall, the global parameters passed to **BC-FB_p** are the length T of the random walk and the number ρ of repetitions of a phase. As in **BC-OB**, each party P_i has input (d_i, b_i, u_i) , where d_i is its number of neighbors, u_i is its the unhappy-bit, and b_i is the input bit.

A formal description of the modified protocol **ProbabilisticRandomWalkPhase_p** for the random-walk phase can be found in Appendix 3.9.1. Note that this protocol should be repeated ρ times in the actual protocol.

Theorem 1. *Let κ be the security parameter. For $\tau = \log(n) + \kappa$, $T = 8n^3\tau$, and $\rho = \tau/(p' - 2^{-\tau})$, where $p' = 1/\lfloor 1/p \rfloor$, protocol **BC-FB_p**($T, \rho, (d_i, b_i)_{P_i \in \mathcal{P}}$) topology-hidingly realizes $\mathcal{F}_{\text{INFO}}^{FB_p} || \mathcal{F}_{\text{BC}}$ (with abort) in the \mathcal{F}_{NET} hybrid-world, where the leakage*

function \mathcal{L}_{FB_p} is the one defined as above. If no crashes occur, then there is no abort and there is no leakage.

Proof. Completeness. We first show that the protocol is complete. That is, that if the adversary does not crash any party, then every party gets the correct output (the OR of all input bits) with overwhelming probability. More specifically, we show that if no crashes occur, then after ρ repetitions of a phase, the party P_o outputs the correct value with probability at least $1 - 2^{-(\kappa + \log(n))}$. The overall completeness follows from the union bound: the probability that all n parties output the correct value is at least $1 - 2^{-\kappa}$.

Notice that if the output of any of the ρ repetitions intended for P_o is correct, then the overall output of P_o is correct. A given repetition can only give an incorrect output when either the random walk does not reach all parties, which happens with probability at most $2^{-\tau}$, or when the repetition fails, which happens with probability $1 - p'$. Hence, the probability that a repetition gives the incorrect result is at most $1 - p' + 2^{-\tau}$. The probability that all repetitions are incorrect is then at most $(1 - p' + 2^{-\tau})^\rho \leq 2^{-(\kappa + \log(n))}$ (the inequality holds for $0 \leq p' - 2^{-\tau} \leq 1$).

Soundness. We show that no environment can distinguish between the real world and the simulated world, when given access to the adversarially-corrupted nodes. The simulator \mathcal{S}_{FB} for BC-FB $_p$ is a modification of \mathcal{S}_{OB} . Here we only sketch the changes and argue why \mathcal{S}_{FB} simulates the real world. A formal proof of the soundness of BC-FB $_p$ can be found in Appendix 3.9.2.

In each of the ρ repetitions of a phase, \mathcal{S}_{FB} executes a protocol very similar to the one for \mathcal{S}_{OB} . In the Aggregate Stage, \mathcal{S}_{FB} proceeds almost identically to \mathcal{S}_{OB} (except that it sends encryptions of vectors $(1, \dots, 1)$ instead of only two values). In the Decrypt Stage the only difference between \mathcal{S}_{FB} and \mathcal{S}_{OB} is in computing the output for the party P_o (as already discussed in the proof of Theorem 5, \mathcal{S}_{FB} does this only when P_o is corrupted and the walk carrying the output enters an area of corrupted parties). In the case when there were no crashes before or during given repetition of a phase, \mathcal{S}_{OB} would simply send the encrypted output. On the other hand, \mathcal{S}_{FB} samples a value from the Bernoulli distribution with parameter p and sends the encrypted output only with probability p , while with probability $1 - p$ it sends the encryption of $(1, 0)$. Otherwise, the simulation is the same as for \mathcal{S}_{OB} .

It can be easily seen that \mathcal{S}_{FB} simulates the real world in the Aggregate Stage and in the Decrypt Stage in every message other than the one encrypting the output. But even this message comes from the same distribution as the corresponding message sent in the real world. This is because in the real world, if the walk was not broken by a crash, this message contains the output with probability p . The simulator encrypts the output also with probability p in the two possible cases: when there was no crash (\mathcal{S}_{FB} samples from the Bernoulli distribution) and when there was a crash but the walk was not broken (\mathcal{L}_{FB} is defined in this way). □

□

3.5 Getting Topology-Hiding Computation

We have just shown how to get topology-hiding broadcasts. To get additional functionality (e.g. for compiling MPC protocols), we have to be able to compose these broadcasts. When there is no leakage, this is straightforward: we can run as many broadcasts in parallel or in sequence as we want and they will not affect each other. However, if we consider a broadcast secure in the fail-stop model that leaks at most 1 bit, composing t of these broadcasts could lead to leaking t bits.

Naive composition of broadcast. The first thing to try is to naively compose broadcasts. It turns out that this will not work — we will incur too much leakage. Given black-box access to a fail-stop secure topology-hiding broadcast with a leakage function, the naive thing to do to compose broadcasts is run both broadcasts, either in parallel or sequentially. So, consider composing two broadcasts together, first in parallel. Each protocol is running independently, and so if there is an abort, the simulator will need to query the leakage function twice, unless we can make the specific claim that the leakage function will output a correlated bit for independent instances given the same abort (note that our construction does not have this property).

If we run the protocols sequentially, we'll need to make a similar claim. If we are simulating this composition and there is both an abort in the first broadcast and the second, then we definitely need to query the leakage function for the first abort. Then, unless we can make specific claims about how we could start a broadcast protocol *after* there has already been an abort, we will need to query the leakage oracle again.

Topology-hiding computation without aggregated leakage. In this section we show how to transform any protocol secure in the presence of passive and fail-stop corruptions into one that additionally hides the network topology and leaks at most an arbitrary fraction p of a bit *in total*. To achieve this, we first consider a variant of parallel composition of the broadcast protocol BC-FB_p . That is, we show how to modify BC-FB_p to construct an all-to-all multibit broadcast protocol (where each party simultaneously broadcasts a k bit message), which leaks only a fraction p of a bit and has the same round complexity as BC-FB_p . Then, we consider sequential composition of all-to-all broadcasts and note that the technique we use is in fact more general and can be applied to a wider class of protocols which leak information. With the above statements, we conclude that any protocol can be compiled into one that leaks only a fraction p of a bit in total. As a corollary, any functionality \mathcal{F} can be implemented by a topology hiding protocol leaking any fraction p of a bit.

3.5.1 All-to-all Multibit Broadcast

We show how to edit the protocol BC-FB_p to implement all-to-all multibit broadcasts, meaning we can broadcast k multibit messages from k not-necessarily distinct parties in a single broadcast. The edited protocol leaks a fraction p of a bit in total. While this transformation is not essential to compile MPC protocols to topology-hiding ones, it will cut down the round complexity by a factor of n times the size of a message.

First observe that BC-FB_p actually works also to broadcast multiple bits. Instead

of sending a single bit during the random-walk protocol, it is enough that parties send vectors of ciphertexts. That is, in each round parties send a vector $[\mathbf{b}_1, \dots, \mathbf{b}_\ell, u]$.

Now we show how to achieve an all-to-all broadcast. Assume each party P_i wants to broadcast some k -bit message, (b_1, \dots, b_k) . We consider a vector of length nk , where each of the n parties is assigned to k slots for k bits of its message. Each of the vectors \mathbf{b}_i in the vector $[\mathbf{b}_1, \dots, \mathbf{b}_\ell, u]$ described above will be of this form. P_i will use the slots from $n(i-1)$ to ni to communicate its message. This means that P_i will have as input vector $\mathbf{b}_i = (0, \dots, 0, b_1, \dots, b_k, 0, \dots, 0)$. Then, in the Aggregate Stage, the parties will input their input message into their corresponding slots (by homomorphically OR'ing the received vector with its input message). At the end of the protocol, each party will receive the output containing the broadcast message of each party P_j in the slots $n(j-1)$ to nj .

More formally, we have the following Lemma, proven in Appendix 3.10.1.

Lemma 2. *Protocol $BC\text{-}FB_p$ can be edited to an all-to-all multi-bit broadcast $\text{Multibit}BC_p$, which is secure against an adversary, who statically passively corrupts and adaptively crashes any number of parties and leaks at most a fraction p of a bit. The round complexity of $\text{Multibit}BC_p$ is the same as for $BC\text{-}FB_p$.*

3.5.2 Sequential Execution Without Aggregated Leakage

We show how to construct a protocol, which implements any number of sequential executions of the protocol $\text{Multibit}BC_p$, while preserving the leakage of a fraction p of a bit in total. The construction makes non-black-box use of the unhappy bits used in $\text{Multibit}BC_p$. The idea is simply to preserve the state of the unhappy bits between sequential executions. That is, once some party sees a crash, it will cause all subsequent executions to abort. For a proof of the following lemma, we refer to Appendix 3.10.2.

Lemma 3. *There exists a protocol, which implements any number k of executions of $\text{Multibit}BC_p$, is secure against an adversary, who statically passively corrupts and adaptively crashes any number of parties and leaks at most a fraction p of a bit in total. The complexity of the constructed protocol is k times the complexity of $\text{Multibit}BC_p$.*

Remark 1. *We note that the above technique to sequentially execute protocols which leak p bits and are secure with abort can be applied to a more general class of protocols (in particular, not only to our topology-hiding broadcast). The idea is that if a protocol satisfies the property that any abort before it begins implies that the protocol does not leak any information, then it can be executed sequentially leaking at most p bits.*

3.5.3 Topology-Hiding Computation

We are now ready to compile any MPC protocol (secure against an adversary, who statically passively corrupts and adaptively crashes any number of parties) into one that is topology-hiding and leaks at most a fraction p of a bit.

To do this, it is enough to do a standard transformation using public key infrastructure. Let Π_{MPC} be a protocol that runs in M rounds. First, the parties use one all-to-all multi-bit topology hiding broadcast protocol to send each public key to every other party. Then, each round of Π_{MPC} is simulated: the parties run n all-to-all multi-bit topology hiding broadcasts simultaneously to send the messages sent in that round encrypted under the corresponding public keys. After the broadcasts, each party can use their secret key to decrypt their corresponding messages. For a proof of the following theorem, we refer to Appendix 3.10.3.

Theorem 6. *Assume PKCR exists. Then, we can compile any MPC protocol Π_{MPC} that runs in M rounds into a topology-hiding protocol with leakage function \mathcal{L}_{FB_p} , that runs in $MR + 1$ rounds, where R is the round complexity of $BC-FB_p$.²*

We can now conclude that any MPC functionality can be implemented by a topology-hiding protocol. Since PKCR is implied by either DDH, QR or LWE, we get the following theorem as a corollary.

Theorem 2. *If DDH, QR or LWE is hard, then any MPC functionality \mathcal{F} can be realized by a topology-hiding protocol which is secure against an adversary that does any number of static passive corruptions and adaptive crashes, leaking an arbitrarily small fraction p of a bit. The round and communication complexity is polynomial in κ and $1/p$.*

Proof. Because every poly-time computable functionality \mathcal{F} has an MPC protocol [17], we get that Theorem 6 implies we can get topology-hiding computation. The round and communication complexity is implied by Theorem 6 and the complexity of MultibitBC_p. □ □

3.6 Efficient Topology-Hiding Computation with FHE

One thing to note is that compiling MPC from broadcast is rather expensive, especially in the fail-stop model; we need a broadcast for every round. However, we will show that an FHE scheme with additive overhead can be used to evaluate any polynomial-time function f in a topology-hiding manner. Additive overhead applies to ciphertext versus plaintext sizes and to error with respect to all homomorphic operations if necessary. We will employ an altered random walk protocol, and the total number of rounds in this protocol will amount to that of a single broadcast. We remark that FHE with additive overhead can be obtained from subexponential iO and subexponentially secure OWFs (probabilistic iO), as shown in [16].

Intuitively, we use the FHE scheme to add a layer by encrypting the ciphertext (and corresponding public key). Because the scheme is fully homomorphic, no matter

²In particular, the complexity of $BC-FB_p$ is $n \cdot \rho \cdot 2T$, where κ is the security parameter, $\tau = \log(n) + \kappa$, $T = 8n^3\tau$ is the length of a walk and $\rho = \tau/(p' - 2^{-\tau})$ is the number of repetitions of a phase (with $p' = 1/\lceil 1/p \rceil$).

how many layers there are, we are able to homomorphically evaluate any functionality through all of the layers.

To describe our method in more detail, we define a variant of a PKCR scheme, called DFH-PKE, and show how to instantiate such a scheme using FHE.

3.6.1 Deeply Fully-Homomorphic Public-Key Encryption

A deeply fully-homomorphic PKE scheme is an enhanced PKE scheme \mathcal{E} where (a) one can add and remove layers of encryption, while (b) one can homomorphically compute any function on encrypted bits (independent of the number of layers). This will be captured by three additional algorithms: AddLayer_r , DelLayer_r , and HomOp_r , operating on ciphertexts with r layers of encryption (we will call such ciphertexts level- r ciphertexts). A level- r ciphertext is encrypted under a level- r public key (each level can have different key space).

Adding a layer requires a new secret key \mathbf{sk} for \mathcal{E} . The algorithm AddLayer_r takes as input a vector of level- r ciphertexts $\llbracket \mathbf{m} \rrbracket_{\mathbf{pk}}$ encrypted under a level- r public key, the corresponding level- r public key \mathbf{pk} , and a new secret key \mathbf{sk} . It outputs a vector of level- $(r+1)$ ciphertexts and the level- $(r+1)$ public key, under which it is encrypted.

Deleting a layer is the opposite of adding a layer. The algorithm DelLayer_r deletes a layer from a level- $(r+1)$ ciphertext. It takes as input a vector of level- $(r+1)$ ciphertexts $\llbracket m \rrbracket_{\mathbf{pk}}$, the corresponding level- $(r+1)$ public key \mathbf{pk} , and the secret key \mathbf{sk} that corresponds to the secret key used to add the last layer. It then outputs a vector of level- r ciphertexts and the level- r public key.

With HomOp_r , one can compute any function on a vector of encrypted messages. It takes a vector of level- r ciphertexts encrypted under a level- r public key, the corresponding level- r public key \mathbf{pk} and a function from a permitted set \mathcal{F} of functions. It outputs a level- r ciphertext that contains the output of the function applied to the encrypted messages.

To describe the security of our enhanced encryption scheme, we will require that there exists an algorithm Leveled-Encrypt_r , which takes as input a plain message and a level- r public key, and outputs a level- r ciphertext. We will then require that from the output of AddLayer_r (DelLayer_r) one cannot obtain any information on the underlying layers of encryption. That is, that the output of AddLayer_r (DelLayer_r) is indistinguishable from a level- $(r+1)$ (level- r) encryption of the message, i.e., the output of $\text{Leveled-Encrypt}_{r+1}$ (Leveled-Encrypt_r) under a level- $(r+1)$ (level- r) public key. We will also require that the output of HomOp_r is indistinguishable from a level- r encryption of the output of the functions applied to the messages.

We refer to Appendix 3.11 for a formal definition of a DFH-PKE scheme and to Appendix 3.11.1 from an instantiation from FHE.

3.6.2 Topology Hiding Computation from DFH-PKE.

We will use DFH-PKE to alter the RandomWalkPhase protocol (and by extension we can also alter the protocol $\text{ProbabilisticRandomWalkPhase}_p$). Then, executing protocols BC-OB and BC-FB_p that leak one bit and a fraction of a bit respectively will be

able to evaluate any poly-time function instead, while still leaking the same amount of information as a broadcast using these random walk protocols. The concept is simple. During the Aggregate Stage, parties will add a leveled encryption of their input and identifying information to a vector of ciphertexts, while adding a layer — we will not need sequential id's if each party knows where their input should go in the function. Then, at the end of the Aggregate Stage, nodes homomorphically evaluate f' , which is the composition of a parsing function, to get one of each inputs in the right place, and f , to evaluate the function on the parsed inputs. The result is a leveled ciphertext of the output of f . This ciphertext is un-layered in the Decrypt Stage so that by the end, the relevant parties get the output. We refer to Appendix 3.12 for a more detailed description of the protocol DFH-THC.

We will provide the formal theorem statement for the soundness of the FHE topology-hiding computation statement, and sketch the proof.

Theorem 7. *For security parameter κ , $\tau = \log(n) + \kappa$, $T = 8n^3\tau$, and $\rho = \tau/(p' - 2^{-\tau})$, where $p' = 1/\lfloor 1/p \rfloor$, the protocol $\text{DFH-THC}(T, \rho, (d_i, \text{input}_i)_{P_i \in \mathcal{P}})$ topology-hidingly evaluates any poly-time function f , $\mathcal{F}_{\text{INFO}}^{FB_p} || f$ in the \mathcal{F}_{NET} hybrid-world.*

Sketch. This proof will look almost exactly like the proof of Theorem 1. The simulator and its use of the leakage oracle will behave in nearly the same manner as before.

- During the Aggregate Stage, the simulator sends leveled encryptions of 1 of the appropriate size with the appropriate number of layers.
- During the Decrypt Stage, the simulator sends the output encrypted with the appropriate leveled keys.

Because Leveled-Encrypt_r is able to produce a distribution of ciphertexts that looks identical to AddLayer_r , and by semantic security of the FHE scheme, no party can tell what other public keys were used except the most recently added one, the simulated ciphertexts and public keys are computationally indistinguishable from those in the real walk.

It is also worth pointing out that as long as the FHE scheme only incurs additive blowup in error and size, and $T = \text{poly}(\kappa)$, the ciphertexts being passed around are only $\text{poly}(\kappa)$ in size. □ □

Remarks. The definition of a DFH-PKE scheme can be relaxed to only being able to homomorphically evaluate the OR. It turns out that this relaxation is implied by any OR-homomorphic PKCR scheme and is strong enough to prove the security of the protocols BC-OB and BC-FB_p. However, for simplicity, we decided to describe our protocols BC-OB and BC-FB_p from a OR-homomorphic PKCR scheme.

3.7 Security Against Semi-malicious Adversaries

In this section, we present a generic idea for turning any topology-hiding protocol secure against an adversary, who can statically passively corrupt and adaptively crash

parties into one secure against a semi-malicious adversary (who can also adaptively crash parties). The core idea of our transformation is to first generate “good” random tapes for all parties and then execute the given protocol with parties using the pre-generated random tapes instead of generating randomness on the fly. Hence, the transformed protocol proceeds in two phases: Randomness Generation and Deterministic Execution.

Randomness Generation. The goal of the first phase is to generate for each party P_i a uniform random value r_i , which can then be used as randomness tape of P_i in the phase of Deterministic Execution. To generate those random values the parties use the following flooding scheme over n rounds.

Protocol GenerateRandomness

- 1: Each party P_i generates $n + 1$ uniform random values $s_i^{(0)}, s_i^{(1)}, \dots, s_i^{(n)}$ and sets $r_i^{(0)} := s_i^{(0)}$.
- 2: **for** any round r from 1 to n **do**
- 3: Each party P_i sends $r_i^{(r-1)}$ to all its neighbors.
- 4: Each party P_i computes $r_i^{(r)}$ as the sum of all values received from its (non-crashed) neighbors in the current round and the value $s_i^{(r)}$.
- 5: Each party P_i outputs $r_i := r_i^{(n)}$.

Lemma 4. *Let G' be the network graph without parties which crashed during the execution of the protocol **GenerateRandomness**. Any party P_i whose connected component in G' contains at least one honest party will output a uniform value r_i . The output of any honest party is not known to the adversary. The protocol **GenerateRandomness** does not leak any information about the network-graph (even if crashes occur).*

Proof. First observe that all randomness is chosen at the beginning of the first round. The rest of the protocol is completely deterministic. This implies that the adversary has to choose the randomness of corrupted parties independently of the randomness chosen by honest parties.

If party P_i at the end of the protocol execution is in a connected component with honest party P_j , the output r_i is a sum which contains at least one of the values $s_j^{(r)}$ from P_j . That summand is independent of the rest of the summands and uniform random. Thus, r_i is uniform random as well.

Any honest party will (in the last round) compute its output as a sum which contains a locally generated truly random value, which is not known to the adversary. Thus, the output is also not known to the adversary.

Finally, observe that the message pattern seen by a party is determined by its neighborhood. Moreover, the messages received by corrupted parties from honest parties are uniform random values. This implies, that the view of the adversary in this protocol can be easily simulated given the neighborhood of corrupted parties. Thus, the protocol does not leak any information about the network topology. $\square \quad \square$

Transformation to Semi-Malicious Security. In the second phase of Deterministic Execution, the parties execute the protocol secure only against passive and fail-stop corruptions, but instead of generating fresh randomness during the protocol execution, the parties use the random tape generated in the first phase. If a party witnessed a crashed in the first phase, it pretends the this crash happens in the first round of the actual protocol and proceeds accordingly.

Protocol EnhanceProtocol(Π)

1. The parties execute **GenerateRandomness** to generate random tapes.
2. The parties execute the actual protocol:
 - Parties use the generated randomness tapes instead of generating randomness on the fly.
 - If a party witnessed a crashed in the **GenerateRandomness**, it pretends that it witnessed this crash in the first round of the protocol Π (and proceeds in the protocol accordingly).

Theorem 4. *Let \mathcal{F} be an MPC functionality and let Π be a protocol that topology-hidingly realizes \mathcal{F} in the presence of static passive corruptions and adaptive crashes. Then, the protocol **EnhanceProtocol**(Π) topology-hidingly realizes \mathcal{F} in the presence of static semi-malicious corruption and adaptive crashes. The leakage stays the same.*

Proof. (sketch) The randomness generation protocol **GenerateRandomness** used in the first phase is secure against a semi-malicious fail-stopping adversary. Lemma 4 implies that the random tape of any semi-malicious party that can interact with honest parties is truly uniform random. Moreover, the adversary has no information on the random tapes of honest parties. This implies that the capability of the adversary in the execution of the actual protocol in the second phase (which for fixed random tapes is deterministic) is the same as for an semi-honest fail-stopping adversary. This implies that the leakage of **EnhanceProtocol**(Π) is the same as for Π as the randomness generation protocol does not leak information (even if crashes occur). \square \square

Remark 2. *To improve overall communication complexity of the protocol the values generated in the first phase could be used as local seeds for a PRG which is then used to generate the actual random tapes.*

As a corollary of Theorem 1 and 4, we obtain that any MPC functionality can be realized in a topology hiding manner against an adversary that does any number of static semi-malicious corruptions and adaptive crashes, leaking at most an arbitrary small fraction p of information about the topology.

Corollary 2. *cor:FinalIf DDH, QR or LWE is hard, then for any MPC functionality \mathcal{F} there is a topology-hiding protocol realizing \mathcal{F} leaking at most an arbitrarily small fraction p of a bit, which is secure against an adversary that does any number of static semi-malicious corruptions and adaptive crashes. The round and communication complexity is polynomial in κ and $1/p$.*

3.8 LWE based OR-Homomorphic PKCR Encryption

In this section we show how to get a PKCR encryption scheme from the LWE assumption. Basis of our PKCR scheme is the public-key crypto-system proposed in [66]. Let us briefly recall the public-key crypto-system:

LWE PKE scheme [66] Let κ be the security parameter of the cryptosystem. The cryptosystem is parameterized by two integers m, q and a probability distribution χ on \mathbb{Z}_q . To guarantee security and correctness of the encryption scheme, one can choose $q \geq 2$ to be some prime number between κ^2 and $2\kappa^2$, and let $m = (1+\epsilon)(\kappa+1) \log q$ for some arbitrary constant $\epsilon > 0$. The distribution χ is a discrete gaussian distribution with standard deviation $\alpha(\kappa) := \frac{1}{\sqrt{\kappa \log^2 \kappa}}$.

Key Generation: *Setup:* For $i = 1, \dots, m$, choose m vectors $\mathbf{a}_1, \dots, \mathbf{a}_m \in \mathbb{Z}_q^\kappa$ independently from the uniform distribution. Let us denote $A \in \mathbb{Z}_q^{m \times \kappa}$ the matrix that contains the vectors \mathbf{a}_i as rows.

Secret Key: Choose $\mathbf{s} \in \mathbb{Z}_q^\kappa$ uniformly at random. The secret key is $\mathbf{sk} = \mathbf{s}$.

Public Key: Choose the error coefficients $e_1, \dots, e_m \in \mathbb{Z}_q$ independently according to χ . The public key is given by the vectors $b_i = \langle \mathbf{a}_i, \mathbf{sk} \rangle + e_i$. In matrix notation, $\mathbf{pk} = A \cdot \mathbf{sk} + \mathbf{e}$.

Encryption: To encrypt a bit b , we choose uniformly at random $\mathbf{x} \in \{0, 1\}^m$. The ciphertext is $c = (\mathbf{x}^\top A, \mathbf{x}^\top \mathbf{pk} + b \frac{q}{2})$.

Decryption: Given a ciphertext $c = (c_1, c_2)$, the decryption of c is 0 if $c_2 - c_1 \cdot \mathbf{sk}$ is closer to 0 than to $\lfloor \frac{q}{2} \rfloor$ modulo q . Otherwise, the decryption is 1.

To extend this scheme to a PKCR scheme, we need to provide algorithms to rerandomize ciphertexts, to add and remove layers of encryption, and to homomorphically compute the OR. To obtain the OR-homomorphic property, it is enough to provide a XOR-Homomorphic PKCR encryption scheme, as was shown in [1].

Extension to PKCR We now extend the above PKE scheme to satisfy the requirements of PKCR (cf. Section 3.2.2). For this we show how to rerandomize ciphertexts, how add and remove layers of encryption, and finally how to homomorphically compute XOR.

Rerandomization: We note that a ciphertext can be rerandomized, which is done by homomorphically adding an encryption of 0. The algorithm **Rand** takes as input a ciphertext and the corresponding public key, as well as a (random) vector $\mathbf{x} \in \{0, 1\}^m$.

Algorithm Rand($c = (c_1, c_2), \mathbf{pk}, \mathbf{x}$)

return $(c_1 + \mathbf{x}^\top A, c_2 + \mathbf{x}^\top \mathbf{pk})$.

Adding and Deleting Layers of Encryption: Given an encryption of a bit b under the public key $\mathbf{pk} = A \cdot \mathbf{sk} + \mathbf{e}$, and a secret key \mathbf{sk}' with corresponding public key $\mathbf{pk}' = A \cdot \mathbf{sk}' + \mathbf{e}'$, one can add a layer of encryption, i.e. obtain a ciphertext under the public key $\mathbf{pk} \cdot \mathbf{pk}' := A \cdot (\mathbf{sk} + \mathbf{sk}') + \mathbf{e} + \mathbf{e}'$. Also, one can delete a layer of encryption.

Algorithm AddLayer($c = (c_1, c_2), \mathbf{sk}$)

return $(c_1, c_1 \cdot \mathbf{sk} + c_2)$

Algorithm DelLayer($c = (c_1, c_2), \mathbf{sk}$)

return $(c_1, c_2 - c_1 \cdot \mathbf{sk})$

Error Analysis Every time we add a layer, the error increases. Hence, we need to ensure that the error does not increase too much. After l steps, the error in the public key is $\mathbf{pk}_{0..l} = \sum_{i=0}^l \mathbf{e}_i$, where \mathbf{e}_i is the error added in each step.

The error in the ciphertext is $c_{0..l} = \sum_{i=0}^l \mathbf{x}_i \sum_{j=0}^i \mathbf{e}_j$, where the \mathbf{x}_i is the chosen randomness in each step. Since $\mathbf{x}_i \in \{0, 1\}^m$, the error in the ciphertext can be bounded by $m \cdot \max_i \{\|\mathbf{e}_i\|_\infty\} \cdot l^2$, which is quadratic in the number of steps.

Homomorphic XOR: A PKCR encryption scheme requires a slightly stronger version of homomorphism. In particular, homomorphic operation includes the rerandomization of the ciphertexts. Hence, the algorithm **hXor** also calls **Rand**. The inputs to **hXor** are two ciphertexts encrypted under the same public key and the corresponding public key.

Algorithm hXor($c = (c_1, c_2), c' = (c'_1, c'_2), \mathbf{pk}$)

Set $c'' = (c_1 + c'_1, c_2 + c'_2)$.

Choose $\mathbf{x} \in \{0, 1\}^m$ uniformly at random.

return **Rand**($c'', \mathbf{pk}, \mathbf{x}$)

Failstop Supplementary Material

3.9 Protocols

This section contains supplementary material for Section 3.4.

3.9.1 Protocol Leaking a Fraction of a Bit

In this section we give a formal description of the random-walk phase **ProbabilisticRandomWalkPhase_p** for the broadcast protocol **BC-FB_p** from Section 3.4.2. The boxes indicate the parts

where it differs from the random-walk phase protocol **RandomWalkPhase** for the broadcast protocol leaking one bit (cf. Section 3.4.1).

Protocol ProbabilisticRandomWalkPhase_p($T, P_o, (d_i, b_i, u_i)_{P_i \in \mathcal{P}}$)

Initialization Stage:

Each party P_i generates $T \cdot d_i$ keypairs $(\mathbf{pk}_{i \rightarrow j}^{(r)}, \mathbf{sk}_{i \rightarrow j}^{(r)}) \leftarrow \text{KeyGen}(1^\kappa)$ where $r \in \{1, \dots, T\}$ and $j \in \{1, \dots, d_i\}$.

Each party P_i generates $T - 1$ random permutations on d_i elements $\{\pi_i^{(2)}, \dots, \pi_i^{(T)}\}$.

