

Generic Compiler for Publicly Verifiable Covert Multi-Party Computation

Sebastian Faust¹, Carmit Hazay², David Kretzler¹, and Benjamin Schlosser¹

¹ Technical University of Darmstadt, Germany

{first.last}@tu-darmstadt.de

² Bar-Ilan University, Israel

carmit.hazay@biu.ac.il

Abstract. Covert security has been introduced as a compromise between semi-honest and malicious security. In a nutshell, covert security guarantees that malicious behavior can be detected by the honest parties with some probability, but in case detection fails all bets are off. While the security guarantee offered by covert security is weaker than full-fledged malicious security, it comes with significantly improved efficiency. An important extension of covert security introduced by Asharov and Orlandi (ASIACRYPT’12) is *public verifiability*, which allows the honest parties to create a publicly verifiable certificate of malicious behavior. Public verifiability significantly strengthen covert security as the certificate allows punishment via an external party, e.g., a judge. Most previous work on publicly verifiable covert (PVC) security focuses on the two-party case, and the multi-party case has mostly been neglected. In this work, we introduce a novel compiler for multi-party PVC secure protocols with no private inputs. The class of supported protocols includes the preprocessing of common multi-party computation protocols that are designed in the offline-online model. Our compiler leverages time-lock encryption to offer high probability of cheating detection (often also called deterrence factor) independent of the number of involved parties. Moreover, in contrast to the only earlier work that studies PVC in the multi-party setting (CRYPTO’20), we provide the first full formal security analysis.

Keywords: Covert Security · Multi-Party Computation · Public Verifiability · Time-Lock Puzzles

1 Introduction

Secure multi-party computation (MPC) allows a set of n parties P_i to jointly compute a function f on their inputs such that nothing beyond the output of that function is revealed. Privacy of the inputs and correctness of the outputs need to be guaranteed even if some subset of the parties is corrupted by an adversary. The two most prominent adversarial models considered in the literature are the *semi-honest* and *malicious* adversary model. In the semi-honest model, the adversary is passive and the corrupted parties follow the protocol

description. Hence, the adversary only learns the inputs and incoming/outgoing messages including the internal randomness of the corrupted parties. In contrast, the adversarial controlled parties can arbitrarily deviate from the protocol specification under malicious corruption.

Since in most cases it seems hard (if not impossible) to guarantee that a corrupted party follows the protocol description, malicious security is typically the desired security goal for the design of multi-party computation protocols. Unfortunately, compared to protocols that only guarantee semi-honest security, protection against malicious adversaries results into high overheads in terms of communication and computation complexity. For protocols based on distributed garbling techniques in the oblivious transfer (OT)-hybrid model, the communication complexity is inflated by a factor of $\frac{s}{\log |\mathcal{C}|}$ [WRK17b], where \mathcal{C} is the computed circuit and s is a statistical security parameter. For secret sharing-based protocols, Hazay et al. [HVW20] have recently shown a constant communication overhead over the semi-honest GMW-protocol [GMW87]. In most techniques, the computational overhead grows with an order of s .

In order to mitigate the drawbacks of the overhead required for malicious secure function evaluation, one approach is to split protocols into an input-independent offline and an input-dependent online phase. The input-independent offline protocol carries out pre-computations that are utilized to speed up the input-dependent online protocol which securely evaluates the desired function. Examples for such offline protocols are the circuit generation of garbling schemes as in authenticated garbling [WRK17a, WRK17b] or the generation of correlated randomness in form of Beaver triples [Bea92] in secret sharing-based protocols such as in SPDZ [DPSZ12]. The main idea of this approach is that the offline protocol can be executed continuously *in the background* and the online protocol is executed ad-hoc once input data becomes available or output data is required. Since the performance requirements for the online protocol are usually much stricter, the offline part should cover the most expensive protocol steps, as for example done in [WRK17a, WRK17b] and [DPSZ12].

A middle ground between the design goals of security and efficiency has been proposed with the notion of *covert security*. Introduced by Aumann and Lindell [AL07], covert security allows the adversary to take full control over a party and let her deviate from the protocol specification in an arbitrary way. The protocol, however, is designed in such a way that honest parties can detect cheating with some probability ϵ (often called the deterrence factor). However, if cheating is not detected all bets are off. This weaker security notion comes with the benefit of significantly improved efficiency, when compared to protocols in the full-fledged malicious security model. The motivation behind covert security is that in many real-world scenarios, parties are able to actively deviate from the protocol instructions (and as such are not semi-honest), but due to reputation concerns only do so if they are not caught. In the initial work of Aumann and Lindell, the focus was on the two-party case. This has been first extended to the multi-party case by Goyal et al. [GMS08] and later been adapted to a different line of MPC protocols by Damgård et al. [DKL⁺13].

While the notion of covert security seems appealing at first glance it has one important shortcoming. If an honest party detects cheating, then she cannot reliably transfer her knowledge to other parties, which makes the notion of covert security significantly less attractive for many applications. This shortcoming of covert security was first observed by Asharov and Orlandi [AO12], and addressed with the notion of *public verifiability*. Informally speaking, public verifiability guarantees that if an honest party detects cheating, she can create a certificate that uniquely identifies the cheater, and can be verified by an external party. Said certificate can be used to punish cheaters for misbehavior, e.g., via a smart contract [ZDH19], thereby disincentivizing misbehavior.

Despite being a natural security notion, there has been relatively little work on covert security with public verifiability. In particular, starting with the work of Asharov and Orlandi [AO12] most works have explored publicly verifiable covert security in the two-party setting [KM15, HKK⁺19, ZDH19, DOS20]. These works use a publicly checkable cut-and-choose approach for secure two-party computation based on garbled circuits. Here a random subset of size $t - 1$ out of t garbled circuits is opened to verify if cheating occurred, while the remaining unopened garbled circuit is used for the actual secure function evaluation. The adversary needs to guess which circuit is used for the final evaluation and only cheat in this particular instance. If her guess is false, she will be detected. Hence, there is a deterrence factor of $\frac{t-1}{t}$.

For the extension to the multi-party case of covert security even less is known. Prior work mainly focuses on the restricted version of covert security that does not offer public verifiability [GMS08, DGN10, LOP11, DKL⁺13]. The only work that we are aware of that adds public verifiability to covert secure multi-party computation protocols is the recent work of Damgård et al. [DOS20]. While [DOS20] mainly focuses on a compiler for the two-party case, they also sketch how their construction can be extended to the multi-party setting.

1.1 Our Contribution

In contrast to most prior research, we focus on the multi-party setting. Our main contribution is a novel compiler for transforming input-independent multi-party computation protocols with semi-honest security into protocols that offer covert security with public verifiability. Our construction achieves a high deterrence factor of $\frac{t-1}{t}$, where t is the number of semi-honest instances executed in the cut-and-choose protocol. In contrast, the only prior work that sketches a solution for publicly verifiable covert security for the multi-part setting [DOS20] achieves $\approx \frac{t-1}{nt}$, which in particular for a large number of parties n results in a low deterrence factor. [DOS20] states that the deterrence factor can be increased at the cost of multiple protocol repetitions, which results into higher complexity and can be abused to amplify denial-of-service attacks. A detail discussion of the main differences between [DOS20] and our work is given in Section 6. We emphasize that our work is also the first that provides a full formal security proof of the multi-party case in the model of covert security with public verifiability.

Our results apply to a large class of input-independent offline protocols for carrying out pre-computation. Damgård et al. [DOS20] have shown that an offline-online protocol with a publicly verifiable covert secure offline phase and a maliciously secure online phase constitutes a publicly verifiable covert secure protocol in total. Hence, by applying our compiler to a passively secure offline protocol and combining it with an actively secure online protocol, we obtain a publicly verifiable covert secure protocol in total. Since offline protocols are often the most expensive part of the secure multi-party computation protocol, e.g., in protocols like [YWZ20] and [DPSZ12], our approach has the potential of significantly improving efficiency of multi-party computation protocols in terms of computation and communication overhead.

An additional contribution of our work (which is of independent interest) is to introduce a novel mechanism for achieving public verifiability in protocols with covert security. Our approach is based on *time-lock encryption* [RSW96, MT19, MMV11, BGJ⁺16], a primitive that enables encryption of messages into the future and has previously been discussed in the context of delayed digital cash payments, sealed-bid auctions, key escrow, and e-voting. Time-lock encryption can be used as a building block to guarantee that in case of malicious behavior each honest party can construct a publicly verifiable cheating certificate without further interaction. The use of time-lock puzzles in a simulation-based security proof requires us to overcome several technical challenges that do not occur for proving game-based security notions.

In order to achieve efficient verification of the cheating certificates, we also show how to add verifiability to the notion of time-lock encryption by using techniques from verifiable delay functions [BBBF18]. While our construction can be instantiated with any time-lock encryption satisfying our requirements, we present a concrete extension of the RSW time-lock encryption scheme. Since RSW-based time-lock encryption [RSW96, MT19] requires a one-time trusted setup, an instantiation of our construction using the RSW-based time-lock encryption inherits this assumption. We can implement the one-time trusted setup using a maliciously secure multi-party computation protocol similar to the MPC ceremony used, e.g., by the cryptocurrency ZCash.

1.2 Technical Overview

In this section, we give a high-level overview of the main techniques used in our work. To this end, we start by briefly recalling how covert security is typically achieved. Most covert secure protocols take a semi-honest protocol and execute t instances of it in parallel. They then check the correctness of $t - 1$ randomly chosen instances by essentially revealing the used inputs and randomness and finally take the result of the last unopened execution as protocol output. The above requires that (a) checking the correctness of the $t - 1$ instances can be carried out efficiently, and (b) the private inputs of the parties are not revealed.

