Topology-Hiding Computation for Networks with Unknown Delays

Rio LaVigne[1*], Chen-Da Liu-Zhang[2], Ueli Maurer[2], Tal Moran[3**], Marta Mularczyk[2***], and Daniel Tschudi[4†]

[1] rio@mit.edu, MIT
[2] {lichen,maurer,mumarta}@inf.ethz.ch, ETH Zurich
[3] talm@idc.ac.il, IDC Herzliya
[4] dt@concordium.com, Concordium

Abstract. Topology-Hiding Computation (THC) allows a set of parties to securely compute a function over an incomplete network without revealing information on the network topology. Since its introduction in TCC'15 by Moran et al., the research on THC has focused on reducing the communication complexity, allowing larger graph classes, and tolerating stronger corruption types. All of these results consider a fully synchronous model with a known upper bound on the maximal delay of all communication channels. Unfortunately, in any realistic setting this bound has to be extremely large, which makes all fully synchronous protocols inefficient. In the literature on multi-party computation, this is solved by considering the fully asynchronous model. However, THC is unachievable in this model (and even hard to define), leaving even the definition of a meaningful model as an open problem.

The contributions of this paper are threefold. First, we introduce a meaningful model of unknown and random communication delays for which THC is both definable and achievable. The probability distributions of the delays can be arbitrary for each channel, but one needs to make the (necessary) assumption that the delays are independent. The existing fully-synchronous THC protocols do not work in this setting and would, in particular, leak information about the topology. Second, in the model with trusted stateless hardware boxes introduced at Eurocrypt'18 by Ball et al., we present a THC protocol that works for any graph class. Third, we explore what is achievable in the standard model without trusted hardware and present a THC protocol for specific graph types (cycles and trees) secure under the DDH assumption. The speed of all protocols scales with the actual (unknown) delay times, in contrast to all previously known THC protocols whose speed is determined by the assumed upper bound on the network delay.

1 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. [MOR15], who showed that THC is possible in the setting with passive corruptions and graphs with logarithmic diameter. Further solutions improve the communication efficiency [HMTZ16] or allow for larger classes of graphs

* This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1122374. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors(s) and do not necessarily reflect the views of the National Science Foundation. Research also supported in part by NSF Grants CNS-1350619 and CNS-1414119, and by the Defense Advanced Research Projects Agency (DARPA) and the U.S. Army Research Office under contracts W911NF-15-C-0226 and W911NF-15-C-0236.

** Supported in part by ISF grant no. 1790/13 and by the Bar-Ilan Cyber-center.

*** Research supported by the Zurich Information Security and Privacy Center (ZISC).

† Work was done while author was at Aarhus University, supported by advanced ERC grant MPCPRO.

[AM17, ALM17]. Recent results [BBMM18, LLM$^+$18] 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.

1.1 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. [BOCG93, Can01]). 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 A 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 2. 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) 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 [AM17]): cycles and trees, respectively.[5] Furthermore, observe that the running time of a protocol could itself leak information about the topology. Indeed, this issue seems very difficult to overcome, 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 4, 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 [BBMM18] in order to deal with fail-stop adversaries in THC. Similar assumptions have also been considered before,

[5] Our second protocol works for any graphs, as long as we agree to reveal a spanning tree: the parties know which of their edges are on the tree and execute the protocol, ignoring other edges. See also [AM17].

for example, stateless tamper-proof tokens [CGS08, GIS+10, CKS+14][6], or honestly-generated secure hardware [HMQU05, CT10].

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 [BBMM18], which was later relaxed to standard assumptions [LLM+18], or the Signature Card assumption for proofs-carrying-data schemes [CT10]). We hope that the techniques presented in this paper can be useful to construct protocols in more standard models.

1.2 Related Work

Topology-hiding computation was introduced by Moran et al. in [MOR15]. 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 [HMTZ16], 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 [AM17]. This result was finally extended to arbitrary (connected) graphs in [ALM17]. These results were extended to the fail-stop setting in [BBMM18] based on stateless secure hardware, and [LLM+18] 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 [HJ07]: 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 which leaks no additional information.

2 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.

2.1 Impossibility of Stronger Models

Common models for asynchronous communication [BOCG93, Can01] 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 [MOR15]. We formally prove this in a very weak adversarial model in Appendix A.

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

[6] The 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.

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.

2.2 Adversary

We consider an adversary, who statically and passively corrupts any set $\mathcal{Z} \subseteq \mathcal{P} = \{P_1, \ldots, P_n\}$ 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.

2.3 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 [KMTZ13], 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 $\mathcal{F}_{\text{NET}}$. 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.[7]

[7] In 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.

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 $\mathcal{F}_{\text{NET}}$

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 $P_j \in \mathbf{N}_G(P_i)$, $\mathcal{F}_{\text{NET}}$ 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`. [8]
- On input $(\text{FetchMessages}, i)$ 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 $\mathcal{F}_{\text{NET}}$. 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 $\mathcal{F}_{\text{NET}}$, 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 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 $j \in \mathbf{N}_G(P_i)$.

[8] Technically, 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.

– 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 $\mathcal{L}(\mathsf{aux}, \mathcal{D})$, where $\mathcal{D}$ is the set of all delay distributions in the network.

2.4 Additional Related Work

Katz et al. [KMTZ13] 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, $\mathcal{F}_{\text{NET}}$ 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. [CCGZ16] 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 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 [ALM17]. 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.

3.1 Synchronous THC from Random Walks

Currently, the most efficient fully-synchronous THC protocols are based on the technique of correlated random walks, introduced in [ALM17]. 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 $\mathtt{pk}_{12} = \mathtt{pk}_1 \circledast \mathtt{pk}_2$. The encryption and decryption algorithms are denoted $\mathsf{PKCR.Enc}(m, \mathtt{pk})$ and $\mathsf{PKCR.Dec}(c, \mathtt{sk})$, respectively. Additionally, a party can add a layer of encryption to a ciphertext c encrypted under $\mathtt{pk}_1$, using the algorithm $\mathsf{PKCR.AddLayer}(c, \mathtt{sk}_2)$, which outputs an encryption c' under the combined key $\mathtt{pk}_{12}$. This operation can be undone with $\mathsf{PKCR.DelLayer}(c', \mathtt{sk}_2)$. 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.

3.2 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 key[9]. We formally define PKCR* and give an instantiation based on the DDH assumption in Appendix B.

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 sent[10].This means that at time τ, we are on round $\mathsf{r}_\tau = \lfloor \tau/R \rfloor$; 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 $\mathtt{sk}_i$ and the combined public key $\mathtt{pk} = \mathtt{pk}_1 \circledast \ldots \circledast \mathtt{pk}_n$. 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($b_i, \mathtt{pk}$). 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. That is, every time it receives a ciphertext c from the previous neighbor, it deletes the layer of encryption using its secret key: PKCR*.DelLayer($c, \mathtt{sk}_i$). 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 2, the delays could be variable, but we make the assumption that if messages are sent at least

[9] Its functionality does not matter and is left undefined on encryptions of 1.

