Generic Compiler for Publicly Verifiable Covert Multi-Party Computation

Sebastian Faust[1], Carmit Hazay[2], David Kretzler[1], and Benjamin Schlosser[1]

[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. 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 $\frac{s}{\log |\mathrm{C}|}$ [WRK17b], where 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 8. 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.

Finally, we show how our compiler can be extended in order to reduce the size of cheating certificates and how techniques known from [IPS08] and [DOS20] can be adapted to our compiler in order to be applicable to input-dependent protocols.

Concurrent and independent work [SSS21]. Independently to our work, Scholl et al. [SSS21] presented two compilers where the first one transforms semi-honest protocols to covert protocols with identifiable abort and the second one compiles from semi-honest to covert security with public verifiability. We compare our work with their second compiler. Both works utilize time-lock puzzles as a building block. We point out the differences between the two works in the following.

First, while we realize a relaxed form of covert security where the adversary is able to learn her potential output before deciding to cheat, Scholl et al. consider the original covert security notion by Aumann and Lindell [AL07]. However, they restrict the class of supported functions that can be evaluated by the covert functionality to preprocessing functions where the adversary may choose the output of the corrupted parties. We note that for this restricted class of functions, our security notion implies the original one.

Second, our compiler invokes an actively secure protocol over a circuit with $O(nt\kappa + \log^2(N))$ AND gates, for n parties, t semi-honest instances, a security parameter κ, and an RSA moduls N, while their compiler evaluates an actively secure protocol over a circuit with only $O(nt\kappa)$ AND gates. Third, Scholl et al. require the judge to solve n time-lock puzzles for n parties while our compiler utilizes only a single puzzle. Moreover, we present the notion of a verifiable time-lock puzzle that allows for a simple verification of puzzle solutions which enables a more efficient judge. Finally, we mention that Scholl et al. showed concrete efficiency parameters while we aim for an asymptotic improvement.

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 obliviously 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 8 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 $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$, where $f = (f_1, \ldots, f_n)$, be the function realized by Π. For every input vector $\bar{x} = (x_1, \ldots, x_n)$ the output vector is $\bar{y} = (f_1(\bar{x}), \ldots, f_n(\bar{x}))$ 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 $\mathsf{REAL}_{\Pi,\mathcal{A}(z),I}(\bar{x}, 1^\kappa)$ to be the output vector of the protocol execution of Π on input $\bar{x} = (x_1, \ldots, x_n)$ and security parameter κ, where the adversary $\mathcal{A}$ on auxiliary input z corrupts the parties $I \subseteq [n]$. By $\mathsf{OUTPUT}_i(\mathsf{REAL}_{\Pi,\mathcal{A}(z),I}(\bar{x}, 1^\kappa))$, 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 $\epsilon : \mathbb{N} \to [0, 1]$ be a function, then the execution in the ideal model works as follows.

[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].

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 $(y_1, \ldots, y_m) = f(\bar{w})$ 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 $w_i = \mathsf{corrupted}_i$ to the trusted party, then the trusted party sends $\mathsf{corrupted}_i$ to all of the honest parties and halts. If multiple parties send $\mathsf{corrupted}_i$, then the trusted party disregards all but one of them (say, the one with the smallest index i). If both $\mathsf{corrupted}_i$ and abort messages are sent, then the trusted party ignores the $\mathsf{corrupted}_i$ message.

Attempted cheat option: If a corrupted party sends additional input $w_i = \mathsf{cheat}_i$ to the trusted party (as above: if there are several messages $w_i = \mathsf{cheat}_i$ 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 $\mathsf{corrupted}_i$ to the adversary and all of the honest parties.
2. With probability $1 - \epsilon$, the trusted party sends $\mathsf{undetected}$ 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, $\mathsf{corrupted}_i$ or cheat_i, the ideal execution continues below.