In order to achieve the first goal, one common approach is to derandomize the protocol, i.e., let the parties generate a random seed from which they derive their internal randomness. Once the protocol is derandomized, correctness can

efficiently be checked by the other parties. To achieve the second goal, the protocol is divided into an offline and an online protocol as described above. The output of the offline phase (e.g., a garbling scheme) is just some correlated randomness. As this protocol is input-independent, the offline phase does not leak information about the parties’ private inputs. The online phase (e.g., evaluating a garbled circuit) is maliciously secure and hence protects the private inputs.

Public verifiability. To add public verifiability to the above-described approach, the basic idea is to let the parties sign all transcripts that have been produced during the protocol execution. This makes them accountable for cheating in one of the semi-honest executions. One particular challenge for public verifiability is to ensure that once a malicious party notices that its cheating attempt will be detected it cannot prevent (e.g., by aborting) the creation of a certificate proving its misbehavior. Hence, the trivial idea of running a shared coin tossing protocol to select which of the instances will be checked does not work because the adversary can abort before revealing her randomness and inputs used in the checked instances. To circumvent this problem, the recent work of Damgård et al. [DOS20] proposes the following technique. Each party locally chooses a subset I of the t semi-honest instances whose computation it wants to check (this is often called a watchlist [IPS08]). Next, it obviously asks the parties to explain their execution in those instances (i.e., by revealing the random coins used in the protocol execution). While this approach works well in the two-party case, in the multi-party case it either results in a low deterrence factor or requires that the protocol execution is repeated many times. This is due to the fact that each party chooses its watchlist independently; in the worst case, all watchlists are mutually disjoint. Hence, the size of each watchlist is set to be lower or equal than $\frac{t-1}{n}$ (resulting in a deterrence factor of $\frac{t-1}{nt}$) to guarantee that one instance remains unchecked or parties repeat the protocol several times until there is a protocol execution with an unchecked instance.

Public verifiability from time-lock encryption. Our approach avoids the above shortcomings by using time-lock encryption. Concretely, we follow the shared coin-tossing approach mentioned above but prevent the rushing attack by locking the shared coin (selecting which semi-honest executions shall be opened) and the seeds of the opened executions in time-lock encryption. Since the time-lock ciphertexts are produced before the selection-coin is made public, it will be too late for the adversary to abort the computation. Moreover, since the time-lock encryption can be solved even without the participation of the adversary, the honest parties can produce a publicly verifiable certificate to prove misbehavior. This approach has the advantage that we can always check all but one instance of the semi-honest executions, thereby significantly improving the deterrence factor and the overall complexity. One may object that solving time-lock encryption adds additional computational overhead to the honest parties. We emphasize, however, that the time-lock encryption has to be solved only in the pessimistic case when one party aborts after the puzzle generation. Moreover, in our construction, the time-lock parameter can be chosen rather small, since

the encryption has to hide the selection-coin and the seeds only for two communication rounds. See section 6 for a more detailed analysis of the overhead introduced by the time-lock puzzle generation and a comparison to prior work.

Creating the time-lock encryption. There are multiple technical challenges that we need to address to make the above idea work. First, current constructions of time-lock encryption matching our requirements require a trusted setup for generating the public parameters. In particular, we need to generate a strong RSA modulus N without leaking its factorization, and produce a base-puzzle that later can be used for efficiency reasons. Both of these need to be generated just once and can be re-used for all protocol executions. Hence, one option is to replace the trusted setup by a maliciously secure MPC similar to what has been done for the MPC ceremony used by the cryptocurrency ZCash. Another alternative is to investigate if time-lock puzzles matching the requirements of our compiler can be constructed from hidden order groups with public setup such as ideal class groups of imaginary quadratic fields [BW88] or Jacobians of hyperelliptic curves [DG20]. An additional challenge is that we cannot simply time-lock the seeds of all semi-honest protocol executions (as one instance needs to remain unopened). To address this problem, we use a maliciously secure MPC protocol to carry out the shared coin-tossing protocol and produce the time-lock encryptions of the seeds for the semi-honest protocol instance that are later opened. We emphasize that the complexity of this step only depends on t and n , and is in particular independent of the complexity of the functionality that we want to compute. Hence, for complex functionalities the costs of the maliciously secure puzzle generation are amortized over the protocol costs ³.

2 Secure Multi-Party Computation

Secure computation in the standalone model is defined via the real world/ideal world paradigm. In the real world, all parties interact in order to jointly execute the protocol Π . In the ideal world, the parties send their inputs to a trusted party called ideal functionality and denoted by \mathcal{F} which computes the desired function f and returns the result back to the parties. It is easy to see that in the ideal world the computation is correct and reveals only the intended information by definition. The security of a protocol Π is analyzed by comparing the ideal-world execution with the real-world execution. Informally, protocol Π is said to securely realize \mathcal{F} if for every real-world adversary \mathcal{A} , there exists an ideal-world adversary \mathcal{S} such that the joint output distribution of the honest parties and the adversary \mathcal{A} in the real-world execution of Π is indistinguishable from the joint output distribution of the honest parties and \mathcal{S} in the ideal-world execution.

We denote the number of parties executing a protocol Π by n . Let $f : (\{0, 1\}^*)^n \rightarrow (\{0, 1\}^*)^n$, where $f = (f_1, \dots, f_n)$, be the function realized by Π .

³ Concretely, for each instantiation we require two exponentiations and a small number of symmetric key encryptions. The latter can be realized using tailored MPC-ciphers like LowMC [ARS⁺15].

For every input vector $\bar{x} = (x_1, \dots, x_n)$ the output vector is $\bar{y} = (f_1(\bar{x}), \dots, f_n(\bar{x}))$ and the i -th party P_i with input x_i obtains output $f_i(\bar{x})$.

An adversary can corrupt any subset $I \subseteq [n]$ of parties. We further set $\text{REAL}_{\Pi, \mathcal{A}(z), I}(\bar{x}, 1^\kappa)$ to be the output vector of the protocol execution of Π on input $\bar{x} = (x_1, \dots, x_n)$ and security parameter κ , where the adversary \mathcal{A} on auxiliary input z corrupts the parties $I \subseteq [n]$. By $\text{OUTPUT}_i(\text{REAL}_{\Pi, \mathcal{A}(z), I}(\bar{x}, 1^\kappa))$, we specify the output of party P_i for $i \in [n]$.

2.1 Covert Security

Aumann and Lindell introduced the notion of *covert security with ϵ -deterrence factor* in 2007 [AL07]. We focus on the strongest given formulation of covert security that is the *strong explicit cheat formulation*, where the ideal-world adversary only learns the honest parties' inputs if cheating is undetected. However, we slightly modify the original notion of covert security to capture realistic effects that occur especially in input-independent protocols and are disregarded by the notion of [AL07]. The changes are explained and motivated below.

As in the standard secure computation model, the execution of a real-world protocol is compared to the execution within an ideal world. The real world is exactly the same as in the standard model but the ideal model is slightly adapted in order to allow the adversary to cheat. Cheating will be detected by some fixed probability ϵ , which is called the deterrence factor. Let $\epsilon : \mathbb{N} \rightarrow [0, 1]$ be a function, then the execution in the ideal model works as follows.

Inputs: Each party obtains an input; the i^{th} party's input is denoted by x_i . We assume that all inputs are of the same length. The adversary receives an auxiliary input z .

Send inputs to trusted party: Any honest party P_j sends its received input x_j to the trusted party. The corrupted parties, controlled by \mathcal{S} , may either send their received input, or send some other input of the same length to the trusted party. This decision is made by \mathcal{S} and may depend on the values x_i for $i \in I$ and auxiliary input z . If there are no inputs, the parties send ok_i instead of their inputs to the trusted party.

Trusted party answers adversary: If the trusted party receives inputs from all parties, the trusted party computes $(y_1, \dots, y_m) = f(\bar{w})$ and sends y_i to \mathcal{S} for all $i \in I$.

Abort options: If the adversary sends **abort** to the trusted party as additional input (before or after the trusted party sends the potential output to the adversary), then the trusted party sends **abort** to all the honest parties and halts. If a corrupted party sends additional input $w_i = \text{corrupted}_i$ to the trusted party, then the trusted party sends corrupted_i to all of the honest parties and halts. If multiple parties send corrupted_i , then the trusted party disregards all but one of them (say, the one with the smallest index i). If both corrupted_i and **abort** messages are sent, then the trusted party ignores the corrupted_i message.

Attempted cheat option: If a corrupted party sends additional input $w_i = \text{cheat}_i$ to the trusted party (as above: if there are several messages $w_i = \text{cheat}_i$

ignore all but one - say, the one with the smallest index i), then the trusted party works as follows:

1. With probability ϵ , the trusted party sends corrupted_i to the adversary and all of the honest parties.
2. With probability $1 - \epsilon$, the trusted party sends undetected to the adversary along with the honest parties inputs $\{x_j\}_{j \notin I}$. Following this, the adversary sends the trusted party abort or output values $\{y_j\}_{j \notin I}$ of its choice for the honest parties. If the adversary sends abort , the trusted party sends abort to all honest parties. Otherwise, for every $j \notin I$, the trusted party sends y_j to P_j .

The ideal execution then ends at this point. Otherwise, if no w_i equals abort_i , corrupted_i or cheat_i , the ideal execution continues below.

Trusted party answers honest parties: If the trusted party did not receive corrupted_i , cheat_i or abort from the adversary or a corrupted party then it sends y_j for all honest parties P_j (where $j \notin I$).

Outputs: An honest party always outputs the message it obtained from the trusted party. The corrupted parties outputs nothing. The adversary \mathcal{S} outputs any arbitrary (probabilistic) polynomial-time computable function of the initial inputs $\{x_i\}_{i \in I}$, the auxiliary input z , and the received messages.

We denote by $\text{IDEALC}_{f, \mathcal{S}(z), I}^\epsilon(\bar{x}, 1^\kappa)$ the output of the honest parties and the adversary in the execution of the ideal model as defined above, where \bar{x} is the input vector and the adversary \mathcal{S} runs on auxiliary input z .