For each party P_i , if any of P_i 's neighbors crashed in any phase before the current one, then P_i becomes unhappy, i.e., sets $u_i = 1$.

Aggregate Stage: Each party P_i does the following:

// Send first ciphertexts

if P_i is the recipient P_0 **then**

Party P_i sends to the first neighbor the public key $\mathbf{pk}_{i \rightarrow 1}^{(1)}$ and the ciphertext

$[b_i \vee u_i, 1, \dots, 1, u_i]_{\mathbf{pk}_{i \rightarrow 1}^{(1)}}$ ($\lfloor 1/p \rfloor$ ciphertexts contain $b_i \vee u_i$).

Party P_i sends to any other neighbor P_j ciphertext $[1, \dots, 1, 1]_{\mathbf{pk}_{i \rightarrow j}^{(1)}}$ and the

public key $\mathbf{pk}_{i \rightarrow j}^{(1)}$.

else

Party P_i sends to each neighbor P_j ciphertext $[1, \dots, 1, 1]_{\mathbf{pk}_{i \rightarrow j}^{(1)}}$ and the public

key $\mathbf{pk}_{i \rightarrow j}^{(1)}$.

// Add layer while ORing own input bit

for any round r from 2 to T **do**

For each neighbor P_j of P_i , do the following:

if P_i did not receive a message from P_j **then**

Let $k = \pi_i^{(r)}(j)$.

Party P_i sends ciphertext $[1, \dots, 1, 1]_{\mathbf{pk}_{i \rightarrow k}^{(r)}}$ and public key $\mathbf{pk}_{i \rightarrow k}^{(r)}$ to neighbor

P_k .

else

Let $k = \pi_i^{(r)}(j)$. Let $\mathbf{c}_{j \rightarrow i}^{(r-1)}$ and $\overline{\mathbf{pk}}_{j \rightarrow i}^{(r-1)}$ be the ciphertext and the public key P_i received from P_j .

Party P_i computes $\overline{\mathbf{pk}}_{i \rightarrow k}^{(r)} = \overline{\mathbf{pk}}_{j \rightarrow i}^{(r-1)} \circledast \mathbf{pk}_{i \rightarrow k}^{(r)}$ and $\hat{\mathbf{c}}_{i \rightarrow k}^{(r)} \leftarrow$

$\text{AddLayer}(\mathbf{c}_{j \rightarrow i}^{(r-1)}, \mathbf{pk}_{i \rightarrow k}^{(r)})$.

Party P_i computes $[b_i \vee u_i, \dots, b_i \vee u_i, u_i]_{\overline{\mathbf{pk}}_{i \rightarrow k}^{(r)}}$ and

$\mathbf{c}_{i \rightarrow k}^{(r)} = \text{HomOR}\left([b_i \vee u_i, \dots, b_i \vee u_i, u_i]_{\overline{\mathbf{pk}}_{i \rightarrow k}^{(r)}}, \hat{\mathbf{c}}_{i \rightarrow k}^{(r)}\right)$.

Party P_i sends ciphertext $\mathbf{c}_{i \rightarrow k}^{(r)}$ and public key $\overline{\mathbf{pk}}_{i \rightarrow k}^{(r)}$ to neighbor P_k .

Decrypt Stage: Each party P_i does the following:

// Return ciphertexts For each neighbor P_j of P_i :

if P_i did not receive a message from P_j at round T of the Aggregate Stage **then**
 Party P_i sends ciphertext $\mathbf{e}_{i \rightarrow j}^{(T)} = [1, 1]_{\overline{\text{pk}}_{j \rightarrow i}^{(T)}}$ to P_j .
else
 Party P_i chooses uniformly at random one of the first $\lfloor 1/p \rfloor$ ciphertexts in $\mathbf{c}_{j \rightarrow i}^{(T)}$.
 Let $\bar{\mathbf{c}}_{j \rightarrow i}^{(T)}$ denote the tuple containing the chosen ciphertext and the last element of $\mathbf{c}_{j \rightarrow i}^{(T)}$ (the encryption of the unhappy bit). Party P_i computes and sends $\mathbf{e}_{i \rightarrow j}^{(T)} = \text{HomOR} \left([b_i \vee u_i, u_i]_{\overline{\text{pk}}_{j \rightarrow i}^{(T)}}, \bar{\mathbf{c}}_{j \rightarrow i}^{(T)} \right)$ to neighbor P_j .

// Remove layers
for any round r from T to 2 **do**
 For each neighbor P_k of P_i :
if P_i did not receive a message from P_k **then**
 Party P_i sends $\mathbf{e}_{i \rightarrow j}^{(r-1)} = [1, 1]_{\overline{\text{pk}}_{j \rightarrow i}^{(r-1)}}$ to neighbor P_j , where $k = \pi_i^{(r)}(j)$.
else
 Denote by $\mathbf{e}_{k \rightarrow i}^{(r)}$ the ciphertext P_i received from P_k , where $k = \pi_i^{(r)}(j)$.
 Party P_i sends $\mathbf{e}_{i \rightarrow j}^{(r-1)} = \text{DelLayer} \left(\mathbf{e}_{k \rightarrow i}^{(r)}, \mathbf{sk}_{i \rightarrow k}^{(r)} \right)$ to neighbor P_j .

// Only the recipient has a proper output
if P_i is the recipient P_o and happy **then**
 Party P_i computes $(b, u) = \text{Decrypt}(\mathbf{e}_{1 \rightarrow i}^{(1)}, \mathbf{sk}_{i \rightarrow 1}^{(1)})$.
 Party P_i **outputs** (b, u, u_i) .
else
 Party P_i **outputs** $(1, 0, u_i)$.

3.9.2 Soundness of the Protocol Leaking a Fraction of a Bit

In this Section we show the soundness of BC-FB_p from Section 3.4.2. That is, we present the simulator, the hybrids and the security proof for Theorem 1.

Theorem 1. *Let κ be the security parameter. For $\tau = \log(n) + \kappa$, $T = 8n^3\tau$ and $\rho = \tau/(p' - 2^{-\tau})$, where $p' = 1/\lfloor 1/p \rfloor$, the protocol $\text{BC-FB}_p(T, \rho, (d_i, b_i)_{P_i \in \mathcal{P}})$ topology-hidingly realizes $\mathcal{F}_{\text{INFO}}^{FB_p} || \mathcal{F}_{\text{BC}}$ (with abort) in the \mathcal{F}_{NET} hybrid-world, where the leakage function $\mathcal{L}_{\text{FB}_p}$ is the one defined as above. If no crashes occur, then there is no abort and there is no leakage.*

Proof. (soundness part)

Simulator. The simulator \mathcal{S}_{FB} proceeds almost identically to the simulator \mathcal{S}_{OB} given in the proof of Theorem 5. We only change the algorithm `PhaseSimulation` to `ProbabilisticPhaseSimulation` and execute it ρ times instead of only once.

Algorithm ProbabilisticPhaseSimulation(P_o, P_i)

If $P_o \in \mathcal{Z}^p$, let w denote the random walk generated in the Initialization Stage (at the beginning of the simulation of this phase), which starts at P_o and carries the output bit. Let ℓ denote the number of parties in \mathcal{Z}^p on w before the first correct party. If $P_o \notin \mathcal{Z}^p$, w and ℓ are not defined.

For every $P_j \in \mathcal{Z}^p \cap \mathbf{N}_G(P_i)$, let $\mathbf{pk}_{j \rightarrow i}^{(r)}$ denote the public key generated in the Initialization Stage by P_j for P_i and for round r .

Initialization Stage

For every neighbor $P_j \in \mathcal{Z}^p$ of the correct P_i , \mathcal{S}_{FB} generates \mathbf{T} key pairs $(\mathbf{pk}_{i \rightarrow j}^{(1)}, \mathbf{sk}_{i \rightarrow j}^{(1)}), \dots, (\mathbf{pk}_{i \rightarrow j}^{(\mathbf{T})}, \mathbf{sk}_{i \rightarrow j}^{(\mathbf{T})})$.

Aggregate Stage

In round r , for every neighbor $P_j \in \mathbf{N}_G(P_i) \cap \mathcal{Z}^p$, \mathcal{S}_{FB} sends the tuple $([1, \dots, 1]_{\mathbf{pk}_{i \rightarrow j}^{(r)}}, \mathbf{pk}_{i \rightarrow j}^{(r)})$ (with $\lfloor 1/p \rfloor + 1$ ones) to P_j .

Decrypt Stage

if $P_o \notin \mathcal{Z}^p$ or \mathcal{A} crashed any party in any phase before the current one or in any repetition of the current phase **then**

In every round r and for every neighbor $P_j \in \mathbf{N}_G(P_i) \cap \mathcal{Z}^p$, \mathcal{S}_{FB} sends $[1, 1]_{\mathbf{pk}_{j \rightarrow i}^{(r)}}$ to P_j .

else

In every round r and for every neighbor $P_j \in \mathbf{N}_G(P_i) \cap \mathcal{Z}^p$, \mathcal{S}_{FB} sends $[1, 1]_{\mathbf{pk}_{j \rightarrow i}^{(r)}}$ to P_j unless the following three conditions hold: (a) P_i is the first party not in \mathcal{Z}^p on w , (b) P_j is the last party in \mathcal{Z}^p on w , and (c) $r = 2\mathbf{T} - \ell$.

If the three conditions hold (in particular $r = 2\mathbf{T} - \ell$), \mathcal{S}_{FB} does the following. If \mathcal{A} did not crash any party in a previous round,

\mathcal{S}_{FB} samples a value x from the Bernoulli distribution with parameter p' . If $x = 1$ (with probability p'), \mathcal{S}_{FB} sends to P_j the ciphertext $[b_{out}, 0]_{\mathbf{pk}_{j \rightarrow i}^{(r)}}$ and otherwise it sends $[1, 0]_{\mathbf{pk}_{j \rightarrow i}^{(r)}}$.

Otherwise, let F denote the set of pairs $(P_f, s - \ell + 1)$ such that \mathcal{A} crashed P_f in round s . \mathcal{S}_{FB} queries $\mathcal{F}_{\text{INFO}}^{FB_p}$ for the leakage on input $(F, P_i, \mathbf{T} - \ell)$. If the returned value is 1, it sends $[1, 1]_{\mathbf{pk}_{j \rightarrow i}^{(r)}}$ to P_j . Otherwise it sends $[b^{out}, 0]_{\mathbf{pk}_{j \rightarrow i}^{(r)}}$ to party P_j .

Hybrids and security proof. We consider similar steps as the hybrids from Paragraph 3.4.1.

Hybrid 1. \mathcal{S}_1 simulates the real world exactly. This means, \mathcal{S}_1 has information on the entire topology of the graph, each party's input, and can simulate identically the real world.

Hybrid 2. \mathcal{S}_2 replaces the real keys with the simulated public keys, but still knows everything about the graph as in the first hybrid.

More formally, in each subphase of each random walk phase and for each party $P_i \in \mathcal{P} \setminus \mathcal{Z}^p$ where $\mathbf{N}_G(P_i) \cap \mathcal{Z}^p \neq \emptyset$, \mathcal{S}_2 generates T key pairs $(\mathbf{pk}_{i \rightarrow j}^{(1)}, \mathbf{sk}_{i \rightarrow j}^{(1)}), \dots, (\mathbf{pk}_{i \rightarrow j}^{(T)}, \mathbf{sk}_{i \rightarrow j}^{(T)})$ for every neighbor $P_j \in \mathbf{N}_G(P_i) \cap \mathcal{Z}^p$. Let $\alpha := \lfloor \frac{1}{p} \rfloor$. In each round r of the corresponding Aggregate Stage and for every neighbor $P_j \in \mathbf{N}_G(P_i) \cap \mathcal{Z}^p$, \mathcal{S}_2 does the following: P_i receives ciphertext $[b_1, \dots, b_\alpha, u]_{\mathbf{pk}_{* \rightarrow i}^{(r)}}$ and the public key $\mathbf{pk}_{* \rightarrow i}^{(r)}$ destined for P_j . Instead of adding a layer and homomorphically OR'ing the bit b_i , \mathcal{S}_2 computes $(b'_1, \dots, b'_\alpha, u') = (b_1 \vee b_i \vee u_i, \dots, b_\alpha \vee b_i \vee u_i, u \vee u_i)$, and sends $[b'_{\sigma(1)}, \dots, b'_{\sigma(\alpha)}, u']_{\mathbf{pk}_{i \rightarrow j}^{(r)}}$ to P_j , where σ is a random permutation on α elements. In other words, it sends the same message as \mathcal{S}_1 but encrypted with a fresh public key. In the corresponding Decrypt Stage, P_i will get back a ciphertext from P_j encrypted under this exact fresh public key.

Hybrid 3. \mathcal{S}_3 now simulates the ideal functionality during the Aggregate Stage. It does so by sending encryptions of $(1, \dots, 1)$ instead of the actual messages and unhappy bits. More formally, let $\alpha := \lfloor \frac{1}{p} \rfloor$. In each round r of a subphase of a random walk phase and for all parties $P_i \in \mathcal{P} \setminus \mathcal{Z}^p$ and $P_j \in \mathbf{N}_G(P_i) \cap \mathcal{Z}^p$, \mathcal{S}_3 sends $[1, 1, \dots, 1]_{\mathbf{pk}_{i \rightarrow j}^{(r)}}$ instead of the ciphertext $[b_1, \dots, b_\alpha, u]_{\mathbf{pk}_{i \rightarrow j}^{(r)}}$ sent by \mathcal{S}_2 .

Hybrid 4. \mathcal{S}_4 does the same as \mathcal{S}_{FB} during the Decrypt Stage for all phases and subphases except for the first subphase of a random walk phase in which the adversary crashes a party.

Hybrid 5. \mathcal{S}_5 is the actual simulator \mathcal{S}_{FB} .

The proofs that no efficient distinguisher D can distinguish between Hybrid 1, Hybrid 2 and Hybrid 3 are similar to the Claim 1 and Claim 2. Hence, we prove indistinguishability between Hybrid 3, Hybrid 4 and Hybrid 5.

Claim 5. *No efficient distinguisher D can distinguish between Hybrid 3 and Hybrid 4.*

Proof. The only difference between the two hybrids is in the Decrypt Stage. We differentiate three cases:

- A subphase l of a phase k where the adversary did not crash any party in this phase, any previous subphase, or any previous phase. In this case, \mathcal{S}_3 sends with probability p an encryption of (b_W, u_W) , where $b_W = \bigvee_{u \in W} b_u$ is the OR of all input bits in the walk and $u_W = 0$ (since no crash occurs), and with probability $1 - p$ an encryption of $(1, 0)$. On the other hand, \mathcal{S}_4 samples r from a Bernoulli distribution with parameter p , and if $r = 1$, it sends an encryption of $(b_{out}, 0)$, where $b_{out} = \bigvee_{i \in [n]} b_i$, and if $r = 0$ it sends an encryption of $(1, 0)$. Since the graph is connected, $b_{out} = b_W$ with overwhelming probability, as proven in Lemma 1. Also, the encryption in Hybrid 4 is done with a fresh public key which is indistinguishable with the encryption done in Hybrid 3 by OR'ing many times in the graph, as shown in Claim 2.1. in [1].

- A subphase l of a phase k where the adversary crashed a party in a previous subphase or a previous phase.

In Hybrid 3 the parties send encryptions of $(1, 1)$. This is also the case in Hybrid 4, because even if a crashed party disconnected the graph, each connected component contains a neighbor of a crashed party. Moreover, in Hybrid 4, the messages are encrypted with a fresh public key, and in Hybrid 3, the encryptions are obtained by the homomorphic OR operation. Both encryptions are indistinguishable, as shown in Claim 2.1. in [1].

■

Claim 6. *No efficient distinguisher D can distinguish between Hybrid 4 and Hybrid 5.*

Proof. The only difference between the two hybrids is in the Decrypt Stage of the first subphase of a phase where the adversary crashes.

Let F be the set of pairs (P_f, r) such that \mathcal{A} crashed P_f at round r of the phase. In Hybrid 4, a walk W of length T is generated from party P_o . Let W_1 be the region of W from P_o to the first not passively corrupted party and let W_2 be the rest of the walk. Then, the adversary's view at this step is the encryption of $(1, 1)$ if one of the crashed parties breaks W_2 or if the walk became dummy (which happens with probability $1 - p$, since the ciphertexts are permuted randomly and only one ciphertext out of $\frac{1}{p}$ contains b_W). Otherwise, the adversary's view is an encryption of $(b_W, 0)$. In both cases, the message is encrypted under a public key for which the adversary knows the secret key.

In Hybrid 5, a walk W'_1 is generated from P_o of length $\ell \leq T$ ending at the first not passively corrupted party P_i . Then, the simulator queries the leakage function on input $(F, P_i, T - \ell)$. Then, with probability p it generates a walk W'_2 of length $T - \ell$ from P_i , and checks whether W'_2 is broken by any party in F . If W'_2 is broken, P_i sends an encryption of $(1, 1)$, and otherwise an encryption of $(b_W, 0)$. Since the walk W' defined as W'_1 followed by W'_2 follows the same distribution as W , $b_W = b_W^{out}$ with overwhelming probability, and the encryption with a fresh public key which is indistinguishable with the encryption done by OR'ing many times in the graph, then it is impossible to distinguish between Hybrid 4 and Hybrid 5.

■

This concludes the proof of soundness. □

□

□

3.10 Getting Topology-Hiding Computation

This section contains supplementary material for Section 3.5.

3.10.1 All-to-all Multibit Broadcast

In this section we prove Lemma 2 from Section 3.5.1 which shows how to modify protocol BC-FB _{p} to achieve an all-to-all multibit broadcast.

Lemma 2. *Protocol $BC\text{-}FB_p$ can be edited to an all-to-all multi-bit broadcast $\text{Multibit}BC_p$, which is secure against an adversary, who statically passively corrupts and adaptively crashes any number of parties and leaks at most a fraction p of a bit. The round complexity of $\text{Multibit}BC_p$ is the same as for $BC\text{-}FB_p$.*

Proof. This involves the following simple transformation of protocol $BC\text{-}FB_p$. Note that $BC\text{-}FB_p$ is already multibit; during the random-walk protocol, parties send around vectors of ciphertexts: $[\mathbf{b}, u] := [b_1, \dots, b_\ell, u]$. In the transformed protocol we will substitute each ciphertext encrypting a bit b_i with a vector of ciphertexts of length m , containing encryptions of a vector of bits \mathbf{b}_i . That is, we now think of parties sending a vector of vectors $[\mathbf{b}_1, \dots, \mathbf{b}_\ell, u]$. Technically, we “flatten” these vectors, that is, the parties will send vectors of length $m\ell + 1$ of ciphertexts.

Let us now explain the transformation. For an all-to-all broadcast, each party, P_i , wants to broadcast some k -bit message, (b_1, \dots, b_k) . Consider a vector of ciphertexts of length nk , where each of the n parties is assigned to k slots for k bits of its message. Each of the vectors \mathbf{b}_i in the vector $[\mathbf{b}_1, \dots, \mathbf{b}_\ell, u]$ described above will be of this form. P_i will use the slots from $n(i - 1)$ to ni to communicate its message.

We now have a look at the Aggregate Stage in the transformed protocol $\text{Multibit}BC_p$.

- Every party P_i that wants to send the k bit message (b_1, \dots, b_k) prepares its input vector

$$\mathbf{b}_i = (0, \dots, 0, b_1, \dots, b_k, 0, \dots, 0)$$

by placing the bits b_1, \dots, b_k in positions from $n(i - 1)$ to ni .

- At the beginning of the Aggregate Stage, the recipient P_o with the input vector \mathbf{b}_o sends the ciphertext $[\mathbf{b}_o \vee u_o, \mathbf{1}, \dots, \mathbf{1}, u_o]_{\text{pk}_{i \rightarrow 1}^{(1)}}$ to its first neighbor. All other ciphertexts to all other neighbors j are just $[\mathbf{1}, \dots, \mathbf{1}, 1]_{\text{pk}_{i \rightarrow j}^{(1)}}$ ³.

Every other party P_i starts the protocol with sending the ciphertext tuple $[\mathbf{1}, \dots, \mathbf{1}, 1]_{\text{pk}_{i \rightarrow j}^{(1)}}$ to every neighbor j .

- Upon receiving a ciphertext at round r from a neighbor j , $[\mathbf{b}_1, \dots, \mathbf{b}_\ell, u]_{\text{pk}_{j \rightarrow i}^{(t)}}$, party P_i takes its input vector \mathbf{b}_i and homomorphically OR's the vector $(\mathbf{b}_i \vee u_i, \dots, \mathbf{b}_i \vee u_i, u_i)$ containing ℓ copies of the vector $\mathbf{b}_i \vee u_i$ to the ciphertext. The result is sent along the walk.

The rest of the protocol $\text{Multibit}BC_p$ proceeds analogously to $BC\text{-}FB_p$.

A quick check of correctness tells us that when a message is not made unhappy, and starts with 0's in the appropriate places, every party's broadcast message eventually gets OR'd in a different spot in the message vector, and so every party will get that broadcast.

A quick check of soundness tells us that the simulator works just as before: it simulates with the encrypted output (all nk bits) when there was no abort, and with a query to the leakage function if there was one. \square \square

³We are abusing notation: $\mathbf{b}_o \vee u_o$ means that we OR u_i with every coordinate in \mathbf{b} .

3.10.2 Sequential Execution Without Aggregated Leakage

In this section we prove Lemma 3 from Section 3.5.2 which states that there is a protocol that executes MultibitBC_p sequentially any number of times leaking at most a fraction p of a bit.

Lemma 3. *There exists a protocol, which implements any number k of executions of MultibitBC_p , is secure against an adversary, who statically passively corrupts and adaptively crashes any number of parties and leaks at most a fraction p of a bit in total. The complexity of the constructed protocol is k times the complexity of MultibitBC_p .*

Proof. The construction makes non-black-box use of the unhappy bits used in MultibitBC_p . The idea is simply to preserve the state of the unhappy bits between sequential executions. That is, once some party sees a crash, it will cause all subsequent executions to abort.

Correctness and complexity of the above construction are trivial, since it simply executes the protocol MultibitBC_p k times.

We now claim that any leakage happens only in the one execution of protocol MultibitBC_p , in which the first crash occurs. Once we show this, it is easy to see that the constructed protocol executing MultibitBC_p k times leaks at most a fraction p of a bit.

By Theorem 1, any execution without crashes causes no leakage (it can be easily simulated as in the setting with only passive corruptions and no fail-stop adversary). Further, assume that any party P_c crashes before BC-FB_p starts. Let $\mathbf{N}_G(a)$ be all of P_a 's neighbors; all of them will have their unhappy bit set to 1. Because of the correctness of the random-walk protocol embedded within BC-FB_p , the random walk will hit every node in the connected component, and so is guaranteed to visit a node in $\mathbf{N}_G(a)$. Therefore, every walk will become a dummy walk, which is easily simulated. \square \square

3.10.3 Topology-Hiding Computation

In this section we prove Theorem 6 from Section 3.5.3 which compiles any MPC protocol secure against an adversary who can statically passively corrupt and adaptively crash any number of parties into a protocol that is in addition topology-hiding and leaks at most a fraction p of a bit of information about the topology.

Theorem 6. *Assume PKCR exists. Then, we can compile any MPC protocol Π_{MPC} that runs in M rounds into a topology-hiding protocol with leakage function $\mathcal{L}_{\text{FB}_p}$, that runs in $MR + 1$ rounds, where R is the round complexity of BC-FB_p .⁴*

Proof. Recall the generic transformation for taking UC-secure topology-hiding broadcast and compiling it into UC-secure topology-hiding MPC using a public key infrastructure. Every MPC protocol with M rounds, Π_{MPC} , has at each round each party

⁴In particular, the complexity of BC-FB_p is $n \cdot \rho \cdot 2T$, where κ is the security parameter, $\tau = \log(n) + \kappa$, $T = 8n^3\tau$ is the length of a walk and $\rho = \tau/(p' - 2^{-\tau})$ is the number of repetitions of a phase (with $p' = 1/\lceil 1/p \rceil$).

sending possibly different messages to every other party. This is a total of $O(n^2)$ messages sent at each round, but we can simulate this with n separate multi-bit broadcasts.

To transform Π_{MPC} into a topology-hiding protocol in the fail-stop model, given a multi-bit topology-hiding broadcast, we do the following:

- Setup phase. The parties use one multi-bit topology-hiding broadcast to give their public key to every other party.
- Each round of Π_{MPC} . For each party P_i that needs to send a message of k bits to party P_j , P_i encrypts that message under P_j 's public key. Then, each party P_i broadcasts the $n - 1$ messages it would send in that round of Π_{MPC} , one for each $j \neq i$, encrypted under the appropriate public keys. That is, P_i is the source for one multi-bit broadcast. All these multi-bit broadcasts are simultaneously executed via an all-to-all multi-bit broadcast, where each party broadcast a message of size $(n - 1)k$ times.
After the broadcasts, each node can use their secret key to decrypt the messages that were for them and continue with the protocol.
- At the end of the protocol, each party now has the output it would have received from running Π_{MPC} , and can compute its respective output.

First, this is a correct construction. We will prove this by inducting on the rounds of Π_{MPC} . To start, all nodes have all information they would have had at the beginning of Π_{MPC} as well as public keys for all other parties and their own secret key. Assume that the graph has just simulated round $r - 1$ of Π_{MPC} and each party has the information it would have had at the end of round $r - 1$ of Π_{MPC} (as well as the public keys etc). At the end of the r 'th simulated round, each party P_i gets encryptions of messages sent from every other party P_j encrypted under P_i 's public key. These messages were all computed correctly according to Π_{MPC} because all other parties had the required information by the inductive hypothesis. P_i can then decrypt those messages to get the information it needs to run the next round. So, by the end of simulating all rounds of Π_{MPC} , each party has the information it needs to complete the protocol and get its respective output.

Security of this construction (and, in particular, the fact that it only leaks a fraction p of a bit) follows directly from Lemma 2 and Lemma 3. □

□

3.11 Deeply Fully-Homomorphic Public-Key Encryption

In this section we present the formal definition of deeply fully-homomorphic public-key encryption from Section 3.6.1. For completeness, we first give the definition of a public-key encryption scheme.

Definition 2. A public-key encryption (PKE) scheme with public-key space \mathcal{PK} , secret-key space \mathcal{SK} , plaintext message space \mathcal{M} , and ciphertext space \mathcal{C} consists of three algorithms (*KeyGen*, *Encrypt*, *Decrypt*) where:

1. The (probabilistic) key-generation algorithm $\text{KeyGen} : \{0, 1\}^* \rightarrow \mathcal{PK} \times \mathcal{SK}$ takes a security parameter and outputs a public key $\mathbf{pk} \in \mathcal{PK}$ and a secret key $\mathbf{sk} \in \mathcal{SK}$.
2. The (probabilistic) encryption algorithm $\text{Encrypt} : \mathcal{PK} \times \mathcal{M} \rightarrow \mathcal{C}$ takes a public key $\mathbf{pk} \in \mathcal{PK}$ and a message $m \in \mathcal{M}$ and outputs a ciphertext $c \in \mathcal{C}$.
3. The decryption algorithm $\text{Decrypt} : \mathcal{SK} \times \mathcal{C} \rightarrow \mathcal{M}$ takes a secret key $\mathbf{sk} \in \mathcal{SK}$ and a ciphertext $c \in \mathcal{C}$ and outputs a message $m \in \mathcal{M}$.

A PKE scheme is correct if for any key pair $(\mathbf{pk}, \mathbf{sk}) \leftarrow \text{KeyGen}$ and any message $m \in \mathcal{M}$ it holds (with probability 1 over the randomness of *Encrypt*) that $m = \text{Decrypt}(\mathbf{sk}, \text{Encrypt}(\mathbf{pk}, m))$.

Our protocol requires a PKE scheme \mathcal{E} where (a) one can add and remove layers of encryption, while (b) one can homomorphically compute any function on encrypted bits (independent of the number of layers). This will be captured by three additional algorithms: AddLayer_r , DelLayer_r , and HomOp_r , operating on ciphertexts with r layers of encryption (we will call such ciphertexts level- r ciphertexts). A level- r ciphertext is encrypted under a level- r public key (we assume that each level can have different key space).

Definition 3. A deeply fully-homomorphic public-key encryption (DFH-PKE) scheme is a PKE scheme with additional algorithms AddLayer_r , DelLayer_r , and HomOp_r . We define additional public-key spaces \mathcal{PK}_r and ciphertext spaces \mathcal{C}_r , for public keys and ciphertexts on level r . We require that $\mathcal{PK}_1 = \mathcal{PK}$ and $\mathcal{C}_1 = \mathcal{C}$. Let \mathcal{F} be the family of efficiently computable functions.

- The algorithm $\text{AddLayer}_r : \mathcal{C}_r^* \times \mathcal{PK}_r \times \mathcal{SK} \rightarrow \mathcal{C}_{r+1}^* \times \mathcal{PK}_{r+1}$ takes as input a level- r ciphertext $\llbracket m \rrbracket_{\mathbf{pk}}$, the corresponding level- r public key \mathbf{pk} , and a new secret key \mathbf{sk} . It outputs a level- $(r+1)$ ciphertext and the level- $(r+1)$ public key, under which it is encrypted.
- The algorithm $\text{DelLayer}_r : \mathcal{C}_{r+1}^* \times \mathcal{PK}_{r+1} \times \mathcal{SK} \rightarrow \mathcal{C}_r^* \times \mathcal{PK}_r$ deletes a layer from a level- $(r+1)$ ciphertext.
- The algorithm $\text{HomOp}_r : \mathcal{C}_r^* \times \mathcal{PK}_r \times \mathcal{F} \rightarrow \mathcal{C}_r$ takes as input some k level- r ciphertexts encrypted under the same level- r public key, the corresponding public key, and a k -ary function f . It outputs a level- r ciphertext that contains f of the encrypted messages.