Trusted party answers honest parties: If the trusted party did not receive $\mathsf{corrupted}_i$, 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 $\mathsf{IDEALC}^{\epsilon}_{f,\mathcal{S}(z),I}(\bar{x}, 1^{\kappa})$ 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 $z \in \{0,1\}^*$:*

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

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 $\mathcal{F}_{\mathsf{Cov}}$.

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 $\mathsf{corrupted}_j$ in the protocol execution for $j \in I$ and returns a certificate Cert, i.e., $\mathsf{Cert} := \mathsf{Blame}(\mathsf{view}_i)$. 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 $\mathsf{Cert} = \mathsf{Blame}(\mathsf{view}_i)$ 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 $(\Pi', \mathsf{Blame}, \mathsf{Judge})$* 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 $\mathsf{OUTPUT}_j(\mathsf{REAL}_{\Pi,\mathcal{A}(z),I}(\bar{x}, 1^\kappa)) = \mathsf{corrupted}_i$ for $j \in [n] \setminus I$ and $i \in I$ then:
   $$\Pr[\mathsf{Judge}(\mathsf{Cert}) = \mathsf{id}_i] > 1 - \mu(n),$$
   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 $j \in [n] \setminus I$:*
   $$\Pr[\mathsf{Cert}^* \leftarrow \mathcal{A}; \mathsf{Judge}(\mathsf{Cert}^*) = \mathsf{id}_j] < \mu(n).$$

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 $\delta_{\mathsf{Sq}}(\kappa) = T(\mathcal{S}, (N, g))$, 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* ($\mathsf{TL.Setup}, \mathsf{TL.Generate}, \mathsf{TL.Solve}, \mathsf{TL.Verify}$) *defined as follows.*

– $(pp) \leftarrow \mathsf{TL.Setup}(1^\lambda, \mathcal{T})$ *takes as input the security parameter* 1^λ *and a hardness parameter* $\mathcal{T}$*, and outputs public parameter pp.*
– $p \leftarrow \mathsf{TL.Generate}(pp, s)$ *takes as input public parameters pp and a solution* $s \in \mathcal{S}$ *and outputs a puzzle p.*
– $(s, \pi) \leftarrow \mathsf{TL.Solve}(pp, p)$ *is a deterministic algorithm that takes as input public parameters pp and a puzzle p and outputs a solution s and a proof* π*.*
– $b := \mathsf{TL.Verify}(pp, p, s, \pi)$ *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* $\mathsf{TL.Verify}$ *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* $pp \leftarrow \mathsf{TL.Setup}(1^\lambda, \mathcal{T})$*, and for all s, it holds that*

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

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

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

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

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

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*

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

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 $\mathsf{Primes}(2k)$. The set $\mathsf{Primes}(2k)$ contains the first 2^{2k} prime numbers, where k denotes the security level (typically 128, 192 or 256).

The $\mathsf{TL.Setup}$-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 know. 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 $\mathsf{TL.Setup}$-algorithm should be executed in a trusted way.

The $\mathsf{TL.Generate}$-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 $\mathsf{TL.Solve}$-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 $\mathsf{TL.Verify}$-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

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

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

$\mathsf{TL.Generate}(pp, s)$:

- Parse $pp := (\mathcal{T}', N, Z)$ and $Z := (g, h)$.

```
  – Sample uniform $r \xleftarrow{\$} \{1, \ldots, N^2\}$.
  – Compute $g^* := g^r$ and $h^* := h^r$.
  – Set $c^* := h^* \cdot s \mod N$.
  – Output $p := (g^*, c^*)$.

$\mathsf{TL.Solve}(pp, p)$:

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

$\mathsf{TL.Verify}(pp, p, s, \pi)$:

  – Parse $pp := (\mathcal{T}', N, Z)$.
  – Parse $p := (g^*, c^*)$.
  – Compute $\ell = H_{\mathsf{prime}}(\mathsf{bin}(g^*)||\star||\mathsf{bin}(s)) \in \mathsf{Primes}(2k)$ as the challenge.
  – Compute $r = 2^{\mathcal{T}'} \mod \ell$.
  – 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 $\mathsf{Primes}(2k)$ 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 $(c, d) \leftarrow \mathsf{Commit}(m)$ to commit to message m, where c is the commitment value and d denotes the decommitment or opening value. Similarly, we use $m' \leftarrow \mathsf{Open}(c, d)$ 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 $\mathsf{Commit}(x) = H(i, x, r)$ where i is the identity of the committer, $H : \{0,1\}^* \to \{0,1\}^{2\kappa}$ is a random oracle and $r \xleftarrow{\$} \{0,1\}^\kappa$.

3.4 Signature Scheme

We use a signature scheme $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ 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 $(\mathsf{pk}, \mathsf{sk})$. 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 $(y^1, \ldots, y^n) \leftarrow f()$, 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 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 $\mathcal{F}_{\mathsf{coin}}$ 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 $\mathsf{seed} \xleftarrow{\$} \{0,1\}^\kappa$ to all parties.

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

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

- Sample $\mathsf{seed} \xleftarrow{\$} \{0,1\}^\kappa$.
- 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.

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

3.7 Puzzle Generation Functionality

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

Functionality $\mathcal{F}_{\mathsf{PG}}$

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 $r := \sum_{i=1}^n r^i \mod t$ and $u := \bigoplus_{i=1}^n u^i$.
- Generate puzzle $p \leftarrow \mathsf{TL.Generate}(pp, (r, \{d_j^i\}_{i\in[n],j\in[t]\setminus r}))$ 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 $\mathsf{continue}$ 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 $\mathsf{continue}$, 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

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

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 $\mathsf{seed}^{(1,h)}$. Additionally, P_h and all other parties get uniform κ-bit strings $\{\mathsf{seed}^{(2,i)}\}_{i\in[n]}$, which are the public seed shares of all parties. The randomness used by P_h in the semi-honest protocol will be derived from $\mathsf{seed}^h := \mathsf{seed}^{(1,h)} \oplus \mathsf{seed}^{(2,h)}$. 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 $\mathsf{seed}^{(1,h)}$, sets $(c^h, d^h) \leftarrow \mathsf{Commit}(\mathsf{seed}^{(1,h)})$, and sends c^h to all parties.

(b) **Public coin-phase**
For each $i \in [n]$, party P_h sends ok to $\mathcal{F}_{\mathsf{coin}}$ and receives $\mathsf{seed}^{(2,i)}$.
Output
If P_h has not received all messages in the expected communication rounds or any $\mathsf{seed}^{(2,i)} = \perp$, it sends abort to all parties and outputs abort.