[10] If 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.

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 $\mathrm{Med}[D_e]$. Now, since the protocol will handle messages in rounds, the actual values we need to consider are all in rounds: $\lceil \mathrm{Med}[D_e]/R \rceil$.

Now, if over κ rounds, P_1 sends a message c each round, the probability that none of the copies arrives after $\kappa + \lceil \mathrm{Med}[D_e]/R \rceil$ rounds is negligible in terms of κ, the security parameter (see Lemma 1 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 $(\kappa + \lceil \mathrm{Med}[D_e]/R \rceil) \cdot R$ has passed from when the original message was sent before processing it.[11]

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 $\mathtt{pk} = \mathtt{pk}_1 \circledast \ldots \circledast \mathtt{pk}_n$. At each step, processing a message will involve using the $\mathsf{PKCR.DelLayer}$ 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 $\mathtt{pk}$ to P_2, let's call this message $c_1^{(1)}$. P_1 will wait $\mathsf{w} = \kappa + \lceil \mathrm{Med}[D_e]/R \rceil$ 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 $c_1^{(1)}, \ldots c_\kappa^{(1)}$ 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, and so on.

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

$$R \cdot \sum_{i=1}^{n} \mathsf{w}_i = R\left(n\kappa + \mathrm{MedRSum}[\mathrm{D}]\right).$$

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

Formally, the protocol $\mathsf{CycleProt}$ is described as follows.

Protocol $\mathsf{CycleProt}$

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

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

Initialization for each P_i:

- Send (GetInfo) to the functionality $\mathcal{F}_{\text{NET}}$ and assign randomly the labels P^0, P^1 to the two neighbors.
- Let $\mathsf{Rec}^0, \mathsf{Rec}^1$ 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 $\varnothing$; 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 $\mathsf{w}^\ell = \kappa + \left\lceil \mathrm{Med}[D_{(i,\ell)}]/R \right\rceil$ 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 $\tau \mod R \neq 0$, send (Ready) to the functionality $\mathcal{F}_{\text{CLOCK}}$. Otherwise, let $\mathsf{r} = \tau/R$ be the current round number and do the following:
2: *Receive messages:* Send (FetchMessages, i) to the functionality $\mathcal{F}_{\text{NET}}$. For each message (r_c, c) received from a neighbor P^ℓ, set $\mathsf{Rec}^\ell[\mathsf{r}_c + \mathsf{w}^\ell] = c$.

[11] Note 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.

3: *Process if no messages received:* For each neighbor P^ℓ such that $\mathsf{Rec}^\ell[\mathsf{r}] = \bot$, start a new cycle traversal in the direction of $P^{1-\ell}$:
  – If P_i is sender (i.e. $i = s$) then add $(\kappa, \mathsf{r}, \mathsf{PKCR^*.Enc}(b_s, \mathtt{pk}))$ to $\mathsf{Send}^{1-\ell}$.
  – Otherwise, add $(\kappa, \mathsf{r}, \mathsf{PKCR^*.Enc}(0, \mathtt{pk}))$ to $\mathsf{Send}^{1-\ell}$.

4: *Process received messages:* For each P^ℓ such that $\mathsf{Rec}^\ell[\mathsf{r}] \neq \bot$ (we have received a message from P^ℓ), set $\mathsf{d} = \mathsf{PKCR^*.DelLayer}(R^\ell[\mathsf{r}], \mathtt{sk}_i)$, and do the following:
  – If $\mathsf{d} \in \{0, 1\}$, output d and halt (we have decrypted the source bit).
  – Otherwise, if $i = s$ and $b_s = 1$, then set $\mathsf{d} = \mathsf{PKCR^*.ToOne}(\mathsf{d})$. Then, in either case, add $(\kappa, \mathsf{r}, \mathsf{PKCR^*.Rand}(\mathsf{d}))$ to $\mathsf{Send}^{1-\ell}$.

5: *Send message:* For each $\ell \in \{0, 1\}$, let $\mathsf{Sending}^\ell = \{(k, \mathsf{r}_c, c) \in \mathsf{Send}^\ell : k > 0\}$. For each $(k, \mathsf{r}_c, c) \in \mathsf{Sending}^\ell$, send (r_c, c) to P^ℓ.

6: *Update* Send *set:* For each $(k, \mathsf{r}_c, c) \in \mathsf{Sending}^\ell$, remove (k, r_c, c) from Send^ℓ and insert $(k-1, \mathsf{r}_c, c)$ to Send^ℓ.

7: Send (READY) to the functionality $\mathcal{F}_{\text{CLOCK}}$.

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

Theorem 1. *The protocol* $\mathsf{CycleProt}$ *UC-realizes* $(\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{INFO}}^{\mathcal{L}_{\mathsf{median}}}, \mathcal{F}_{\text{BC}})$ *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* $\mathcal{L}_{\mathsf{median}}(R, \mathcal{D}) = \mathrm{MedRSum}[\mathrm{D}]$.[12]

3.3 Protocol for Trees

We show how to modify the cycle protocol presented in the previous section to securely realize the broadcast functionality $\mathcal{F}_{\text{BC}}$ in any tree. As observed in [AM17], 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 $\mathtt{sk}_i$ 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 D we give a formal description of the protocol $\mathsf{TreeProt}$. The proof of the following theorem is a straightforward extension of the proof of Theorem 1.

Theorem 2. *The protocol* $\mathsf{TreeProt}$ *UC-realizes* $(\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{INFO}}^{\mathcal{L}_{\mathsf{median}}}, \mathcal{F}_{\text{BC}})$ *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* $\mathcal{L}_{\mathsf{median}}(R, \mathcal{D}) = \mathrm{MedRSum}[\mathrm{D}]$.

4 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 [BBMM18]: we assume that the parties have access to secure hardware boxes, initialized with the same secret key, and executing the same functionality $\mathcal{F}_{\text{HW}}$, independent of the graph and the realized functionality (see [BBMM18] for details of this model).

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.[13] 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

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

[13] In 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.

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 $\mathcal{F}_{\mathrm{HW}}$ executed by the hardware boxes, to Appendix E, and for now only give an informal description of its behavior whenever $\mathcal{F}_{\mathrm{HW}}$ is invoked.

4.1 Preprocessing

The preprocessing is executed without any inputs. The output is a pair $(\mathtt{id}_i, c)$, where $\mathtt{id}_i$ 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 $\mathtt{id}_i$ 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 $\mathcal{F}_{\mathrm{HW}}$ box to update c with the information from c_j. That is, $\mathcal{F}_{\mathrm{HW}}$ 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 $\mathcal{F}_{\mathrm{HW}}$.