For convenience, it will be easy to describe the security of our enhanced encryption scheme with the help of an algorithm Leveled-Encrypt_r , which takes as input a vector of plain messages and a level- r public key, and outputs a vector of level- r ciphertexts⁵.

⁵This algorithm can be obtained by keeping an encryption of 0 and 1 as part of the leveled public key and rerandomizing the ciphertext using HomOp_r .

Definition 4. For a DFH-PKE scheme, we additionally define the algorithm *Leveled-Encrypt_r* : $\mathcal{M}^* \times \mathcal{PK}_r \rightarrow \mathcal{C}_r^* \times \mathcal{PK}_r$ that outputs the level- r encryptions of the messages \mathbf{m} and the corresponding level- r public key.

Intuitively, we will require that from the output of *AddLayer_r* (*DelLayer_r*) one cannot obtain any information on the underlying layers of encryption. That is, that the output of *AddLayer_r* (*DelLayer_r*) is indistinguishable from a level- $(r + 1)$ (level- r) encryption of the message. We will also require that the output of *HomOp_r* is indistinguishable from a level- r encryption of the output of the functions applied to the messages.

Definition 5. We require that a DFH-PKE scheme satisfies the following properties:

Aggregate Soundness. For every r , every vector of messages \mathbf{m} and every efficiently computable pair of level- r public keys \mathbf{pk}_1 and \mathbf{pk}_2 ,

$$\begin{aligned} & \{ \text{AddLayer}_r(\llbracket \mathbf{m} \rrbracket_{\mathbf{pk}_1}, \mathbf{pk}_1, \mathbf{sk}; U^*) : (\mathbf{pk}, \mathbf{sk}) \leftarrow \text{KeyGen}(1^\kappa; U^*) \} \\ & \approx_c \\ & \left\{ (\text{Leveled-Encrypt}_{r+1}(\mathbf{m}, \mathbf{pk}'_2; U^*), \mathbf{pk}'_2) : \begin{array}{l} (\mathbf{pk}, \mathbf{sk}) \leftarrow \text{KeyGen}(1^\kappa; U^*), \\ (\llbracket 0 \rrbracket_{\mathbf{pk}'_2}, \mathbf{pk}'_2) \leftarrow \text{AddLayer}_r(\llbracket 0 \rrbracket_{\mathbf{pk}_2}, \mathbf{pk}_2, \mathbf{sk}; U^*) \end{array} \right\} \end{aligned}$$

Decrypt Soundness. For every r , every vector \mathbf{m} and every efficiently computable level- r public key \mathbf{pk}_1 ,

$$\begin{aligned} & \left\{ \text{DelLayer}_r(\llbracket \mathbf{m} \rrbracket_{\mathbf{pk}}, \mathbf{pk}, \mathbf{sk}; U^*) : \begin{array}{l} (\mathbf{pk}, \mathbf{sk}) \leftarrow \text{KeyGen}(1^\kappa; U^*), \\ (\llbracket 0 \rrbracket_{\mathbf{pk}}, \mathbf{pk}) \leftarrow \text{AddLayer}_r(\llbracket 0 \rrbracket_{\mathbf{pk}_1}, \mathbf{pk}_1, \mathbf{sk}; U^*) \end{array} \right\} \\ & \approx_c \\ & \{ (\text{Leveled-Encrypt}_r(\mathbf{m}, \mathbf{pk}_1; U^*), \mathbf{pk}_1) \} \end{aligned}$$

Full-Homomorphism. For every vector of messages $\mathbf{m} \in \mathcal{M}^*$, every level- r public key \mathbf{pk} , every vector of ciphertexts $\mathbf{c} \in \mathcal{C}^*$ and every function $f \in \mathcal{F}$ such that *Leveled-Encrypt_r*(\mathbf{m}, \mathbf{pk}) = \mathbf{c} ,

$$\begin{aligned} & \{ (\mathbf{m}, \mathbf{c}, \mathbf{pk}, f, \text{Leveled-Encrypt}_r(f(\mathbf{m}), \mathbf{pk}; U^*)) \} \\ & \approx_c \\ & \{ (\mathbf{m}, \mathbf{c}, \mathbf{pk}, f, \text{HomOp}_r(\mathbf{c}, \mathbf{pk}, f; U^*)) \} \end{aligned}$$

Note that *AddLayer_r* and *DelLayer_r* produce both the level- r encrypted messages and the level- r public key. In the case where we only need the public key, we will just call *AddLayer_r*($\llbracket 0 \rrbracket_{\mathbf{pk}}, \mathbf{pk}, \mathbf{sk}$), since the encrypted message does not matter for producing a new public key — the same applies for *DelLayer_r*.

Also note that one can create a level- r public key generating r level-1 key pairs $(\mathbf{pk}_i, \mathbf{sk}_i) \leftarrow \text{KeyGen}(1^\kappa)$ and using *AddLayer* to add the public keys one by one. Furthermore, with all secret keys $(\mathbf{sk}_1, \dots, \mathbf{sk}_r)$ used in the creation of some level- r public key \mathbf{pk} , we can define a combined level- r secret key $\mathbf{sk} = (\mathbf{sk}_1, \dots, \mathbf{sk}_r)$, which we can use to decrypt a level- r ciphertext by calling *DelLayer* r times.

3.11.1 Instantiation of DFH-PKE from FHE

We show how to instantiate DFH-PKE from FHE. As required from the DFH-PKE scheme, the level-1 public key space and ciphertext space are the FHE public key space and FHE ciphertext space respectively, i.e., $\mathcal{PK}_1 = \mathcal{PK}$ and $\mathcal{C}_1 = \mathcal{C}$. For $r > 1$, a level- r public key and ciphertext spaces are $\mathcal{PK}_r = \mathcal{PK} \times \mathcal{C}$ and $\mathcal{C}_r = \mathcal{C}$, respectively.

Notation. We denote by $\text{FHE.Encrypt}(m, \mathbf{pk})$ the FHE encryption algorithm that takes message m and encrypts under public key \mathbf{pk} . In the same way, the FHE decryption algorithm is denoted by FHE.Decrypt . The FHE evaluation algorithm is defined as $\text{FHE.HomOp}([m_1, \dots, m_n]_{\mathbf{pk}}, \mathbf{pk}, f) := [f(m_1, \dots, m_n)]_{\mathbf{pk}}$. It gets as input a vector of encrypted messages under \mathbf{pk} , the public key \mathbf{pk} and the function to evaluate, and it returns the output of f applied to the messages.

In the following we define the algorithms to add and remove layers:

Algorithm $\text{AddLayer}_r((c_1, \dots, c_n), \mathbf{pk}, \mathbf{sk})$

Let \mathbf{pk} be the corresponding public key of \mathbf{sk} .
 $c'_i \leftarrow \text{FHE.Encrypt}(c_i, \mathbf{pk})$.
 $\mathbf{pk}' \leftarrow (\mathbf{pk}, \text{FHE.Encrypt}(\mathbf{pk}, \mathbf{pk}))$.
return $((c'_1, \dots, c'_n), \mathbf{pk}')$.

Algorithm $\text{DelLayer}_r((c'_1, \dots, c'_n), \mathbf{pk}', \mathbf{sk})$

Parse $\mathbf{pk}' = (\mathbf{pk}, [\mathbf{pk}]_{\mathbf{pk}})$.
 $\mathbf{pk} \leftarrow \text{FHE.Decrypt}([\mathbf{pk}]_{\mathbf{pk}}, \mathbf{sk})$.
 $c_i \leftarrow \text{FHE.Decrypt}(c'_i, \mathbf{sk})$.
return $((c_1, \dots, c_n), \mathbf{pk})$.

Notice the recursive nature of leveling; let us write $\mathbf{pk}_r = (\mathbf{pk}_r, [\mathbf{pk}_{r-1}, [\dots [\mathbf{pk}_1]_{\mathbf{pk}_2} \dots]_{\mathbf{pk}_{r-1}}]_{\mathbf{pk}_r})$, and $\llbracket m \rrbracket_{\mathbf{pk}_r}$ denotes the leveled ciphertext $[[\dots [m]_{\mathbf{pk}_1} \dots]_{\mathbf{pk}_{r-1}}]_{\mathbf{pk}_r}$. Hence, it is easy to see that the two algorithms above accomplish the following:

$$\text{AddLayer}_r(\llbracket \mathbf{m} \rrbracket_{\mathbf{pk}_r}, \mathbf{pk}_r, \mathbf{sk}_{r+1}) = (\llbracket \mathbf{m} \rrbracket_{\mathbf{pk}_{r+1}}, \mathbf{pk}_{r+1})$$

and

$$\text{DelLayer}_r(\llbracket \mathbf{m} \rrbracket_{\mathbf{pk}_{r+1}}, \mathbf{pk}_{r+1}, \mathbf{sk}_r) = (\llbracket \mathbf{m} \rrbracket_{\mathbf{pk}_r}, \mathbf{pk}_r)$$

In the following, we show how to apply any function f on any vector of level- r ciphertexts. It is clear that if the ciphertexts are level-1 ciphertexts, we can apply f using FHE directly. If the ciphertexts are level- r ciphertexts for $r > 1$, we FHE evaluate the ciphertexts and public key with a recursive function call on the previous level. More concretely, we use the following recursive algorithm to apply f to any vector of level- r ciphertexts:

Algorithm $\text{HomOp}_r((c_1, \dots, c_n), \mathbf{pk}, f)$

```

if  $r = 1$  then
  Parse  $\mathbf{pk} = \mathbf{pk}$ .
  return  $\text{FHE.HomOp}((c_1, \dots, c_n), \mathbf{pk}, f)$ .
Parse  $\mathbf{pk} = (\mathbf{pk}, [\mathbf{pk}_{r-1}]_{\mathbf{pk}})$ ,  $c_i = [c'_i]_{\mathbf{pk}}$ .
Let  $f'(\cdot, \cdot) := \text{HomOp}_{r-1}(\cdot, \cdot, f)$ .
return  $\text{FHE.HomOp}([(c'_1, \dots, c'_n)]_{\mathbf{pk}}, [\mathbf{pk}_{r-1}]_{\mathbf{pk}}, \mathbf{pk}, f')$ .

```

Lemma 5. *For any r , algorithm HomOp_r is correct on leveled ciphertexts.*

Proof. We want to show that for a vector of level- r ciphertexts $\mathbf{c} = \llbracket \mathbf{m} \rrbracket_{\mathbf{pk}}$, $\text{HomOp}_r(c, \mathbf{pk}, f) = \llbracket f(\mathbf{m}) \rrbracket_{\mathbf{pk}}$. We will prove this via induction on r .

For the base case, consider $r = 1$. Here we go into the if statement, and the algorithm returns

$$\text{FHE.HomOp}([\mathbf{m}]_{\mathbf{pk}}, \mathbf{pk}, f) = [f(\mathbf{m})]_{\mathbf{pk}}$$

by the correctness of the FHE scheme.

Now, assume that $\text{HomOp}_{r-1}(\llbracket \mathbf{m} \rrbracket_{\mathbf{pk}_{r-1}}, \mathbf{pk}_{r-1}, f) = \llbracket f(\mathbf{m}) \rrbracket_{\mathbf{pk}_{r-1}}$ for all messages \mathbf{m} encrypted under $r-1$ levels of keys. Calling HomOp_r on $\llbracket \mathbf{m} \rrbracket_{\mathbf{pk}_r}$ results in returning

$$\begin{aligned}
& \text{FHE.HomOp}((\llbracket \mathbf{m} \rrbracket_{\mathbf{pk}_r}, [\mathbf{pk}_{r-1}]_{\mathbf{pk}_r}), \mathbf{pk}_r, \text{HomOp}_{r-1}(\cdot, \cdot, f)) \\
&= [\text{HomOp}_{r-1}(\llbracket \mathbf{m} \rrbracket_{\mathbf{pk}_{r-1}}, \mathbf{pk}_{r-1}, f)]_{\mathbf{pk}_r} \\
&= \llbracket f(\mathbf{m}) \rrbracket_{\mathbf{pk}_r}
\end{aligned}$$

by correctness of the FHE homomorphic evaluation. \square

We are also able to encrypt in a leveled way by exploiting the fully-homomorphic properties of the scheme, using the FHE.HomOp algorithm to apply encryption.

Algorithm $\text{Leveled-Encrypt}_r(\mathbf{m}, \mathbf{pk})$

```

if  $r = 1$  then
  Parse  $\mathbf{pk} = \mathbf{pk}$ 
  return  $(\text{FHE.Encrypt}(m_i, \mathbf{pk}))_i$ 
Parse  $\mathbf{pk} = (\mathbf{pk}, [\mathbf{pk}_{r-1}]_{\mathbf{pk}})$ .
Let  $[\mathbf{m}]_{\mathbf{pk}} = (\text{FHE.Encrypt}(m_i, \mathbf{pk}))_i$ .
return  $\text{FHE.HomOp}([\mathbf{m}]_{\mathbf{pk}}, [\mathbf{pk}_{r-1}]_{\mathbf{pk}}, \mathbf{pk}, \text{Leveled-Encrypt}_{r-1})$ 

```

Finally, we need to prove that adding a fresh layer is equivalent to looking like a fresh random encryption.

Lemma 6. $\text{Leveled-Encrypt}_r(\mathbf{m}, \mathbf{pk}_r) = \llbracket \mathbf{m} \rrbracket_{\mathbf{pk}_r}$.

Proof. We will prove this by induction on r . First, when $r = 1$, we have that $\text{Leveled-Encrypt}_1(\mathbf{m}, \mathbf{pk}_1) = \text{FHE.Encrypt}(\mathbf{m}, \mathbf{pk}_1) = \llbracket \mathbf{m} \rrbracket_{\mathbf{pk}_1}$ from the base case.

Now, assume that for $r - 1$, $\text{Leveled-Encrypt}_{r-1}(\mathbf{m}, \mathbf{pk}_{r-1}) = \llbracket \mathbf{m} \rrbracket_{\mathbf{pk}_{r-1}}$. Hence, when we invoke $\text{Leveled-Encrypt}_r(\mathbf{m}, \mathbf{pk}_r)$, we have that $f(\mathbf{m}, \mathbf{pk}_{r-1}) = \llbracket \mathbf{m} \rrbracket_{\mathbf{pk}_{r-1}}$ from the inductive hypothesis. Therefore, we return

$$\text{FHE.HomOp}(\llbracket \mathbf{m} \rrbracket_{\mathbf{pk}_r}, \llbracket \mathbf{pk}_{r-1} \rrbracket_{\mathbf{pk}_r}, \mathbf{pk}_r, f) = \llbracket \llbracket \mathbf{m} \rrbracket_{\mathbf{pk}_{r-1}} \rrbracket_{\mathbf{pk}_r} = \llbracket \mathbf{m} \rrbracket_{\mathbf{pk}_r}$$

as desired. \square

Lemma 7. *The instantiation of DFH-PKE from FHE presented above satisfies the properties Aggregate Soundness, Decrypt Soundness and Full-Homomorphism, presented in Definition 5.*

Proof. Aggregate Soundness. The algorithm **AddLayer** returns a tuple that contains $(\llbracket \mathbf{m} \rrbracket_{\mathbf{pk}}, \mathbf{pk})$, where \mathbf{m} is a vector of messages, and \mathbf{pk} is a pair containing a fresh key \mathbf{pk} and an encryption of a level- r key under \mathbf{pk} . The tuple that consists of $(\text{Leveled-Encrypt}_r(\mathbf{m}, \mathbf{pk}_1; U^*), \mathbf{pk}_1)$, where \mathbf{pk}_1 is a level- $(r + 1)$ public key obtained from adding a fresh layer to a level- r public key, has the same distribution: the first part of both tuples contain fresh FHE encryptions of level- r ciphertexts.

Decrypt Soundness. This property is trivially achieved given the correctness of the FHE decryption algorithm and Leveled-Encrypt_r .

Full-Homomorphism. The Leveled-Encrypt_r algorithm returns a level- r encryption of $f(\mathbf{m})$ which is the result of applying FHE homomorphic operations on a level- r ciphertext. The algorithm HomOp_r also returns a level- r ciphertext output by the FHE homomorphic operation. \square

3.12 Topology Hiding Computation from DFH-PKE

In this section, we present a detailed description of protocol DFH-THC from Section 3.6.2.

We will use DFH-PKE to alter the **RandomWalkPhase** protocol (and by extension we can also alter the protocol **ProbabilisticRandomWalkPhase_p**). Then, executing protocols **BC-OB** and **BC-FB_p** that leak one bit and a fraction of a bit respectively will be able to evaluate any poly-time function instead, while still leaking the same amount of information as a broadcast using these random walk protocols. The concept is simple. During the Aggregate Stage, parties will add a leveled encryption of their input and identifying information to a vector of ciphertexts, while adding a layer — we will not need sequential id's if each party knows where their input should go in the function. Then, at the end of the Aggregate Stage, nodes homomorphically evaluate f' , which is the composition of a parsing function, to get one of each input in the right place, and f , to evaluate the function on the parsed inputs. The result is a leveled ciphertext of the output of f . This ciphertext is un-layered in the Decrypt Stage so that by the end, the relevant parties get the output.

For completeness, here is a more detailed description of the modified protocol **RandomWalkPhase** leaking one bit, which we call **DFH-RandomWalkPhase**:

Initialization Stage. Each party P_i has its own input bit b_i and unhappiness bit u_i . Each party P_i knows the function f on n variables that the graph wants to compute, and generates $T \cdot d_i$ keypairs and $T - 1$ permutations on d_i elements (d_i is the number of neighbors for party i). P_i also generates a unique ID (or uses a given sequential or other ID) p_i . If party P_i witnessed an abort from the last phase, it becomes unhappy, setting its unhappy bit $u_i = 1$.

Aggregate Stage. Round 1. Each party P_i sends to each neighbor P_j a vector of level-1 ciphertexts under $\mathbf{pk}_{i \rightarrow j}^{(1)}$ containing the input bit b_i , id p_i , unhappy bit u_i and a bit v_i indicating whether the walk is dummy or not.

If P_i is the party that gets the output in that phase, i.e., $P_i = P_o$, then it sends to the first neighbor an encryption of b_i , p_i , u_i and a bit $v_i = 0$ indicating that the walk should not be dummy. To all other neighbors, $v_i = 1$. In the case where $P_i \neq P_o$, $v_i = 1$ as well.

Round $r \in [2, T]$. Let $k = \pi_i^{(r)}(j)$. Upon receiving a vector of level- $(r - 1)$ ciphertexts from P_j . Party P_i uses $\mathbf{sk}_{i \rightarrow k}^{(r)}$ to add a fresh layer with **AddLayer** to the vector of ciphertexts. The function **AddLayer** will return the vector \mathbf{c} of level- r ciphertexts with the corresponding level- r public key \mathbf{pk} . Then, P_i will encrypt its own input, id and unhappybit via **Leveled-Encrypt** under \mathbf{pk} and appends these ciphertexts to \mathbf{c} . It then sends to P_k the level- r public key and all the level- r ciphertexts.

If no vector of ciphertexts was received from P_j (i.e. P_j aborted), P_i generates a fresh level- r public key \mathbf{pk} and secret key \mathbf{sk} . It then generates a vector of level- r ciphertexts containing the bit 1 using **Leveled-Encrypt** under \mathbf{pk} . The size of this vector corresponds to the size of the vector containing the dummy bit, r input bits, r ids, and r unhappy bits.

Evaluation. We are now at the last step in the walk. If P_i received an encrypted vector of level- T ciphertexts from P_j , it evaluates the vector using **HomOp_T** on the function f' which does the following: if the dummy bit is 1 or any unhappy bit set to 1, the function evaluates to \perp . Otherwise, it arranges the inputs by ids and evaluates f on the arranged inputs. That is, it evaluates $f \circ \mathbf{parse}$, where $\mathbf{parse}((m_{i_1}, p_{i_1}), \dots, (m_{i_T}, p_{i_T})) = (m_1, \dots, m_n)$. More concretely, for the vector of ciphertexts \mathbf{c} and level- T public key \mathbf{pk} received from P_j , P_i evaluates $\hat{c} \leftarrow \mathbf{HomOp}(\mathbf{c}, \mathbf{pk}, f')$, and sends \hat{c} to P_j .

If P_i did not receive a message from P_j , or u_i has been set to 1, P_i sends a ciphertext containing \perp : it generates a fresh level- T public key \mathbf{pk} and secret key \mathbf{sk} , and uses **Leveled-Encrypt** under \mathbf{pk} to send to P_j a level- T ciphertext containing \perp .

Decrypt Stage. Round $r \in [T, 2]$ If P_i receives a level- r ciphertext \mathbf{c} from P_j , party P_i will delete a layer using the secret key $\mathbf{sk}_{i \rightarrow j}^{(r)}$ that was used to add a layer of encryption at round r of the Aggregate Stage. Otherwise, it uses **Leveled-Encrypt**

to encrypt the message \perp under the level- $(r - 1)$ public key that was received in round r during the Aggregate Stage.

Output. If P_i is the party that gets the output in that phase, i.e., $P_i = P_o$ and it receives a level-1 ciphertext c from its first neighbour, P_i computes the output message using **Decrypt** using the secret key $\mathbf{sk}_{i \rightarrow 1}^{(1)}$. In any other case, P_i outputs \perp . P_i also outputs its unhappy bit u_i .

Now, DFH-THC runs the protocol DFH-RandomWalkPhase n times, similarly to BC-OB.

Protocol DFH-THC($T, (d_i, \text{input}_i)_{P_i \in \mathcal{P}}, f$)

Each party P_i sets $\text{output}_i = \mathbf{1}$ and $u_i = 0$.

for o from 1 to n **do**

Parties jointly execute $((\text{input}_i^{\text{temp}}, u_i^{\text{temp}})_{P_i \in \mathcal{P}}) = \text{DFH-RandomWalkPhase}(T, P_o, (d_i, \text{input}_i, u_i)_{P_i \in \mathcal{P}}, f)$.

Party P_o sets $\text{output}_o = \text{input}_o^{\text{temp}}$.

Each party P_i sets $u_i = u_i^{\text{temp}} \vee u_i$.

Each party P_i **outputs** output_i if $\text{output}_i \neq \perp$.

3.13 Introduction

In the wake of GDPR and other privacy laws, companies need ways to process data in a way such that the trust is distributed among several parties. A fundamental solution to this problem is secure multiparty computation. Here, one commonly assumes that all parties have pairwise communication channels. In contrast, for many real-world scenarios, the communication network is not complete, and parties can only communicate with a subset of other parties. A natural question is whether a set of parties can successfully perform a joint computation over an incomplete communication network while revealing no information about the network topology.

The problem of *topology-hiding computation* (THC) was introduced by Moran et al. [60], who showed that THC is possible in the setting with passive corruptions and graphs with logarithmic diameter. Further solutions improve the communication efficiency [43] or allow for larger classes of graphs [4, 2]. Recent results [9, 53] even provide THC for fail-stop or semi-malicious adversaries (although at the price of leaking some small amount of information about the topology).

However, all those results consider the *fully synchronous* model, where a protocol proceeds in rounds. This model makes two assumptions: first, the parties have access to synchronized clocks, and second, every message is guaranteed to be delivered within one round. While the first assumption is reasonable in practice, as nowadays computers usually stay synchronized with milliseconds of variation, the second assumption makes protocols inherently impractical. This is because the running time of a protocol is always counted in the number of rounds, and the round length must be chosen based on the most pessimistic bound on the message delivery time. For

concreteness, consider a network where most of the time messages are delivered within milliseconds, but one of the connections, once in a while, may slow down to a couple of hours. In this case, a round would have to take a couple of hours.

3.13.1 Contributions

This motivates the goal of this work, which is to construct THC protocols for more realistic settings, where messages are not guaranteed to be delivered within a fixed time bound.

Model. A natural starting point would be to consider the strongest possible adversary, i.e. one who fully controls message delivery (this is the standard setting considered by asynchronous MPC, e.g. [12, 15]). First, note that this standard model is not well suited for our setting, since in order to decide when messages are delivered, the adversary must know the network, which we attempt to hide. The next logical step is to consider a model where the adversary can only interfere with delays between parties he controls, but unfortunately, even this grants the adversary too much power. In fact, we prove in Appendix 3.17 that it is impossible to get a topology-hiding broadcast in this model.

This forces us to define a slightly weaker model. We call it the Probabilistic Unknown Delay Model and we formally define it in Section 3.14. In this model the messages are delayed independently of the adversary, but different connections have different, unbounded probabilistic delays. This means that we throw off the assumption that makes the synchronous protocols impractical. Still, parties have access to synchronized clocks.

Protocols. We remark that it is not easy to modify synchronous THC protocols (even those tolerating fail-stop adversaries) to remain secure in the Probabilistic Unknown Delay Model. For example, consider the standard technique of letting each party attach to each message a round number r , and then wait until it receives all round- r messages before proceeding to the next round. This seems to inherently leak the topology, as the time at which a party receives a message for round r reveals information about the neighborhood of the sender (e.g., that it contains an edge with very long delays).

This forces us to develop new techniques, which result in three new protocols, secure in the Probabilistic Unknown Delay Model against any number of passive corruptions. We require a setup, but this setup is independent of the network topology (it only depends on the number of parties), and it can be used to run multiple instances of the protocols, with different communication graphs.

Our first two protocols (Section 3.15) implement topology-hiding broadcast (any functionality can then be realized using standard techniques, by executing a sequence of broadcasts). The protocols are based on standard assumptions, but can only be used in limited classes of graphs (the same ones as in [4]): cycles and trees, respectively.⁶ Furthermore, observe that the running time of a protocol could itself leak information about the topology. Indeed, this issue seems very difficult to overcome,

⁶Our second protocol works for any graphs, as long as we agree to reveal a spanning tree: the

since, intuitively, making the running time fully independent of the graph delays conflicts with our goal to design protocols that run as fast as the *actual* network. We deal with this by making the running time of our protocols depend only on the sum of all the delays in the network.

Then, in Section 3.16, we introduce a protocol that implements any functionality, works on arbitrary connected graphs, and its running time corresponds to (one sample of) the sum of all delays. On the other hand, we assume stateless secure hardware. Intuitively, a hardware box is a stateless program with an embedded secret key (the same for all parties). This assumption was introduced in [9] in order to deal with fail-stop adversaries in THC. Similar assumptions have also been considered before, for example, stateless tamper-proof tokens [20, 38, 27]⁷, or honestly-generated secure hardware [44, 26].

While secure hardware is a very strong assumption, the paradigm of constructing protocols with the help of a hardware oracle and then replacing the hardware oracle by more standard assumptions is common in the literature (see for example the secure hardware box assumption for the case of synchronous topology-hiding computation (with known upper bounds on the delays) for fail-stop adversaries [9], which was later relaxed to standard assumptions [53], or the Signature Card assumption for proofs-carrying-data schemes [26]). We hope that the techniques presented in this paper can be useful to construct protocols in more standard models.

3.13.2 Related Work

Topology-hiding computation was introduced by Moran et al. in [60]. The authors propose a broadcast protocol tolerating any number of passive corruptions. The construction uses a series of nested multi-party computations, in which each node is emulated by its neighbors. This broadcast protocol can then be used to achieve topology-hiding MPC using standard techniques to transform broadcast channels into secure point-to-point channels. In [43], the authors provide a more efficient construction based on the DDH assumption. However, both results are only feasible for graphs with logarithmic diameter. Topology-hiding communication for certain classes of graphs with large diameter was described in [4]. This result was finally extended to arbitrary (connected) graphs in [2]. These results were extended to the fail-stop setting in [9] based on stateless secure hardware, and [53] based on standard assumptions. All of the results mentioned above are in the cryptographic setting. Moreover, all results are stated in the synchronous communication model with known upper bounds on the delays.

In the information-theoretic setting, the main result is negative [41]: any topology-hiding MPC protocol inherently leaks information about the network graph. This work also shows that if the routing table is leaked, one can construct an MPC protocol

parties know which of their edges are on the tree and execute the protocol, ignoring other edges. See also [4].

⁷The difference here is that a token typically needs to be passed around during the protocol and the parties can embed their own programs in it, whereas a secure hardware box is used only by one party and is initialized with the correct program.

which leaks no additional information.

3.14 The Probabilistic Unknown Delay Model

At a high level, we assume loosely synchronized clocks, which allow the parties to proceed in rounds. However, we do not assume that the messages are always delivered within one round. Rather, we model channels that have delays drawn from some distributions each time a message is sent along (a different distribution for each channel). These delays are a property of the network. As already mentioned, this allows to achieve a significant speedup, comparable to that of asynchronous protocols and impossible in the fully synchronous model.

3.14.1 Impossibility of Stronger Models

Common models for asynchronous communication [12, 15] consider a worst-case scenario and give the adversary the power to schedule the messages. By scheduling the messages, the adversary automatically learns which parties are communicating. As a consequence, it is unavoidable that the adversary learns the topology of the communication graph, which we want to hide.

A natural definition, then, would be to give the adversary control over scheduling on channels from his corrupted parties. However, any reasonable model in which the adversary has the ability to delay messages for an unbounded amount of time allows him to learn something about the topology of the graph. In essence, a very long delay from a party behaves almost like an abort, and an adversary can exploit this much like a fail-stop adversary in the impossibility result of [59]. We formally prove this in a very weak adversarial model in Appendix 3.17.