Otherwise, it outputs $(\mathsf{seed}^{(1,h)}, d^h, \{\mathsf{seed}^{(2,i)}, c^i\}_{i\in[n]})$.

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 $(\mathsf{sk}, \mathsf{pk})$ and all parties know the public keys of the other parties. Furthermore, we suppose the setup of the verifiable time-lock puzzle scheme $\mathsf{TL.Setup}$ 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 $\mathsf{TL.Setup}$-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 $\mathcal{F}_{\mathsf{PG}}$ 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 $\mathcal{F}_{\mathsf{PG}}$ and can verify that opening the commitments is successful. In the pessimistic case, solving the time-lock puzzle reveals the input to $\mathcal{F}_{\mathsf{PG}}$. 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
$$(\mathsf{seed}_j^{(1,h)}, d_j^h, \{\mathsf{seed}_j^{(2,i)}, c_j^i\}_{i\in[n]}) \leftarrow \mathit{\Pi}_{\mathsf{SG}}$$
and computes $\mathsf{seed}_j^h := \mathsf{seed}_j^{(1,h)} \oplus \mathsf{seed}_j^{(2,h)}$.

Semi-honest protocol execution:

2. Party P_h engages in t instances of the protocol $\mathit{\Pi}_{\mathsf{SH}}$ 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:
$$(\mathsf{trans}_j, y_j^h) \leftarrow \mathit{\Pi}_{\mathsf{SH}}.$$

Create publicly verifiable evidence:

3. Party P_h samples a coin share $r^h \overset{\$}{\leftarrow} [t]$, a randomness share $u^h \overset{\$}{\leftarrow} \{0,1\}^\kappa$, sends the message $(r^h, u^h, \{d_j^h\}_{j\in[t]})$ to $\mathcal{F}_{\mathsf{PG}}$ and receives time-lock puzzle p as response.
4. For each $j \in [t]$, Party P_h creates a signature $\sigma_j^h \leftarrow \mathsf{Sign}_{\mathsf{sk}_h}(\mathsf{data}_j)$, where the signed data is defined as
$$\mathsf{data}_j := (h, j, \{\mathsf{seed}_j^{(2,i)}\}_{i\in[n]}, \{c_j^i\}_{i\in[n]}, p, \mathsf{trans}_j).$$
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.
   - $\mathcal{F}_{\mathsf{PG}}$ returned abort, or
   - any other party has sent abort

   party P_h broadcasts and outputs abort.
6. Otherwise, P_h sends $\mathsf{continue}_h$ to $\mathcal{F}_{\mathsf{PG}}$, receives $(r, \{d_j^{*i}\}_{i\in[n],j\in[t]\setminus r})$ as response and calculates
$$(m, \mathsf{cert}) := \mathsf{Blame}(\mathsf{view}^h)$$
where view^h is the view of P_h.
If $\mathsf{cert} \neq \perp$, broadcast cert and output $\mathsf{corrupted}_m$. Otherwise, P_h outputs y_r^h.

Pessimistic case:

7. If $\mathcal{F}_{\mathsf{PG}}$ returned abort in step 6, P_h solves the time-lock puzzle
$$((r, \{d_j^{*i}\}_{i\in[n],j\in[t]\setminus r}), \pi) := \mathsf{TL.Solve}(pp, p)$$
and calculates
$$(m, \mathsf{cert}) := \mathsf{Blame}(\mathsf{view}^h)$$
where view^h is the view of P_h.
If $\mathsf{cert} \neq \perp$, broadcast cert and output $\mathsf{corrupted}_m$. 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 $\{\mathsf{seed}_j^{(2,i)}\}_{i\in[n]}$
- 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 $(\{\mathsf{pk}_j\}_{i\in[n]}, \{\mathsf{data}_j\}_{j\in[t]}, \pi, \{\sigma_j^i\}_{i\in[n],j\in[t]})$

do:

1. Calculate $\mathsf{seed}_j^{(1,i)} := \mathsf{Open}(c_j^i, d_j^i)$ for each $i \in [n], j \in [t] \setminus r$.
2. Let $M_1 := \{(i,j) \in ([n], [t] \setminus r) : \mathsf{seed}_j^{(1,i)} = \bot\}$. If $M_1 \neq \emptyset$, choose the tuple $(m,l) \in M_1$ with minimal m and l, prioritized by m, compute $(\cdot, \pi) := \mathsf{TL.Solve}(pp, p)$, if $\pi = \bot$, set $\mathsf{cert} := (\mathsf{pk}_m, \mathsf{data}_j, \pi, r, \{d_j^i\}_{i\in[n],j\in[t]\setminus r}, \sigma_l^m)$ and output (m, cert).