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

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 (FetchMessages, i) to $\mathcal{F}_{\mathrm{NET}}$. For each received message c', input $(\mathtt{id}_i, c, c')$ to $\mathcal{F}_{\mathrm{HW}}$ and set the updated state c to the result.

Termination for each P_i: Upon receiving the signal, output $(\mathtt{id}_i, c)$.

4.2 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 $\mathtt{id}_i$ and c_i, outputted by preprocessing.

The high-level idea is that the hardware box $\mathcal{F}_{\mathrm{HW}}$ 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 $\texttt{id}$'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 $\texttt{id}_i$. 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 $\texttt{id}_i$'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 4.3 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 $(x_i, f_i, \texttt{id}_i, c_i)$, where $\texttt{id}_i$ 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 $\mathcal{F}_{\text{HW}}$.

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

Execution for each P_i at every r clock ticks:

1: Send (FetchMessages) to $\mathcal{F}_{\text{NET}}$ and receive the messages $(E_1, \ldots, E_\nu)$.
2: Choose r at random and input $(i, \texttt{id}_i, c_i, \bigcup_j E_j, x_i, f_i, r)$ to $\mathcal{F}_{\text{HW}}$. Get the result $(\texttt{val}, \{(E'_1, \texttt{next}_1), \ldots, (E'_k, \texttt{next}_k)\})$. If $\texttt{val} \neq \perp$, output $\texttt{val}$, but continue running.
3: For each $(E'_j, \texttt{next}_j)$, for each $e \in E'_j$, send e to $\texttt{next}_j$ via $(\text{Send}, i, \texttt{next}_j, e)$.[14]

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 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

[14] We 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.

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, 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 $\mathsf{Hardware}$ 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 F we prove the following theorem.

Theorem 3. *For any efficiently computable and well-formed*[15] *functionality $\mathcal{F}$, the protocol* $\mathsf{Hardware}$ *UC-realizes $(\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{INFO}}^{\mathcal{L}_{\mathsf{sum}}}, \mathcal{F})$ 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 $\mathcal{L}_{\mathsf{sum}} := R \sum_{D_e \in \mathcal{D}} \lceil D_e/R \rceil$.*

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 [DI05, LPSY15].

4.3 Computing the Eulerian Cycle

It turns out that not every algorithm computing an Eulerian cycle can be used in $\mathcal{F}_{\text{HW}}$ 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.

Common algorithms, such as Fleury or Hierholzer [Fle83, Fle91], 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 [Mak97] 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 1 for an example execution.

[15] Intuitively, a functionality is well-formed if its code does not depend on the ID's of the corrupted parties. We refer to [CLOS02] for a detailed description.

Algorithm $\text{EulerianCycle}(u, G = (E, V))$

// 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.

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

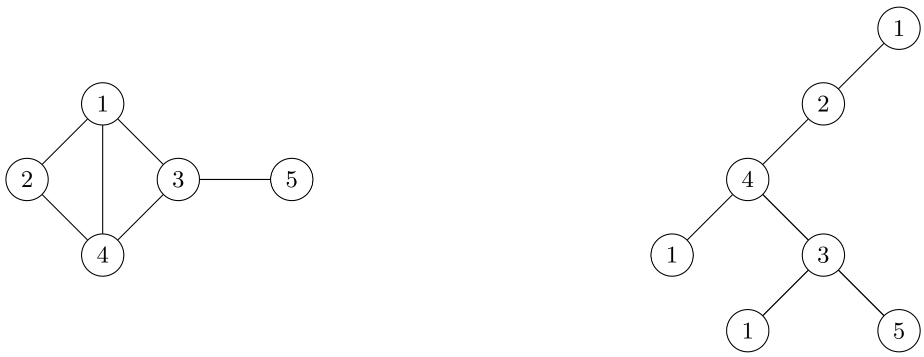

Fig. 1. An example of a graph G (on the left) and the corresponding tree $\mathcal{T}$, computed by $\mathsf{EulerianCycle}(1, G)$ (on the right). The eulerian cycle (on the graph with doubled edges) is $(1, 2, 4, 1, 3, 1, 3, 5, 3, 4, 2, 1)$.

References

[ALM17] 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.

[AM17] 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.

[BBDP01] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-privacy in public-key encryption. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 566–582. Springer, Heidelberg, December 2001.

[BBMM18] Marshall Ball, Elette Boyle, Tal Malkin, and Tal Moran. Exploring the boundaries of topology-hiding computation. In *Eurocrypt'18*, 2018.

[BOCG93] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *25th ACM STOC*, pages 52–61. ACM Press, May 1993.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[CCGZ16] Ran Cohen, Sandro Coretti, Juan A. Garay, and Vassilis Zikas. Probabilistic termination and composability of cryptographic protocols. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 240–269. Springer, Heidelberg, August 2016.

[CGS08] Nishanth Chandran, Vipul Goyal, and Amit Sahai. New constructions for uc secure computation using tamper-proof hardware. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 545–562. Springer, 2008.

[CKS$^+$14] 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 *Theory of Cryptography Conference*, pages 638–662. Springer, 2014.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 494–503. ACM, 2002.

[CT10] Alessandro Chiesa and Eran Tromer. Proof-carrying data and hearsay arguments from signature cards. In *ICS*, volume 10, pages 310–331, 2010.

[DI05] Ivan Damgård and Yuval Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, pages 378–394, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[Fle83] M Fleury. Deux problèmes de géométrie de situation. *Journal de mathématiques élémentaires*, 2:257–261, 1883.

[Fle91] H Fleischner. X. 1 algorithms for eulerian trails. *Eulerian Graphs and Related Topics: Part 1 (Annals of Discrete Mathematics)*, 2(50):1–13, 1991.

[GIS+10] Vipul Goyal, Yuval Ishai, Amit Sahai, Ramarathnam Venkatesan, and Akshay Wadia. Founding cryptography on tamper-proof hardware tokens. In *Theory of Cryptography Conference*, pages 308–326. Springer, 2010.

[HJ07] Markus Hinkelmann and Andreas Jakoby. Communications in unknown networks: Preserving the secret of topology. *Theoretical Computer Science*, 384(2-3):184–200, 2007.

[HMQU05] Dennis Hofheinz, Jörn Müller-Quade, and Dominique Unruh. Universally composable zero-knowledge arguments and commitments from signature cards. In *5th Central European Conference on Cryptology*, 2005.

[HMTZ16] 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.

[KMTZ13] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 477–498. Springer, Heidelberg, March 2013.

[LLM+18] 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 Conference, to appear*, 2018.