Definition 1 (Covert security with ϵ -deterrent). *Let f, Π , and ϵ be as above. Protocol Π is said to securely compute f in the presence of covert adversaries with ϵ -deterrent if for every non-uniform probabilistic polynomial-time adversary \mathcal{A} for the real model, there exists a non-uniform probabilistic polynomial-time adversary \mathcal{S} for the ideal model such that for every $I \subseteq [n]$, every balanced vector $\bar{x} \in (\{0, 1\}^*)^n$, and every auxiliary input $z \in \{0, 1\}^*$:*

$$\{\text{IDEALC}_{f, \mathcal{S}(z), I}^\epsilon(\bar{x}, 1^\kappa)\}_{\kappa \in \mathbb{N}} \stackrel{c}{\equiv} \{\text{REAL}_{\Pi, \mathcal{A}(z), I}(\bar{x}, 1^\kappa)\}_{\kappa \in \mathbb{N}}$$

Notice that the definition of the ideal world given above differs from the original definition of Aumann and Lindell in four aspects. First, we add the support of functions with no private inputs from the parties to model input-independent functionalities. In this case, the parties send ok instead of their inputs to the trusted party. Second, whenever a corrupted party aborts, the trusted party sends abort to all honest parties. Note that this message does not include the index of the aborting party which differs from the original model. The security notion of *identifiable abort* [IOZ14], where the aborting party is identified, is an independent research area, and is not achieved by our compiler. Third, we allow a corrupted party to abort after undetected cheating, which does not weaken the security guarantees.

Finally, we allow the adversary to learn the output of the function f before it decides to cheat or to act honestly. In the original notion the adversary has

to make this decision without seeing the potential output. Although this modification gives the adversary additional power, it captures the real world more reliably in regard to standalone input-independent protocols.

Covert security is typically achieved by executing several semi-honest instances and checking some of them via cut-and-choose while utilizing an unchecked instance for the actual output generation. The result of the semi-honest instances is often an input-independent precomputation in the form of correlated randomness, e.g., a garbled circuit or multiplication triples, which is consumed in a maliciously secure input-dependent online phase, e.g., the circuit evaluation or a SPDZ-style [DKL⁺13] online phase. Typically, the precomputation is explicitly designed not to leak any information about the actual output of the online phase, e.g., a garbled circuit obfuscates the actual circuit gate tables and multiplication triples are just random values without any relation to the output or even the function computed in the online phase. Thus, in such protocols, the adversary does not learn anything about the output when executing the semi-honest instances and therefore when deciding to cheat, which makes the original notion of covert security realistic for such input-dependent protocols.

However, if covert security is applied to the standalone input-independent precomputation phase, as done by our compiler, the actual output is the correlated randomness provided by one of the semi-honest instances. Hence, the adversary learns potential outputs when executing the semi-honest instances. Considering a rushing adversary that learns the output of a semi-honest instance first and still is capable to cheat with its last message, the adversary can base its decision to cheat on potential outputs of the protocol. Although this scenario is simplified and there is often a trade-off between output determination and cheating opportunities, the adversary potentially learns something about the output before deciding to cheat. This is a power that the adversary might have in all cut-and-choose-based protocols that do not further process the output of the semi-honest instances, also in the input-independent covert protocols compiled by Damgård et al. [DOS20].

Additionally, as we have highlighted above, the result of the precomputation typically does not leak any information about an input-dependent phase which uses this precomputation. Hence, in such offline-online protocols, the adversary has only little benefit of seeing the result of the precomputation before deciding to cheat or to act honestly.

Instead of adapting the notion of covert security, we could also focus on protocols that first obfuscate the output of the semi-honest instances, e.g., by secret sharing it, and then de-obfuscate the output in a later stage. However, this restricts the compiler to a special class of protocols but has basically the same effect. If we execute such a protocol with our notion of security up to the obfuscation stage but without de-obfuscating, the adversary learns the potential output, that is just some obfuscated output and therefore does not provide any benefit to the adversary's cheat decision. Next, we only have to ensure that the de-obfuscating is done in a malicious or covert secure way, which can be achieved,

e.g., by committing to all output shares after the semi-honest instances and then open them when the cut-and-choose selection is done.

For the above reasons, we think it is a realistic modification to the covert notion to allow the adversary to learn the output of the function f before she decides to cheat or to act honestly. Note that the real-world adversary in cut-and-choose-based protocols does only see a list of potential outputs but the ideal-world adversary receives a single output which is going to be the protocol output if the adversary does not cheat or abort. However, we have chosen to be more generous to the adversary and model the ideal world like this in order to keep it simpler and more general. For the same reason we ignore the trade-off between output determination and cheating opportunities observed in real-world protocols.

In the rest of this work, we denote the trusted party computing function f in the ideal-world description by \mathcal{F}_{Cov} .

2.2 Covert Security with Public Verifiability

As discussed in the introduction Asharov and Orlandi introduced to notion of *covert security with ϵ -deterrent and public verifiability* (PVC) in the two-party setting [AO12]. We give an extension of their formal definition to the multi-party setting in the following.

In addition to the covert secure protocol Π , we define two algorithms **Blame** and **Judge**. **Blame** takes as input the view of an honest party P_i after P_i outputs corrupted_j in the protocol execution for $j \in I$ and returns a certificate Cert , i.e., $\text{Cert} := \text{Blame}(\text{view}_i)$. The **Judge**-algorithm takes as input a certificate Cert and outputs the identity id_j if the certificate is valid and states that party P_j behaved maliciously; otherwise, it returns **none** to indicate that the certificate was invalid.

Moreover, we require that the protocol Π is slightly adapted such that an honest party P_i computes $\text{Cert} = \text{Blame}(\text{view}_i)$ and broadcasts it after cheating has been detected. We denote the modified protocol by Π' . Notice that due to this change, the adversary gets access to the certificate. By requiring simulatability, it is guaranteed that the certificate does not reveal any private information.

We now continue with the definition of covert security with ϵ -deterrent and public verifiability in the multi-party case.

Definition 2 (Covert security with ϵ -deterrent and public verifiability in the multi-party case (PVC-MPC)). *Let f, Π', Blame , and Judge be as above. The triple $(\Pi', \text{Blame}, \text{Judge})$ securely computes f in the presence of covert adversaries with ϵ -deterrent and public verifiability if the following conditions hold:*

1. *(Simulatability) The protocol Π' securely computes f in the presence of covert adversaries with ϵ -deterrent according to the strong explicit cheat formulation (see Definition 1).*

2. (*Public Verifiability*) For every PPT adversary \mathcal{A} corrupting parties P_i for $i \in I \subseteq [n]$, there exists a negligible function $\mu(\cdot)$ such that for all $(\bar{x}, z) \in (\{0, 1\}^*)^{n+1}$ the following holds:
If $\text{OUTPUT}_j(\text{REAL}_{\Pi, \mathcal{A}(z), I}(\bar{x}, 1^\kappa)) = \text{corrupted}_i$ for $j \in [n] \setminus I$ and $i \in I$ then:

$$\Pr[\text{Judge}(\text{Cert}) = \text{id}_i] > 1 - \mu(n),$$

where Cert is the output certificate of the honest party P_j in the execution.

3. (*Defamation Freeness*) For every PPT adversary \mathcal{A} corrupting parties P_i for $i \in I \subseteq [n]$, there exists a negligible function $\mu(\cdot)$ such that for all $(\bar{x}, z) \in (\{0, 1\}^*)^{n+1}$ and all $j \in [n] \setminus I$:

$$\Pr[\text{Cert}^* \leftarrow \mathcal{A}; \text{Judge}(\text{Cert}^*) = \text{id}_j] < \mu(n).$$

3 Preliminaries

3.1 Communication Model & Notion of Time

We assume the existence of authenticated channels between every pair of parties. Further, we assume synchronous communication between all parties participating in the protocol execution. This means the computation proceeds in rounds, where each party is aware of the current round. All messages sent in one round are guaranteed to arrive at the other parties at the end of this round. We further consider rushing adversaries which in each round are able to learn the messages sent by other parties before creating and sending their own messages. This allows an adversary to create messages depending on messages sent by other parties in the same round.

We denote the time for a single communication round by T_c . In order to model the time, it takes to compute algorithms, we use the approach presented by Wesolowski [Wes19]. Suppose the adversary works in computation model \mathcal{M} . The model defines a cost function C and a time-cost function T . $C(\mathcal{A}, x)$ denotes the overall cost to execute algorithm \mathcal{A} on input x . Similar, the time-cost function $T(\mathcal{A}, x)$ abstracts the notion of time of running $\mathcal{A}(x)$. Considering circuits as computational model, one may consider the cost function denoting the overall number of gates of the circuit and the time-cost function being the circuit's depth.

Let \mathcal{S} be an algorithm that for any RSA modulus N generated with respect to the security parameter κ on input N and some element $g \in \mathbb{Z}_N$ outputs the square of g . We define the time-cost function $\delta_{\text{sq}}(\kappa) = T(\mathcal{S}, (N, g))$, i.e., the time it takes for the adversary to compute a single squaring modulo N .

3.2 Verifiable Time-Lock Puzzle

Time-lock puzzles (TLP) provide a mean to encrypt messages to the future. The message is kept secret at least for some predefined time. The concept of a time-lock puzzle was first introduced by Rivest et al. [RSW96] presenting an elegant

construction using sequential squaring modulo a composite integer $N = p \cdot q$, where p and q are primes. The puzzle is some $x \in \mathbb{Z}_N^*$ with corresponding solution $y = x^{2^T}$. The conjecture about this construction is that it requires T sequential squaring to find the solution. Based on the time to compute a single squaring modulo N , the hardness parameter \mathcal{T} denotes the amount of time required to decrypt the message. (See Section 3.1 for a notion of time.)