Since delays cannot depend on the adversary without leaking topology, delays are an inherent property of the given network, much like in real life. As stated before, we give each edge a delay distribution, and the delays of messages traveling along that edge are sampled from this distribution. This allows us to model real-life networks where the adversary cannot tamper with the network connections. For example, on the Internet, delays between two directly connected nodes depend on their distance and the reliability of their connection.

3.14.2 Adversary

We consider an adversary, who statically and passively corrupts any set $\mathcal{Z} \subseteq \mathcal{P} = \{P_1, \dots, P_n\}$ of parties, with $|\mathcal{Z}| < n$. Static corruptions mean that the set \mathcal{Z} is chosen before the protocol execution. Passively corrupted parties follow the protocol instructions, but the adversary can access their internal states during the execution.

The setting with passive corruptions and secure hardware boxes is somewhat subtle. In particular, the adversary is allowed to input to the box of a corrupted party any messages of his choice, even based on secret states of other corrupted parties; he can even replay messages from honest parties with different corrupted inputs. This

will be why we need authenticated encryption, for example. Importantly, in the passive model, the messages actually sent by a corrupted party are produced using the box with valid inputs.

3.14.3 Communication Network and Clocks

Clocks. Each party has access to a clock that *ticks* at the same rate as every other clock. These ticks are fast; one can think of them as being milliseconds long or even faster (essentially, the smallest measurable unit of time).

We model the clocks by the clock functionality $\mathcal{F}_{\text{CLOCK}}$ of [50], which we recall here for completeness. The functionality keeps the absolute time τ , which is just the number of ticks that have passed since the initialization. Every single tick, a party is activated, given the time, and runs a part of the protocol. To ensure that honest parties are activated at least once every clock tick, the absolute time is increased according to “Ready” messages from honest parties.

Functionality $\mathcal{F}_{\text{CLOCK}}$

The clock functionality stores a counter τ , initially set to 0. For each honest party P_i it stores a flag d_i , initialized to 0.

ReadClock: On input (READCLOCK) from party P_i return τ .

Ready: On input (READY) from honest party P_i set $d_i = 1$.

ClockUpdate: On every activation the functionality runs this code before doing anything else.

- 1: **if** for every honest party P_i it holds $d_i = 1$ **then**
- 2: Set $d_i = 0$ for every honest party P_i .
- 3: Set $\tau = \tau + 1$.

Because clocks wait for “Ready” messages, computation is instant, happening within a single clock-tick. While this is not exactly what happens in the real world, our protocols do not abuse this property. In particular, they proceed in rounds, where each round takes a number (e.g., one million) clock-ticks. Parties process and send messages only once in a round, and remain passive at other times (in real world, this would be the time they perform the computation).

Network. The (incomplete) network with delays is modeled by the network functionality \mathcal{F}_{NET} . Similar to the synchronous models for THC, the description of the communication graph is inputted before the protocol execution by a special party P_{setting} . In our case, this description also contains a (possibly different) probability distribution for each edge indicating its delay. Each party can ask the functionality for its neighborhood in the communication graph and the delay distributions on the

edges to its neighbors.⁸ During the protocol execution, at every clock tick, parties can send to each neighbor a message, which is delivered after a delay sampled from a given distribution.

Functionality \mathcal{F}_{NET}

The functionality is connected to a clock functionality $\mathcal{F}_{\text{CLOCK}}$. The functionality stores a communication graph G and, for each edge e , a distribution D_e from which delays are sampled. Initially, G contains no edges. The functionality also stores the current time τ and a set of message tuples **buffer** which initially is empty.

Clock Update: Each time the functionality is activated, it first queries $\mathcal{F}_{\text{CLOCK}}$ for the current time and updates τ accordingly.

Initialization Step: // This is done at most once, before the protocol starts.

The party P_{setting} inputs a communication graph G and, for each edge e , a distribution D_e . The functionality stores G and D_e .

Graph Info: On input (GETINFO) from an honest party P_i , the functionality outputs to P_i its neighborhood $\mathbf{N}_G(P_i)$ and the delay distribution $D_{(i,j)}$ for all $j \in \mathbf{N}_G(P_i)$.

Communication Step:

- On input (SEND, i, j, m) from party P_i , where $P_j \in \mathbf{N}_G(P_i)$, \mathcal{F}_{NET} samples the delay d_{ij} for the edge (i, j) from $D_{(i,j)}$ and records the tuple $(\tau + d_{ij}, P_i, P_j, m)$ in **buffer**.⁹
- On input (FETCHMESSAGES, i) from P_i , for each message tuple (T, P_k, P_i, m) from **buffer** where $T \leq \tau$, the functionality removes the tuple from **buffer** and outputs (k, m) to P_i .

Leakage in the ideal world. During the protocol execution the adversary can learn local neighborhoods from \mathcal{F}_{NET} . Therefore, any ideal-world adversary should also have access to this information. This is ensured by the ideal-world functionality $\mathcal{F}_{\text{INFO}}^{\mathcal{L}}$, which has the same initialization step and the same graph information as \mathcal{F}_{NET} , but does not allow for actual communication.

Moreover, in any protocol it is unavoidable that the adversary learns the time at which the output is revealed. In previous synchronous THC protocols, this quantity corresponded to a fixed number of rounds (depending on an upper bound on the graph size or its diameter). This can no longer be the case in our model, where the number of rounds it takes to deliver a message is unbounded. Hence, it is necessary

⁸In fact, our hardware-based protocol does not use this information, and our protocols for cycles and trees only need upper bounds on the expected values of the delays. This bound can be easily established, e.g. by probing the connection.

⁹Technically, our model allows to send in one round multiple independent messages. However, our protocols do not exploit this property; we only assume that messages are independent if they are sent in different rounds.

to parameterize $\mathcal{F}_{\text{INFO}}^{\mathcal{L}}$ by a leakage function \mathcal{L} , that allows the adversary to compute the output time. \mathcal{L} depends on the set \mathcal{D} of all delay distributions in the network, but it does not on the communication graph itself. Additionally, we allow the adversary to pass to \mathcal{L} an auxiliary input, that will accommodate any protocol parameters that influence the output time.

For example, in our protocol based on secure hardware, \mathcal{L} will return the distribution of the sum of all network delays, rounded to the next multiple of the round length R (where R is provided as auxiliary input by the adversary).

Functionality $\mathcal{F}_{\text{INFO}}^{\mathcal{L}}$

Initialization Step: // This is done at most once, before the protocol starts.

The party P_{setting} inputs a communication graph G and, for each edge e , a distribution D_e . The functionality stores G and D_e .

Graph Info:

- On input (GETINFO) from an honest party P_i , the functionality outputs to P_i its neighborhood $\mathbf{N}_G(P_i)$ and the delay distribution $D_{(i,j)}$ for all $j \in \mathbf{N}_G(P_i)$.
- On the first input (GETINFO, aux) from the adversary the functionality outputs: the neighborhood of all corrupted parties, the delay distribution of every edge where at least one of the nodes is corrupted, and the leakage $\mathcal{L}(\text{aux}, \mathcal{D})$, where \mathcal{D} is the set of all delay distributions in the network.

3.14.4 Additional Related Work

Katz et al. [50] introduce eventual-delivery and channels with a fixed known upper bound. These functionalities implement communication between two parties, where the adversary can set, for each message, the delay after which it is delivered. For reasons stated at the beginning of this section, such functionalities cannot be used directly to model topology-hiding computation. Instead of point-to-point channels we need to model the whole communication network, and we cannot allow the adversary to set the delays. Intuitively, \mathcal{F}_{NET} implements a number of bounded-delay channels, each of which is modified so that the delay is chosen once and independently of the adversary. If we did not consider hiding the topology, our modified channels would be a stronger assumption.

Cohen et al. [28] define different channels with probabilistic delays, for example point-to-point channels (the SMT functionalities) and an all-to-all channel (parallel SMT, or PSMT). However, their PSMT functionality cannot be easily modified to model THC, since the delivery time is sampled once for all parties. One could modify the SMT functionalities and use their parallel composition, but we find our formulation simpler and much better suited for THC.

3.15 Protocols for Restricted Classes of Graphs

This section considers protocols that realize topology-hiding broadcast in the Probabilistic Unknown Delay Model under standard assumptions (in particular, we give an instantiation based on DDH), but in the limited setting where graphs are trees or cycles. We stress that we can deal with any graphs if a spanning tree is revealed. In the following, we first recall the known technique to achieve fully-synchronous THC using random walks and so-called PKCR encryption [2]. Then, we extend PKCR by certain additional properties, which allows us to construct a broadcast protocol for cycles in the Probabilistic Unknown Delay Model. Finally, we extend this protocol to trees.

3.15.1 Synchronous THC from Random Walks

Currently, the most efficient fully-synchronous THC protocols are based on the technique of correlated random walks, introduced in [2]. Intuitively, a PKCR scheme is assumed, which is an enhanced public-key encryption scheme on group elements, where the public keys come with a group operation: we write $\mathbf{pk}_{12} = \mathbf{pk}_1 \otimes \mathbf{pk}_2$. The encryption and decryption algorithms are denoted $\text{PKCR.Enc}(m, \mathbf{pk})$ and $\text{PKCR.Dec}(c, \mathbf{sk})$, respectively. Additionally, a party can add a layer of encryption to a ciphertext c encrypted under \mathbf{pk}_1 , using the algorithm $\text{PKCR.AddLayer}(c, \mathbf{sk}_2)$, which outputs an encryption c' under the combined key \mathbf{pk}_{12} . This operation can be undone with $\text{PKCR.DelLayer}(c', \mathbf{sk}_2)$. We also require that PKCR is homomorphic and rerandomizable (note that the latter is implied).

The goal is to broadcast one bit. However, we instead realize the OR functionality, which can then be used for broadcast (in the semi-honest setting) by having the sender input his bit, and all other parties input 0. The protocol proceeds as follows. A party starts by encrypting 0 if its input bit is 0, and a random group element otherwise, under a fresh key. In the first, so-called aggregate phase, this ciphertext travels along a random walk for a fixed number of rounds R (collecting the input bits of each party until it has traversed the whole graph with high probability). In each round, each party adds a layer of encryption to the received ciphertext (using a freshly generated key) and homomorphically adds its input. After R rounds, the parties start the decrypt phase, in which they send the final ciphertext back through the same walk it traversed in the first phase, and the layers of encryption are removed (using the secret keys stored during the aggregate phase). It is important that the ciphertext is sent via the same walk, to remove exactly the same layers of encryption that were added in the first phase. The parties determine this walk based on how they routed the ciphertext in the corresponding round of the aggregate phase. After another R rounds, each party interprets the group element as a 0-bit (the 0 element) or as a 1-bit (any other element).

This technique breaks down in the Probabilistic Unknown Delay Model. For example, it is not clear how to choose R such that the walk traverses the whole graph since it would depend on an upper bound on the delays. Moreover, in the decrypt phase, parties no longer know how to route a ciphertext back via the same walk it

took in the aggregate phase. This is because they do not know the number of steps it already made in the backward walk (this depends on the actual delays). Furthermore, it is not straightforward to modify the random walk technique to deal with this. For instance, the standard method of attaching a round number to every message (to count the number of encryption layers) reveals information about the topology.

3.15.2 Protocol for Cycles

We assume an enhanced PKCR scheme, denoted PKCR*. The main differences from PKCR are as follows. First, the message space in PKCR* is now the set $\{0, 1\}$, and it is disjoint from the ciphertext space. This allows to distinguish between a layered ciphertext and a plaintext. Moreover, we no longer require explicit homomorphism, but instead use the algorithm PKCR*.ToOne(c) that transforms an encryption of 0 into an encryption of 1 without knowing the public key¹⁰. We formally define PKCR* and give an instantiation based on the DDH assumption in Appendix 3.18.

Rounds. Although we are striving for a protocol that behaves in a somewhat asynchronous way, we still have a notion of rounds defined by a certain number of clock ticks. Even though each party is activated in every clock tick, each party receives, processes and sends a message only every R clock ticks — this keeps parties in sync despite delays, without clogging the network. Even if no message is received, a message is sent¹¹. This means that at time τ , we are on round $r_\tau = \lfloor \tau/R \rfloor$; the τ parameter will be dropped if obvious from context. Moreover, observe that the message complexity increases as R decreases. For reference, R can be thought of as relatively large, say 1,000 or more; this is also so that parties are able to completely process messages every round.

A protocol with constant delays. To better explain our ideas, we first describe our protocol in the setting with constant delays, and then modify it to deal with any delay distributions.

The high-level idea is to execute directly the decrypt phase of the random-walk protocol, where the walk is simply the cycle traversal, and the combined public key corresponding to the ciphertext resulting from the aggregate phase is given as the setup (note that this is independent of the order of parties on the graph). More concretely, we assume that each party P_i holds a secret key \mathbf{sk}_i and the combined public key $\mathbf{pk} = \mathbf{pk}_1 \circledast \dots \circledast \mathbf{pk}_n$. Assume for the moment that each party knows who the next clockwise party is in the cycle. At the beginning, a party P_i , every round (i.e., every R clock ticks), starts a new cycle traversal by sending to the next party a fresh encryption of its input PKCR*.Enc(b_i, \mathbf{pk}). Once P_i starts receiving ciphertexts from its neighbor (note that since the delays are fixed, there is at most one ciphertext arriving in a given round), it instead continues the cycle traversals.

¹⁰Its functionality does not matter and is left undefined on encryptions of 1.

¹¹If the parties do not send at every round, the topology would leak. Intuitively, when a party P_i sends the initial message to its right neighbor P_j , the right neighbor of P_j learns how big the delay from P_i to P_j was. We can extend this to larger neighborhood, eventually revealing information about relative positions of corrupted parties.

That is, every time it receives a ciphertext c from the previous neighbor, it deletes the layer of encryption using its secret key: $\text{PKCR}^*.\text{DelLayer}(c, \text{sk}_i)$. It then rerandomizes the result and sends it to the next party. The sender additionally transforms the ciphertext it receives to a 1-ciphertext in case its bit is 1. After traversing the whole cycle, all layers of encryption are removed and the parties can recognize a plaintext bit. This happens at the same time for every party.

In order to remove the assumption that each party knows who the next clockwise party is, we simply traverse the cycle in both directions.

A protocol accounting for variable delays. The above approach breaks down with arbitrary delays, where many messages can arrive at the same round. We deal with this by additionally ensuring that every message is received in a predictable timely manner: we will be repeating message sends. As stated in Section 3.14, the delays could be variable, but we make the assumption that if messages are sent at least R clock-ticks from each other, then the delay for each message is independent. We also assume that the median value of the delay along each edge is polynomial, denoted as $\text{Med}[D_e]$. Now, since the protocol will handle messages in rounds, the actual values we need to consider are all in rounds: $\lceil \text{Med}[D_e]/R \rceil$.

Now, if over κ rounds, P_1 sends a message c each round, the probability that none of the copies arrives after $\kappa + \lceil \text{Med}[D_e]/R \rceil$ rounds is negligible in terms of κ , the security parameter (see Lemma 8 for the proof). Because we are guaranteed to have the message by that time (and we believe with reasonable network delays, median delay is small), we wait until time $(\kappa + \lceil \text{Med}[D_e]/R \rceil) \cdot R$ has passed from when the original message was sent before processing it.¹²

For the purposes of this sketch, we will just consider sending messages one way around the protocol. We will also focus on P_1 (with neighbors P_n and P_2) since all parties will behave in an identical manner. First, the setup phase gives every party the combined public key $\text{pk} = \text{pk}_1 \otimes \dots \otimes \text{pk}_n$. At each step, processing a message will involve using the $\text{PKCR}.\text{DelLayer}$ functionality for their key.

In the first round, P_1 sends its bit (0 if not the source node, b_s if the source node) encrypted under pk to P_2 , let's call this message $c_1^{(1)}$. P_1 will wait $w = \kappa + \lceil \text{Med}[D_e]/R \rceil$ rounds to receive P_n 's first message during this time. Now, because P_1 needs to make sure $c_1^{(1)}$ makes it to P_2 , for the next κ rounds, P_1 continues to send $c_1^{(1)}$. However, because P_1 also needs to hide w (and thus cannot reveal when it starts sending its processed message from P_n), P_1 starts sending a new ciphertext encrypting the same message, $c_2^{(1)}$ (again κ times over κ rounds), until it has waited w rounds — so, P_1 is sending $c_1^{(1)}$ and $c_2^{(1)}$ in the second round, $c_1^{(1)}$, $c_2^{(1)}$ and $c_3^{(1)}$ the third round and so forth until it sends $c_1^{(1)}, \dots, c_\kappa^{(1)}$ in round κ . Then it stops sending $c_1^{(1)}$ and starts sending $c_{\kappa+1}^{(1)}$. P_1 will only ever send κ messages at once per round. Once it has waited w rounds, P_1 is guaranteed to have received the message from P_n and can process and forward that message, again sending it κ times over κ rounds. In the next round, P_1 will then be guaranteed to receive the next message from P_n ,

¹²Note that delays between rounds are independent, but not within the round. This means we need to send copies of the message over multiple rounds for this strategy to work.

and so on.

Denote the median-round-sum as $\text{MedRSum}[D] = \sum_{i=1}^n \lceil \text{Med}[D_{(i, (i+1 \bmod n)+1)}] / R \rceil$. Because each party waits like this, the protocol has a guaranteed time to end, the same for all parties:

$$R \cdot \sum_{i=1}^n w_i = R(n\kappa + \text{MedRSum}[D]).$$

This is the only information ‘leaked’ from the protocol: all parties learn the sum of ceiling’d medians, $\text{MedRSum}[D]$. Additionally, parties all know the (real, not a round-delay) distribution of delays for messages to reach them, and thus can compute $\lceil \text{Med}[D_e] / R \rceil$ for their adjacent edges.

Formally, the protocol **CycleProt** is described as follows.

Protocol CycleProt

// The common input of all parties is the round length R . Additionally, the sender P_s has the input bit b_s .

Setup: For $i \in \{1, \dots, n\}$, let $(\text{pk}_i, \text{sk}_i) = \text{PKCR}^*. \text{KGen}(1^\kappa)$. Let $\text{pk} = \text{pk}_1 \otimes \dots \otimes \text{pk}_n$. The setup outputs to each party P_i its secret key sk_i and the product public key pk .

Initialization for each P_i :

- Send (GETINFO) to the functionality \mathcal{F}_{NET} and assign randomly the labels P^0 , P^1 to the two neighbors.
- Let $\text{Rec}^0, \text{Rec}^1$ be lists of received messages from P^0 and P^1 respectively, both initialized to \perp . Let Send^0 and Send^1 be sets initialized to \emptyset ; these are the sets of messages that are ready to be sent.
- For each $\ell \in \{0, 1\}$, $D_{(i, \ell)}$ is the delay distribution on the edge between P_i and P^ℓ , obtained from $\mathcal{F}_{\text{INFO}}$.
- Let $w^\ell = \kappa + \lceil \text{Med}[D_{(i, \ell)}] / R \rceil$ be the time P_i waits before sending a message from P^ℓ to $P^{1-\ell}$.

Execution for each P_i :

- 1: Send (READCLOCK) to the functionality $\mathcal{F}_{\text{CLOCK}}$ and let τ be the output. If $\tau \bmod R \neq 0$, send (READY) to the functionality $\mathcal{F}_{\text{CLOCK}}$. Otherwise, let $r = \tau / R$ be the current round number and do the following:
- 2: *Receive messages:* Send (FETCHMESSAGES, i) to the functionality \mathcal{F}_{NET} . For each message (r_c, c) received from a neighbor P^ℓ , set $\text{Rec}^\ell[r_c + w^\ell] = c$.
- 3: *Process if no messages received:* For each neighbor P^ℓ such that $\text{Rec}^\ell[r] = \perp$, start a new cycle traversal in the direction of $P^{1-\ell}$:
 - If P_i is sender (i.e. $i = s$) then add $(\kappa, r, \text{PKCR}^*. \text{Enc}(b_s, \text{pk}))$ to $\text{Send}^{1-\ell}$.
 - Otherwise, add $(\kappa, r, \text{PKCR}^*. \text{Enc}(0, \text{pk}))$ to $\text{Send}^{1-\ell}$.
- 4: *Process received messages:* For each P^ℓ such that $\text{Rec}^\ell[r] \neq \perp$ (we have received a message from P^ℓ), set $d = \text{PKCR}^*. \text{DeLayer}(R^\ell[r], \text{sk}_i)$, and do the following:

- If $d \in \{0, 1\}$, output d and halt (we have decrypted the source bit).
 - Otherwise, if $i = s$ and $b_s = 1$, then set $d = \text{PKCR}^*. \text{ToOne}(d)$. Then, in either case, add $(\kappa, r, \text{PKCR}^*. \text{Rand}(d))$ to $\text{Send}^{1-\ell}$.
- 5: *Send message:* For each $\ell \in \{0, 1\}$, let $\text{Sending}^\ell = \{(k, r_c, c) \in \text{Send}^\ell : k > 0\}$. For each $(k, r_c, c) \in \text{Sending}^\ell$, send (r_c, c) to P^ℓ .
- 6: *Update Send set:* For each $(k, r_c, c) \in \text{Sending}^\ell$, remove (k, r_c, c) from Send^ℓ and insert $(k - 1, r_c, c)$ to Send^ℓ .
- 7: Send (READY) to the functionality $\mathcal{F}_{\text{CLOCK}}$.

In Appendix 3.19 we prove the following theorem. (\mathcal{F}_{BC} denotes the broadcast functionality.)

Theorem 8. *The protocol CycleProt UC-realizes $(\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{INFO}}^{\mathcal{L}_{\text{median}}}, \mathcal{F}_{\text{BC}})$ in the $(\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{NET}})$ -hybrid model with an adversary who statically passively corrupts any number of parties, where the leakage function is defined as $\mathcal{L}_{\text{median}}(R, \mathcal{D}) = \text{MedRSum}[\mathcal{D}]$.¹³*

3.15.3 Protocol for Trees

We show how to modify the cycle protocol presented in the previous section to securely realize the broadcast functionality \mathcal{F}_{BC} in any tree. As observed in [4], given a tree, nodes can locally compute their local views of a cycle-traversal of the tree. However, to apply the cycle protocol to this cycle-traversal, we would need as setup a combined public key that has each secret key sk_i as many times as P_i appears in the cycle-traversal. To handle that, each party simply removes its secret key from the ciphertexts received from the first neighbor, and we can assume the same setup as in the cycle protocol.

In Appendix 3.20 we give a formal description of the protocol TreeProt. The proof of the following theorem is a straightforward extension of the proof of Theorem 8.

Theorem 9. *The protocol TreeProt UC-realizes $(\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{INFO}}^{\mathcal{L}_{\text{median}}}, \mathcal{F}_{\text{BC}})$ in the $(\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{NET}})$ -hybrid model with an adversary who statically passively corrupts any number of parties, where the leakage function is defined as $\mathcal{L}_{\text{median}}(R, \mathcal{D}) = \text{MedRSum}[\mathcal{D}]$.*

3.16 Protocol for General Graphs

We present a protocol that allows us to securely realize any functionality in any connected communication graph with unknown delay distributions on the edges. For that, we use the same setup as [9]: we assume that the parties have access to secure hardware boxes, initialized with the same secret key, and executing the same functionality \mathcal{F}_{HW} , independent of the graph and the realized functionality (see [9] for details of this model).

¹³Note that the round length R is a parameter of the protocol, so we allow the adversary to provide it.

Our protocol is divided into two sub-protocols: preprocessing and computation. Both sub-protocols do not terminate on their own. Rather, we assume that each party gets a signal when it can finish each sub-protocol.¹⁴ The preprocessing is executed only once, before any input is specified and can be re-used. Intuitively, it outputs, for each party, an encryption of the entire communication graph under the secret key embedded in the hardware boxes. The computation allows to evaluate any function, with the help of the encrypted information outputted by the preprocessing. One output of preprocessing can be used to execute the computation any number of times, each time with different function and different inputs.

In the following, we formally describe both protocols. To make the exposition easier to follow, we postpone the precise definition of the functionality \mathcal{F}_{HW} executed by the hardware boxes, to Appendix 3.21, and for now only give an informal description of its behavior whenever \mathcal{F}_{HW} is invoked.

3.16.1 Preprocessing

The preprocessing is executed without any inputs. The output is a pair (id_i, c) , where id_i is a (secret) random string used to identify a party, and c is a ciphertext that contains an encrypted state with the whole graph. This output pair will be inputted to the computation protocol.

At a high level, the protocol floods the network with encrypted partial images of the graph, until the signal to terminate occurs. We assume that the signal occurs late enough for all parties to collect all information. In more detail, throughout the protocol, a party P_i keeps an encrypted state c , containing information about the graph and parties' id 's, that it collected up to a given point. Initially, c contains only the local neighborhood and id_i chosen at random by P_i . Then, every round, P_i sends c to all its neighbors. When it receives a state c_j from a neighbor P_j , it uses the functionality \mathcal{F}_{HW} box to update c with the information from c_j . That is, \mathcal{F}_{HW} gets as input two encrypted states containing partial images on the graph, respectively, decrypts both states and merges the information into a new state, which is encrypted and output.

Protocol Hw-Preprocessing

// The common input of all parties is the round length R .

Setup: Each party P_i has access to a secure hardware box functionality \mathcal{F}_{HW} .

Initialization for each P_i : Choose an identifier id_i at random and send (GETINFO) to \mathcal{F}_{NET} , to obtain the neighborhood $\mathbf{N}_G(P_i)$. Input $(i, \text{id}_i, \mathbf{N}_G(P_i))$ to \mathcal{F}_{HW} and store the resulting encrypted state c .

¹⁴In practice, this is not an unrealistic assumption. It would be enough, for example, if each party was given a very rough upper bound on the time it takes to flood the network and traverse all edges of the graph (for instance, a constant number proportional to the sum of delays on all edges). This is still faster than assuming worst-case upper bounds on the delays along edges, as one would need to do to adapt a fully synchronous protocol.

Execution for each P_i at every round (every R clock ticks):

- 1: Send c to each $P_j \in \mathbf{N}_G(i)$.
- 2: Send $(\text{FETCHMESSAGES}, i)$ to \mathcal{F}_{NET} . For each received message c' , input (id_i, c, c') to \mathcal{F}_{HW} and set the updated state c to the result.

Termination for each P_i : Upon receiving the signal, output (id_i, c) .

3.16.2 Computation

The inputs to the computation protocol are, for every P_i , its input x_i , a description of the function f_i that evaluates P_i 's output of the computed function, and the values id_i and c_i , outputted by preprocessing.

The high-level idea is that the hardware box \mathcal{F}_{HW} gets as part of its input the value c_i , containing, among others, the encrypted communication graph. This allows it to deterministically compute an Eulerian cycle, which visits every edge exactly twice. Then, every party starts a traversal of the Eulerian cycle, in order to collect the inputs from all parties. Once all inputs are collected, the box computes the function and gives the output to the party. Traversing each edge exactly twice allows all parties to learn the output at a time that does not depend on the graph topology but (roughly) on the distribution of the sum of the delays. Of course, all messages are encrypted under the secret key embedded in the hardware boxes.

This means that at any time during the protocol there are n cycle traversals going through the graph (one per a starting party). Each of the traversals visits all edges in the graph twice. So in each round a party P_i processes messages for up to n traversals. To hide the number of actual traversal processed P_i sends n messages to each each of its neighbors. This means that each round, P_i receives from each neighbor n messages. It inputs all of them to its hardware box (together with its input to the computed function) and receives back, for each neighbor, a set of n messages that it then sends to him.

A party receives the output once the cycle has been traversed, which takes time proportional to the sum of the rounded delays. Once the parties receive output, they continue executing the protocol until they receive the termination signal, which we assume occurs late enough for *all* parties to get their outputs.

There are still some subtle issues, that the above sketch does not address. First, the adversary could try to tamper with the ciphertexts. For example, in our protocol a message contains a list of id 's that identifies the path it already traversed. This is done so that the adversary cannot extend the traversal on behalf of an honest party P_i without knowing its secret id_i . Now the adversary could try to extend this list nevertheless, by copying part of the encrypted state of a corrupted party — recall that this state contains all id_i 's. To prevent such situations, we use authenticated encryption.

Second, we need to specify when the parties input the function they are evaluating into the box. Doing this at the very end would allow the adversary to evaluate many

functions of her choice, including the identity. So instead, in our protocol the function is inputted once, when the cycle traversal is started, and it is always a part of the message. This way, when the output is computed, the function is taken from a message that has been already processed by all honest parties. Since honest parties only process messages that are actually sent to them, and even corrupted parties only send correctly generated messages, this function must be the correct one. In some sense, when sending the first message to an honest party, the adversary commits herself to the correct function.

A similar problem occurs when the parties input to their boxes the inputs to the computed function. A sequence of corrupted parties at the end of the traversal can emulate the last steps of the protocol many times, with different inputs. To prevent this, we traverse the cycle twice. After the first traversal, the inputs are collected and the function is evaluated. Then, the (still encrypted) output traverses the cycle for the second time, and only then is given to the parties.

Finally, we observe that at the end of the protocol, a graph component of neighboring corrupted parties learns where the traversal enters their component (this can be done by fast-forwarding the protocol). Depending on how the eulerian cycle is computed, this could leak information about the topology. To address this, we introduce in Section 3.16.3 an algorithm for computing the traversal that does not have this issue (formally, the last part of the cycle can be simulated).