[LPSY15] 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 Robshaw, editors, *Advances in Cryptology – CRYPTO 2015*, pages 319–338, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[Mak97] SAM Makki. A distributed algorithm for constructing an eulerian tour. In *Performance, Computing, and Communications Conference, 1997. IPCCC 1997., IEEE International*, pages 94–100. IEEE, 1997.

[MOR15] 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.

Supplementary Material

A Adversarially-Controlled Delays Leak Topology

Much like how adversarially-controlled aborts were shown to leak topological information in [MOR15], we can show that adversarially-controlled delays also leak topological 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 [MOR15], using a similar game-based definition and even using the same adversarially-chosen graphs (see figure 2). 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 [MOR15], 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.

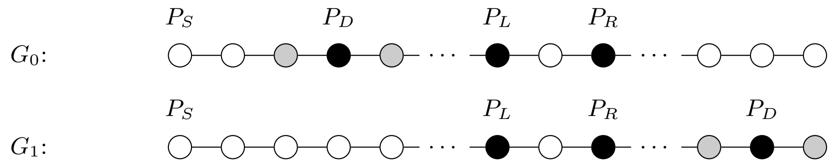

Fig. 2. Graphs used to prove the impossibility of THC with adversarial delays. P_S is the sender. The corrupted parties (black dots) are: P_L and P_R (they delay messages), and the detective P_D. The adversary determines whether P_D (and its two neighbors) are on the left or on the right.

A.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.

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 1. *A* indistinguishability-delay algorithm (IDA) *for a protocol Π,* $\mathsf{DelayAlgorithm}_{\Pi}$*, 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. Explicitely, for any graph $G = (V, E)$,* $\mathsf{DelayAlgorithm}(G)$ *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 [MOR15]. Let $\mathsf{DelayAlgorithm}$ 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 $\mathsf{DelayAlgorithm}(G_b) = \mathcal{T}$ 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 2. *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*

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

A.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 3. *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 4. *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 2.

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 adverary decides to delay along some edge, all subsequent messages along that edge cannot arrive for at least T clock-ticks.

Claim. Consider any party P_v whose neighbors do not add any extra delay as described by the long-delay paragraph above. As in [MOR15], 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

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

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 $\Pr[H_{v,b}|E_0] \leq \frac{1}{2} - \mathrm{negl}(n)$ 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, $\Pr[H_{v,b}|E_0] = \Pr[H_{v,b}|\text{no extra delays}] = 1 - \mathrm{negl}(n)$ for all honest parties without a delaying neighbor by the definition of weakly realizing broadcast.

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

Proof (Theorem 4). This just follows from the previous claim. A simple hybrid argument shows that there exists a pair $(\tau^*, b) \in \{0, \ldots, T-1\} \times \{0, 1\}$ such that

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

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 2 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 $\mathcal{T}_{\mathcal{Z}}$ where $\mathcal{T}_{\mathcal{Z}}((i,j),t) = 0$ for $t < \tau^*$, and for $t \geq \tau^*$: $\mathcal{T}_{\mathcal{Z}}((L, n/2), t) = \mathcal{T}_{\mathcal{Z}}((L, n/2-2), t)T$ and $\mathcal{T}_{\mathcal{Z}}((R, n/2), t+1) = \mathcal{T}_{\mathcal{Z}}((R, n/2+2), t+1) = T$.

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 trasncript.

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. □

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.

B PKCR* Encryption

This section formally defines PKCR*—the extended Privately Key Commutative and Rerandomizable (PKCR) encryption of [AM17].

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, $\mathcal{C} \cap \{0, 1\} = \varnothing$. As in any public-key encryption scheme, we have the algorithms PKCR*.KGen : $\{0,1\}^* \to \mathcal{PK} \times \mathcal{SK}$ and PKCR*.Enc : $\{0,1\} \times \mathcal{PK} \to \mathcal{C}$ 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 $\circledast$. In particular, given any $\mathtt{pk}_1, \mathtt{pk}_2 \in \mathcal{PK}$ and the secret key $\mathtt{sk}_1$ corresponding to $\mathtt{pk}_1$, we can efficiently compute $\mathtt{pk}_3 = \mathtt{pk}_1 \circledast \mathtt{pk}_2 \in \mathcal{PK}$ (note that $\mathtt{sk}_1$ can be replaced by $\mathtt{sk}_2$, since $\mathcal{PK}$ is commutative).
This group must interact well with ciphertexts; there exists a pair of deterministic efficiently computable algorithms $\mathsf{PKCR^*.AddLayer} : \mathcal{C} \times \mathcal{SK} \to \mathcal{C}$ and $\mathsf{PKCR^*.DelLayer} : \mathcal{C} \times \mathcal{SK} \to \mathcal{C} \cup \{0, 1\}$ such that for every pair of public keys $\mathtt{pk}_1, \mathtt{pk}_2 \in \mathcal{PK}$ with corresponding secret keys $\mathtt{sk}_1$ and $\mathtt{sk}_2$, for every bit $b \in \{0, 1\}$, and every ciphertext $c = \mathsf{PKCR^*.Enc}(b, \mathtt{pk}_1)$, with overwhelming probability it holds that:
- The ciphertext $\mathsf{PKCR^*.AddLayer}(c, \mathtt{sk}_2)$ is an encryption of b under the public key $\mathtt{pk}_1 \circledast \mathtt{pk}_2$.
- $\mathsf{PKCR^*.DelLayer}(c, \mathtt{sk}_2)$ is an encryption of b under the public key $\mathtt{pk}_1 \circledast \mathtt{pk}_2^{-1}$.
- $\mathsf{PKCR^*.DelLayer}(c, \mathtt{sk}_1) = b$.

Notice that we need the secret key to perform these operations.[16]

Rerandomizable. There exists an efficient probabilistic algorithm $\mathsf{PKCR^*.Rand} : \mathcal{C} \to \mathcal{C}$, which rerandomizes a ciphertext.[17] Formally, we require that for every public key $\mathtt{pk} \in \mathcal{PK}$, every bit b, and every $c = \mathsf{PKCR^*.Enc}(b, \mathtt{pk})$, the following distributions are computationally indistinguishable:

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

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

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. [BBDP01].

B.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 $\mathtt{sk}$, one simply raises the second element to $\mathtt{sk}$. 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).

PKCR*.KGen

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

PKCR*.Enc(b, y)

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

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

1: Output $(c_1, c_2^{\mathtt{sk}})$.

PKCR*.Rand$((c_1, c_2))$

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

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

1: Set $c_2' = c_2^{\mathtt{sk}^{-1}}$.
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').

PKCR*.ToOne$((c_1, c_2))$

1: Output (c_1^2, c_2).

[16] In PKCR of [ALM17], computing $\mathtt{pk}_1 \circledast \mathtt{pk}_2$ does not require the secret key. Moreover, PKCR requires perfect correctness.