We extend the notion of time-lock puzzle by a verifiability notion. This property allows a party who solved a puzzle to generate a proof which can be efficiently verified by any third party. Hence, a solver is able to create a verifiable statement about the solution of a puzzle. Boneh et al. [BBBF18] introduced the notion of verifiable delay functions (VDF). Similar to solving a TLP, the evaluation of a VDF on some input x takes a predefined number of sequential steps. Together with the output y , the evaluator obtains a short proof π . Any other party can use π to verify that y was obtained by evaluating the VDF on input x . Besides the sequential evaluation, a VDF provides no means to obtain the output more efficiently. Since we require a primitive that allows a party using some trapdoor information to perform the operation more efficiently, we cannot use a VDF but start with a TLP scheme and add verifiability using known techniques.

We present a definition of verifiable time-lock puzzles. We include a setup algorithm in the definition which generates public parameters required to efficiently construct a new puzzle. This way, we separate expensive computation required as a one-time setup from the generation of puzzles.

Definition 3. *Verifiable time-lock puzzle (VTLP)* A verifiable time-lock puzzle scheme over some finite domain \mathcal{S} consists of four probabilistic polynomial-time algorithms (TL.Setup, TL.Generate, TL.Solve, TL.Verify) defined as follows.

- $(pp) \leftarrow \text{TL.Setup}(1^\lambda, \mathcal{T})$ takes as input the security parameter 1^λ and a hardness parameter \mathcal{T} , and outputs public parameter pp .
- $p \leftarrow \text{TL.Generate}(pp, s)$ takes as input public parameters pp and a solution $s \in \mathcal{S}$ and outputs a puzzle p .
- $(s, \pi) \leftarrow \text{TL.Solve}(pp, p)$ is a deterministic algorithm that takes as input public parameters pp and a puzzle p and outputs a solution s and a proof π .
- $b := \text{TL.Verify}(pp, p, s, \pi)$ is a deterministic algorithm that takes as input public parameters pp , a puzzle p , a solution s , and a proof π and outputs a bit b , with $b = 1$ meaning valid and $b = 0$ meaning invalid. Algorithm TL.Verify must run in total time polynomial in $\log \mathcal{T}$ and λ .

We require the following properties of a verifiable time-lock puzzle scheme.

Completeness For all $\lambda \in \mathbb{N}$, for all \mathcal{T} , for all $pp \leftarrow \text{TL.Setup}(1^\lambda, \mathcal{T})$, and for all s , it holds that

$$(s, \cdot) \leftarrow \text{TL.Solve}(\text{TL.Generate}(pp, s)).$$

Correctness For all $\lambda \in \mathbb{N}$, for all \mathcal{T} , for all $pp \leftarrow \text{TL.Setup}(1^\lambda, \mathcal{T})$, for all s , and for all $p \leftarrow \text{TL.Generate}(pp, s)$, if $(s, \pi) \leftarrow \text{TL.Solve}(p)$, then

$$\text{TL.Verify}(pp, p, s, \pi) = 1.$$

Soundness For all $\lambda \in \mathbb{N}$, for all \mathcal{T} , and for all PPT algorithms \mathcal{A}

$$\Pr \left[\text{TL.Verify}(pp, p', s', \pi') = 1 \mid \begin{array}{l} pp \leftarrow \text{TL.Setup}(1^\lambda, \mathcal{T}) \\ (p', s', \pi') \leftarrow \mathcal{A}(1^\lambda, pp, \mathcal{T}) \\ (s, \cdot) \leftarrow \text{TL.Solve}(pp, p') \end{array} \right] \leq \text{negl}(\lambda)$$

Security A VTLP scheme is secure with gap $\epsilon < 1$ if there exists a polynomial $\tilde{\mathcal{T}}(\cdot)$ such that for all polynomials $\mathcal{T}(\cdot) \geq \tilde{\mathcal{T}}(\cdot)$ and every polynomial-size adversary $(\mathcal{A}_1, \mathcal{A}_2) = \{(\mathcal{A}_1, \mathcal{A}_2)_\lambda\}_{\lambda \in \mathbb{N}}$ where the depth of \mathcal{A}_2 is bounded from above by $\mathcal{T}^\epsilon(\lambda)$, there exists a negligible function $\mu(\cdot)$, such that for all $\lambda \in \mathbb{N}$ it holds that

$$\Pr \left[b \leftarrow \mathcal{A}_2(pp, p, \tau) \mid \begin{array}{l} (\tau, s_0, s_1) \leftarrow \mathcal{A}_1(1^\lambda) \\ pp \leftarrow \text{TL.Setup}(1^\lambda, \mathcal{T}(\lambda)) \\ b \stackrel{\$}{\leftarrow} \{0, 1\} \\ p \leftarrow \text{TL.Generate}(pp, s_b) \end{array} \right] \leq \frac{1}{2} + \mu(\lambda)$$

and $(s_0, s_1) \in \mathcal{S}^2$.

Although our compiler can be instantiated with any TLP scheme satisfying Definition 3, we present a concrete construction based on the RSW time-lock puzzle [RSW96]. We leave it to further research to investigate if a time-lock puzzle scheme matching our requirements, i.e., verifiability and efficient puzzle generation, can be constructed based on hidden order groups with public setup such as ideal class groups of imaginary quadratic fields [BW88] or Jacobians of hyperelliptic curves [DG20]. Due to the public setup, such constructions might be more efficient than our RSW-based solution.

In order to make the decrypted value verifiable we integrate the generation of a proof as introduced by Wesolowski [Wes19] for verifiable delay functions. The technique presented by Wesolowski provides a way to generate a small proof which can be efficiently verified. However, proof generation techniques from other verifiable delay functions, e.g., presented by Pietrzak [Pie19] can be used as well. The approach of Wesolowski utilizes a function `bin`, which maps an integer to its binary representation, and a hash function H_{prime} that maps any string to an element of $\text{Primes}(2k)$. The set $\text{Primes}(2k)$ contains the first 2^{2k} prime numbers, where k denotes the security level (typically 128, 192 or 256).

The `TL.Setup`-algorithm takes the security and hardness parameter and outputs public parameter. This includes an RSA modulus of two strong primes, the number of sequential squares corresponding to the hardness parameter, and a base puzzle. The computation can be executed efficiently if the prime numbers are known. Afterwards, the primes are not needed anymore and can be thrown away. Note that any party knowing the factorization of the RSA modulus can efficiently solve puzzles. Hence, the `TL.Setup`-algorithm should be executed in a trusted way.

The `TL.Generate`-algorithm allows any party to generate a time-lock puzzle over some secret s . In the construction given below, we assume s to be an element in \mathbb{Z}_N^* . However, one can use a hybrid approach where the secret is encrypted

with some symmetric key which is then mapped to an element in \mathbb{Z}_N^* . This allows the generator to time-lock large secrets as well. Note that the puzzle generation can be done efficiently and does not depend on the hardness parameter \mathcal{T} .

The `TL.Solve`-algorithm solves a time-lock puzzle p by performing sequential squaring, where the number of steps depend on the hardness parameter \mathcal{T} . Along with the solution, it outputs a verifiable proof π . This proof is used as additional input to the `TL.Verify`-algorithm outputting true if the given secret was time-locked by the given puzzle.

We state the formal definition of our construction next.

Construction Verifiable Time-Lock Puzzle

TL.Setup($1^\lambda, \mathcal{T}$):

- Sample two strong primes (p, q) and set $N := p \cdot q$.
- Set $\mathcal{T}' := \mathcal{T} / \delta_{\text{Sq}}(\lambda)$.
- Sample uniform $\tilde{g} \xleftarrow{\$} \mathbb{Z}_N^*$ and set $g := -\tilde{g}^2 \pmod{N}$.
- Compute $h := g^{2^{\mathcal{T}'}}$, which can be optimized by reducing $2^{\mathcal{T}'}$ module $\phi(N)$ first.
- Set $Z := (g, h)$.
- Output (\mathcal{T}', N, Z) .

TL.Generate(pp, s):

- Parse $pp := (\mathcal{T}', N, Z)$ and $Z := (g, h)$.
- Sample uniform $r \xleftarrow{\$} \{1, \dots, N^2\}$.
- Compute $g^* := g^r$ and $h^* := h^r$.
- Set $c^* := h^* \cdot s \pmod{N}$.
- Output $p := (g^*, c^*)$.

TL.Solve(pp, p):

- Parse $pp := (\mathcal{T}', N, Z)$ and $p := (g^*, c^*)$.
- Compute $h := g^{*2^{\mathcal{T}'}} \pmod{N}$ by repeated squaring.
- Compute $s := \frac{c^*}{h} \pmod{N}$ as the solution.
- Compute $\ell = H_{\text{prime}}(\text{bin}(g^*) || * || \text{bin}(s)) \in \text{Primes}(2k)$ as the challenge.
- Compute $\pi = g^{* \lfloor 2^{\mathcal{T}' / \ell} \rfloor}$ as the proof.
- Output (s, π) .

TL.Verify(pp, p, s, π):

- Parse $pp := (\mathcal{T}', N, Z)$.
- Parse $p := (g^*, c^*)$.
- Compute $\ell = H_{\text{prime}}(\text{bin}(g^*) || * || \text{bin}(s)) \in \text{Primes}(2k)$ as the challenge.
- Compute $r = 2^{\mathcal{T}' / \ell} \pmod{\ell}$.
- Compute $h' = \pi^\ell g^{*r}$.
- Compute $s' := \frac{c^*}{h'}$.
- If $s = s'$, output 1, otherwise output 0.

The security of the presented construction is based on the conjecture that it requires \mathcal{T}' sequential squarings to solve a puzzle. Moreover, the soundness of the proof generation is based on the number-theoretic assumption that it is hard to find the ℓ -th root modulo an RSA modulus N of an integer $x \notin \{-1, 0, +1\}$ where ℓ is uniformly sampled from $\text{Primes}(2k)$ and the factorization of N is unknown. See [Wes19] for a detailed description of the security assumption.