Protocol Hw-Computation

// The common input of all parties is the round length R . Additionally, each P_i has input $(x_i, f_i, \text{id}_i, c_i)$, where id_i is the identifier chosen in Hw-Preprocessing, and c_i is the encrypted state outputted by Hw-Preprocessing.

Setup: Each party P_i has access to a secure hardware box functionality \mathcal{F}_{HW} .

Initialization for each P_i : For each neighbor P_j , let $E_j = \emptyset$.

Execution for each P_i at every r clock ticks:

- 1: Send (FETCHMESSAGES) to \mathcal{F}_{NET} and receive the messages (E_1, \dots, E_ν) .
- 2: Choose r at random and input $(i, \text{id}_i, c_i, \bigcup_j E_j, x_i, f_i, r)$ to \mathcal{F}_{HW} . Get the result $(\text{val}, \{(E'_1, \text{next}_1), \dots, (E'_k, \text{next}_k)\})$. If $\text{val} \neq \perp$, output val , but continue running.
- 3: For each (E'_j, next_j) , for each $e \in E'_j$, send e to next_j via $(\text{SEND}, i, \text{next}_j, e)$.¹⁵

Termination for each P_i : Upon receiving the signal, terminate.

Realizing reactive functionalities. Reactive functionalities are those which require explicit interaction between parties, e.g. if the function we realize is very simple

¹⁵We will assume that every message sent in this round is independent. In this case this is equivalent to assuming only independence between rounds — since there is an upper bound n on the number of messages sent at once, one can always make the round longer, partition it into slots separated by a sufficient time interval, and send one message in every slot.

but we want to evaluate a complex function, parties may need to run this protocol multiple times in sequence, using previous outputs to generate the next inputs. Our current hardware protocol allows us to realize secure function evaluation. In the synchronous setting, this can be easily extended to reactive functionalities by invoking many function evaluations in sequence. However, in the setting with unknown delays this is no longer clear. For example, if our protocol is composed sequentially in the naive way, then parties start the second execution at different times, which leaks topology.

So, to get reactive functionalities or composition to work for this hardware protocol we can do one of two things. First, we could add a synchronization point before each ‘round’ of the reactive function. Second, we could employ the same trick as for the cycle/tree protocol in Section 3.15, sending the same message many times so that with high probability it arrives to the next node within some reasonable time interval. With this method, every party ends the protocol at exactly the same time, and so can start the next protocol at the same time, despite the delays.

The running time of the protocol **Hardware** depends only on the sum of all delays in the network, each rounded to the next multiple of the round length R , which is the only information leaked in the ideal world. In Appendix 3.22 we prove the following theorem.

Theorem 10. *For any efficiently computable and well-formed¹⁶ functionality \mathcal{F} , the protocol **Hardware** UC-realizes $(\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{INFO}}^{\mathcal{L}_{\text{sum}}}, \mathcal{F})$ in the $(\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{NET}}, \mathcal{F}_{\text{HW}})$ -hybrid model with an adversary who statically passively corrupts any number of parties, where $\mathcal{L}_{\text{sum}} := R \sum_{D_e \in \mathcal{D}} \lceil D_e / R \rceil$.*

Remark. One can observe that in our protocol the hardware boxes must be able to evaluate a complex function. This can be resolved at the cost of efficiency, by computing the functionality by many calls to the simple broadcast functionality. Note that even if we require one synchronization point per broadcast, this still seems reasonable, since it is possible to evaluate any function with constant number of broadcasts [30, 54].

3.16.3 Computing the Eulerian Cycle

It turns out that not every algorithm computing an Eulerian cycle can be used in \mathcal{F}_{HW} to achieve THC. In particular, during the execution of our protocol the adversary learns some information about a part of the cycle, which for some algorithms depends on the graph. More technically, during the simulation, it is necessary to compute the time when the adversary learns the output, and this happens as soon as the Eulerian cycle traversal enters a fragment of consecutive corrupted parties containing the output party. This is because it can “fast-forward” the protocol (without communication). Hence, we need an algorithm for computing such a cycle on a graph with doubled edges, for which the “entry point” to a connected component (of corrupted parties) can be simulated with only the knowledge of the component.

¹⁶Intuitively, a functionality is well-formed if its code does not depend on the ID’s of the corrupted parties. We refer to [17] for a detailed description.

Common algorithms, such as Fleury or Hierholzer [34, 33], check a global property of the graph and hence cannot be used without the knowledge of the entire graph topology. Moreover, a distributed algorithm in the local model (where the parties only have knowledge of its neighbors) such as [56] is also not enough, since the algorithm has to be executed until the end in order to know what is the last part of the cycle.

We present the algorithm `EulerianCycle`, which, if executed from a node u on a connected neighborhood containing u , leads to the same starting path as if it was executed on the whole graph. This property is enough to simulate, since the simulator can compute the last fragment of the Eulerian Cycle in the corrupted neighborhood. We note that the start of the cycle generated by our algorithm can be simulated, however, the simulator needs to compute the end. Hence, the hardware boxes will traverse the path outputted by `EulerianCycle` from the end.

The idea is to generate a tree from the graph, in such a way that the generated tree contains exactly the same edges as the graph. To do that, the tree is generated in a DFS-manner from a source u . At every step, a new edge (the one that leads to the smallest id according to a DFS order, and without repeating nodes) is added to the tree. Since the graph is connected, all edges are eventually added. Moreover, each edge is added exactly once, since no repeated nodes are expanded. See Figure 3-2 for an example execution.

Algorithm `EulerianCycle($u, G = (E, V)$)`

// Computes an eulerian cycle on the graph G with the set of nodes V and the set of edges E (where each edge is considered doubled), starting at node $u \in V$. We assume some ordering on V .

- 1: Let \mathcal{T} be the tree with a single root node u .
- 2: **while** $E \neq \emptyset$ **do**
- 3: **if** there is no $v \in V$ such that $(u, v) \in E$ **then**
- 4: Set $u = \text{parent}(\mathcal{T}, u)$
- 5: **else**
- 6: Pick the smallest v such that $(u, v) \in E$ and append v to the children of u in \mathcal{T} .
- 7: Set $E = E \setminus \{(u, v)\}$.
- 8: If $v \notin \text{nodes}(\mathcal{T})$, then set $u = v$.
- 9: Output the path corresponding to the in-order traversal of \mathcal{T} .

Supplementary Material

3.17 Adversarially-Controlled Delays Leak Topology

Much like how adversarially-controlled aborts were shown to leak topological information in [59], we can show that adversarially-controlled delays also leak topological

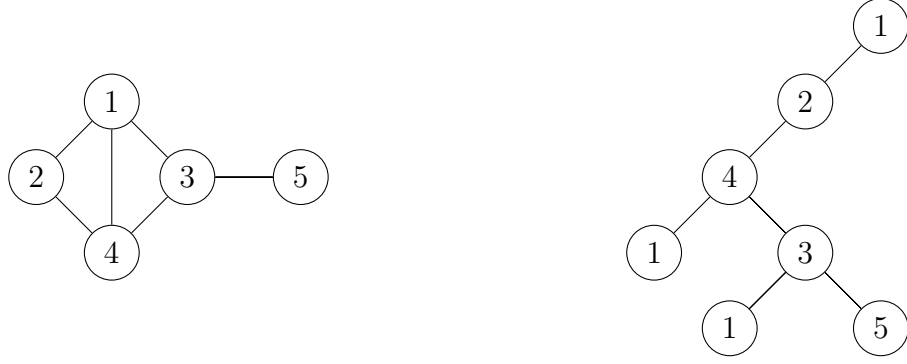


Figure 3-2: An example of a graph G (on the left) and the corresponding tree \mathcal{T} , computed by $\text{EulerianCycle}(1, G)$ (on the right). The eulerian cycle (on the graph with doubled edges) is $(1, 2, 4, 1, 3, 1, 3, 5, 3, 4, 2, 1)$.

information. First, note that if we have bounded delays, we can always use a synchronous protocol, starting the next round after waiting the maximum delay. So, in order for this model to be interesting, we must assume the adversary has unbounded delays. In order to be as general as possible, we prove this with the weakest model we can while still giving the adversary some control over its delays: the adversary can only add delay to messages leaving corrupt nodes.

Our proof will follow the structure of [59], using a similar game-based definition and even using the same adversarially-chosen graphs (see figure 3-3). Our game is straightforward. The adversary gives the challenger two graphs and a set of corrupt nodes so that the corrupt neighborhoods are identical when there is no adversarially added delay. The challenger then chooses one of those graphs at random, runs the protocol, and gives the views of all corrupt nodes to the adversary. The adversary wins if she can tell which graph was used. In [59], the adversary would choose a round to failstop one of its corrupt parties. In our model, the adversary will instead choose a time (clock-tick) to add what we call a long-delay (which is just a very long delay on sending that and all subsequent messages). The adversary will be able to detect the delay based on when the protocol ends: if the delay was early in the protocol, the protocol takes longer to finish for all parties, and if it was late, the protocol will still finish quickly for most parties.

This impossibility result translates to an impossibility in the simulation-based setting since a secure protocol for the simulation-based setting would imply a secure protocol for the game-based setting.

3.17.1 Adversarially-Controlled Delay Indistinguishability-based Security Definition

Before proving the impossibility result, we first formally define our model. This model is as weak as possible while still assuming delays are somewhat controlled by the adversary. We will assume a minimum delay along edges: it takes at least one clock-tick for a message to get from one party to another.

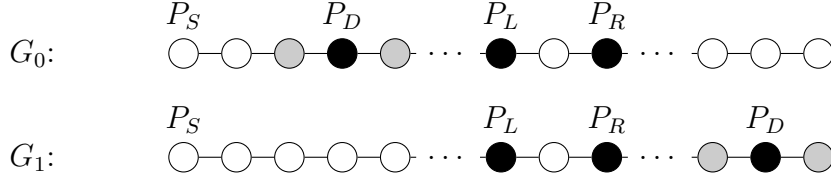


Figure 3-3: Graphs used to prove the impossibility of THC with adversarial delays. P_S is the sender. The corrupted parties (black dots) are: P_L and P_R (they delay messages), and the detective P_D . The adversary determines whether P_D (and its two neighbors) are on the left or on the right.

Delay Algorithms In order to give the adversary as little power as possible, we define a public (and arbitrary) randomized algorithm that outputs the delays for a graph for protocol Π . Both the adversary and challenger have access to this algorithm and can sample from it.

Definition 6. A indistinguishability-delay algorithm (IDA) for a protocol Π , $\text{DelayAlgorithm}_\Pi$, is a probabilistic polynomial-time algorithm that takes as input an arbitrary graph outputs unbounded polynomial delays for every time τ and every edge in the graph. Explicitly, for any graph $G = (V, E)$, $\text{DelayAlgorithm}(G)$ outputs \mathcal{T} such that for every edge $(i, j) \in E_b$ and time τ , $\mathcal{T}((i, j), \tau) = d_{(i, j), \tau}$ is a delay that is at least one.

The Indistinguishability Game This indistinguishability definition is a game between an adversary \mathcal{A} and challenger \mathcal{C} adapted from [59]. Let DelayAlgorithm be an IDA as defined above.

- **Setup:** Let \mathcal{G} be a class of graphs and Π a topology-hiding broadcast protocol that works on any of the networks described by \mathcal{G} according to our adversarial delay model, and let DelayAlgorithm be a public, fixed IDA algorithm. Without loss of generality, let P_1 have input $x \in \{0, 1\}$, the broadcast bit.
- \mathcal{A} chooses two graphs $G_0 = (V_0, E_0)$ and $G_1 = (V_1, E_1)$ from \mathcal{G} and then a subset \mathcal{Z} of the parties to corrupt. \mathcal{Z} must look locally the same in both G_0 and G_1 . Formally, $\mathcal{Z} \subset V_0 \cap V_1$ and $\mathbf{N}_{G_0}(\mathcal{Z}) = \mathbf{N}_{G_1}(\mathcal{Z})$. If this doesn't hold, \mathcal{C} wins automatically.
 \mathcal{A} then generates $\mathcal{T}_{\mathcal{Z}}$, a function defining delays for every edge at every time-step controlled by the adversary. That is, $\mathcal{T}_{\mathcal{Z}}((i, j), \tau) = d_{(i, j), \tau}$, and if $P_i \in \mathcal{Z}$, then every message sent from P_i to P_j at time τ is delayed by an extra $d_{(i, j), \tau}$. \mathcal{A} sends G_0, G_1, \mathcal{Z} , and $\mathcal{T}_{\mathcal{Z}}$ to \mathcal{C} .
- \mathcal{C} chooses a random $b \in \{0, 1\}$ and executes Π in G_b with delays according to $\text{DelayAlgorithm}(G_b) = \mathcal{T}$ for all messages sent from honest parties. For messages sent from corrupt parties, delay is determined by the time and parties as follows: for time τ a message sent from party $P_i \in \mathcal{Z}$ to P_j has delay $\mathcal{T}((i, j), \tau) + \mathcal{T}_{\mathcal{Z}}((i, j), \tau)$ in reaching P_j . \mathcal{A} receives the view of all parties in \mathcal{Z} during the execution.

- \mathcal{A} then outputs $b' \in \{0, 1\}$ and wins if $b' = b$ and loses otherwise.

Notice that in this model, the adversary statically and passively corrupts any set of parties, and statically determines what delays to add to the protocol.

Definition 7. A protocol Π is indistinguishable under chosen delay attack (IND-CDA) over a class of graphs \mathcal{G} if for any PPT adversary \mathcal{A} , there exists an IDA DelayAlgorithm such that

$$\Pr[\mathcal{A} \text{ wins}] \leq \frac{1}{2} + \text{negl}(n).$$

3.17.2 Proof that Adversarially-Controlled Delays Leak Topology

First, we will define what we mean when we say a protocol is ‘weakly’ realized in the adversarial delay model. Intuitively, it is just that the protocol outputs the correct bit to all parties if there is no adversarial delay.

Definition 8. A protocol Π weakly realizes the broadcast functionality if Π is such that when all parties execute honestly with delays determined by any IDA, all parties get the broadcast bit within polynomial time (with all but negligible probability).

Theorem 11. There does not exist an IND-CDA secure protocol Π that weakly realizes the broadcast functionality of any class of graphs \mathcal{G} that contains line graphs.

Throughout the proof and associated claim, we refer to a specific pair of graphs that the adversary has chosen to distinguish between, winning the IND-CDA game. Both graphs will be a line of n vertices: $G = (V, E)$ where $E = \{(P_i, P_{i+1})\}_{i=1, \dots, n-1}$. We will let Π be a protocol executed on G that weakly realizes broadcast when P_1 is the broadcaster, see Figure 3-3.

Our adversary in this model will either add no delay, or will add a very long polynomial delay to every message sent after some time τ .

Notice that \mathcal{A} is given access to DelayAlgorithm at the start of the protocol. One can sample from DelayAlgorithm using G_0 , G_1 , and \mathcal{Z} to get an upper bound T on the time it takes Π to terminate with all but negligible probability. Since Π weakly realizes broadcast, T is polynomial. So, \mathcal{A} has access to this upper bound T .

Long-delays. Let a long-delay be a delay that lasts for T clock-ticks. Consider an adversary that will only add long-delays to a protocol, and once an adversary has long-delayed a message, he must continue to long-delay messages along that edge until the end of the protocol. That is, once the adversary decides to delay along some edge, all subsequent messages along that edge cannot arrive for at least T clock-ticks.

Claim 7. Consider any party P_v whose neighbors do not add any extra delay as described by the long-delay paragraph above. As in [59], let $H_{v,b}$ be the event that P_v outputs the broadcast bit by time T (P_v may still be running the protocol by time T

or terminate by guessing a bit by T). Let E_τ be the event that the first long-delay is at time τ . Then either Π is not IND-CDA secure, or there exists a bit b such that

$$|\Pr[H_{v,b}|E_{T-1}] - \Pr[H_{v,b}|E_0]| \geq \frac{1}{2} - \text{negl}(n).$$

Proof. If some P_i long-delays at time 0, then the first message it sends is at time T , and so the graph is disconnected until time T . This makes it impossible for parties separated from P_1 to learn about the output bit by time T . So, by that time, these parties must either guess an output bit (and be right with probability at most $1/2$) or output nothing and keep running the protocol (which is still not $H_{v,b}$). If Π is IND-CDA secure, then all honest parties must have the same probability of outputting the output bit by time T , and so there exists a b such that $\Pr[H_{v,b}|E_0] \leq \frac{1}{2} - \text{negl}(n)$ for all honest parties P_v .

However, if P_i long-delays at time $T-1$, then the only parties possibly affected by P_i are P_{i-1} and P_{i+1} ; all other parties will get the output by time T and the information that P_i delayed cannot reach them (recall we assumed a minimum delay of at least one clock-tick in the **DelayAlgorithm**). So, $\Pr[H_{v,b}|E_0] = \Pr[H_{v,b}|\text{no extra delays}] = 1 - \text{negl}(n)$ for all honest parties without a delaying neighbor by the definition of weakly realizing broadcast.

The claim follows: $|\Pr[H_{v,b}|E_{T-1}] - \Pr[H_{v,b}|E_0]| \geq |\frac{1}{2} - \text{negl}(n) - 1| \geq \frac{1}{2} - \text{negl}(n)$. \square

Theorem 11. This just follows from the previous claim. A simple hybrid argument shows that there exists a pair $(\tau^*, b) \in \{0, \dots, T-1\} \times \{0, 1\}$ such that

$$|\Pr[H_{v,b}|E_{\tau^*}] - \Pr[H_{v,b}|E_{\tau^*+1}]| \geq \frac{1}{2T} - \text{negl}(n)$$

for all P_v who do not have a neighbor delaying. Since T is polynomial, this is a non-negligible value. Without loss of generality, assume $\Pr[H_{v,b}|E_{\tau^*}] > \Pr[H_{v,b}|E_{\tau^*+1}]$. Leveraging this difference, we will construct an adversary \mathcal{A} that can win the IND-CDA game with non-negligible probability.

\mathcal{A} chooses two graphs G_0 and G_1 . $G = G_0$ and G_1 is G except parties 3, 4, and 5 are exchanged with parties $n-2$, $n-1$, and n respectively. \mathcal{A} corrupts the source part $P_S := P_1$, a left party $P_L := P_{n/2-1}$, a right party $P_R := P_{n/2+1}$, and the detective party $P_D := P_4$. See figure 3-3 for how this looks. The goal of \mathcal{A} will be to determine if P_D is to the left or right side of the network (close to the broadcaster or far).

\mathcal{A} computes the upper bound T using **DelayAlgorithm** and randomly guesses (τ^*, b) that satisfy the inequality above. At time τ , \mathcal{A} initiates a long-delay at party P_L , and at time $\tau+1$, \mathcal{A} initiates a long-delay at party P_R . So, \mathcal{A} gives the challenger \mathcal{T}_Z where $\mathcal{T}_Z((i, j), t) = 0$ for $t < \tau^*$, and for $t \geq \tau^*$: $\mathcal{T}_Z((L, n/2), t) = \mathcal{T}_Z((L, n/2 - 2), t)T$ and $\mathcal{T}_Z((R, n/2), t+1) = \mathcal{T}_Z((R, n/2 + 2), t+1) = T$.

Notice that news of P_L 's delay at time τ^* cannot reach P_R or any other party on the right side of the graph by time T . Also note that the time \mathcal{A} gets output for each of its corrupt parties is noted in the transcript.

If \mathcal{C} chooses G_0 , then P_D is on the left side of the graph and has probability $\Pr[H_{D,b}|E_{\tau^*}]$ of having the output bit by time T because its view is consistent with

P_L delaying at time τ^* . If \mathcal{C} chooses G_1 , then P_D is on the right side of the graph, and has a view consistent with the first long delay happening at time $\tau^* + 1$ and therefore has $\Pr[H_{D,b}|E_{\tau^*}]$ of having the output bit by time T . Because there is a noticeable difference in these probabilities, \mathcal{A} can distinguish between these two cases with $\frac{1}{2}$ plus some non-negligible probability. \square \square

Consequences of this lower bound. We note that this is just one model where we prove it is impossible for the adversary to control delays. However, we restrict the adversary a great deal, to the point of saying that regardless of what the natural network delays are, the adversary can learn something about the topology of the graph. The lower bound proved in this model seems to rule out any possible model (simulation or game-based) where the adversary has power over delays.

3.18 PKCR* Encryption

This section formally defines PKCR*—the extended Privately Key Commutative and Rerandomizable (PKCR) encryption of [4].

Let \mathcal{PK} , \mathcal{SK} and \mathcal{C} denote the public key, secret key and ciphertext spaces. In contrast to PKCR, the message space is $\{0, 1\}$. Moreover, $\mathcal{C} \cap \{0, 1\} = \emptyset$. As in any public-key encryption scheme, we have the algorithms $\text{PKCR}^*.\text{KGen} : \{0, 1\}^* \rightarrow \mathcal{PK} \times \mathcal{SK}$ and $\text{PKCR}^*.\text{Enc} : \{0, 1\} \times \mathcal{PK} \rightarrow \mathcal{C}$ for key generation and encryption, respectively (decryption can be implemented via deleting layers). Moreover, we require the following properties, where only the first two are provided (with minor differences) by PKCR.

Key-Commutative. \mathcal{PK} forms a commutative group under the operation \otimes . In particular, given any $\text{pk}_1, \text{pk}_2 \in \mathcal{PK}$ and the secret key sk_1 corresponding to pk_1 , we can efficiently compute $\text{pk}_3 = \text{pk}_1 \otimes \text{pk}_2 \in \mathcal{PK}$ (note that sk_1 can be replaced by sk_2 , since \mathcal{PK} is commutative).

This group must interact well with ciphertexts; there exists a pair of deterministic efficiently computable algorithms $\text{PKCR}^*.\text{AddLayer} : \mathcal{C} \times \mathcal{SK} \rightarrow \mathcal{C}$ and $\text{PKCR}^*.\text{DelLayer} : \mathcal{C} \times \mathcal{SK} \rightarrow \mathcal{C} \cup \{0, 1\}$ such that for every pair of public keys $\text{pk}_1, \text{pk}_2 \in \mathcal{PK}$ with corresponding secret keys sk_1 and sk_2 , for every bit $b \in \{0, 1\}$, and every ciphertext $c = \text{PKCR}^*.\text{Enc}(b, \text{pk}_1)$, with overwhelming probability it holds that:

- The ciphertext $\text{PKCR}^*.\text{AddLayer}(c, \text{sk}_2)$ is an encryption of b under the public key $\text{pk}_1 \otimes \text{pk}_2$.
- $\text{PKCR}^*.\text{DelLayer}(c, \text{sk}_2)$ is an encryption of b under the public key $\text{pk}_1 \otimes \text{pk}_2^{-1}$.
- $\text{PKCR}^*.\text{DelLayer}(c, \text{sk}_1) = b$.

Notice that we need the secret key to perform these operations.¹⁷

¹⁷In PKCR of [2], computing $\text{pk}_1 \otimes \text{pk}_2$ does not require the secret key. Moreover, PKCR requires perfect correctness.

Rerandomizable. There exists an efficient probabilistic algorithm $\text{PKCR}^*.\text{Rand} : \mathcal{C} \rightarrow \mathcal{C}$, which re-randomizes a ciphertext.¹⁸ Formally, we require that for every public key $\text{pk} \in \mathcal{PK}$, every bit b , and every $c = \text{PKCR}^*.\text{Enc}(b, \text{pk})$, the following distributions are computationally indistinguishable:

$$\{(b, c, \text{pk}, \text{PKCR}^*.\text{Enc}(b, \text{pk}))\} \approx \{(b, c, \text{pk}, \text{PKCR}^*.\text{Rand}(c, \text{pk}))\}$$

Transforming a 0-ciphertext to a 1-ciphertext. There exists an efficient algorithm $\text{PKCR}^*.\text{ToOne} : \mathcal{C} \rightarrow \mathcal{C}$, such that for every $\text{pk} \in \mathcal{PK}$ and for every $c = \text{PKCR}^*.\text{Enc}(0, \text{pk})$, the output of $\text{PKCR}^*.\text{ToOne}(c)$ is an encryption of 1 under pk .

Key anonymity. A ciphertext reveals no information about which public key was used in encryption. Formally, we require that PKCR^* is key-indistinguishable (or IK-CPA secure), as defined by Bellare et al. [11].

3.18.1 Construction of PKCR^* Based on DDH

We use a cyclic group $G = \langle g \rangle$. We keep as ciphertext a pair of group elements (c_1, c_2) . The first group element contains the message. The second group element contains the secret keys of each layer of encryption. All information is contained in the exponent.

To add a layer of encryption with a secret key sk , one simply raises the second element to sk . Similarly, one can remove layers of encryption. When all layers of encryption are removed, both group elements are either equal $c_1 = c_2$ (the message is 0) or $c_1 = c_2^2$ (the message is 1). To transform an encryption of 0 to an encryption of 1, one simply squares the first group element.

Algorithm PKCR^*

We let G be a group of order p , generated by g . These parameters are implicitly passed to all algorithms (formally, they are part of each ciphertext and an input to key generation).

$\text{PKCR}^*.\text{KGen}$

- 1: Sample the secret key sk uniform at random from \mathbb{Z}_p .
- 2: Output $(g^{\text{sk}}, \text{sk})$.

$\text{PKCR}^*.\text{Enc}(b, y)$

- 1: Sample r at random from \mathbb{Z}_p .
- 2: Output $c = (g^{(b+1)r}, y^r)$.

$\text{PKCR}^*.\text{AddLayer}((c_1, c_2), \text{sk})$

- 1: Output (c_1, c_2^{sk}) .

$\text{PKCR}^*.\text{Rand}((c_1, c_2))$

- 1: Sample r at random from \mathbb{Z}_p .
- 2: Output (c_1^r, c_2^r) .

$\text{PKCR}^*.\text{DelLayer}((c_1, c_2), \text{sk})$

- 1: Set $c'_2 = c_2^{\text{sk}^{-1}}$.
- 2: **if** $c_1 = c'_2$ **then** Output 0.
- 3: **else if** $c_1 = c'^2_2$ **then** Output 1.
- 4: **else** Output (c_1, c'_2) .

$\text{PKCR}^*.\text{ToOne}((c_1, c_2))$

- 1: Output (c_1^2, c_2) .

¹⁸In [2] the rerandomization algorithm is given the public key as input. We also note that they require public keys to be re-randomizable, while we do not need this property.

Security. Semantic security and KI-CPA security of our scheme follow from the respective properties of the ElGamal encryption (for the proof of KI-CPA security, see [11]). Further, the proof that it satisfies the requirements of rerandomizability and key commutativity is analogous to the proof that the DDH-based construction of [2] satisfies these properties. We refer to [2] for details.

It remains to prove the correctness of $\text{PKCR}^*.\text{ToOne}$ and $\text{PKCR}^*.\text{DelLayer}$. The former follows trivially from inspection of the protocol.

For the latter, we need to show that the probability of $\text{PKCR}^*.\text{DelLayer}$ giving the wrong output (either from the wrong domain, or the incorrect decryption) is negligible. Observe that, by correctness of the ElGamal cryptosystem, whenever $\text{PKCR}^*.\text{DelLayer}$ should output a bit, it indeed outputs the correct value. Now, for a fixed secret key sk , and a public key pk , consider the probability of $\text{PKCR}^*.\text{DelLayer}$ outputting a bit when it should output a ciphertext. This event happens only when $c_1 = c_2^{\text{sk}^{-1}}$ or $c_1 = c_2^{2\text{sk}^{-1}}$, which happens with probability $2/p$.

3.19 Proof of Theorem 8

Simulator. We simulate the outputs of \mathcal{F}_{NET} on inputs $(\text{FETCHMESSAGES}, i)$ from the corrupted parties (note that everything else can be simulated trivially). The messages sent by the corrupted parties can be easily generated by executing the protocol. Hence, the challenge is to generate the messages sent by honest parties to their corrupted neighbors.

We first deal with the problem of outputting the messages at correct times. That is, the simulator generates all messages upfront. The messages are then stored in `buffer`, and the simulator outputs them by executing the algorithm of \mathcal{F}_{NET} .

What remains is to show how to compute the messages. This will be done per a corrupted arc of the cycle. Observe that a sequence of corrupted parties can fast-forward the protocol and learn the output before the protocol terminates. Concretely, consider an honest party neighboring the corrupted arc. Right before the end of the protocol, it sends messages, that can be read by its direct corrupted neighbor. Before that, it sends messages, that can be read by its colluding two-neighborhood. This continues until time t , before which the messages carry the output for a party outside of the corrupted arc. The messages sent before time t are computed as encryptions of 0 under a fresh public key, since the corrupted arc cannot decrypt these messages. The messages sent after t are encryptions of the output bit.

Finally, we need a way to compute the time t , after which the messages sent by an honest party carry output for a party in the corrupted arc. As noted in Section 3.15, the protocol has a *deterministic* end time of $T = R(n\kappa + \text{MedRSum}[\text{D}])$. Consider a single corrupted arc P_1, \dots, P_k (all corrupted arcs can be handled independently since there is at least one honest party between them). A message sent from P_k of that arc to an honest node can be read by the corrupted arc when it reaches P_1 of the arc. Since the corrupted arc knows the waiting time for its parties (w_1, \dots, w_k) , the simulator also knows these values, and so the time at which the message is revealed

to the arc is T minus the time it would have taken for that message to traverse from P_1 to P_k : $T - \sum_{i=1}^k \mathbf{w}_i$. This is how we compute t .