[17] In [ALM17] 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 [BBDP01]). Further, the proof that it satisfies the requirements of rerandomizability and key commutativity is analogous to the proof that the DDH-based construction of [ALM17] satisfies these properties. We refer to [ALM17] for details.

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

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

C Proof of Theorem 1

Simulator. We simulate the outputs of $\mathcal{F}_{\text{NET}}$ on inputs $(\textsc{FetchMessages}, i)$ 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 $\mathtt{buffer}$, and the simulator outputs them by executing the algorithm of $\mathcal{F}_{\text{NET}}$.

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, the protocol has a *deterministic* end time of $T = R\,(n\kappa + \mathrm{MedRSum}[\mathrm{D}])$. Consider a single corrupted arc $P_1, \ldots, P_k$ (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 $(\mathsf{w}_1, \ldots, \mathsf{w}_k)$, 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: $T - \sum_{i=1}^{k} \mathsf{w}_i$. 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 $\mathcal{F}_{\text{BC}}$ and receives the output bit b^{out}.
3. It sends $(\textsc{GetInfo}, R)$ to $\mathcal{F}_{\text{INFO}}^{\mathcal{L}_{\mathsf{median}}}$ 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 $\mathsf{CycleProt}$. To simulate the messages sent by honest parties to their corrupted neighbors, $\mathcal{S}_{cycle}$ proceeds as follows.
5. First, it prepares a set $\mathtt{buffer}$, containing all messages which will be sent by the honest parties throughout the simulation (recall the variable $\mathtt{buffer}$ in $\mathcal{F}_{\text{NET}}$). $\mathcal{S}_{cycle}$ initializes $\mathtt{buffer} = \varnothing$.
6. $\mathcal{S}_{cycle}$ generates the messages per a connected corrupted arc $P^1, P^2, \ldots, P^K$ of the cycle. We will use the following notation:
   - P^0 and P^{K+1}: the neighboring honest parties.
   - $P^{K+2}, \ldots, P^{n-1}$: (the labels of) the rest of the parties on the cycle (their identities are unknown to $\mathcal{S}_{cycle}$).
   - $\mathrm{MedRSum}[\mathrm{D}]$: the distribution corresponding to the median-round-sum of all delays, obtained from $\mathcal{F}_{\text{INFO}}^{\mathcal{L}_{\mathsf{median}}}$.

– for $0 \leq k \leq n-1$, denote by D_k the delay distribution on the edge from $P^{(k-1) \mod n}$ to P^k (for $1 \leq k \leq K$, D_k was obtained from $\mathcal{F}_{\text{INFO}}^{\mathcal{L}_{\mathsf{median}}}$).
– for $1 \leq k \leq K$, define $\mathsf{w}^k = \kappa + \lceil \text{Med}[D_{(k,k+1)}]/R \rceil$ (recall the initialization step of $\mathsf{CycleProt}$).
– for $1 \leq k \leq K$, denote by $\mathtt{pk}^k$ the public key corresponding to the secret keys of the corrupted parties $P^1, \ldots, P^k$.
– $\mathtt{pk}_{sim}$: 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 $t^k = R\big(n\kappa + \text{MedRSum[D]} - (\mathsf{w}^1 + \cdots + \mathsf{w}^k)\big)$.
2: Let $t^0 = R\,(n\kappa + \text{MedRSum[D]})$.
3: **for** $\tau = t^0$ to 0 and $\tau \equiv 0 \mod R$ **do**
4: **if** $\tau < t^K$ **then**
5: Compute $c = \mathsf{PKCR^*.Enc}(0, \mathtt{pk}_{sim})$.
6: **else**
7: Find k such that $t^{k+1} \leq \tau < t^k$.
8: Compute $c = \mathsf{PKCR^*.Enc}(b^{out}, \mathtt{pk}^k)$.
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. $\mathcal{S}_{cycle}$ simulates the messages received by corrupted parties from $\mathcal{F}_{\text{NET}}$ by executing the algorithm of $\mathcal{F}_{\text{NET}}$. On every input ($\textsc{FetchMessages}, j$) 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 $\mathsf{CycleProt}$: 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 1. *In the real execution of the protocol* $\mathsf{CycleProt}$*, 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 $\mathsf{r}_c + \kappa + \lceil \text{Med}[D_{(i,\ell)}]/R \rceil$ 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 $T = R(\mathsf{r}_c + \kappa + \lceil \text{Med}[D_{(i,\ell)}]/R \rceil)$. Since the message was sent at time $t^j_{sent} = R(\mathsf{r}_c + j)$, with probability 1/2 the message arrives at P_i by time $t^j_{sent} + \text{Med}[D_{(i,\ell)}]$, and is officially delivered at the next round, time $t^j_{sent} + \lceil \text{Med}[D_{(i,\ell)}]/R \rceil \cdot R$. Note that for every j, time T is greater than or equal to $t^j_{sent} + \lceil \text{Med}[D_{(i,\ell)}]/R \rceil \cdot R$. 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

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

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

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 2. *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 1, 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 $0, \ldots, \mathsf{w}^k - 1$. A walk started at a given round is processed by the party $P^{k+k'}$ after $\sum_{i=1}^{k'} \mathsf{w}^{k+i}$ rounds. So it is processed by P^0 after $\sum_{i=0}^{n-1} \mathsf{w}^i - (\mathsf{w}^1 + \cdots + \mathsf{w}^k)$ rounds. We have $\sum_{i=0}^{n-1} \mathsf{w}^i = \text{MedRSum}[\text{D}]$. Hence, P^0 processes messages from the walk started by P^k after time $R \cdot (\text{MedRSum}[\text{D}] - \sum_{i=1}^{k} \mathsf{w}^i) = t^k$. ■

Remark 1. The protocol is correct. Note that Lemma 2 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 $\mathcal{S}_{cycle}$ 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. $\mathcal{S}^1_{cycle}$ simulates the real world exactly. That is, $\mathcal{S}^1_{cycle}$ 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. $\mathcal{S}^2_{cycle}$ generates all messages upfront the same way $\mathcal{S}_{cycle}$ does, but the messages are still generated according to the protocol.

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

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

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

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

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 $\mathcal{S}^6_{cycle}$ is the original simulator $\mathcal{S}_{cycle}$.