3.3 Commitment

Our protocol makes use of an extractable commitment scheme which is *computationally binding and hiding*. For ease of description, we assume the scheme to be non-interactive. We will use the notation $(c, d) \leftarrow \text{Commit}(m)$ to commit to message m , where c is the commitment value and d denotes the decommitment or opening value. Similarly, we use $m' \leftarrow \text{Open}(c, d)$ to open commitment c with opening value d to $m' = m$ or $m' = \perp$ in case of incorrect opening. The extractability property allows the simulator to extract the committed message m and the opening value d from the commitment c by using some trapdoor information.

Such a scheme can be implemented in the random oracle model by defining $\text{Commit}(x) = H(i, x, r)$ where i is the identity of the committer, $H : \{0, 1\}^* \rightarrow \{0, 1\}^{2\kappa}$ is a random oracle and $r \xleftarrow{\$} \{0, 1\}^\kappa$.

3.4 Signature Scheme

We use a signature scheme ($\text{Gen}, \text{Sign}, \text{Verify}$) that is *existentially unforgeable under chosen-message attacks*. Before the start of our protocol, each party executes the Gen -algorithm to obtain a key pair (pk, sk) . While the secret key sk is kept private, we assume that each other party is aware of the party's public key pk .

3.5 Semi-Honest Base Protocol

Our compiler is designed to transform a semi-honest secure n -party protocol with no private input tolerating $n - 1$ corruptions, Π_{SH} , that computes a probabilistic function $(y^1, \dots, y^n) \leftarrow f()$, where y^i is the output for party P_i , into a publicly verifiable covert protocol, Π_{PVC} , that computes the same function. In order to compile Π_{SH} , it is necessary that all parties that engage in the protocol Π_{SH} receive a protocol transcript, which is the same if all parties act honestly. This means that there needs to be a fixed ordering for the sent messages and that each message needs to be sent to all involved parties ⁴.

We stress that any protocol can be adapted to fulfill the compilation requirements. Adding a fixed order to the protocol messages is trivial and just a matter of specification. Furthermore, parties can send all of their outgoing messages to all other parties without harming the security. This is due to the fact, that the

⁴ This requirement is inherent to all known publicly verifiable covert secure protocols.

protocol tolerates $n - 1$ corruptions which implies that the adversary is allowed to learn all messages sent by the honest party anyway. Note that the transferred messages do not need to be securely broadcasted, because our compiler requires the protocol to produce a consistent transcript only if all parties act honestly.

3.6 Coin Tossing Functionality

We utilize a maliciously secure coin tossing functionality $\mathcal{F}_{\text{coin}}$ parameterized with the security parameter κ and the number of parties n . The functionality receives ok_i from each party P_i for $i \in [n]$ and outputs a random κ -bit string $\text{seed} \xleftarrow{\$} \{0, 1\}^\kappa$ to all parties.

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

Inputs: Each party P_i with $i \in [n]$ inputs ok_i .

- Sample $\text{seed} \xleftarrow{\$} \{0, 1\}^\kappa$.
- Send seed to \mathcal{A} .
 - If \mathcal{A} returns **abort**, send **abort** to all honest parties and stop.
 - Otherwise, send seed to all honest parties.

3.7 Puzzle Generation Functionality

The maliciously secure puzzle generation functionality \mathcal{F}_{PG} is parameterized with the computational security parameter κ , the number of involved parties n , the cut-and-choose parameter t and public TLP parameters pp . It receives a coin share r^i , a puzzle randomness share u^i , and the seed-share decommitments for all instances $\{d_j^i\}_{j \in [t]}$ as input from each party P_i . \mathcal{F}_{PG} calculates the random coin r and the puzzle randomness u using the shares of all parties. Then, it generates a time-lock puzzle p of r and all seed-share decommitments expect the ones with index r . In the first output round it sends p to all parties. In the second output round it reveals the values locked within p to all parties. As we assume a rushing adversary, \mathcal{A} receives the outputs first in both rounds and can decide if the other parties should receive the outputs as well.

The functionality \mathcal{F}_{PG} can be instantiated with a general purpose maliciously secure MPC-protocol such as the ones specified by [DKL⁺13] or [YWZ20].

Functionality \mathcal{F}_{PG}

Inputs: Each party P_i with $i \in [n]$ inputs $(r^i, u^i, \{d_j^i\}_{j \in [t]})$, where $r^i \in [t]$, $u^i \in \{0, 1\}^\kappa$, and $d_j^i \in \{0, 1\}^\kappa$.

- Compute $r := \sum_{i=1}^n r^i \bmod t$ and $u := \bigoplus_{i=1}^n u^i$.
- Generate puzzle $p \leftarrow \text{TL.Generate}(pp, (r, \{d_j^i\}_{i \in [n], j \in [t] \setminus r}))$ using randomness u .
- Send p to \mathcal{A} .

- If \mathcal{A} returns **abort**, send **abort** to all honest parties and stop.
- Otherwise, send p to all honest parties.⁵
- Upon receiving **continue** from each party, send $(r, \{d_j^i\}_{i \in [n], j \in [t] \setminus r})$ to \mathcal{A} .
 - If \mathcal{A} returns **abort** or some party does not send **continue**, send **abort** to all honest parties and stop.
 - Otherwise, send $(r, \{d_j^i\}_{i \in [n], j \in [t] \setminus r})$ to all honest parties.

4 PVC Compiler

In the following, we present our compiler for multi-party protocols with no private input from semi-honest to publicly verifiable covert security. We start with presenting a distributed seed computation which is used as subprotocol in our compiler. Next, we state the detailed description of our compiler. Lastly, we provide information about the Blame- and Judge-algorithm required by the notion of publicly verifiable covert security.

4.1 Distributed Seed Computation

The execution of the semi-honest protocol instances Π_{SH} within our PVC compiler requires each party to use a random tape that is uniform at random. In order to ensure this requirement, the parties execute several instances of a distributed seed computation subprotocol Π_{SG} at the beginning. During this subprotocol, each party P_h selects a uniform κ -bit string as private seed share $\text{seed}^{(1,h)}$. Additionally, P_h and all other parties get uniform κ -bit strings $\{\text{seed}^{(2,i)}\}_{i \in [n]}$, which are the public seed shares of all parties. The randomness used by P_h in the semi-honest protocol will be derived from $\text{seed}^h := \text{seed}^{(1,h)} \oplus \text{seed}^{(2,h)}$. This way seed^h is distributed uniformly. Note that if protocol Π_{SH} is semi-malicious instead of semi-honest secure then each party may choose the randomness arbitrarily and there is no need to run the seed generation.

As the output, party P_h obtains its own private seed, commitments to all private seeds, a decommitment for its own private seed, and all public seed shares. We state the detailed protocol steps next. The protocol is executed by each party P_h , parameterized with the number of parties n and the security parameter κ .

Protocol Π_{SG}

- (a) **Commit-phase**
Party P_h chooses a uniform κ -bit string $\text{seed}^{(1,h)}$, sets $(c^h, d^h) \leftarrow \text{Commit}(\text{seed}^{(1,h)})$, and sends c^h to all parties.
 - (b) **Public coin-phase**
For each $i \in [n]$, party P_h sends **ok** to $\mathcal{F}_{\text{coin}}$ and receives $\text{seed}^{(2,i)}$.
- Output**

⁵ The honest parties receive p or **abort** in the same communication round as \mathcal{A} .

If P_h has not received all messages in the expected communication rounds or any $\text{seed}^{(2,i)} = \perp$, it sends **abort** to all parties and outputs **abort**. Otherwise, it outputs $(\text{seed}^{(1,h)}, d^h, \{\text{seed}^{(2,i)}, c^i\}_{i \in [n]})$.

4.2 The PVC Compiler

Starting with a n -party semi-honest secure protocol Π_{SH} we compile a publicly verifiable covert secure protocol Π_{PVC} . The compiler works for protocols that receive no private input.

The compiler uses a signature scheme, a verifiable time-lock puzzle scheme, and a commitment scheme as building blocks. Moreover, the communication model is as defined in Section 3.1. We assume each party generated a signature key pair (sk, pk) and all parties know the public keys of the other parties. Furthermore, we suppose the setup of the verifiable time-lock puzzle scheme TL.Setup was executed in a trusted way beforehand. This means in particular that all parties are aware of the public parameters pp . We stress that this setup needs to be executed once and may be used by many protocol executions. The hardness parameter \mathcal{T} used as input to the TL.Setup -algorithm needs to be defined as $\mathcal{T} > 2 \cdot T_c$, where T_c denotes the time for a single communication round (see Section 3.1). In particular, the hardness parameter is independent of the complexity of Π_{SH} .

From a high-level perspective, our compiler works in five phases. At the beginning, all parties jointly execute the seed generation to set up seeds from which the randomness in the semi-honest protocol instances is derived. Second, the parties execute t instances of the semi-honest protocol Π_{SH} . By executing several instances, the parties' honest behavior can be later on checked in all but one instance. Since checking reveals the confidential outputs of the other parties, there must be one instance that is unchecked. The index of this one is jointly selected in a random way in the third phase. Moreover, publicly verifiable evidence is generated such that an honest party can blame any malicious behavior afterwards. To this end, we use the puzzle generation functionality \mathcal{F}_{PG} to generate a time-lock puzzle first. Next, each party signs all information required for the other parties to blame this party. In the fourth phase, the parties either honestly reveal secret information for all but one semi-honest execution or abort. In case of abort, the honest parties execute the fifth phase. By solving the time-lock puzzle, the honest parties obtain the required information to create a certificate about malicious behavior. Since this phase is only required to be executed in case any party aborted before revealing the information, we call this the pessimistic case. We stress that no honest party is required to solve a time-lock puzzle in case all parties behave honestly.

A corrupted party may cheat in two different ways in the compiled protocol. Either the party inputs decommitment values into the puzzle generation functionality which open the commitments created during the seed generation to \perp or the party misbehaved in the execution of Π_{SH} . The later means that a

party uses different randomness than derived from the seeds generated at the beginning.