Simulator \mathcal{S}_{cycle}

1. \mathcal{S}_{cycle} corrupts the parties in the set \mathcal{Z} .
2. \mathcal{S}_{cycle} sends inputs for all parties in \mathcal{Z} to \mathcal{F}_{BC} and receives the output bit b^{out} .
3. It sends $(\text{GETINFO}, R)$ to $\mathcal{F}_{INFO}^{\mathcal{L}_{median}}$ and receives the neighborhoods of corrupted parties.
4. Now \mathcal{S}_{cycle} has to simulate the view of all parties in \mathcal{Z} . The messages sent by corrupted parties can be easily generated by executing the protocol **CycleProt**. To simulate the messages sent by honest parties to their corrupted neighbors, \mathcal{S}_{cycle} proceeds as follows.
5. First, it prepares a set **buffer**, containing all messages which will be sent by the honest parties throughout the simulation (recall the variable **buffer** in \mathcal{F}_{NET}). \mathcal{S}_{cycle} initializes **buffer** = \emptyset .
6. \mathcal{S}_{cycle} generates the messages per a connected corrupted arc P^1, P^2, \dots, P^K of the cycle. We will use the following notation:

- P^0 and P^{K+1} : the neighboring honest parties.
- P^{K+2}, \dots, P^{n-1} : (the labels of) the rest of the parties on the cycle (their identities are unknown to \mathcal{S}_{cycle}).
- $\text{MedRSum}[D]$: the distribution corresponding to the median-round-sum of all delays, obtained from $\mathcal{F}_{INFO}^{\mathcal{L}_{median}}$.
- for $0 \leq k \leq n-1$, denote by D_k the delay distribution on the edge from $P^{(k-1) \bmod n}$ to P^k (for $1 \leq k \leq K$, D_k was obtained from $\mathcal{F}_{INFO}^{\mathcal{L}_{median}}$).
- for $1 \leq k \leq K$, define $\mathbf{w}^k = \kappa + \lceil \text{Med}[D_{(k,k+1)}] / R \rceil$ (recall the initialization step of **CycleProt**).
- for $1 \leq k \leq K$, denote by \mathbf{pk}^k the public key corresponding to the secret keys of the corrupted parties P^1, \dots, P^k .
- \mathbf{pk}_{sim} : a public key freshly sampled by \mathcal{S}_{cycle} at the beginning of the simulation.

\mathcal{S}_{cycle} has to compute the messages sent by P^0 to P^1 and by P^{K+1} to P^K . To compute the former messages, it does as follows (the latter messages are computed analogously):

- 1: For $1 \leq k \leq K$, compute the time after which P^0 starts processing messages from the walk started by P^k as $t^k = R(n\kappa + \text{MedRSum}[D] - (\mathbf{w}^1 + \dots + \mathbf{w}^k))$.
- 2: Let $t^0 = R(n\kappa + \text{MedRSum}[D])$.
- 3: **for** $\tau = t^0$ to 0 and $\tau \equiv 0 \pmod R$ **do**
- 4: **if** $\tau < t^K$ **then**
- 5: Compute $c = \text{PKCR}^*. \text{Enc}(0, \mathbf{pk}_{sim})$.

```

6:   else
7:     Find  $k$  such that  $t^{k+1} \leq \tau < t^k$ .
8:     Compute  $c = \text{PKCR}^*. \text{Enc}(b^{\text{out}}, \text{pk}^k)$ .
9:     for  $i = 0$  to  $\kappa - 1$  do
10:      Sample  $d$  from  $D_0$ .
11:      Record the tuple  $(\tau + iR + d, P^0, P^1, (\tau/R + i, c))$  in buffer.

```

7. $\mathcal{S}_{\text{cycle}}$ simulates the messages received by corrupted parties from \mathcal{F}_{NET} by executing the algorithm of \mathcal{F}_{NET} . On every input (`FETCHMESSAGES`, j) from a corrupted P_j , it gets the current time τ from $\mathcal{F}_{\text{CLOCK}}$. Then, for each message tuple (t, P_i, P_j, c) from **buffer** where $t \leq \tau$, it removes the tuple from **buffer** and outputs (i, c) to P_j .

We first prove a fact about the protocol **CycleProt**: with overwhelming probability one of the κ copies of a message generated by P_i for P^ℓ in a given round is delivered within w^ℓ rounds.

Lemma 8. *In the real execution of the protocol **CycleProt**, the probability that none of the κ messages (r_c, c) sent by P_i to P^ℓ for round r_c is delivered by round $r_c + \kappa + \lceil \text{Med}[D_{(i,\ell)}]/R \rceil$ is negligible.*

Proof. For the distribution $D_{(i,\ell)}$ on the edge between P_i and P^ℓ and for $1 \leq j \leq \kappa$, let X_j be the indicator variable that message (r_c, c) arrived after time $T = R(r_c + \kappa + \lceil \text{Med}[D_{(i,\ell)}]/R \rceil)$. Since the message was sent at time $t_{\text{sent}}^j = R(r_c + j)$, with probability $1/2$ the message arrives at P_i by time $t_{\text{sent}}^j + \text{Med}[D_{(i,\ell)}]$, and is officially delivered at the next round, time $t_{\text{sent}}^j + \lceil \text{Med}[D_{(i,\ell)}]/R \rceil \cdot R$. Note that for every j , time T is greater than or equal to $t_{\text{sent}}^j + \lceil \text{Med}[D_{(i,\ell)}]/R \rceil \cdot R$. Therefore, for every j , we have

$$\Pr_{D_{(i,\ell)}} [X_j = 1] \geq \frac{1}{2}.$$

Finally, given independence between messages sent at different rounds, the probability that all messages arrive *after* time T is upper bounded by

$$\Pr_{D_{(i,\ell)}} \left[\sum_{j=1}^{\kappa} X_j = 0 \right] = \prod_{j=1}^{\kappa} \Pr_{D_{(i,\ell)}} [X_j \neq 1] \leq \frac{1}{2^\kappa} = \text{negl}(\kappa).$$

As a consequence, a message sent at round r_c arrives with all but negligible probability at round $r_c + (\kappa + \lceil \text{Med}[D_{(i,\ell)}]/R \rceil)$. ■ □

We are now ready to prove that the messages from the walks initiated by P^k are acknowledged and processed by P^0 after exactly t^k clock ticks. This also proves correctness of the protocol: P^0 will get his walk back after t^0 clock ticks.

Lemma 9. *In the real execution of the protocol, at time t^k , the party P^0 starts sending to P^1 a message from the walk started by P^k .*

Proof. We assume that for each P^i , one of the copies of a message generated by P^{i-1} in a given round arrives within w^i rounds. By Lemma 8, this happens with overwhelming probability.

Consider each message-walk started by party P^k going to P^{k+1} (we ignore the repetitions and count the number of distinct messages sent). There are w^k such walks, started at rounds $0, \dots, w^k - 1$. A walk started at a given round is processed by the party $P^{k+k'}$ after $\sum_{i=1}^{k'} w^{k+i}$ rounds. So it is processed by P^0 after $\sum_{i=0}^{n-1} w^i - (w^1 + \dots + w^k)$ rounds. We have $\sum_{i=0}^{n-1} w^i = \text{MedRSum}[D]$. Hence, P^0 processes messages from the walk started by P^k after time $R \cdot (\text{MedRSum}[D] - \sum_{i=1}^k w^i) = t^k$. ■ □

Remark 3. *The protocol is correct. Note that Lemma 9 implies the correctness of the protocol. A party “sends” a message first by processing it (and seeing if it can decrypt) and then forwarding it if it did not decrypt. Without loss of generality, consider party P_1 . The walk started from P_1 arrives back at P_1 at time t^n , at which point, $n - 1$ layers of encryption will have been removed, and the message is guaranteed to have passed the source party (and thus have the output bit). P_1 decrypts this message and gets the output from the protocol.*

Finally, we show that the execution with $\mathcal{S}_{\text{cycle}}$ is indistinguishable from the real execution by presenting a sequence of hybrids. In the following, we only consider the messages sent by an honest $P_i = P^0$ to its corrupted neighbor $P_j = P^1$ (all other messages are trivial to simulate).

Hybrid 1. $\mathcal{S}_{\text{cycle}}^1$ simulates the real world exactly. That is, $\mathcal{S}_{\text{cycle}}^1$ has information on the entire communication graph and all edge delays. It generates messages according to the protocol, at the time they are sent.

Hybrid 2. $\mathcal{S}_{\text{cycle}}^2$ generates all messages upfront the same way $\mathcal{S}_{\text{cycle}}$ does, but the messages are still generated according to the protocol.

Hybrid 3. $\mathcal{S}_{\text{cycle}}^3$ replaces the real ciphertexts $\text{PKCR}^*. \text{DelLayer}(c, sk_i)$ sent by P^0 by fresh encryptions $\text{PKCR}^*. \text{Enc}(m, pk')$ (where m is the message c encrypts, and pk' is the public key corresponding to the secret keys of the parties remaining on the cycle).

Hybrid 4. For messages generated at time $\tau < t^K$, $\mathcal{S}_{\text{cycle}}^4$ changes the encryption key to pk_{sim} , that is, it computes $\text{PKCR}^*. \text{Enc}(m, pk_{\text{sim}})$ instead of $\text{PKCR}^*. \text{Enc}(m, pk')$. For messages generated at time $\tau \geq t^K$, $\mathcal{S}_{\text{cycle}}^4$ uses fresh encryptions under the public key $\text{PKCR}^*. \text{Enc}(m, pk^k)$, where k is chosen as in $\mathcal{S}_{\text{cycle}}$.

Hybrid 5. For messages generated at time $\tau < t^K$, $\mathcal{S}_{\text{cycle}}^5$ changes the message to 0, i.e., it computes $\text{PKCR}^*. \text{Enc}(0, pk_{\text{sim}})$ instead of $\text{PKCR}^*. \text{Enc}(m, pk_{\text{sim}})$.

Hybrid 6. For messages generated at time $\tau \geq t^K$, $\mathcal{S}_{\text{cycle}}^6$ computes the encrypted message using the output of \mathcal{F}_{BC} .

Observe that Hybrids 1 and 2 are trivially identical. Moreover, indistinguishability of Hybrids 5 and 6 follows from the correctness of the protocol, allowing the adversary to decrypt not the bit generated by traversing the graph, but the bit generated by the ideal functionality. These two will be equivalent since each message traverses the entire graph. It is also easy to see that \mathcal{S}_{cycle}^6 is the original simulator \mathcal{S}_{cycle} .

Claim 8. *No efficient distinguisher can distinguish between Hybrids 2 and 3.*

Proof. For an honest party P_i , \mathcal{S}_{cycle}^3 generates all messages sent by it as fresh encryptions, while a message generated by \mathcal{S}_{cycle}^2 can be one of the following:

- An initial ciphertext (starting the cycle): this is the same as in \mathcal{S}_{cycle}^3 .
- A ciphertext c , which results from applying to another ciphertext c' , in order, PKCR*.DelLayer, and PKCR*.Rand. By correctness, c is an encryption of the same message as c' with overwhelming probability. Moreover, re-randomizability of PKCR*, guarantees that the distribution of c is indistinguishable from the distribution of a fresh encryption of the same message, as generated by \mathcal{S}_{cycle}^3 .
- A ciphertext c , which results from applying to another ciphertext c' , in order, PKCR*.DelLayer, PKCR*.ToOne, and PKCR*.Rand. As in the previous case, c is an encryption of one with overwhelming probability, and the distribution of c is indistinguishable from the distribution of a fresh encryption of 1.

■

Claim 9. *No efficient distinguisher can distinguish between Hybrids 3 and 4.*

Proof. We will prove first that messages sent after time t^K are indistinguishable between the two hybrids, and then show that before time t^K , they are also indistinguishable.

After t^K : Consider each message walk started by corrupted party P^k going to P^{k+1} . By Lemma 9, by time t^k , in the real world, the messages from that walk would have traversed all parties except P_1, \dots, P_{k-1} . So, by that time, all key-layers except those from corrupted parties P_1, \dots, P_{k-1} will have been removed, meaning that message is now encrypted under public key \mathbf{pk}^k . Hybrids 3 and 4 are equivalent in this case, then, because we are actually encrypting under the same key as one would encrypt in the real world.

Before t^K : By Lemma 9, the messages sent by P^0 before t^K are encrypted under the public key, for which the honest party P^{K+1} holds a part of the secret key. So, the proof for time before t^K is a reduction to a version of KI-CPA security of PKCR*, where the adversary is allowed to ask many encryption queries. We note that [11] defines only the game for one query, but the reduction follows by a standard hybrid argument. For completeness, we recall their game against an adversary \mathcal{A} :

Game IK-CPA

- 1: $\mathbf{pk}_0, \mathbf{sk}_0 = \text{PKCR}^*. \text{KGen}(1^\kappa)$
- 2: $\mathbf{pk}_1, \mathbf{sk}_1 = \text{PKCR}^*. \text{KGen}(1^\kappa)$
- 3: Choose $b \in \{0, 1\}$ at random.
- 4: $b' = \mathcal{A}^{\text{PKCR}^*. \text{Enc}(\cdot, \mathbf{pk}_b)}(\mathbf{pk}_0, \mathbf{pk}_1)$
- 5: Output $b = b'$.

Recall that, for each honest party P_i , $\mathcal{S}_{\text{cycle}}^3$ uses the real key corresponding to a number of parties on the cycle, while $\mathcal{S}_{\text{cycle}}^4$ uses a different key \mathbf{pk}_{sim} . We will introduce a sequence of intermediate hybrids H_1 to H_n , where H_i uses the real key for P_1, \dots, P_i , and the simulated key for the other parties (in case they are honest and have corrupted neighbors).

Assume that \mathcal{D} is a distinguisher for Hybrids H_{i-1} and H_i . A winner \mathcal{A} for the above game can be constructed as follows. Let $\mathbf{pk}_0, \mathbf{pk}_1$ be the keys obtained from the game. If \mathcal{D} corrupts P_i or it does not corrupt any of its neighbors, we abort (the hybrids are trivially indistinguishable). Otherwise, let P^1, \dots, P^K be the corrupted arc starting at P_i 's neighbor P^1 . \mathcal{A} will simulate the protocol for \mathcal{D} using freshly generated key pairs, except that for P^{K+1} it will use its challenge key \mathbf{pk}_0 . Note that it can now efficiently compute the joint key. Ciphertexts sent by the parties can now be generated using \mathcal{A} 's encryption oracle. \mathcal{A} outputs whatever \mathcal{D} outputs. ■

Claim 10. *No efficient distinguisher can distinguish between Hybrids 4 and 5.*

Proof. In both Hybrid 4 and Hybrid 5, the messages sent before D_k are encrypted under a fresh public key \mathbf{pk}_{sim} chosen by the simulator. Hence, the indistinguishability follows by semantic security. ■

3.20 Details of the Protocol for Trees

Protocol TreeProt

// The common input of all parties is the round length R . Additionally, the sender P_s has the input bit b_s .

Setup: For $i \in \{1, \dots, n\}$, let $(\mathbf{pk}_i, \mathbf{sk}_i) = \text{PKCR}^*. \text{KGen}(1^\kappa)$. Let $\mathbf{pk} = \mathbf{pk}_1 \otimes \dots \otimes \mathbf{pk}_n$.

The setup outputs to each party P_i its secret key \mathbf{sk}_i and the product public key \mathbf{pk} .

Initialization for each P_i :

- Send (GETINFO) to the functionality \mathcal{F}_{NET} to obtain the distributions of the local edges.
- Assign randomly the labels $P^0, \dots, P^{\nu-1}$ to its neighbors. Let $\text{succ}(\ell) = \ell + 1 \bmod \nu$ denote the index of the successor of neighbor P^ℓ on the tree traversal.
- For each neighbor P^ℓ , initialize a list $\text{Rec}^\ell = \perp$ and a set $\text{Send}^\ell = \emptyset$.
- For each ℓ , $D_{(i,\ell)}$ is the delay distribution on the edge between P_i and neighbor

P^ℓ , obtained from $\mathcal{F}_{\text{INFO}}$.

- Let $w^\ell = \kappa + \lceil \text{Med}[D_{(i,\ell)}]/R \rceil$ be the time P_i waits before sending a message from P^ℓ to $P^{\ell+1}$.

Execution for each P_i :

- 1: Send (READCLOCK) to the functionality $\mathcal{F}_{\text{CLOCK}}$ and let τ be the output. If $\tau \bmod R \neq 0$, send (READY) to the functionality $\mathcal{F}_{\text{CLOCK}}$. Otherwise, let $r = \tau/R$ be the current round number and do the following:
- 2: Receive messages: Send (FETCHMESSAGES, i) to the functionality \mathcal{F}_{NET} . For each message (r_c, c) received from a neighbor P^ℓ , set $\text{Rec}^\ell[r_c + w^\ell] = c$.
- 3: Process messages from P^ℓ with $\ell \neq 0$:
 - For each $P^\ell \neq P^0$ such that $\text{Rec}^\ell[r] = \perp$, add $(\kappa, r, \text{PKCR}^*. \text{Enc}(0, pk))$ to $\text{Send}^{\text{succ}(\ell)}$.
 - For each $P^\ell \neq P^0$ such that $\text{Rec}^\ell[r] \neq \perp$, add $(\kappa, r, \text{PKCR}^*. \text{Rand}(\text{Rec}^\ell[r]))$ to $\text{Send}^{\text{succ}(\ell)}$.
- 4: Process if no messages received from P^0 : If $\text{Rec}^0[r] = \perp$, start a new cycle traversal in the direction of P^1 :
 - If P_i is sender (i.e. $i = s$) then add $(\kappa, r, \text{PKCR}^*. \text{Enc}(b_s, pk))$ to Send^1 .
 - Otherwise, add $(\kappa, r, \text{PKCR}^*. \text{Enc}(0, pk))$ to Send^1 .
- 5: Process received messages from P^0 : Set $d = \text{PKCR}^*. \text{DelLayer}(\text{Rec}^0[r], sk_i)$, and do the following:
 - If $d \in \{0, 1\}$, output d and halt (we have decrypted the source bit).
 - Otherwise, if $i = s$ and $b_s = 1$, then set $d = \text{PKCR}^*. \text{ToOne}(d)$. Then, in either case, add $(\kappa, r, \text{PKCR}^*. \text{Rand}(d))$ to Send^1 .
- 6: Send messages: For each $\ell \in \{0, \dots, \nu - 1\}$, let $\text{Sending}^\ell = \{(k, r_c, c) \in \text{Send}^\ell : k > 0\}$. For each $(k, r_c, c) \in \text{Sending}^\ell$, send (r_c, c) to P^ℓ .
- 7: Update Send sets: For each $(k, r_c, c) \in \text{Sending}^\ell$, remove (k, r_c, c) from Send^ℓ and insert $(k - 1, r_c, c)$ to Send^ℓ .
- 8: Send (READY) to the functionality $\mathcal{F}_{\text{CLOCK}}$.

3.21 The Function Executed by the Hardware Boxes

The functionality \mathcal{F}_{HW} contains hard-wired the following values: a symmetric encryption key pk , and a key rk for a pseudo-random function prf . Whenever it outputs an encryption, it uses an authenticated encryption scheme AE with key pk , and with encryption randomness computed as $\text{prf}_{rk}(x)$, where x is the whole input of \mathcal{F}_{HW} . \mathcal{F}_{HW} can receive three types of input, depending on the current stage of the protocol: the initial input and an intermediate input during *Hw-Preprocessing*, and an intermediate input during *Hw-Computation*. On any other inputs, \mathcal{F}_{HW} outputs \perp .

Behavior during preprocessing. During the preprocessing, the first input is a

triple $(i, \mathbf{id}_i, \mathbf{N}_G(P_i))$, and next inputs are triples (\mathbf{id}, c, c_j) , where c and c_j are states of parties, encrypted under pk . In particular, the state of a party P_i consists of the following information:

- i : the index of P_i ,
- G : the current image of the graph (stored in an n -by- n matrix),
- $ID = (\mathbf{id}_1, \dots, \mathbf{id}_n)$: a vector, containing the currently known identifiers of parties.

On the first input, \mathcal{F}_{HW} outputs an encryption of the initial state, that is, the state where the graph G contains only the direct neighborhood of P_i , and ID contains only the value \mathbf{id}_i chosen by P_i . For the inputs of the form (\mathbf{id}, c, c_j) , \mathcal{F}_{HW} decrypts the states c and c_j and merges the information they contain into a new state s , which it then encrypts and outputs.

Behavior during computation. Recall that the goal of \mathcal{F}_{HW} at this stage is to compute the next encrypted messages, which a party P_i will send to its neighbors. That is, it takes as input a set of encrypted messages received by P_i and, for each neighbor of P_i , outputs a set of n messages to be sent.

Each encrypted message contains information about which graph traversal it is a part of, about the current progress of the traversal, and about all the inputs collected so far. Moreover, we include the information from the encrypted state: (i, G, ID) and the function f of the party starting the cycle. Intuitively, the reason for including f and the encrypted state is that, since the adversary is passive, the information taken from the message must be correct (for example, now a corrupted party cannot use its box to evaluate any function of its choice). Formally, an encrypted message from another node decrypts to a message m_j containing the following elements:

- j is the party number (the publicly known number between 1 and n , not the party's \mathbf{id})
- ID_j is the vector of unique random \mathbf{id} 's. Carrying this in the message allows us to ensure that inputs are all consistent with the same parties.
- G_j is the adjacency matrix of the network graph. It is also used to check consistency.
- $Path_j = (\mathbf{id}^1, \dots, \mathbf{id}^{4n^2})$: a vector of length $4n^2$, containing the current set of identifiers of parties visited so far along the graph traversal starting at P_j (recall that the eulerian cycle of length at most $2n^2$ is traversed twice).
- f_j is the function that parties will compute.
- \mathbf{x}_j is a vector that has a slot for every party to put its input. It starts as being completely empty, but gains an entry when it visits a new node on the graph. We also check this for consistency (a party trying to input a different value from the one they started with will not be able to use the hardware).

At a high level, \mathcal{F}_{HW} first discards any dummy or repeated messages (a party can receive many messages, but the hardware box needs to continue at most n Eulerian cycles), and then processes each remaining message. If a message has traversed the whole Eulerian cycle, \mathcal{F}_{HW} computes and reveals the function applied to the inputs. Otherwise, it creates an encryption of a new message with the current party's id added to the current path, and its input added to the list of inputs, and **next** contains the id of the destination neighbor. After processing all messages, for each destination neighbor, it adds correctly formatted dummy encryptions, so that exactly n encryptions are sent to each neighbor.

The functionality \mathcal{F}_{HW} is formally described below. It calls the following subroutines:

- **AggregateTours** takes as input a set of messages M . Each of these messages contain information about a Eulerian Cycle, the party that started that Eulerian Cycle, and the path traversed so far. The subroutine selects the (at most n) messages that start from different parties. It is expected that Eulerian Cycles starting from the same party, are exactly the same message.
- **ContinueTour** takes as input a specific message, a Eulerian Cycle that the message must traverse, and a current party's input and number. If the Eulerian Cycle has not been traversed, it then creates a new message containing a path with the current party's input and id appended to the corresponding variables, and also the id of the party where the message should be sent. Otherwise, it outputs a flag indicating that the Eulerian Cycle has ended and the output must be revealed.
- **EncryptAndFormatOutput** takes as input a set of pairs message-destination, and appends to each possible destination parsable messages until there are n messages. It then encrypts each message and outputs, for each possible destination a set of encryptions and the id of the party where the encryptions must be sent.

Functionality \mathcal{F}_{HW}

Setup: The hardware box is initialized with a symmetric encryption key pk and a PRF key rk .

Initial input during Hw-Preprocessing

Input: $x = (i, id_i, N_G(P_i))$

- 1: Compute the initial vector ID as a vector of n \perp 's except with id_i in the i -th position.
- 2: Compute a new adjacency matrix G_i with the only entries being the local neighborhood of P_i .
- 3: Compute the initial state $s = (i, ID, G_i)$

Output: the encrypted initial state $AE.Enc_{pk}(s; prf_{rk}(x))$.

Intermediate input during Hw-Preprocessing

Input: $x = (id, c, c_j)$, where id is the identifier of P_i , c is the encrypted state of P_i , and c_j is the state of a neighbor P_j .

- 1: Compute the states $(i, ID, G) = AE.Dec_{pk}(c)$ and $(j, ID_j, G_j) = AE.Dec_{pk}(c_j)$.
- 2: Compute the new state $s = (i, ID', G')$, where ID' contains all identifiers which appear in ID_j or ID , and G' is the union of G and G_j .

Output: the encrypted state $AE.Enc_{pk}(s; prf_{rk}(x))$.

Intermediate input during Hw-Computation

Input: $x = (i, id, c, E, x_i, f_i, r)$, where i is the party's index, id is the identifier of P_i , c is the encrypted state of P_i , E is the set of encrypted messages (freshly gotten from the buffer), x_i is the input, f_i is the evaluated function and r is a fresh random value.

- 1: Decrypt the messages $M = \{AE.Dec_{pk}(e) \mid e \in E\}$ (output \perp if any decryption fails).
- 2: Let $L = \text{AggregateTours}(M)$, and output \perp if AggregateTours outputs \perp .
- 3: Let $S = \emptyset$, $val = \perp$.
- 4: **if** $L = \emptyset$ **then** // Start the traversal.
- 5: Decrypt the state $(i, ID, G) = AE.Dec_{pk}(c)$ (output \perp if the decryption fails). // The graph and the ID-vector are taken from the encrypted state.
- 6: Let $Path = (id, \perp, \dots, \perp)$ be a vector of length $4n^2$. Let \mathbf{x} be the vector of length n , initialized to \perp and set $\mathbf{x}[i] = x_i$.
- 7: Compute $Tour_i$ as the reverse Euler Cycle for G starting at party P_i .
- 8: Let $m = (i, ID, G, Path, f_i, \mathbf{x})$.
- 9: Add $(m, Tour_i[2])$ to S .
- 10: **else** // Continue traversals.
- 11: **for** $m \in L$ **do**
- 12: Parse $m = (j, ID_j, G_j, Path_j, f_j, \mathbf{x}_j)$. // The graph and the ID-vector are taken from the message.
- 13: Compute $Tour_j$ as the reverse Euler Cycle for G starting at party P_j .
- 14: Parse $Path_j = (p_1, \dots, p_{\ell_j}, \perp, \dots, \perp)$. Output \perp if any of the following conditions holds:
 - $id \neq ID_j[i]$
 - $p_{\ell_j} \neq i$
 - for any $l \in [\ell_j]$, $p_l \neq Tour_j[l \bmod 2m]$
- 15: Let $(m', next) = \text{ContinueTour}(m, x_i, i, Tour_j)$.
- 16: **if** $m' = \text{Output}$ **then**
- 17: Let $val = f_i(\mathbf{x}_j)$.
- 18: **else**
- 19: Add $(m', next)$ to S .
- 20: **Output** : $(val, \text{EncryptAndFormatOutput}(i, G, r, S, 0))$

Functionality \mathcal{F}_{HW} -subroutines

AggregateTours (M)

// Takes a set of messages and for each party outputs a message that corresponds to its Euler Cycles.

- 1: If any $m \in M$ does not parse properly, return \perp .
- 2: Let $L = \emptyset$.
- 3: **for each** $m \in M$ **do**
- 4: Parse $m = (j, ID, G, Path, f, \mathbf{x})$.
- 5: **if** $\exists m' := (j, *, *, *, *, *) \in L$ and $m' \neq m$ **then**
- 6: Output \perp .
- 7: **if** $m \notin L$ **then**
- 8: Add m to L .
- 9: **return** L

ContinueTour ($m_j, x_i, i, Tour_j$)

- 1: Parse $m_j = (j, ID_j, G_j, Path_j, f_j, \mathbf{x}_j)$.
- 2: Parse $Path_j = (p_1, \dots, p_{\ell_j}, \perp, \dots, \perp)$.
- 3: **if** $\ell_j = 4m - 1$ and $Tour_j[(\ell_j + 1) \bmod 2m] = i$ **then**
- 4: **return** ($Output, 0$).
- 5: Set $Path_j = (p_1, \dots, p_{\ell_j}, Tour_j[(\ell_j + 1) \bmod 2m], \perp, \dots, \perp)$.
- 6: If $\mathbf{x}_j[i] = \perp$, then set $\mathbf{x}_j[i] = x_i$.
- 7: **return** ($m_j, Tour_j[(\ell_j + 1) \bmod 2m]$).

EncryptAndFormatOutput (i, G, r, S, sim)¹⁹

- 1: For each $d \in \mathbf{N}_G(i)$, let $M_d = \{m : (m, d) \in S\}$.
- 2: **for** $d \in \mathbf{N}_G(i)$ **do**
- 3: If $|M_d| < n$, pad M_d with fake, but parsable, messages until it is length n (messages that start with the party number being 0).
- 4: **for** $d \in \mathbf{N}_G(i)$ **do**
- 5: Let $k = 0$, $E_d = \emptyset$.
- 6: **for** $m \in M_d$ **do**
- 7: **if** $\text{sim} = 0$ **then**
- 8: Add $\text{AE.Enc}_{pk}(m; \text{prf}_{rk}(M_d, k, r))$ to E_d . // Used in protocol
- 9: **else**
- 10: Add $\text{AE.Enc}_{pk}(m; r)$ to E_d . // Used in simulator
- 11: **return** $\{(E_d, d) : d \in \mathbf{N}_G(i)\}$

3.22 Proof of Theorem 10

Proof. Simulator. The simulator has to simulate the view of all corrupted parties. It knows the neighborhood of corrupted parties, its delay distributions and its clock

¹⁹The additional input $\text{sim} \in \{0, 1\}$ will be used by the simulator and can be ignored at this point.

rates. The view of the corrupted parties in the real world, consist of messages received from the network functionality \mathcal{F}_{NET} , and messages received from the hardware functionality \mathcal{F}_{HW} .