Claim 1. *No efficient distinguisher can distinguish between Hybrids 2 and 3.*

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

- An initial ciphertext (starting the cycle): this is the same as in $\mathcal{S}^3_{cycle}$.
- A ciphertext c, which results from applying to another ciphertext c', in order, $\mathsf{PKCR^*.DelLayer}$, and $\mathsf{PKCR^*.Rand}$. By correctness, c is an encryption of the same message as c' with overwhelming probability. Moreover, re-randomizability of $\mathsf{PKCR^*}$, guarantees that the distribution of c is indistinguishable from the distribution of a fresh encryption of the same message, as generated by $\mathcal{S}^3_{cycle}$.
- A ciphertext c, which results from applying to another ciphertext c', in order, $\mathsf{PKCR^*.DelLayer}$, $\mathsf{PKCR^*.ToOne}$, and $\mathsf{PKCR^*.Rand}$. 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 2. *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 2, by time t^k, in the real world, the messages from that walk would have traversed all parties except $P_1, \ldots, P_{k-1}$. So, by that time, all key-layers except those from corrupted parties $P_1, \ldots, P_{k-1}$ will have been removed, meaning that message is now encrypted under public key $\mathtt{pk}^k$. 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 2, 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 [BBDP01] 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: $\mathtt{pk}_0, \mathtt{sk}_0 = \mathsf{PKCR^*.KGen}(1^\kappa)$
2: $\mathtt{pk}_1, \mathtt{sk}_1 = \mathsf{PKCR^*.KGen}(1^\kappa)$
3: Choose $b \in \{0,1\}$ at random.
4: $b' = \mathcal{A}^{\mathsf{PKCR^*.Enc}(\cdot, \mathtt{pk}_b)}(\mathtt{pk}_0, \mathtt{pk}_1)$
5: Output $b = b'$.

Recall that, for each honest party P_i, $\mathcal{S}^3_{cycle}$ uses the real key corresponding to a number of parties on the cycle, while $\mathcal{S}^4_{cycle}$ uses a different key $\mathtt{pk}_{sim}$. We will introduce a sequence of intermediate hybrids H_1 to H_n, where H_i uses the real key for $P_1, \ldots, P_i$, 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 $\mathtt{pk}_0, \mathtt{pk}_1$ 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 $P^1, \ldots, P^K$ 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 $\mathtt{pk}_0$. 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 3. *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 $\mathtt{pk}_{sim}$ chosen by the simulator. Hence, the indistinguishability follows by semantic security. ■

D 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 $i \in \{1, \ldots, n\}$, let $(\mathtt{pk}_i, \mathtt{sk}_i) = \mathsf{PKCR^*.KGen}(1^\kappa)$. Let $\mathtt{pk} = \mathtt{pk}_1 \circledast \ldots \circledast \mathtt{pk}_n$. The setup outputs to each party P_i its secret key $\mathtt{sk}_i$ and the product public key $\mathtt{pk}$.

Initialization for each P_i:

- Send (GetInfo) to the functionality $\mathcal{F}_{\text{NET}}$ to obtain the distributions of the local edges.
- Assign randomly the labels $P^0, \ldots, P^{\nu-1}$ to its neighbors. Let $\mathtt{succ}(\ell) = \ell + 1 \mod \nu$ denote the index of the successor of neighbor P^ℓ on the tree traversal.
- For each neighbor P^ℓ, initialize a list $\mathsf{Rec}^\ell = \perp$ and a set $\mathsf{Send}^\ell = \varnothing$.
- 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 $\mathsf{w}^\ell = \kappa + \lceil \mathrm{Med}[D_{(i,\ell)}]/R \rceil$ 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 $\tau \mod R \neq 0$, send (Ready) to the functionality $\mathcal{F}_{\text{CLOCK}}$. Otherwise, let $\mathsf{r} = \tau/R$ be the current round number and do the following:
2: *Receive messages:* Send (FetchMessages, i) to the functionality $\mathcal{F}_{\text{NET}}$. For each message (r_c, c) received from a neighbor P^ℓ, set $\mathsf{Rec}^\ell[\mathsf{r}_c + \mathsf{w}^\ell] = c$.
3: *Process messages from P^ℓ with $\ell \neq 0$:*
  - For each $P^\ell \neq P^0$ such that $\mathsf{Rec}^\ell[\mathsf{r}] = \perp$, add $(\kappa, \mathsf{r}, \mathsf{PKCR^*.Enc}(0, \mathtt{pk}))$ to $\mathsf{Send}^{\mathsf{succ}(\ell)}$.
  - For each $P^\ell \neq P^0$ such that $\mathsf{Rec}^\ell[\mathsf{r}] \neq \perp$, add $(\kappa, \mathsf{r}, \mathsf{PKCR^*.Rand}(\mathsf{Rec}^\ell[\mathsf{r}]))$ to $\mathsf{Send}^{\mathsf{succ}(\ell)}$.
4: *Process if no messages received from P^0:* If $\mathsf{Rec}^0[\mathsf{r}] = \perp$, start a new cycle traversal in the direction of P^1:
  - If P_i is sender (i.e. $i = s$) then add $(\kappa, \mathsf{r}, \mathsf{PKCR^*.Enc}(b_s, \mathtt{pk}))$ to Send^1.
  - Otherwise, add $(\kappa, \mathsf{r}, \mathsf{PKCR^*.Enc}(0, \mathtt{pk}))$ to Send^1.
5: *Process received messages from P^0:* Set $\mathsf{d} = \mathsf{PKCR^*.DelLayer}(\mathsf{Rec}^0[\mathsf{r}], \mathtt{sk}_i)$, and do the following:
  - If $\mathsf{d} \in \{0, 1\}$, output d and halt (we have decrypted the source bit).
  - Otherwise, if $i = s$ and $b_s = 1$, then set $\mathsf{d} = \mathsf{PKCR^*.ToOne}(\mathsf{d})$. Then, in either case, add $(\kappa, \mathsf{r}, \mathsf{PKCR^*.Rand}(\mathsf{d}))$ to Send^1.
6: *Send messages:* For each $\ell \in \{0, \ldots, \nu - 1\}$, let $\mathsf{Sending}^\ell = \{(k, \mathsf{r}_c, c) \in \mathsf{Send}^\ell : k > 0\}$. For each $(k, \mathsf{r}_c, c) \in \mathsf{Sending}^\ell$, send (r_c, c) to P^ℓ.
7: *Update* Send *sets:* For each $(k, \mathsf{r}_c, c) \in \mathsf{Sending}^\ell$, remove (k, r_c, c) from Send^ℓ and insert $(k - 1, \mathsf{r}_c, c)$ to Send^ℓ.
8: Send (Ready) to the functionality $\mathcal{F}_{\text{CLOCK}}$.

E The Function Executed by the Hardware Boxes

The functionality $\mathcal{F}_{\text{HW}}$ 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 $\mathsf{prf}_{rk}(x)$, where x is the whole input of $\mathcal{F}_{\text{HW}}$. $\mathcal{F}_{\text{HW}}$ can receive three types of input, depending on the current stage of the protocol: the initial input and an intermediate input during $\mathsf{Hw\text{-}Preprocessing}$, and an intermediate input during $\mathsf{Hw\text{-}Computation}$. On any other inputs, $\mathcal{F}_{\text{HW}}$ outputs $\perp$.