The first cheat attempt may be detected in two ways. In the optimistic execution, all parties receive the inputs to \mathcal{F}_{PG} and can verify that opening the commitments is successful. In the pessimistic case, solving the time-lock puzzle reveals the input to \mathcal{F}_{PG} . Since we do not want the **Judge** to solve the puzzle itself, we provide a proof along with the solution of the time-lock puzzle. To this end, we require a verifiable time-lock puzzle as modeled in Section 3. Even in the optimistic case, if an honest party detects cheating, the time-lock puzzle needs to be solved in order to generate a publicly verifiable certificate.

If all decommitments open the commitments successfully, an honest party can recompute the seeds used by all other parties in an execution of Π_{SH} and re-run the execution. The resulting transcript is compared with the one signed by all parties beforehand. In case any party misbehaved, a publicly verifiable certificate can be created. For the sake of exposition, we compress the detection of malicious behavior and the generation of the certificate into the **Blame**-algorithm.

The protocol defined as follows is executed by each honest party P_h .

Protocol Π_{PVC}

Public input: All parties agree on κ , n , t , Π_{SH} and pp and know all parties' public keys $\{pk_i\}_{i \in [n]}$.

Private input: P_h knows its own secret key sk_h .

Distributed seed computation:

We abuse notation here and assume that the parties execute the seed generation protocol from above.

1. For each instance $j \in [t]$ party P_h interacts with all other parties to receive

$$(\text{seed}_j^{(1,h)}, d_j^h, \{\text{seed}_j^{(2,i)}, c_j^i\}_{i \in [n]}) \leftarrow \Pi_{\text{SG}}$$

and computes $\text{seed}_j^h := \text{seed}_j^{(1,h)} \oplus \text{seed}_j^{(2,h)}$.

Semi-honest protocol execution:

2. Party P_h engages in t instances of the protocol Π_{SH} with all other parties. In the j -th instance, party P_h uses randomness derived from seed_j^h and receives a transcript and output:

$$(\text{trans}_j, y_j^h) \leftarrow \Pi_{\text{SH}}.$$

Create publicly verifiable evidence:

3. Party P_h samples a coin share $r^h \xleftarrow{\$} [t]$, a randomness share $u^h \xleftarrow{\$} \{0, 1\}^\kappa$, sends the message $(r^h, u^h, \{d_j^h\}_{j \in [t]})$ to \mathcal{F}_{PG} and receives time-lock puzzle p as response.
4. For each $j \in [t]$, Party P_h creates a signature $\sigma_j^h \leftarrow \text{Sign}_{sk_h}(\text{data}_j)$, where the signed data is defined as

$$\text{data}_j := (h, j, \{\text{seed}_j^{(2,i)}\}_{i \in [n]}, \{c_j^i\}_{i \in [n]}, p, \text{trans}_j).$$

P_h broadcasts its signatures and verifies the received signatures.

Optimistic case:

5. If any of the following cases happens
 - P_h has not received valid messages in the first protocol steps in the expected communication round.
 - \mathcal{F}_{PG} returned **abort**, or
 - any other party has sent **abort**party P_h broadcasts and outputs **abort**.
6. Otherwise, P_h sends **continue_n** to \mathcal{F}_{PG} , receives $(r, \{d_j^{*i}\}_{i \in [n], j \in [t] \setminus r})$ as response and calculates

$$(m, \text{cert}) := \text{Blame}(\text{view}^h)$$

where view^h is the view of P_h .

If $\text{cert} \neq \perp$, broadcast **cert** and output **corrupted_m**. Otherwise, P_h outputs y_r^h .

Pessimistic case:

7. If \mathcal{F}_{PG} returned **abort** in step 6, P_h solves the time-lock puzzle

$$((r, \{d_j^{*i}\}_{i \in [n], j \in [t] \setminus r}), \pi) := \text{TL.Solve}(pp, p)$$

and calculates

$$(m, \text{cert}) := \text{Blame}(\text{view}^h)$$

where view^h is the view of P_h .

If $\text{cert} \neq \perp$, broadcast **cert** and output **corrupted_m**. Otherwise, output **abort**.

4.3 Blame-Algorithm

Our PVC compiler uses an algorithm **Blame** in order to verify the behavior of all parties in the opened protocol instances and to generate a certificate of misbehavior if cheating has been detected. It takes the view of a party as input and outputs the index of the corrupted party in addition to the certificate. If there are several malicious parties the algorithm selects the one with the minimal index.

Algorithm Blame

On input the view view of a party which contains:

- public parameters (n, t)
- public seed shares $\{\text{seed}_j^{(2,i)}\}_{i \in [n]}$
- shared coin r
- private seed share commitments and decommitments $\{c_j^i, d_j^i\}_{i \in [n], j \in [t] \setminus r}$
- additional certificate information
 $(\{\text{pk}_j\}_{i \in [n]}, \{\text{data}_j\}_{j \in [t]}, \pi, \{\sigma_j^i\}_{i \in [n], j \in [t]})$

do:

1. Calculate $\text{seed}_j^{(1,i)} := \text{Open}(c_j^i, d_j^i)$ for each $i \in [n], j \in [t] \setminus r$.
2. Let $M_1 := \{(i, j) \in ([n], [t] \setminus r) : \text{seed}_j^{(1,i)} = \perp\}$. If $M_1 \neq \emptyset$, choose the tuple $(m, l) \in M_1$ with minimal m and l , prioritized by m , compute $(\cdot, \pi) := \text{TL.Solve}(pp, p)$, if $\pi = \perp$, set $\text{cert} := (\text{pk}_m, \text{data}_j, \pi, r, \{d_j^i\}_{i \in [n], j \in [t] \setminus r}, \sigma_l^m)$ and output (m, cert) .
3. Set $\text{seed}_j^i := \text{seed}_j^{(1,i)} \oplus \text{seed}_j^{(2,i)}$ for all $i \in [n]$ and $j \in [t] \setminus r$.
4. Re-run Π_{SH} for all $j \in [t] \setminus r$ by simulating the view of all other parties: In the j -th instance simulate all parties P_i with randomness seed_j^i for $i \in [n]$ and receive (trans'_j, \cdot) .
5. Let $M_2 := \{j \in [t] \setminus r : \text{trans}'_j \neq \text{trans}_j\}$. If $M_2 \neq \emptyset$, determine the minimal index m such that P_m is the first party that has deviated from the protocol description in an instance $l \in M_2$. If P_m has deviated from the protocol description in several instances $l \in M_2$, choose the smallest such l . Then, set $\text{cert} := (\text{pk}_m, \text{data}_l, \{d_l^i\}_{i \in [n]}, \sigma_l^m)$ and output (m, cert) .
6. Output $(0, \perp)$.

4.4 Judge-Algorithm

The Judge-algorithm receives the certificate and outputs either the identity of the corrupted party or \perp . The execution of this algorithm requires no interaction with the parties participating in the protocol execution. Therefore, it can also be executed by any third party which possesses a certificate cert . If the output is pk_m for $m \in [n]$, the executing party is convinced that party P_m misbehaved during the protocol execution. The Judge-algorithm is parameterized with n, t, pp , and Π_{SH} .

Algorithm Judge(cert)

Inconsistency certificate:

On input $\text{cert} = (\text{pk}_m, \text{data}, \pi, r, \{d_j^i\}_{i \in [n], j \in [t] \setminus r}, \sigma_l^m)$ do:

- If $\text{Verify}_{\text{pk}_m}(\text{data}; \sigma_l^m) = \perp$, output \perp .
- Parse data to $(m, l, \cdot, \{c_i^i\}_{i \in [n]}, p, \cdot)$.
- If $\text{TL.Verify}(pp, p, (r, \{d_j^i\}_{i,j}), \pi) = 0$ output \perp .
- If $r = l$, output \perp .
- If $\text{Open}(c_l^m, d_l^m) \neq \perp$, output \perp . Else output pk_m .

Deviation certificate:

On input $\text{cert} = (\text{pk}_m, \text{data}, \{d_l^i\}_{i \in [n]}, \sigma_l^m)$.

- If $\text{Verify}_{\text{pk}_m}(\text{data}; \sigma_l^m) = \perp$, output \perp .
- Parse data to $(m, l, \{\text{seed}_l^{(2,i)}\}_{i \in [n]}, \{c_i^i\}_{i \in [n]}, \cdot, \text{trans}_l)$.
- Set $\text{seed}_l^{(1,i)} \leftarrow \text{Open}(c_i^i, d_l^i)$ for each $i \in [n]$. If any $\text{seed}_l^{(1,i)} = \perp$, output \perp .
- Set $\text{seed}_l^i := \text{seed}_l^{(1,i)} \oplus \text{seed}_l^{(2,i)}$ for each i .
- Simulate Π_{SH} using the seeds seed_l^i as randomness of party P_i and get result (trans'_l, \cdot) .

- If $\text{trans}'_i = \text{trans}_i$, output \perp . Otherwise, determine the index m' of the first party that has deviated from the protocol description. If $m \neq m'$, output \perp . Otherwise, output pk_m .

Ill formatted: If the cert cannot be parsed to neither of the two above cases, output (\perp) .

5 Security

In this section, we show the security of the compiled protocol described in Section 4. To this end, we state the security guarantee in Theorem 1 and prove its correctness in the following.

Theorem 1. *Let Π_{SH} be a n -party protocol, receiving no private inputs, which is secure against a passive adversary that corrupts up to $n - 1$ parties. Let the signature scheme $(\text{Gen}, \text{Sign}, \text{Verify})$ be existentially unforgeable under chosen-message attacks and let the verifiable time-lock puzzle scheme TL be secure with hardness parameter $\mathcal{T} > 2 \cdot T_c$. Let $(\text{Commit}, \text{Open})$ be an extractable commitment scheme which is computationally binding and hiding. Then protocol Π_{PVC} along with algorithms **Blame** and **Judge** is secure against a covert adversary that corrupts up to $n - 1$ parties with deterrence $\epsilon = 1 - \frac{1}{t}$ and public verifiability according to definition 2 in the $(\mathcal{F}_{\text{coin}}, \mathcal{F}_{\text{PG}})$ -hybrid model.⁶*

Proof. We prove security of the compiled protocol Π_{PVC} by showing simulatability, public verifiability, and defamation freeness according to Definition 2 separately.