Consider a corrupted component C and the subgraph G_C , which contains C and the honest parties in the immediate neighborhood of C (but not the edges between honest parties). Observe that \mathcal{S}_{HW} has complete knowledge of the topology of G_C .

To simulate the preprocessing phase, we do as follows: Each honest party in G_C starts with a state where its only neighbors are in G_C , and the simulated hardware box answers the queries exactly the same way as the real hardware box \mathcal{F}_{HW} . As a result, at the end of the preprocessing phase, all parties in G_C have an encrypted state containing the graph G_C .

In the computation phase, the simulator generates all local delays upfront. It then computes, for each corrupted party P_j in G_C , the number of traversals it initiates (recall that the party initiates a traversal every round, until the first message is received). For each of these traversals, it computes the last honest party P_i in G_C , before the traversal enters G_C (by executing the algorithm `EulerianCycle` on G_C), the next (corrupted) party P_k on the traversal, and samples and records the time at which the message arrives to P_i (corresponding to four times the total rounded delay minus the sum of rounded delays of the last fragment of corrupted parties in G_C). Then, the simulator checks at every round whether he has to send to a neighbor P_k a message containing the output of any corrupted party $P_j \in G_C$. It then sends the corresponding encryptions containing outputs, and appends encryptions so that every round, P_i sends n encryptions to each (corrupted) neighbor in G_C . The simulated hardware messages are as in the real protocol, except that the vector of inputs is 0, and, once the traversal is completed, it gives directly the output (ignoring the inputs). Moreover, it generates truly random values instead of using a PRF.

Simulator \mathcal{S}_{HW}

1. \mathcal{S}_{HW} corrupts \mathcal{Z} .
2. \mathcal{S}_{HW} sends inputs for all parties in \mathcal{Z} to \mathcal{F}_{BC} and for each party $P_i \in \mathcal{Z}$ receives the output value v_i^{out} .
3. Let R be the round length. \mathcal{S}_{HW} receives from $\mathcal{F}_{\text{INFO}}^{\mathcal{L}_{\text{sum}}}$ the distribution $D = \sum_e [D_e/R]R$ and, for each $P_i \in \mathcal{Z}$, the neighborhood $\mathbf{N}_G(P_i)$ and its delay distributions.
4. \mathcal{S}_{HW} generates an authenticated encryption key ek .
5. Now, \mathcal{S}_{HW} has to simulate the view of all parties in \mathcal{Z} . The view of corrupted parties consist of messages received via the network \mathcal{F}_{NET} and the queries to the hardware functionality \mathcal{F}_{HW} .

Network messages: The messages sent by corrupted parties can be easily generated by executing the protocol `Hardware`. To simulate the messages sent by an honest party to the corrupted neighbors, \mathcal{S}_{HW} proceeds as follows.

First, it prepares a set **buffer**, containing all messages which will be sent by the honest parties to corrupted neighbors throughout the simulation (recall the variable **buffer** in \mathcal{F}_{NET}).

\mathcal{S}_{HW} sets **buffer** = \emptyset . We simulate the messages per corrupted connected component. Let $G_C = (V_C, E_C) \subset G$ be a corrupted connected component including the honest parties in its immediate neighborhood. Then, \mathcal{S}_{HW} does the following:

Preprocessing: Let P_j be a corrupted party in the component, who has an honest neighbor P_i .

- 1: Let t_{prep} be the time when the preprocessing finishing signal is received by P_i .
- 2: Choose a random value id_i and compute the initial vector **ID** as a vector of n \perp 's with id_i in the i -th position. Let $s_0 = (i, \text{ID}, \mathbf{N}_{G_C}(P_i))$.
- 3: **for** $\tau \in \{0, R, 2R, \dots, \lfloor t_{\text{prep}}/R \rfloor R\}$ **do**
- 4: Sample the delay d_{ij} from the distribution $D_{(i,j)}$ and record the tuple $(\tau + d_{ij}, P_i, P_j, \text{AE.Enc}_{ek}(s_0))$ in **buffer**.

Computation:

- 1: Let t_{start} be the start time of the computation (rounded to the next multiple of R), and for each party $P_i \in G_C$, let t_{end}^i be the signal to terminate the execution of the computation phase.
- 2: For each honest party $P_i \in G_C$ and corrupted neighbor $P_j \in G_C$, set $S_{ij} = \emptyset$.
// Local delays generated upfront.
- 3: For each party $P_i \in G_C$ and neighbor $P_j \in G_C$, let $L_{(i,j)}$ be a list containing $n \cdot \left(\left\lfloor \frac{t_{\text{end}}^i - t_{\text{start}}}{R} \right\rfloor + 1 \right)$ samples from $D_{(i,j)}$. The messages sent by corrupted parties use these delays.
- 4: For each P_j , let $t_{\text{stop}}^j := \min\{t \in L_{(i,j)} : P_i \in \mathbf{N}_G(P_j)\}$ the time at which P_j obtains the first message from any neighbor (stops initiating Eulerian Cycles).
- 5: Compute the number of Eulerian Cycles initiated from each corrupted $P_j \in G_C$ as $N_j := \left\lfloor \frac{t_{\text{stop}}^j}{R} \right\rfloor$.
- 6: For each corrupted $P_j \in G_C$, do as follows: Run $\mathcal{E}^j = \text{EulerianCycle}(P_j, G_C)$. Let \mathcal{E}_C^j be the starting path of the eulerian cycle from P_j until the first honest party P_i . Let P_k be the last corrupted party in \mathcal{E}_C^j . Let P'_j be the first party after P_j in the path.
For each $v \in [N_j]$, for each edge $e \in \mathcal{E}_C^j$, let d_e be the next unused delay. Then, sample t_{out}^v from the distribution $4D - \sum_{e \in \mathcal{E}_C^j} \lfloor d_e/R \rfloor R$. Add (P_j, t_{out}^v) to S_{ik} .
- 7: Let P_i be an honest party, and let P_k be a corrupted neighbor. We simulate all messages sent by P_i to P_k . Let $S = \emptyset$.
- 8: // Message sent at Round 0
- 9: For each $w \in [n]$, record the tuple $(t_{\text{start}} + L_{(i,k)}[w], P_i, P_k, \text{AE.Enc}_{ek}(\perp))$ in **buffer**, where \perp is a parsable fake state.
- 10: // Messages sent at Round r
- 11: Let $r = 1$.
- 12: **while** $t_{\text{start}} + rR < t_{\text{end}}^i$ **do**
- 13: Let $S = \emptyset$ and let \mathbf{x} be a vector of length n initialized to 0.

- 14: If there exists $(P_j, t_{\text{out}}) \in S_{ik}$ such that $(r-1)R \leq t_{\text{out}} < rR$, add $\text{AE.Enc}_{ek}((j, \text{ID}, G_C, \text{Path}, \mathbf{x}, f_j))$ to S , where ID is the vector of IDs that parties in G_C generated during the preprocessing, and $\text{Path} = \mathcal{E}^j \setminus \mathcal{E}_C^j$.
- 15: **while** $|S| < n$ **do**
- 16: Add an encryption of a parsable fake state $\text{AE.Enc}_{ek}(\perp; z)$ to S .
- 17: Let $S = \{c_1, \dots, c_n\}$. For each $v \in [n]$, record the tuple $(t_{\text{start}} + rR + L_{(i,k)}[rn + v], P_i, P_k, c_v)$ in **buffer**.
- 18: $r = r + 1$.

\mathcal{S}_{HW} simulates the messages received by corrupted parties from \mathcal{F}_{NET} as follows. On every input $(\text{FETCHMESSAGES}, j)$ from a corrupted P_j , it gets the current time τ from $\mathcal{F}_{\text{CLOCK}}$. Then, for each message tuple (T, P_i, P_j, c) from **buffer** where $T \leq \tau$, it removes the tuple from **buffer** and outputs (i, c) to P_j .

Hardware messages: \mathcal{S}_{HW} has to simulate the replies to queries of corrupted parties to the hardware box functionality \mathcal{F}_{HW} . \mathcal{S}_{HW} simulates the queries of the preprocessing phase doing exactly the same as the functionality \mathcal{F}_{HW} .

In the computation phase, on input $x = (i, \text{id}, c, E, x_i, f_i, z)$, \mathcal{S}_{HW} simulates the query as follows: It executes the code of \mathcal{F}_{HW} , except that in Step 17, instead of evaluating f_i , it sets $\text{val} = v_j^{\text{out}}$, and in Step 20, it outputs $(\text{val}, \text{EncryptAndFormatOutput}(i, G, z, S, 1))$ (i.e., it sets $\text{sim} = 1$).

We now show that the execution with \mathcal{S}_{HW} is indistinguishable from the real execution. For that, we present a sequence of hybrids. In the following, we only consider the messages sent by an honest P_i to its corrupted neighbor P_k (messages between corrupted neighbors are trivial to simulate).

Hybrid 1. \mathcal{S}_{HW}^1 simulates the real world exactly. That is, \mathcal{S}_{HW}^1 has information on the entire communication graph, all edge delays and all clock rates. It simulates the messages exactly.

Hybrid 2. \mathcal{S}_{HW}^2 generates fresh random values instead of using the outputs from the **prf**.

Hybrid 3. \mathcal{S}_{HW}^3 is exactly as \mathcal{S}_{HW}^2 , except the way the output value is generated upon querying the hardware box. It returns the output v_j^{out} of the corresponding party (ignoring the input vector), instead of evaluating its function $f_j(\mathbf{x})$ on the input vector.

Hybrid 4. \mathcal{S}_{HW}^4 generates the local delays of the messages during the computation phase upfront, but still generates all messages as in \mathcal{S}_{HW}^3 . That is, instead of sampling the delays when the message is going to be sent, it calculates the number of delays that are needed for the entire computation, and generates all samples upfront.

Hybrid 5. \mathcal{S}_{HW}^5 generates all messages in the computation phase by sending artificial ciphertexts containing either the output or parsable fake encryptions.

More concretely, instead of following the path according to the message received $(j, \text{ID}, G, \text{Path}, \mathbf{x}, f_j)$ from the previous neighbor, it generates a new message $(j, \text{ID}, G_C, \text{Path}_j, \mathbf{x}_j, f_j)$ containing the graph G_C , the ID's in G_C , and the fake path on G_C started from P_j , but with the same ending. The input vector \mathbf{x}_j is the 0 vector.

Hybrid 6. \mathcal{S}_{HW}^6 , generates all messages in the computation phase at the correct times at follows: for each corrupted party P_j that has an Eulerian Cycle \mathcal{E}^j where the first honest party is P_i and the previous party is P_k , it computes the time t_{stop}^j at which P_j received the first message, the number of Eulerian Cycles N_j initiated by P_j , and a sample delay for the delay it takes for each initiated cycle message to arrive to P_i .

Hybrid 7. \mathcal{S}_{HW}^5 : P_i starts the preprocessing phase with the state where its only neighbors are the corrupted neighbors in G_C . This Hybrid corresponds to \mathcal{S}_{HW} .

- Hybrids 1 and 2 are indistinguishable by the security of the prf.
- For Hybrids 2 and 3 to be indistinguishable, we need that $f_j(\mathbf{x}) = v_j^{\text{out}}$, where v_j^{out} is received from the ideal functionality, and \mathbf{x} and f_j are the values contained in the message. Hence, we have to show that the message contains the actual function and inputs (and not modified values from the adversary). To see why this holds, observe that any message that enters the corrupted component and can be decrypted has been processed by at least one honest party during the *second* cycle traversal. This means that this message was actually sent to this party. Now these sent messages are always correct (recall the adversary is passive) and the box never changes them. Moreover, the adversary cannot produce any forged or modified messages due to the security of the authenticated encryption. Hence, the value f_j is correct and \mathbf{x} contains inputs (again, correctly) provided by parties during the first traversal.
- Hybrids 3 and 4 are trivially identical.
- The only difference between the Hybrids 4 and 5 is in the content of the encrypted messages generated by the simulator. We argue that the output of the simulated hardware box is indistinguishable for these messages. Observe that the outputs of the hardware box can be either encryptions, or the output. Both are trivially indistinguishable, as long as the box outputs an encryption in Hybrid 4 if and only if it outputs an encryption in Hybrid 5. This is the case, because (1) the graph G in any message from an honest party is correct (by an argument analogous to the one we used to argue that the function f_j is correct), and (2) the part of Path remaining to traverse when the Eulerian cycle is computed on j and G is the same as the part of Path_j remaining to traverse when it is computed on j and G_C .
- Hybrids 5 and 6 are trivially identical, since the preprocessing messages are always encryptions under the secret key of \mathcal{F}_{HW} .

- Hybrids 6 and 7 are indistinguishable by semantic security.

□

□

Chapter 4

Property Preserving Hashing

TODO

4.1 Introduction

The problem of property-preserving hashing, namely how to compress a large input in a way that preserves a class of its properties, is an important one in the modern age of massive data. In particular, the idea of property-preserving hashing underlies sketching [61, 58, 5, 29, 21], compressed sensing [25], locality-sensitive hashing [46], and in a broad sense, much of machine learning.

As two concrete examples in theoretical computer science, consider universal hash functions [19] which can be used to test the equality of data points, and locality-sensitive hash functions [46, 45] which can be used to test the ℓ_p -distance between vectors. In both cases, we trade off accuracy in exchange for compression. For example, in the use of universal hash functions to test for equality of data points, one stores the hash $h(x)$ of a point x together with the description of the hash function h . Later, upon obtaining a point y , one computes $h(y)$ and checks if $h(y) = h(x)$. The pigeonhole principle tells us that mistakes are inevitable; all one can guarantee is that they happen with an acceptably small probability. More precisely, universal hash functions tell us that

$$\forall x \neq y \in D, \Pr[h \leftarrow \mathcal{H} : h(x) \neq h(y)] \geq 1 - \epsilon$$

for some small ϵ . A cryptographer's way of looking at such a statement is that it asks the adversary to pick x and y first; and evaluates her success w.r.t. a hash function chosen randomly from the family \mathcal{H} . In particular, the adversary has no information about the hash function when she comes up with the (potentially) offending inputs x and y . Locality-sensitive hash functions have a similar flavor of correctness guarantee.

The starting point of this work is that this definition of correctness is too weak in the face of adversaries with access to the hash function (either the description of the function itself or perhaps simply oracle access to its evaluation). Indeed, in the context of equality testing, we have by now developed several notions of robustness against such adversaries, in the form of pseudorandom functions (PRF) [36], universal one-way

hash functions (UOWHF) [63] and collision-resistant hash functions (CRHF). Our goal in this work is to expand the reach of these notions beyond testing equality; that is, our aim is to do unto property-preserving hashing what CRHFs did to universal hashing.

Several works have observed the deficiency of the universal hash-type definition in adversarial settings, including a wide range of recent attacks within machine learning in adversarial environments (e.g., [55, 48, 68, 64, 49]). Such findings motivate a rigorous approach to combatting adversarial behavior in these settings, a direction in which significantly less progress has been made. Mironov, Naor and Segev [57] showed interactive protocols for sketching in such an adversarial environment; in contrast, we focus on non-interactive hash functions. Hardt and Woodruff [39] showed negative results which say that linear functions cannot be robust (even against computationally bounded adversaries) for certain natural ℓ_p distance properties; our work will use non-linearity and computational assumptions to overcome the [39] attack. Finally, Naor and Yagev [62] study adversarial Bloom filters which compress a set in a way that supports checking set membership; we will use their lower bound techniques in Section ??.

Motivating Robustness: Facial Recognition. In the context of facial recognition, authorities A and B store the captured images x of suspects. At various points in time, say authority A wishes to look up B 's database for a suspect with face x . A can do so by comparing $h(x)$ with $h(y)$ for all y in B 's database.

This application scenario motivated prior notions of fuzzy extractors and secure sketching. As with secure sketches and fuzzy extractors, a locality-sensitive property-preserving hash guarantees that close inputs (facial images) remain close when hashed [31]; this ensures that small changes in one's appearance do not affect whether or not that person is authenticated. However, neither fuzzy extractors nor secure sketching guarantees that far inputs remain far when hashed. Consider an adversarial setting, not where a person wishes to evade detection, but where she wishes to be mistaken for someone else. Her face x' will undoubtedly be different (far) from her target x , but there is nothing preventing her from slightly altering her face and passing as a completely different person when using a system with such a one-sided guarantee. This is where our notion of robustness comes in (as well as the need for cryptography): not only will adversarially chosen close x and x' map to close $h(x)$ and $h(x')$, but if adversarially chosen x and x' are far, they will be mapped to far outputs, unless the adversary is able to break a cryptographic assumption.

Comparison to Secure Sketches and Fuzzy Extractors. It is worth explicitly comparing fuzzy extractors and secure sketching to this primitive [31], as they aim to achieve similar goals. Both of these seek to preserve the privacy of their inputs. Secure sketches generate random-looking sketches that hide information about the original input so that the original input can be reconstructed when given something close to it. Fuzzy extractors generate uniform-looking keys based off of fuzzy (biometric) data also using entropy: as long as the input has enough entropy, so will the output. As stated above, both guarantee that if inputs are close, they will 'sketch' or 'extract' to the same object. Now, the entropy of the sketch or key guarantees that randomly

generated far inputs will not collide, but there are no guarantees about adversarially generated far inputs. To use the example above, it could be that once an adversary sees a sketch or representation, she can generate two far inputs that will reconstruct to the correct input.

Robust Property-Preserving Hash Functions. We put forth several notions of robustness for property-preserving hash (PPH) functions which capture adversaries with increasing power and access to the hash function. We then ask which properties admit robust property-preserving hash functions, and show positive and negative results.

- On the negative side, using a connection to communication complexity, we show that most properties and even simple ones such as set disjointness, inner product and greater-than do not admit non-trivial property-preserving hash functions.
- On the positive side, we provide two constructions of robust property-preserving hash functions (satisfying the strongest of our notions). The first is based on the standard cryptographic assumption of collision-resistant hash functions, and the second achieves more aggressive parameters under a new assumption related to the hardness of syndrome decoding on low density parity-check (LDPC) codes.
- Finally, we show that for essentially any non-trivial predicate (which we call collision-sensitive), achieving even a mild form of robustness requires cryptographic assumptions.

We proceed to describe our contributions in more detail.

4.1.1 Our Results and Techniques

We explore two notions of properties. The first is that of property classes $\mathcal{P} = \{P : D \rightarrow \{0, 1\}\}$, sets of single-input predicates. This notion is the most general, and is the one in which we prove lower bounds. The second is that of two-input properties $P : D \times D \rightarrow \{0, 1\}$, which compares two inputs. This second notion is more similar to standard notions of universal hashing and collision-resistance, stronger than the first, and where we get our constructions. We note that a two-input predicate has an analogous predicate-class $\mathcal{P} = \{P_x\}_{x \in D}$, where $P_{x_1}(x_2) = P(x_1, x_2)$.

The notion of a property can be generalized in many ways, allowing for promise properties which output 0, 1 or \otimes (a don't care symbol), and allowing for more than 2 inputs. The simplest notion of correctness for property-preserving hash functions requires that, analogously to universal hash functions,

$$\forall x, y \in D \Pr[h \leftarrow \mathcal{H} : \mathcal{H}.\text{Eval}(h, h(x), h(y)) \neq P(x, y)] = \text{negl}(\kappa)$$

or for single-input predicate-classes

$$\forall x \in D \text{ and } P \in \mathcal{P} \Pr[h \leftarrow \mathcal{H} : \mathcal{H}.\text{Eval}(h, h(x), P) \neq P(x)] = \text{negl}(\kappa)$$

where κ is a security parameter.

For the sake of simplicity in our overview, we will focus on two-input predicates.

Defining Robust Property-Preserving Hashing.

We define several notions of robustness for PPH, each one stronger than the last. Here, we describe the strongest of all, called direct-access PPH.

In a direct-access PPH, the (polynomial-time) adversary is given the hash function and is asked to find a pair of bad inputs, namely $x, y \in D$ such that $\mathcal{H}.\text{Eval}(h, h(x), h(y)) \neq P(x, y)$. That is, we require that

$$\forall \text{ p.p.t. } \mathcal{A}, \Pr[h \leftarrow \mathcal{H}; (x, y) \leftarrow \mathcal{A}(h) : \mathcal{H}.\text{Eval}(h, h(x), h(y)) \neq P(x, y)] = \text{negl}(\kappa).$$

The direct-access definition is the analog of collision-resistant hashing for general properties.

Our other definitions vary by how much access the adversary is given to the hash function, and are motivated by different application scenarios. From the strong to weak, these include double-oracle PPH where the adversary is given access to a hash oracle and a hash evaluation oracle, and evaluation-oracle PPH where the adversary is given only a combined oracle. Definitions similar to double-oracle PPH have been proposed in the context of adversarial bloom filters [62], and ones similar to evaluation-oracle PPH have been proposed in the context of showing attacks against property-preserving hash functions [39]. For more details, we refer the reader to Section ??.

Connections to Communication Complexity and Negative Results. Property-preserving hash functions for a property P , even without robustness, imply communication-efficient protocols for P in several models. For example, any PPH for P implies a protocol for P in the simultaneous messages model of Babai, Gal, Kimmel and Lokam [8] wherein Alice and Bob share a common random string h , and hold inputs x and y respectively. Their goal is to send a single message to Charlie who should be able to compute $P(x, y)$ except with small error. Similarly, another formalization of PPH that we present, called PPH for single-input predicate classes (see Section ??) implies efficient protocols in the one-way communication model [70].

We use known lower bounds in these communication models to rule out PPHs for several interesting predicates (even without robustness). There are two major differences between the PPH setting and the communication setting, however: (a) in the PPH setting, we demand an error that is negligible (in a security parameter); and (b) we are happy with protocols that communicate $n - 1$ bits (or the equivalent bound in the case of promise properties) whereas the communication lower bounds typically come in the form of $\Omega(n)$ bits. In other words, the communication lower bounds as-is do not rule out PPH.

At first thought, one might be tempted to think that the negligible-error setting is the same as the deterministic setting where there are typically lower bounds of n (and not just $\Omega(n)$); however, this is not the case. For example, the equality function which has a negligible error public-coin simultaneous messages protocol (simply using universal hashing) with communication complexity $CC = O(\kappa)$ and deterministic protocols require $CC \geq n$. Thus, deterministic lower bounds do not (indeed, cannot)

do the job, and we must better analyze the randomized lower bounds. Our refined analysis shows the following lower bounds:

- PPH for the Gap-Hamming (promise) predicate with a gap of $\sqrt{n}/2$ is impossible by refining the analysis of a proof by Jayram, Kumar and Sivakumar [47]. The Gap-Hamming predicate takes two vectors in $\{0,1\}^n$ as input, outputs 1 if the vectors are very far, 0 if they are very close, and we do not care what it outputs for inputs in the middle.
- We provide a framework for proving PPHs are impossible for some total predicates, characterizing these classes as reconstructing. A predicate-class is reconstructing if, when only given oracle access to the predicates of a certain value x , we can efficiently determine x with all but negligible probability.¹ With this framework, we show that PPH for the Greater-Than (GT) function is impossible. It was known that GT required $\Omega(n)$ bits (for constant error) [65], but we show a lower bound of exactly n if we want negligible error. Index and Exact-Hamming are also reconstructing predicates.
- We also obtain a lower bound for a variant of GT: the (promise) Gap- k GT predicate which on inputs $x, y \in [N = 2^n]$, outputs 1 if $x - y > k$, 0 if $y - x > k$, and we do not care what it outputs for inputs in between. Here, exactly $n - \log(k) - 1$ bits are required for a perfect PPH. This is tight: we show that with fewer bits, one cannot even have a non-robust PPH, whereas there is a perfect robust PPH that compresses to $n - \log(k) - 1$ bits.

New Constructions. Our positive results are two constructions of a direct-access PPH for gap-Hamming for n -length vectors for large gaps of the form $\sim O(n/\log n)$ (as opposed to an $O(\sqrt{n})$ -gap for which we have a lower bound). Let us recall the setting: the gap Hamming predicate P_{ham} , parameterized by n, d and ϵ , takes as input two n -bit vectors x and y , and outputs 1 if the Hamming distance between x and y is greater than $d(1 + \epsilon)$, 0 if it is smaller than $d(1 - \epsilon)$ and a don't care symbol \otimes otherwise. To construct a direct-access PPH for this (promise) predicate, one has to construct a compressing family of functions \mathcal{H} such that

$$\begin{aligned} \forall p.p.t. \mathcal{A}, \Pr[h \leftarrow \mathcal{H}; (x, y) \leftarrow \mathcal{A}(h) : P_{\text{ham}}(x, y) \neq \otimes \\ \wedge \mathcal{H}.\text{Eval}(h, h(x), h(y)) \neq P_{\text{ham}}(x, y)] = \text{negl}(\kappa). \end{aligned} \quad (4.1)$$

Our two constructions offer different benefits. The first provides a clean general approach, and relies on the standard cryptographic assumption of collision-resistant hash functions. The second builds atop an existing one-way communication protocol, supports a smaller gap and better efficiency, and ultimately relies on a (new) variant of the syndrome decoding assumption on low-density parity check codes.

Construction 1. The core idea of the first construction is to reduce the goal of robust Hamming PPH to the simpler one of robust equality testing; or, in a word,

¹In the single-predicate language of above, the predicate class corresponds to $\mathcal{P} = \{P(x, \cdot)\}$.

“subsampling.” The intuition is to notice that if $\mathbf{x}_1 \in \{0,1\}^n$ and $\mathbf{x}_2 \in \{0,1\}^n$ are close, then most small enough subsets of indices of \mathbf{x}_1 and \mathbf{x}_2 will match identically. On the other hand, if \mathbf{x}_1 and \mathbf{x}_2 are far, then most large enough subsets of indices will differ.

The hash function construction will thus fix a collection of sets $\mathcal{S} = \{S_1, \dots, S_k\}$, where each $S_i \subseteq [n]$ is a subset of appropriately chosen size s . The desired structure can be achieved by defining the subsets S_i as the neighbor sets of a bipartite expander. On input $\mathbf{x} \in \{0,1\}^n$, the hash function will consider the vector $\mathbf{y} = (\mathbf{x}|_{S_1}, \dots, \mathbf{x}|_{S_k})$ where $\mathbf{x}|_S$ denotes the substring of \mathbf{x} indexed by the set S . The observation above tells us that if \mathbf{x}_1 and \mathbf{x}_2 are close (resp. far), then so are \mathbf{y}_1 and \mathbf{y}_2 .

Up to now, it is not clear that progress has been made: indeed, the vector \mathbf{y} is not compressing (in which case, why not stick with $\mathbf{x}_1, \mathbf{x}_2$ themselves?). However, $\mathbf{y}_1, \mathbf{y}_2$ satisfy the desired Hamming distance properties with fewer symbols over a large alphabet, $\{0,1\}^s$. As a final step, we can then leverage (standard) collision-resistant hash functions (CRHF) to compress these symbols. Namely, the final output of our hash function $h(\mathbf{x})$ will be the vector $(g(\mathbf{x}|_{S_1}), \dots, g(\mathbf{x}|_{S_k}))$, where each substring of \mathbf{x} is individually compressed by a CRHF g .

The analysis of the combined hash construction then follows cleanly via two steps. The (computational) collision-resistance property of g guarantees that any efficiently found pair of inputs $\mathbf{x}_1, \mathbf{x}_2$ will satisfy that their hash outputs

$$h(\mathbf{x}_1) = (g(\mathbf{x}_1|_{S_1}), \dots, g(\mathbf{x}_1|_{S_k})) \quad \text{and} \quad h(\mathbf{x}_2) = (g(\mathbf{x}_2|_{S_1}), \dots, g(\mathbf{x}_2|_{S_k}))$$

are close if and only if it holds that

$$(\mathbf{x}_1|_{S_1}, \dots, \mathbf{x}_1|_{S_k}) \quad \text{and} \quad (\mathbf{x}_2|_{S_1}, \dots, \mathbf{x}_2|_{S_k})$$

are close as well; that is, $\mathbf{x}_1|_{S_i} = \mathbf{x}_2|_{S_i}$ for most S_i . (Anything to the contrary would imply finding a collision in g .) Then, the combinatorial properties of the chosen index subsets S_i ensures (unconditionally) that any such inputs $\mathbf{x}_1, \mathbf{x}_2$ must themselves be close. The remainder of the work is to specify appropriate parameter regimes for which the CRHF can be used and the necessary bipartite expander graphs exist.

Construction 2. The starting point for our second construction is a simple non-robust hash function derived from a one-way communication protocol for gap-Hamming due to Kushilevitz, Ostrovsky, and Rabani [51]. In a nutshell, the hash function is parameterized by a random sparse $m \times n$ matrix A with 1’s in $1/d$ of its entries and 0’s elsewhere; multiplying this matrix by a vector \mathbf{z} “captures” information about the Hamming weight of \mathbf{z} . However, this can be seen to be trivially not robust when the hash function is given to the adversary. The adversary simply performs Gaussian elimination, discovering a “random collision” (x, y) in the function, where, with high probability $x \oplus y$ will have large Hamming weight. This already breaks equation (4.1).