Behavior during preprocessing. During the preprocessing, the first input is a triple $(i, \mathtt{id}_i, \mathbf{N}_G(P_i))$, and next inputs are triples $(\mathtt{id}, c, c_j)$, 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),
- $\mathtt{ID} = (\mathtt{id}_1, \ldots, \mathtt{id}_n)$: a vector, containing the currently known identifiers of parties.

On the first input, $\mathcal{F}_{\text{HW}}$ outputs an encryption of the initial state, that is, the state where the graph G contains only the direct neighborhood of P_i, and $\mathtt{ID}$ contains only the value $\mathtt{id}_i$ chosen by P_i. For the inputs of the form $(\mathtt{id}, c, c_j)$, $\mathcal{F}_{\text{HW}}$ 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 $\mathcal{F}_{\text{HW}}$ 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: $(i, G, \mathtt{ID})$ 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 $\mathtt{id}$)

– $\texttt{ID}_j$ is the vector of unique random 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.
– $\texttt{Path}_j = (\texttt{id}^1, \ldots, \texttt{id}^{4n^2})$: 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.
– $\vec{x}_j$ 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, $\mathcal{F}_{\text{HW}}$ 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, $\mathcal{F}_{\text{HW}}$ 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 formated dummy encryptions, so that exactly n encryptions are sent to each neighbor.

The functionality $\mathcal{F}_{\text{HW}}$ 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-dest- ination, 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 $\mathcal{F}_{\text{HW}}$

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

Initial input during Hw-Preprocessing

Input: $x = (i, \texttt{id}_i, \mathbf{N}_G(P_i))$
1: Compute the initial vector $\texttt{ID}$ as a vector of n $\perp$'s except with $\texttt{id}_i$ 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 $s = (i, \texttt{ID}, G_i)$
Output: the encrypted initial state $\mathsf{AE.Enc}_{pk}(s; \mathsf{prf}_{rk}(x))$.

Intermediate input during Hw-Preprocessing

Input: $x = (\texttt{id}, c, c_j)$, where $\texttt{id}$ 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 $(i, \texttt{ID}, G) = \mathsf{AE.Dec}_{pk}(c)$ and $(j, \texttt{ID}_j, G_j) = \mathsf{AE.Dec}_{pk}(c_j)$.
2: Compute the new state $s = (i, \texttt{ID}', G')$, where $\texttt{ID}'$ contains all identifiers which appear in $\texttt{ID}_j$ or $\texttt{ID}$, and G' is the union of G and G_j.
Output: the encrypted state $\mathsf{AE.Enc}_{pk}(s; \mathsf{prf}_{rk}(x))$.

Intermediate input during Hw-Computation

Input: $x = (i, \texttt{id}, c, E, x_i, f_i, r)$, where i is the party's index, $\texttt{id}$ 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.

Decrypt the messages $M = \{\mathsf{AE.Dec}_{pk}(e) \mid e \in E\}$ (output $\perp$ if any decryption fails).
Let $L = \text{AggregateTours}(M)$, and output $\perp$ if AggregateTours outputs $\perp$.
Let $S = \varnothing$, $\texttt{val} = \perp$.
if $L = \varnothing$ **then**// Start the traversal.
    Decrypt the state $(i, \texttt{ID}, G) = \mathsf{AE.Dec}_{pk}(c)$ (output $\perp$ if the decryption fails). // The graph and the ID-vector are taken from the encrypted state.
    Let $\texttt{Path} = (\texttt{id}, \perp, \ldots, \perp)$ be a vector of length $4n^2$. Let $\texttt{x}$ be the vector of length n, initialized to $\perp$ and set $\texttt{x}[i] = x_i$.
    Compute $\texttt{Tour}_i$ as the reverse Euler Cycle for G starting at party P_i.
    Let $m = (i, \texttt{ID}, G, \texttt{Path}, f_i, \texttt{x})$.
    Add $(m, \texttt{Tour}_i[2])$ to S.
else// Continue traversals.
    for $m \in L$ **do**
        Parse $m = (j, \texttt{ID}_j, G_j, \texttt{Path}_j, f_j, \vec{x}_j)$. // The graph and the ID-vector are taken from the message.
        Compute $\texttt{Tour}_j$ as the reverse Euler Cycle for G starting at party P_j.
        Parse $\texttt{Path}_j = (p_1, \ldots, p_{\ell_j}, \perp, \ldots, \perp)$. Output $\perp$ if any of the following conditions holds:
        – $\texttt{id} \neq \texttt{ID}_j[i]$
        – $p_{\ell_j} \neq i$
        – for any $l \in [\ell_j]$, $p_l \neq \texttt{Tour}_j[l \mod 2m]$
        Let $(m', \texttt{next}) = \text{ContinueTour}(m, x_i, i, \texttt{Tour}_j)$.
        if $m' = \texttt{Output}$ **then**
            Let $\texttt{val} = f_i(\vec{x}_j)$.
        else
            Add $(m', \texttt{next})$ to S.
Output : $(\texttt{val}, \text{EncryptAndFormatOutput}(i, G, r, S, 0))$

Functionality $\mathcal{F}_{\text{HW}}$**-subroutines**

AggregateTours (M)
// Takes a set of messages and for each party outputs a message that corresponds to its Euler Cycles.
If any $m \in M$ does not parse properly, return $\perp$.
Let $L = \varnothing$.
for each $m \in M$ **do**
    Parse $m = (j, \texttt{ID}, G, \texttt{Path}, f, \vec{x})$.
    if $\exists m' := (j, *, *, *, *, *) \in L$ and $m' \neq m$ **then**
        Output $\perp$.
    if $m \notin L$ **then**
        Add m to L.
return L

ContinueTour $(m_j, x_i, i, \texttt{Tour}_j)$
Parse $m_j = (j, \texttt{ID}_j, G_j, \texttt{Path}_j, f_j, \vec{x}_j)$.
Parse $\texttt{Path}_j = (p_1, \ldots, p_{\ell_j}, \perp, \ldots, \perp)$.
if $\ell_j = 4m - 1$ and $\texttt{Tour}_j[(\ell_j + 1) \mod 2m] = i$ **then**
    return $(\texttt{Output}, 0)$.
Set $\texttt{Path}_j = (p_1, \ldots, p_{\ell_j}, \texttt{Tour}_j[(\ell_j + 1) \mod 2m], \perp, \ldots, \perp)$.
If $\vec{x}_j[i] = \perp$, then set $\vec{x}_j[i] = x_i$.
return $(m_j, \texttt{Tour}_j[(\ell_j + 1) \mod 2m])$.