5.1 Simulatability

In order to prove that Π_{PVC} meets covert security with ϵ -deterrent, we define an ideal-world simulator \mathcal{S} using the adversary \mathcal{A} in a black-box way as a subroutine and playing the role of the parties corrupted by \mathcal{A} when interacting with the ideal covert-functionality \mathcal{F}_{Cov} .

The simulator and the proof that the joint distribution of the honest parties' outputs and the view of \mathcal{A} in the ideal world is computationally indistinguishable from the honest parties' outputs and the view of \mathcal{A} in the real world are given in the full version of the paper.

5.2 Public Verifiability

We first argue that an adversary is not able to perform what we call a *detection dependent abort*. This means that once an adversary learns if its cheating will be detected, it can no longer prevent honest parties from generating a certificate.

⁶ See section 3.1, for details on the notion of time and the communication model.

In order to see this, note that withholding valid signatures by corrupted parties in step 4 results in an abort of all honest parties. In contrast, if all honest parties receive valid signatures from all other parties in step 4, then they are guaranteed to obtain the information encapsulated in the time-lock puzzle, i.e., the coin r and the decommitments of all parties $\{d_j^i\}_{i \in [n], j \in [t] \setminus r}$. Either, all parties jointly trigger the puzzle generation functionality \mathcal{F}_{PG} to output the values or in case any corrupted party aborts, an honest party can solve the time-lock puzzle without interaction. Thus, it is not possible for a rushing adversary that gets the output of \mathcal{F}_{PG} in step 6 first, to prevent the other parties from learning it at some time as well. Moreover, the adversary also cannot extract the values from the puzzles before making the decision if it wants to continue or abort, as the decision has to be made in time smaller than the time required to solve the puzzle. Thus, the adversary's decision to continue or abort is independent from the coin r and therefore independent from the event of being detected or not.

Secondly, we show that the Judge-algorithm will accept a certificate, created by an honest party, except with negligible probability. Assume without loss of generality that some malicious party P_m has cheated, cheating has been detected and a certificate (blaming party P_m) has been generated. As we have two types of certificates, we will look at them separately.

If an honest party outputs an *inconsistency certificate*, it has received an inconsistent commitment-opening pair (c_l^m, d_l^m) for some $l \neq r$. The value c_l^m is signed directly by P_m and d_l^m indirectly via the signed time-lock puzzle p . Hence, Judge can verify the signatures and detect the inconsistent commitment of P_m as well. Note that due to the verifiability of our time-lock construction, the Judge-algorithm does not have to solve the time-lock puzzle itself but just needs to verify a given solution. This enables the algorithm to be executed efficiently.

If an honest party outputs a *deviation certificate*, it has received consistent openings for all $j \neq r$ from all other parties, but party P_m was the first party who deviated from the specification of Π_{SH} in some instance $l \in [t] \setminus r$. Since Π_{SH} requires no input from the parties, deviating from its specification means using different randomness than derived from the seeds generated at the beginning of the compiled protocol. As P_m has signed the transcript trans_l , the private seed-commitments of all parties $\{c_l^i\}_{i \in [n]}$, the public seeds $\{\text{seed}^{(2,i)}\}_{i \in [n]}$, and the certificate contains the valid openings $\{d_l^i\}_{i \in [n]}$, the Judge-algorithm can verify that P_m was the first party who misbehaved in instance l the same way the honest party does. Note that it is not necessary for Judge to verify that $j \neq r$, because the certificate generating party can only gain valid openings $\{d_l^i\}_{i \in [n]}$ for $j \neq r$.

5.3 Defamation Freeness

Assume, without loss of generality, that some honest party P_h is blamed by the adversary. We show defamation freeness for the two types of certificates separately via a reduction to the security of the commitment scheme, the signature scheme and the time-lock puzzle scheme.

First, assume there is a valid *inconsistency certificate* cert^* blaming P_h . This means that there is a valid signatures of P_h on a commitment c_j^{*h} and a time-lock puzzle p^* that has a solution s^* which contains an opening d_j^{*h} such that $\text{Open}(c_j^{*h}, d_j^{*h}) = \perp$ and $j \neq r$. As P_h is honest, P_h only signs a commitment c_j^{*h} which equals the commitment honestly generated by P_h during the seed generation. We call such a c_j^{*h} *correct*. Thus, c_j^{*h} is either correct or the adversary can forge signatures. Similar, P_h does only sign the puzzle p^* received by \mathcal{F}_{PG} . This puzzle is generated on the opening value provided by all parties. Since P_h is honest, correct opening values are inserted. Therefore, the signed puzzle p^* either contains the correct opening value or the adversary can forge signatures. Due to the security guarantees of the puzzle, the adversary has to either provide the correct solution s^* or can break the soundness of the time-lock puzzle scheme. To sum it up, an adversary creating a valid *inconsistency certificate* contradicts to the security assumptions specified in Theorem 1.

Second, assume there is a valid *deviation certificate* cert^* blaming P_h . This means, there is a protocol transcript trans_j^* in which P_h is the first party that has sent a message which does not correspond to the next-message function of Π_{SH} and the randomness, seed_j^h used by the judge to simulate P_h . As P_h is honest, either trans_j^* or seed_j^h needs to be incorrect. Also, P_h does not create a signature for an invalid trans_j^* . Thus, trans_j^* is either correct or the adversary can forge signatures. The seed_j^h is calculated as $\text{seed}_j^h := \text{seed}_j^{(1,h)} \oplus \text{seed}_j^{(2,h)}$. The public seed $\text{seed}_j^{(2,h)}$ is signed by P_h and provided directly. The private seed of P_h is provided via a commitment-opening pair (c_j^h, d_j^h) , where c_j^h is signed by P_h . As above, c_j^h and $\text{seed}_j^{(2,h)}$ are either correct or the adversary can forge signatures. Similar, d_j^h is either correct or the adversary can break the binding property of the commitment scheme. If the certificate contains correct $(\text{trans}_j^*, c_j^h, d_j^h, \text{seed}_j^{(2,h)})$ the certificate is not valid. Thus, when creating an accepting cert^* , the adversary has either broken the signature or the commitment scheme which contradicts to the assumption of Theorem 1. □

6 Evaluation

6.1 Efficiency of our Compiler

In Section 4, we presented a generic compiler for transforming input-independent multi-party computation protocols with semi-honest security into protocols that offer covert security with public verifiability. We elaborate on efficiency parameters of our construction in the following.

The deterrence factor $\epsilon = \frac{t-1}{t}$ only depends on the number of semi-honest protocol executions t . In particular, ϵ is independent of the number of parties. This property allows for achieving the same deterrence factor for a fixed number of semi-honest executions while the number of parties increases. Our compiler

therefore facilitates secure computation with a large number of parties. Furthermore, the deterrence factor grows with the number of semi-honest instances (t), similar to previous work based on cut-and-choose (e.g., [AL07, AO12, DOS20]). Concretely, this means that for only five semi-honest instances, our compiler achieves a cheating detection probability of 80%. Moreover, the semi-honest instances are independent of each other and, hence, can be executed in parallel. This means, that the communication and computation complexity in comparison to a semi-honest protocol increases by factor t . However, our compiler preserves the round complexity of the semi-honest protocol. Hence, it is particularly useful for settings and protocols in which the round complexity constitutes the major efficiency bottleneck. Similarly, the requirement of sending all messages to all parties further increases the communication overhead by a factor of $n - 1$ but does not affect the round complexity. Since this requirement is inherent to all known publicly verifiable covert secure protocols, e.g., [DOS20], these protocols incur a similar communication overhead.

While our compiler requires a maliciously secure puzzle generation functionality, we stress that the complexity of the puzzle generation is independent of the cost of the semi-honest protocol. Therefore, the relative overhead of the puzzle generation shrinks for more complex semi-honest protocols. One application where our result may be particularly useful is for the preprocessing phase of multi-party computation, e.g., protocols for generating garbled circuits or multiplication triples. In such protocols, one can generate several circuits resp. triples that are used in several online instances but require just one puzzle generation.

For the sake of concreteness, we constructed a boolean circuit for the puzzle generation functionality and estimated its complexity in terms of the number of AND-gates. The construction follows a naive design and should not constitute an efficient solution but should give a first impression on the circuit complexity. We present some intuition on how to improve the circuit complexity afterwards.

We utilize the RSW VTLP construction described in Section 3.2 with a hybrid construction, in which a symmetric encryption key is locked within the actual time-lock puzzle and is used to encrypt the actual secret. Note that the RSW VTLP is not optimized for MPC scenarios. Since our compiler can be instantiated with an arbitrary VTLP satisfying Definition 3, any achievements in the area of MPC-friendly TLP can result into an improved puzzle generation functionality for our compiler. To instantiate the symmetric encryption operation, we use the LowMC [ARS⁺15] cipher, an MPC-friendly cipher tailored for boolean circuits.

Let n be the number of parties, t being the number of semi-honest instances, κ denoting the computational security parameter, and N represents the RSA modulus used for the RSW VTLP. We use the notation $|x|$ to denote the bit length of x . The total number of AND-gates of our naive circuit is calculated as follows:

$$\begin{aligned}
& (n - 1) \cdot (11|t| + 22|N| + 12) \\
& + nt \cdot (4|t| + 2\kappa + 755) \\
& + 192|N|^3 + 112|N|^2 + 22|N|
\end{aligned}$$

It is easy to see that the number of AND-gates is linear in both n and t . The most expensive part of the puzzle generation is the computation of two exponentiations required for the RSW VTLP, since the number of required AND-gates is cubic in $|N|$ for an exponentiation. However, we can slightly adapt our puzzle generation functionality and protocol to remove these exponentiations from the maliciously secure puzzle generation protocol. For the sake of brevity, we just give an intuition here.