The situation is somewhat worse. Even in a very weak, oracle sense, corresponding to our evaluation-oracle-robustness definition, a result of Hardt and Woodruff [39] shows that there are no linear functions h that are robust for the gap- ℓ_2 predicate. While their result does not carry over as-is to the setting of ℓ_0 (Hamming), we conjecture it does, leaving us with two options: (a) make the domain sparse: both the

Gaussian elimination attack and the Hardt-Woodruff attack use the fact that Gaussian elimination is easy on the domain of the hash function; however making the domain sparse (say, the set of all strings of weight at most βn for some constant $\beta < 1$) already rules it out; and (b) make the hash function non-linear: again, both attacks crucially exploit linearity. We will pursue both options, and as we will see, they are related.

But before we get there, let us ask whether we even need computational assumptions to get such a PPH. Can there be information-theoretic constructions? The first observation is that by a packing argument, if the output domain of the hash function has size less than $2^{n - n \cdot H(\frac{d(1+\epsilon)}{n})} \approx 2^{n - d \log n (1+\epsilon)}$ (for small d), there are bound to be “collisions”, namely, two far points (at distance more than $d(1+\epsilon)$) that hash to the same point. So, you really cannot compress much information-theoretically, especially as d becomes smaller. A similar bound holds when restricting the domain to strings of Hamming weight at most βn for constant $\beta < 1$.

With that bit of information, let us proceed to describe in a very high level our construction and the computational assumption. Our construction follows the line of thinking of Applebaum, Haramaty, Ishai, Kushilevitz and Vaikuntanathan [6] where they used the hardness of syndrome decoding problems to construct collision-resistant hash functions. Indeed, in a single sentence, our observation is that their collision-resistant hash functions are locality-sensitive by virtue of being input-local, and thus give us a robust gap-Hamming PPH (albeit under a different assumption).

In slightly more detail, our first step is to simply take the construction of Kushilevitz, Ostrovsky, and Rabani [51], and restrict the domain of the function. We show that finding two close points that get mapped to far points under the hash function is simply impossible (for our setting of parameters). On the other hand, there exist two far points that get mapped to close points under the hash functions (in fact, they even collide). Thus, showing that it is hard to find such points requires a computational assumption.

In a nutshell, our assumption says that given a random matrix \mathbf{A} where each entry is chosen from the Bernoulli distribution with $\text{Ber}(1/d)$ with parameter $1/d$, it is hard to find a large Hamming weight vector \mathbf{x} where $\mathbf{Ax} \pmod{2}$ has small Hamming weight. Of course, “large” and “small” here have to be parameterized correctly (see Section ?? for more details), however we observe that this is a generalization of the syndrome decoding assumption for low-density parity check (LDPC) codes, made by [6].

In our second step, we remove the sparsity requirement on the input domain of the predicate. We show a sparsification transformation which takes arbitrary n -bit vectors and outputs $n' > n$ -bit sparse vectors such that (a) the transformation is injective, and (b) the expansion introduced here does not cancel out the effect of compression achieved by the linear transformation $\mathbf{x} \rightarrow \mathbf{Ax}$. This requires careful tuning of parameters for which we refer the reader to Section ??.

The Necessity of Cryptographic Assumptions. The goal of robust PPH is to compress beyond the information theoretic limits, to a regime where incorrect hash outputs exist but are hard to find. If robustness is required even when the hash function

is given, this inherently necessitates cryptographic hardness assumptions. A natural question is whether weaker forms of robustness (where the adversary sees only oracle access to the hash function) similarly require cryptographic assumptions, and what types of assumptions are required to build non-trivial PPHs of various kinds.

As a final contribution, we identify necessary assumptions for PPH for a kind of predicate we call collision sensitive. In particular, PPH for any such predicate in the double-oracle model implies the existence of one-way functions, and in the direct-access model implies existence of collision-resistant hash functions. In a nutshell, collision-sensitive means that finding a collision in the predicate breaks the property-preserving nature of any hash. The proof uses and expands on techniques from the work of Naor and Yaguev on adversarially robust Bloom Filters [62]. The basic idea is the same: without OWFs, we can invert arbitrary polynomially-computable functions with high probability in polynomial time, and using this we get a representation of the hash function/set, which can be used to find offending inputs.

4.2 Defining Property-Preserving Hash Functions

Our definition of property preserving hash functions (PPHs) comes in several flavors, depending on whether we support total or partial predicates; whether the predicates take a single input or multiple inputs; and depending on the information available to the adversary. We discuss each of these choices in turn.

Total vs. Partial Predicates. We consider total predicates that assign a 0 or 1 output to each element in the domain, and promise (or partial) predicates that assign a 0 or 1 to a subset of the domain and a wildcard (don't-care) symbol \otimes to the rest. More formally, a total predicate P on a domain X is a function $P : X \rightarrow \{0, 1\}$, well-defined as 0 or 1 for every input $x \in X$. A promise predicate P on a domain X is a function $P : X \rightarrow \{0, 1, \otimes\}$. Promise predicates can be used to describe scenarios (such as gap problems) where we only care about providing an exact answer on a subset of the domain.

Our definitions below will deal with the more general case of promise predicates, but we will discuss the distinction between the two notions when warranted.

Single-Input vs Multi-Input Predicates. In the case of single-input predicates, we consider a class of properties \mathcal{P} and hash a single input x into $h(x)$ in a way that given $h(x)$, one can compute $P(x)$ for any $P \in \mathcal{P}$. Here, h is a compressing function. In the multi-input setting, we think of a single fixed property P that acts on a tuple of inputs, and require that given $h(x_1), h(x_2), \dots, h(x_k)$, one can compute $P(x_1, x_2, \dots, x_k)$. The second syntax is more expressive than the first, and so we use the multi-input syntax for constructions and the single-input syntax for lower bounds².

²There is yet a third possibility, namely where there is a *fixed* predicate P that acts on a single input x , and we require that given $h(x)$, one can compute $P(x)$. This makes sense when the computational complexity of h is considerably less than that of P , say when P is the parity function and h is an AC^0 circuit, as in the work of Dubrov and Ishai [?]. We do not explore this third syntax further in this work.

For security parameter λ , fixed predicate class \mathcal{P} , and h sampled from $\mathcal{H}.\text{Samp}$	
Non-Robust PPH	Adversary has no access to hash function or evaluation.
Evaluation-Oracle PPH	Access to the evaluation oracle $\mathcal{O}_h^{\text{Eval}}(x, P) = \mathcal{H}.\text{Eval}(h, P, h(x))$.
Double-Oracle PPH	Access to both $\mathcal{O}_h^{\text{Eval}}$ (as above) and hash oracle $\mathcal{O}_h^{\text{Hash}}(x) = h(x)$.
Robust PPH “Direct Access”	Direct access to the hash function, description of h .

Figure 4-1: A table comparing the adversary’s access to the hash function within different robustness levels of PPHs.

Before we proceed to discuss robustness, we provide a working definition for a property-preserving hash function for the single-input syntax. For the multi-input predicate definition and further discussion, see appendix ??.

Definition 9. A (non-robust) η -compressing Property Preserving Hash (η -PPH) family $\mathcal{H} = \{h : X \rightarrow Y\}$ for a function η and a class of predicates \mathcal{P} requires the following two efficiently computable algorithms:

- $\mathcal{H}.\text{Samp}(1^\kappa) \rightarrow h$ is a randomized p.p.t. algorithm that samples a random hash function from \mathcal{H} with security parameter κ .
- $\mathcal{H}.\text{Eval}(h, P, y)$ is a deterministic polynomial-time algorithm that on input the hash function h , a predicate $P \in \mathcal{P}$ and $y \in Y$ (presumably $h(x)$ for some $x \in X$), outputs a single bit.

Additionally, \mathcal{H} must satisfy the following two properties:

- η -compressing, namely, $\log |Y| \leq \eta(\log |X|)$, and
- robust, according to one of four definitions that we describe below, leading to four notions of PPH: definition ?? (non-robust PPH), ?? (evaluation-oracle-robust PPH or EO-PPH), ?? (double-oracle-robust PPH or DO-PPH), or ?? (direct-access robust PPH or DA-PPH). We will refer to the strongest form, namely direct-access robust PPH as simply robust PPH when the intent is clear from the context. See also figure ?? for a direct comparison between these.

The Many Types of Robustness. We will next describe four definitions of robustness for PPHs, starting from the weakest to the strongest. Each of these definitions, when plugged into the last bullet of Definition ??, gives rise to a different type of property-preserving hash function. In each of these definitions, we will describe an adversary whose goal is to produce an input and a predicate such that the hashed predicate evaluation disagrees with the truth. The difference between the definitions is in what an adversary has access to, summarized in figure ??.

4.2.1 Non-Robust PPH

We will start by defining the weakest notion of robustness which we call non-robust PPH. Here, the adversary has no information at all on the hash function h , and is required to produce a predicate P and a valid input x , namely where $P(x) \neq \circledast$, such that $\mathcal{H}.\text{Eval}(h, P, x) \neq P(x)$ with noticeable probability. When \mathcal{P} is the family of point functions (or equality functions), this coincides with the notion of 2-universal hash families [19]³.

Here and in the following, we use the notation $\Pr[A_1; \dots; A_m : E]$ to denote the probability that event E occurs following an experiment defined by executing the sequence A_1, \dots, A_m in order.

Definition 10. A family of PPH functions $\mathcal{H} = \{h : X \rightarrow Y\}$ for a class of predicates \mathcal{P} is a family of non-robust PPH functions if for any $P \in \mathcal{P}$ and $x \in X$ such that for $P(x) \neq \circledast$,

$$\Pr[h \leftarrow \mathcal{H}.\text{Samp}(1^\kappa) : \mathcal{H}.\text{Eval}(h, P, h(x)) \neq P(x)] \leq \text{negl}(\kappa).$$

4.2.2 Evaluation-Oracle Robust PPH

In this model, the adversary has slightly more power than in the non-robust setting. Namely, she can adaptively query an oracle that has $h \leftarrow \mathcal{H}.\text{Samp}(1^\kappa)$ in its head, on inputs $P \in \mathcal{P}$ and $x \in X$, and obtain as output the hashed evaluation result $\mathcal{H}.\text{Eval}(h, P, h(x))$. Let $\mathcal{O}_h(x, P) = \mathcal{H}.\text{Eval}(h, P, h(x))$.

Definition 11. A family of PPH functions $\mathcal{H} = \{h : X \rightarrow Y\}$ for a class of predicates \mathcal{P} is a family of evaluation-oracle robust (EO-robust) PPH functions if, for any PPT adversary \mathcal{A} ,

$$\Pr[h \leftarrow \mathcal{H}.\text{Samp}(1^\kappa); (x, P) \leftarrow \mathcal{A}^{\mathcal{O}_h}(1^\kappa) : P(x) \neq \circledast \wedge \mathcal{H}.\text{Eval}(h, P, h(x)) \neq P(x)] \leq \text{negl}(\kappa).$$

The reader might wonder if this definition is very weak, and may ask if it follows just from the definition of a non-robust PPH family. In fact, for total predicates, we show that the two definitions are the same. At a high level, simply querying the evaluation oracle on (even adaptively chosen) inputs cannot reveal information about the hash function since with all but negligible probability, the answer from the oracle will be correct and thus simulatable without oracle access. The proof of the following lemma is in Appendix ??.

Lemma 10. Let \mathcal{P} be a class of total predicates on X . A non-robust PPH \mathcal{H} for \mathcal{P} is also an Evaluation-Oracle robust PPH for \mathcal{P} for the same domain X and same codomain Y .

³While 2-universal hashing corresponds with a two-input predicate testing equality, the single-input version ($\{P_{x_1}\}$ where $P_{x_1}(x_2) = (x_1 == x_2)$) is more general, and so it is what we focus on.

However, when dealing with promise predicates, an EO-robustness adversary has the ability to make queries that do not satisfy the promise, and could get information about the hash function, perhaps even reverse-engineering the entire hash function itself. Indeed, Hardt and Woodruff [39] show that there are no EO-robust linear hash functions for a certain promise- ℓ_p distance property; whereas, non-robust linear hash functions for these properties follow from the work of Indyk [46, 45].

4.2.3 Double-Oracle PPH

We continue our line of thought, giving the adversary more power. Namely, she has access to two oracles, both have a hash function $h \leftarrow \mathcal{H}.\text{Samp}(1^\kappa)$ in their head. The hash oracle $\mathcal{O}_h^{\text{Hash}}$, parameterized by $h \in \mathcal{H}$, outputs $h(x)$ on input $x \in X$. The predicate evaluation oracle $\mathcal{O}_h^{\text{Eval}}$, also parameterized by $h \in \mathcal{H}$, takes as input $P \in \mathcal{P}$ and $y \in Y$ and outputs $\mathcal{H}.\text{Eval}(h, P, y)$. When \mathcal{P} is the family of point functions (or equality functions), this coincides with the notion of psuedo-random functions.

Definition 12. A family of PPH functions $\mathcal{H} = \{h : X \rightarrow Y\}$ for a class of predicates \mathcal{P} is a family of double-oracle-robust PPH (DO-PPH) functions if, for any PPT adversary \mathcal{A} ,

$$\Pr[h \leftarrow \mathcal{H}.\text{Samp}(1^\kappa); (x, P) \leftarrow \mathcal{A}^{\mathcal{O}_h^{\text{Hash}}, \mathcal{O}_h^{\text{Eval}}}(1^\kappa) : P(x) \neq \circledast \wedge \mathcal{H}.\text{Eval}(h, P, h(x)) \neq P(x)] \leq \text{negl}(\kappa).$$

We show that any evaluation-oracle-robust PPH can be converted into a double-oracle-robust PPH at the cost of a computational assumption, namely, one-way functions. In a nutshell, the observation is that the output of the hash function can be encrypted using a symmetric key that is stored as part of the hash description, and the evaluation proceeds by first decrypting.

Lemma 11. Let \mathcal{P} be a class of (total or partial) predicates on X . Assume that one-way functions exist. Then, any EO-robust PPH for \mathcal{P} can be converted into a DO-robust PPH for \mathcal{P} .

See appendix ?? for the full proof.

4.2.4 Direct-Access Robust PPH

Finally, we define the strongest notion of robustness where the adversary is given the description of the hash function itself. When \mathcal{P} is the family of point functions (or equality functions), this coincides with the notion of collision-resistant hash families.

Definition 13. A family of PPH functions $\mathcal{H} = \{h : X \rightarrow Y\}$ for a class of predicates \mathcal{P} is a family of direct-access robust PPH functions if, for any PPT adversary \mathcal{A} ,

$$\Pr[h \leftarrow \mathcal{H}.\text{Samp}(1^\kappa); (x, P) \leftarrow \mathcal{A}(h) : P(x) \neq \circledast \wedge \mathcal{H}.\text{Eval}(h, P, h(x)) \neq P(x)] \leq \text{negl}(\kappa).$$

We will henceforth focus on direct-access-robust property-preserving hash functions and refer to them simply as robust PPHs.

Chapter 5

Fine-Grained Cryptography

TODO

Appendix A

Tables

Table A.1: Armadillos

Armadillos	are
our	friends

Appendix B

Figures

Figure B-1: Armadillo slaying lawyer.

Figure B-2: Armadillo eradicating national debt.

Bibliography

- [1] Adi Akavia, Rio LaVigne, and Tal Moran. *Topology-hiding computation on all graphs*. In Jonathan Katz and Hovav Shacham, editors, CRYPTO 2017, Part I, volume 10401 of LNCS, pages 447–467. Springer, Heidelberg, August 2017.
- [2] Adi Akavia, Rio LaVigne, and Tal Moran. *Topology-hiding computation on all graphs*. In Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I, pages 447–467, 2017.
- [3] Adi Akavia, Rio LaVigne, and Tal Moran. *Topology-hiding computation on all graphs*. Cryptology ePrint Archive, Report 2017/296, 2017. <http://eprint.iacr.org/2017/296>.
- [4] Adi Akavia and Tal Moran. *Topology-hiding computation beyond logarithmic diameter*. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, EUROCRYPT 2017, Part II, volume 10211 of LNCS, pages 609–637. Springer, Heidelberg, May 2017.
- [5] Noga Alon, Yossi Matias, and Mario Szegedy. *The space complexity of approximating the frequency moments*. In Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996, pages 20–29, 1996.
- [6] Benny Applebaum, Naama Haramaty, Yuval Ishai, Eyal Kushilevitz, and Vinod Vaikuntanathan. *Low-complexity cryptographic hash functions*. In 8th Innovations in Theoretical Computer Science Conference, ITCS 2017, January 9-11, 2017, Berkeley, CA, USA, pages 7:1–7:31, 2017.
- [7] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. *Multiparty computation with low communication, computation and interaction via threshold FHE*. In David Pointcheval and Thomas Johansson, editors, EUROCRYPT 2012, volume 7237 of LNCS, pages 483–501. Springer, Heidelberg, April 2012.
- [8] László Babai, Anna Gál, Peter G. Kimmel, and Satyanarayana V. Lokam. *Communication complexity of simultaneous messages*. SIAM J. Comput., 33(1):137–166, 2003.

- [9] Marshall Ball, Elette Boyle, Tal Malkin, and Tal Moran. *Exploring the boundaries of topology-hiding computation*. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III, pages 294–325, 2018.
- [10] Boaz Barak and Mohammad Mahmoody-Ghidary. *Merkle puzzles are optimal - an $O(n^2)$ -query attack on any key exchange from a random oracle*. In *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference*, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings, pages 374–390, 2009.
- [11] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. *Key-privacy in public-key encryption*. In *Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security*, Gold Coast, Australia, December 9-13, 2001, Proceedings, pages 566–582, 2001.
- [12] Michael Ben-Or, Ran Canetti, and Oded Goldreich. *Asynchronous secure computation*. In *25th ACM STOC*, pages 52–61. ACM Press, May 1993.
- [13] Jurjen N. Bos and Bert den Boer. *Detection of disrupters in the DC protocol*. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *EUROCRYPT’89*, volume 434 of LNCS, pages 320–327. Springer, Heidelberg, April 1990.
- [14] Elette Boyle, Rio LaVigne, and Vinod Vaikuntanathan. *Adversarially robust property-preserving hash functions*. In *10th Innovations in Theoretical Computer Science Conference, ITCS 2019, January 10-12, 2019, San Diego, California, USA*, pages 16:1–16:20, 2019.
- [15] Ran Canetti. *Universally composable security: A new paradigm for cryptographic protocols*. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.
- [16] Ran Canetti, Huijia Lin, Stefano Tessaro, and Vinod Vaikuntanathan. *Obfuscation of probabilistic circuits and applications*. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *TCC 2015, Part II*, volume 9015 of LNCS, pages 468–497. Springer, Heidelberg, March 2015.
- [17] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. *Universally composable two-party and multi-party secure computation*. In *34th ACM STOC*, pages 494–503. ACM Press, May 2002.
- [18] Marco L. Carmosino, Jiawei Gao, Russell Impagliazzo, Ivan Mihajlin, Ramamohan Paturi, and Stefan Schneider. *Nondeterministic extensions of the strong exponential time hypothesis and consequences for non-reducibility*. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, Cambridge, MA, USA, January 14-16, 2016, pages 261–270, 2016.

- [19] Larry Carter and Mark N. Wegman. *Universal classes of hash functions (extended abstract)*. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, May 4-6, 1977, Boulder, Colorado, USA, pages 106–112, 1977.
- [20] Nishanth Chandran, Vipul Goyal, and Amit Sahai. *New constructions for UC secure computation using tamper-proof hardware*. In *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Istanbul, Turkey, April 13-17, 2008. *Proceedings*, pages 545–562, 2008.
- [21] Moses Charikar, Kevin C. Chen, and Martin Farach-Colton. *Finding frequent items in data streams*. *Theor. Comput. Sci.*, 312(1):3–15, 2004.
- [22] David Chaum. *Untraceable electronic mail, return addresses, and digital pseudonyms*. *Commun. ACM*, 24(2):84–88, 1981.
- [23] David Chaum. *The dining cryptographers problem: Unconditional sender and recipient untraceability*. *Journal of Cryptology*, 1(1):65–75, 1988.
- [24] Stephen Checkoway, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J. Bernstein, Jake Maskiewicz, Hovav Shacham, and Matthew Fredrikson. *On the practical exploitability of dual EC in TLS implementations*. In *Proceedings of the 23rd USENIX Security Symposium*, San Diego, CA, USA, August 20-22, 2014, pages 319–335, 2014.
- [25] Scott Shaobing Chen, David L. Donoho, and Michael A. Saunders. *Atomic decomposition by basis pursuit*. *SIAM Rev.*, 43(1):129–159, 2001.
- [26] Alessandro Chiesa and Eran Tromer. *Proof-carrying data and hearsay arguments from signature cards*. In *Andrew Chi-Chih Yao, editor, ICS 2010*, pages 310–331. *Tsinghua University Press*, January 2010.
- [27] Seung Geol Choi, Jonathan Katz, Dominique Schröder, Arkady Yerukhimovich, and Hong-Sheng Zhou. *(Efficient) universally composable oblivious transfer using a minimal number of stateless tokens*. In *Yehuda Lindell, editor, TCC 2014*, volume 8349 of *LNCS*, pages 638–662. *Springer, Heidelberg*, February 2014.
- [28] Ran Cohen, Sandro Coretti, Juan A. Garay, and Vassilis Zikas. *Probabilistic termination and composability of cryptographic protocols*. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference*, Santa Barbara, CA, USA, August 14-18, 2016, *Proceedings, Part III*, pages 240–269, 2016.
- [29] Graham Cormode and S. Muthukrishnan. *An improved data stream summary: the count-min sketch and its applications*. *J. Algorithms*, 55(1):58–75, 2005.
- [30] Ivan Damgård and Yuval Ishai. *Constant-round multiparty computation using a black-box pseudorandom generator*. In *Advances in Cryptology - CRYPTO 2005*:

25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings, pages 378–394, 2005.

- [31] Yevgeniy Dodis, Rafail Ostrovsky, Leonid Reyzin, and Adam Smith. *Fuzzy extractors: How to generate strong keys from biometrics and other noisy data*. SIAM J. Comput., 38(1):97–139, March 2008.
- [32] Danny Dolev, Cynthia Dwork, Orli Waarts, and Moti Yung. *Perfectly secure message transmission*. In 31st FOCS, pages 36–45. IEEE Computer Society Press, October 1990.
- [33] H Fleischner. *X. 1 algorithms for eulerian trails*. Eulerian Graphs and Related Topics: Part 1 (Annals of Discrete Mathematics), 2(50):1–13, 1991.
- [34] M Fleury. *Deux problèmes de géométrie de situation*. Journal de mathématiques élémentaires, 2:257–261, 1883.
- [35] O. Goldreich and L. A. Levin. *A hard-core predicate for all one-way functions*. In Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing, STOC '89, pages 25–32, New York, NY, USA, 1989. ACM.
- [36] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. *How to construct random functions*. J. ACM, 33(4):792–807, 1986.
- [37] Philippe Golle and Ari Juels. *Dining cryptographers revisited*. In Christian Cachin and Jan Camenisch, editors, EUROCRYPT 2004, volume 3027 of LNCS, pages 456–473. Springer, Heidelberg, May 2004.
- [38] Vipul Goyal, Yuval Ishai, Amit Sahai, Ramarathnam Venkatesan, and Akshay Wadia. *Founding cryptography on tamper-proof hardware tokens*. In Theory of Cryptography, 7th Theory of Cryptography Conference, TCC 2010, Zurich, Switzerland, February 9-11, 2010. Proceedings, pages 308–326, 2010.
- [39] Moritz Hardt and David P. Woodruff. *How robust are linear sketches to adaptive inputs?* In Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013, pages 121–130, 2013.
- [40] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. *Mining your ps and qs: Detection of widespread weak keys in network devices*. In Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012, pages 205–220, 2012.
- [41] Markus Hinkelmann and Andreas Jakoby. *Communications in unknown networks: Preserving the secret of topology*. Theor. Comput. Sci., 384(2-3):184–200, 2007.
- [42] Martin Hirt, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. *Network-hiding communication and applications to multi-party protocols*. In Matthew Robshaw and Jonathan Katz, editors, CRYPTO 2016, Part II, volume 9815 of LNCS, pages 335–365. Springer, Heidelberg, August 2016.

- [43] *Martin Hirt, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Network-hiding communication and applications to multi-party protocols. In Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II, pages 335–365, 2016.*
- [44] *Dennis Hofheinz, Jörn Müller-Quade, and Dominique Unruh. Universally composable zero-knowledge arguments and commitments from signature cards. In IN PROC. OF THE 5TH CENTRAL EUROPEAN CONFERENCE ON CRYPTOLOGY MORAVIACRYPT 2005, 2005.*
- [45] *Piotr Indyk. Stable distributions, pseudorandom generators, embeddings and data stream computation. In 41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA, pages 189–197, 2000.*
- [46] *Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998, pages 604–613, 1998.*
- [47] *T. S. Jayram, Ravi Kumar, and D. Sivakumar. The one-way communication complexity of hamming distance. Theory of Computing, 4(1):129–135, 2008.*
- [48] *Daniel M. Kane and R. Ryan Williams. The orthogonal vectors conjecture for branching programs and formulas. CoRR, abs/1709.05294, 2017.*
- [49] *Harini Kannan, Alexey Kurakin, and Ian J. Goodfellow. Adversarial logit pairing. CoRR, abs/1803.06373, 2018.*
- [50] *Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings, pages 477–498, 2013.*
- [51] *Eyal Kushilevitz, Rafail Ostrovsky, and Yuval Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, STOC '98, pages 614–623, New York, NY, USA, 1998. ACM.*
- [52] *Rio LaVigne, Andrea Lincoln, and Virginia Vassilevska Williams. Public-key cryptography in the fine-grained setting. IACR Cryptology ePrint Archive, 2019:625, 2019.*
- [53] *Rio LaVigne, Chen-Da Liu Zhang, Ueli Maurer, Tal Moran, Marta Mularczyk, and Daniel Tschudi. Topology-hiding computation beyond semi-honest adversaries. In Theory of Cryptography - 16th International Conference, TCC 2018, Panaji, India, November 11-14, 2018, Proceedings, Part II, pages 3–35, 2018.*

- [54] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. *Efficient constant round multi-party computation combining BMR and SPDZ*. In Rosario Gennaro and Matthew J. B. Robshaw, editors, CRYPTO 2015, Part II, volume 9216 of LNCS, pages 319–338. Springer, Heidelberg, August 2015.
- [55] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. *Towards deep learning models resistant to adversarial attacks*. CoRR, abs/1706.06083, 2017.
- [56] S. A. M. Makki. *A distributed algorithm for constructing an eulerian tour*. In 1997 IEEE International Performance, Computing and Communications Conference, pages 94–100, Feb 1997.
- [57] Ilya Mironov, Moni Naor, and Gil Segev. *Sketching in adversarial environments*. In Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008, pages 651–660, 2008.
- [58] Jayadev Misra and David Gries. *Finding repeated elements*. Sci. Comput. Program., 2(2):143–152, 1982.
- [59] Tal Moran, Ilan Orlov, and Silas Richelson. *Topology-hiding computation*. In Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part I, pages 159–181, 2015.
- [60] Tal Moran, Ilan Orlov, and Silas Richelson. *Topology-hiding computation*. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, TCC 2015, Part I, volume 9014 of LNCS, pages 159–181. Springer, Heidelberg, March 2015.
- [61] J. Ian Munro and Mike Paterson. *Selection and sorting with limited storage*. Theor. Comput. Sci., 12:315–323, 1980.
- [62] Moni Naor and Eylon Yogev. *Bloom filters in adversarial environments*. In Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II, pages 565–584, 2015.
- [63] Moni Naor and Moti Yung. *Universal one-way hash functions and their cryptographic applications*. In Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washington, USA, pages 33–43, 1989.
- [64] Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. *Certified defenses against adversarial examples*. CoRR, abs/1801.09344, 2018.
- [65] Sivaramakrishnan Natarajan Ramamoorthy and Makrand Sinha. *On the communication complexity of greater-than*. In 53rd Annual Allerton Conference on Communication, Control, and Computing, Allerton 2015, Allerton Park & Retreat Center, Monticello, IL, USA, September 29 - October 2, 2015, pages 442–444, 2015.

- [66] Oded Regev. *Lattice-based cryptography (invited talk)*. In Cynthia Dwork, editor, CRYPTO 2006, volume 4117 of LNCS, pages 131–141. Springer, Heidelberg, August 2006.
- [67] Michael K. Reiter and Aviel D. Rubin. *Crowds: Anonymity for web transactions*. ACM Trans. Inf. Syst. Secur., 1(1):66–92, 1998.
- [68] Aman Sinha, Hongseok Namkoong, and John C. Duchi. *Certifiable distributional robustness with principled adversarial training*. CoRR, abs/1710.10571, 2017.
- [69] Paul F. Syverson, David M. Goldschlag, and Michael G. Reed. *Anonymous connections and onion routing*. In 1997 IEEE Symposium on Security and Privacy, May 4-7, 1997, Oakland, CA, USA, pages 44–54, 1997.
- [70] Andrew Chi-Chih Yao. *Some complexity questions related to distributive computing (preliminary report)*. In Proceedings of the 11h Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1979, Atlanta, Georgia, USA, pages 209–213, 1979.