EncryptAndFormatOutput $(i, G, r, S, \texttt{sim})$[18]
For each $d \in \mathbf{N}_G(i)$, let $M_d = \{m : (m, d) \in S\}$.
for $d \in \mathbf{N}_G(i)$ **do**

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).
for $d \in \mathbf{N}_G(i)$ **do**
  Let $k = 0$, $E_d = \varnothing$.
  for $m \in M_d$ **do**
    if $\texttt{sim} = 0$ **then**
      Add $\mathsf{AE.Enc}_{pk}(m; \mathsf{prf}_{rk}(M_d, k, r))$ to E_d. // Used in protocol
    else
      Add $\mathsf{AE.Enc}_{pk}(m; r)$ to E_d. // Used in simulator
return $\{(E_d, d) : d \in \mathbf{N}_G(i)\}$

F Proof of Theorem 3

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

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 $\mathcal{F}_{\text{HW}}$. 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 $\mathcal{F}_{\text{BC}}$ 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 $\mathcal{F}_{\text{INFO}}^{\mathcal{L}_{\mathsf{sum}}}$ the distribution $D = \sum_e \lceil D_e/R \rceil R$ 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 $\mathcal{F}_{\text{NET}}$ and the queries to the hardware functionality $\mathcal{F}_{\text{HW}}$.

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 $\mathcal{F}_{\text{NET}}$).

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

$\mathcal{S}_{HW}$ sets $\texttt{buffer} = \varnothing$. 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.

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

Computation:

```
Let $t_{\texttt{start}}$ be the start time of the computation (rounded to the next multiple of $R$), and for each party $P_i \in G_C$, let $t^i_{\texttt{end}}$ be the signal to terminate the execution of the computation phase.
For each honest party $P_i \in G_C$ and corrupted neighbor $P_j \in G_C$, set $S_{ij} = \varnothing$.
// Local delays generated upfront.
For each party $P_i \in G_C$ and neighbor $P_j \in G_C$, let $L_{(i,j)}$ be a list containing $n \cdot \left( \lfloor \frac{t^i_{\texttt{end}} - t_{\texttt{start}}}{R} \rfloor + 1 \right)$ samples from $D_{(i,j)}$. The messages sent by corrupted parties use these delays.
For each $P_j$, let $t^j_{\texttt{stop}} := \min\{t \in L_{(i,j)} : P_i \in \mathbf{N}_G(P_j)\}$ the time at which $P_j$ obtains the first message from any neighbor (stops initiating Eulerian Cycles).
Compute the number of Eulerian Cycles initiated from each corrupted $P_j \in G_C$ as $N_j := \lfloor \frac{t^j_{\texttt{stop}}}{R} \rfloor$.
For each corrupted $P_j \in G_C$, do as follows: Run $\mathcal{E}^j$ =EulerianCycle$(P_j, G_C)$. Let $\mathcal{E}^j_C$ 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 $\mathcal{E}^j_C$. Let $P'_j$ be the first party after $P_j$ in the path.
For each $v \in [N_j]$, for each edge $e \in \mathcal{E}^j_C$, let $d_e$ be the next unused delay. Then, sample $t^v_{\texttt{out}}$ from the distribution $4D - \sum_{e \in \mathcal{E}^j_C} \lceil d_e/R \rceil R$. Add $(P_j, t^v_{\texttt{out}})$ to $S_{ik}$.
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 $S = \varnothing$.
// Message sent at Round 0
For each $w \in [n]$, record the tuple $(t_{\texttt{start}} + L_{(i,k)}[w], P_i, P_k, \mathsf{AE.Enc}_{ek}(\perp))$ in $\texttt{buffer}$, where $\perp$ is a parsable fake state.
// Messages sent at Round r
Let $\mathsf{r} = 1$.
while $t_{\texttt{start}} + \mathsf{r}R < t^i_{\texttt{end}}$ do
    Let $S = \varnothing$ and let $\texttt{x}$ be a vector of length $n$ initialized to 0.
    If there exists $(P_j, t_{\texttt{out}}) \in S_{ik}$ such that $(\mathsf{r} - 1)R \le t_{\texttt{out}} < \mathsf{r}R$, add $\mathsf{AE.Enc}_{ek}((j, \texttt{ID}, G_C, \texttt{Path}, \texttt{x}, f_j))$ to $S$, where $\texttt{ID}$ is the vector of IDs that parties in $G_C$ generated during the preprocessing, and $\texttt{Path} = \mathcal{E}^j \setminus \mathcal{E}^j_C$.
    while $|S| < n$ do
        Add an encryption of a parsable fake state $\mathsf{AE.Enc}_{ek}(\perp; z)$ to $S$.
    Let $S = \{c_1, \ldots, c_n\}$. For each $v \in [n]$, record the tuple $(t_{\texttt{start}} + \mathsf{r}R + L_{(i,k)}[\mathsf{r}n + v], P_i, P_k, c_v)$ in $\texttt{buffer}$.
    $\mathsf{r} = \mathsf{r} + 1$.
```

$\mathcal{S}_{HW}$ simulates the messages received by corrupted parties from $\mathcal{F}_{\text{NET}}$ 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 $\texttt{buffer}$ where $T \le \tau$, it removes the tuple from $\texttt{buffer}$ 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 $\mathcal{F}_{\text{HW}}$. $\mathcal{S}_{HW}$ simulates the queries of the preprocessing phase doing exactly the same as the functionality $\mathcal{F}_{\text{HW}}$.

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

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

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

Hybrid 3. $\mathcal{S}^3_{HW}$ is exactly as $\mathcal{S}^2_{HW}$, 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 $f_j(\vec{x})$ on the input vector.

Hybrid 4. $\mathcal{S}^4_{HW}$, generates the local delays of the messages during the computation phase upfront, but still generates all messages as in $\mathcal{S}^3_{HW}$. 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. $\mathcal{S}^5_{HW}$, 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 $(j, \mathtt{ID}, G, \mathtt{Path}, \mathtt{x}, f_j)$ from the previous neighbor, it generates a new message $(j, \mathtt{ID}, G_C, \mathtt{Path}_j, \mathtt{x}_j, f_j)$ 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 $\mathtt{x}_j$ is the 0 vector.

Hybrid 6. $\mathcal{S}^6_{HW}$, 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 $t^j_{\mathtt{stop}}$ 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. $\mathcal{S}^5_{HW}$: 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 $f_j(\vec{x}) = v_j^{out}$, where v_j^{out} is received from the ideal functionality, and $\vec{x}$ 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 $\vec{x}$ 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 $\mathtt{Path}$ remaining to traverse when the Eulerian cycle is computed on j and G is the same as the part of $\mathtt{Path}_j$ 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 $\mathcal{F}_{\mathrm{HW}}$.
- Hybrids 6 and 7 are indistinguishable by semantic security.

□