Instead of performing the exponentiations g^u and h^u required for the puzzle creation within the puzzle generation functionality, we let each party P_i input a 0-puzzle consisting of the two values $g_i = g^{u_i}$ and $h_i = h^{u_i}$. The products of all g_i respectively h_i are used as g^* and h^* for the VTLP computation. Since we replace the exponentiations with multiplications, the number of AND-gates is quadratic instead of cubic in $|N|$.

Note that this modification enables a malicious party to modify the resulting puzzle by inputting a non-zero puzzle. Intuitively, the attacker can render the puzzle invalid such that no honest party can create a valid certificate or the puzzle can be modified such that a corrupted party can create a valid certificate defaming an honest party. Concretely, one possible attack is to input inconsistent values g_i and h_i , i.e., to use different exponents for the two exponentiations. As such an attack must be executed without knowledge of the coin r , it is sufficient to detect invalid inputs and consider such behavior as an early abort. To this end, parties have to provide u_i to the puzzle generation functionality and the functionality outputs $u = \sum u_i$, g^* and h^* in the second output round together with the coin and the seed openings. By comparing if $g^* = g^u$ and $h^* = h^u$, each party can check the validity of the puzzle. Finally, we need to ensure that a manipulated puzzle cannot be used to create an inconsistency certificate blaming an honest party. Such false accusation can easily be prevented, e.g., by adding some zero padding to the value inside the puzzle such that any invalid puzzle input renders the whole puzzle invalid.

6.2 Comparison with Prior Work

To the best of our knowledge, our work is the first to provide a fully specified publicly verifiable multi-party computation protocol against covert adversaries. Hence, we cannot compare to existing protocols directly. However, Damgård et al. [DOS20] have recently presented two compilers for constructing publicly verifiable covert secure protocols from semi-honest secure protocols in the two-party setting, one for input-independent and one for input-dependent protocols. For the latter, they provide an intuition on how to extend the compiler to the multi-party case. However, there is no full compiler specification for neither

input-dependent nor input-independent protocols. Still, there exist a natural extension for the input-independent compiler, which we can compare to.

The major difference between our input-independent protocol and their input-independent protocol, is the way the protocols prevent *detection dependent abort*. In the natural extension to Damgård et al. [DOS20], which we call the *watchlist approach* in the following, each party independently selects a subset of instances it wants to check and receives the corresponding seeds via oblivious transfer. The transcript of the oblivious transfer together with the receiver’s randomness can be used by the receiver to prove integrity of its watchlist to the judge; similar to the seed commitments and openings used in our protocol. The watchlists are only revealed after each party receives the data required to create a certificate in case of cheating detection, i.e., the signatures by the other parties. Once a party detects which instances are checked, it is too late to prevent the creation of a certificate. Our approach utilizes time-lock puzzles for the same purpose.

In the watchlist approach, all parties have different watchlists. For t semi-honest instances and watchlists of size $s \geq \frac{t}{n}$, there is a constant probability $\Pr[\text{bad}]$ that no semi-honest instance remains unwatched which leads to a failure of the protocol. Thus, parties either need to choose $s < \frac{t}{n}$ and hence $\epsilon = \frac{s}{t} < \frac{1}{n}$ or run several executions of the protocol. For the latter, the probability of a protocol failure $\Pr[\text{bad}]$ and the expected number of protocol runs runs are calculated based on the inclusion-exclusion principle as follows:

$$\begin{aligned} \Pr[\text{bad}] &= 1 - \frac{\sum_{k=1}^t (-1)^{(k-1)} * \binom{t}{k} * (\prod_{j=0}^{s-1} (t-j-k))^n}{\prod_{j=0}^{s-1} (t-j)^n} \\ &= 1 - \sum_{k=1}^t (-1)^{(k-1)} \cdot \binom{t}{k} \cdot \left(\frac{(t-k)! \cdot (t-s)!}{(t-k-s)! \cdot t!} \right)^n \\ \text{runs} &= \Pr[\text{bad}]^{-1} \end{aligned}$$

Setting the watchlist size $s \geq \frac{t}{n}$ such that there is a constant failure probability has the additional drawback that the repetition can be abused to amplify denial-of-service attacks. An adversary can enforce a high failure probability by selecting its watchlists strategically. If $s \geq \frac{t}{(n-1)}$ and $n-1$ parties are corrupted, the adversary can cause an error with probability 1 which enables an infinite DoS-attack.

This restriction of the deterrence factor seems to be a major drawback of the watchlist approach. Although our approach has an additional overhead due to the puzzle generation, which is independent of the complexity of the transformed protocol and thus amortizes over the complexity of the base protocols, it has the benefit that it immediately supports an arbitrary deterrence factor ϵ . This is due to the fact that the hidden shared coin toss determines a single watchlist shared by all parties. In Table 1, we display the maximal deterrence factor of our approach ϵ in comparison to the maximal deterrence factor of the watchlist approach without protocol repetitions ϵ' for different settings. Additionally,

we provide the number of expected runs required to achieve ϵ in the watchlist approach with repetitions.

n	t	Our approach	Watchlist approach		
		ϵ	ϵ'	or	runs
	2	1/2	-		2
2	3	2/3	1/3		3
	10	9/10	4/10		10
	2	1/2	-		4
3	4	3/4	1/4		16
	10	9/10	3/10		100
5	2	1/2	-		16
	6	5/6	1/6		1296

Table 1. Maximal deterrence factor or expected number of runs of the watchlist approach in comparison to our approach.

Acknowledgments

The first, third, and fourth authors were supported by the German Federal Ministry of Education and Research (BMBF) *iBlockchain project* (grant nr. 16KIS0902), by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) *SFB 1119 - 236615297 (CROSSING Project S7)*, by the BMBF and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the *National Research Center for Applied Cybersecurity ATHENE*, and by Robert Bosch GmbH, by the Economy of Things Project. The second author was supported by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister’s Office, and by ISF grant No. 1316/18.

References

- [AL07] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *TCC 2007*.
- [AO12] Gilad Asharov and Claudio Orlandi. Calling out cheaters: Covert security with public verifiability. *ASIACRYPT 2012*.
- [ARS⁺15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. *EUROCRYPT 2015, Part I*.
- [BBBF18] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. *CRYPTO 2018, Part I*.

- [Bea92] Donald Beaver. Efficient multiparty protocols using circuit randomization. *CRYPTO'91*.
- [BGJ⁺16] Nir Bitansky, Shafi Goldwasser, Abhishek Jain, Omer Paneth, Vinod Vaikuntanathan, and Brent Waters. Time-lock puzzles from randomized encodings. *ITCS 2016*.
- [BW88] Johannes Buchmann and Hugh C. Williams. A key-exchange system based on imaginary quadratic fields. *Journal of Cryptology*, June 1988.
- [DG20] Samuel Dobson and Steven D. Galbraith. Trustless groups of unknown order with hyperelliptic curves. *IACR Cryptol. ePrint Arch. 2020*, 2020.
- [DGN10] Ivan Damgård, Martin Geisler, and Jesper Buus Nielsen. From passive to covert security at low cost. *TCC 2010*.
- [DKL⁺13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. *ESORICS 2013*.
- [DOS20] Ivan Damgård, Claudio Orlandi, and Mark Simkin. Black-box transformations from passive to covert security with public verifiability. *CRYPTO 2020, Part II*.
- [DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zarkarias. Multiparty computation from somewhat homomorphic encryption. *CRYPTO 2012*.
- [GMS08] Vipul Goyal, Payman Mohassel, and Adam Smith. Efficient two party and multi party computation against covert adversaries. *EUROCRYPT 2008*.
- [GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. *19th ACM STOC 1987*.
- [HKK⁺19] Cheng Hong, Jonathan Katz, Vladimir Kolesnikov, Wen-jie Lu, and Xiao Wang. Covert security with public verifiability: Faster, leaner, and simpler. *EUROCRYPT 2019, Part III*.
- [HVW20] Carmit Hazay, Muthuramakrishnan Venkatasubramanian, and Mor Weiss. The price of active security in cryptographic protocols. *EUROCRYPT 2020, Part II*.
- [IOZ14] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multi-party computation with identifiable abort. *CRYPTO 2014, Part II*.
- [IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. *CRYPTO 2008*.
- [KM15] Vladimir Kolesnikov and Alex J. Malozemoff. Public verifiability in the covert model (almost) for free. *ASIACRYPT 2015, Part II*.
- [LOP11] Yehuda Lindell, Eli Oxman, and Benny Pinkas. The IPS compiler: Optimizations, variants and concrete efficiency. *CRYPTO 2011*.
- [MMV11] Mohammad Mahmoody, Tal Moran, and Salil P. Vadhan. Time-lock puzzles in the random oracle model. *CRYPTO 2011*.
- [MT19] Giulio Malavolta and Sri Aravinda Krishnan Thyagarajan. Homomorphic time-lock puzzles and applications. *CRYPTO 2019, Part I*.
- [Pie19] Krzysztof Pietrzak. Simple verifiable delay functions. *ITCS 2019*.
- [RSW96] Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. Technical report, Massachusetts Institute of Technology. Laboratory for Computer Science, 1996.
- [Wes19] Benjamin Wesolowski. Efficient verifiable delay functions. *EUROCRYPT 2019, Part III*.

- [WRK17a] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. *ACM CCS 17*.
- [WRK17b] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. *ACM CCS 17*.
- [YWZ20] Kang Yang, Xiao Wang, and Jiang Zhang. More efficient MPC from improved triple generation and authenticated garbling. *ACM CCS 2020*.
- [ZDH19] Ruiyu Zhu, Changchang Ding, and Yan Huang. Efficient publicly verifiable 2pc over a blockchain with applications to financially-secure computations. *ACM CCS 2019*.