3. Set $\mathsf{seed}_j^i := \mathsf{seed}_j^{(1,i)} \oplus \mathsf{seed}_j^{(2,i)}$ 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 $(\mathsf{trans}'_j, \cdot)$.
5. Let $M_2 := \{j \in [t] \setminus r : \mathsf{trans}'_j \neq \mathsf{trans}_j\}$. 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 $\mathsf{cert} := (\mathsf{pk}_m, \mathsf{data}_l, \{d_l^i\}_{i\in[n]}, \sigma_l^m)$ and output (m, cert).
6. Output $(0, \bot)$.

4.4 Judge-Algorithm

The Judge-algorithm receives the certificate and outputs either the identity of the corrupted party or $\bot$. 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 $\mathsf{Judge}(\mathsf{cert})$

Inconsistency certificate:
On input $\mathsf{cert} = (\mathsf{pk}_m, \mathsf{data}, \pi, r, \{d^i_j\}_{i\in[n], j\in[t]\setminus r}, \sigma^m_l)$ do:

- If $\mathsf{Verify}_{\mathsf{pk}_m}(\mathsf{data}; \sigma^m_l) = \perp$, output $\perp$.
- Parse data to $(m, l, \cdot, \{c^i_l\}_{i\in[n]}, p, \cdot)$.
- If $\mathsf{TL.Verify}(pp, p, (r, \{d^i_j\}_{i,j}), \pi) = 0$ output $\perp$.
- If $r = l$, output $\perp$.
- If $\mathsf{Open}(c^m_l, d^m_l) \neq \perp$, output $\perp$. Else output pk_m.

Deviation certificate:
On input $\mathsf{cert} = (\mathsf{pk}_m, \mathsf{data}, \{d^i_l\}_{i\in[n]}, \sigma^m_l)$.

- If $\mathsf{Verify}_{\mathsf{pk}_m}(\mathsf{data}; \sigma^m_l) = \perp$, output $\perp$.
- Parse data to $(m, l, \{\mathsf{seed}^{(2,i)}_l\}_{i\in[n]}, \{c^i_l\}_{i\in[n]}, \cdot, \mathsf{trans}_l)$.
- Set $\mathsf{seed}^{(1,i)}_l \leftarrow \mathsf{Open}(c^i_l, d^i_l)$ for each $i \in [n]$. If any $\mathsf{seed}^{(1,i)}_l = \perp$, output $\perp$.
- Set $\mathsf{seed}^i_l := \mathsf{seed}^{(1,i)}_l \oplus \mathsf{seed}^{(2,i)}_l$ for each i.
- Simulate Π_{SH} using the seeds seed^i_l as randomness of party P_i and get result $(\mathsf{trans}'_l, \cdot)$.
- If $\mathsf{trans}'_l = \mathsf{trans}_l$, 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* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ *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* $(\mathsf{Commit}, \mathsf{Open})$ *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 $(\mathcal{F}_{\mathsf{coin}}, \mathcal{F}_{\mathsf{PG}})$-hybrid model.* [6]

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

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 subroutine and playing the role of the parties corrupted by $\mathcal{A}$ when interacting with the ideal covert-functionality $\mathcal{F}_{\mathsf{Cov}}$.

The proof is given in the $(\mathcal{F}_{\mathsf{coin}}, \mathcal{F}_{\mathsf{PG}})$-hybrid world in which Π_{SH} with according simulators $\mathcal{S}_{\Pi}$ exist. Without loss of generality, we assume that P_h is honest and all other parties are corrupt. Scenarios in which other or less parties are corrupt, are symmetrical. For the sake of simplicity, we ignore messages ok that are sent to $\mathcal{F}_{\mathsf{Cov}}$. When $\mathcal{S}$ sends abort, cheat_i or $\mathsf{corrupted}_i$ on behalf of any corrupted party P_i to $\mathcal{F}_{\mathsf{Cov}}$, $\mathcal{S}$ also sends ok on behalf of all other corrupted parties. The simulator $\mathcal{S}$ is specified as follows:

0. $\mathcal{S}$ sets $r \xleftarrow{\$} [t]$, $M_1, M_2, L = \emptyset$ and sends ok to $\mathcal{F}_{\mathsf{Cov}}$ for $i \in [n] \setminus h$ and receives output $\{y^i\}_{i\in[n]\setminus h}$ from $\mathcal{F}_{\mathsf{Cov}}$. Then, $\mathcal{S}$ generates keys $(\mathsf{pk}_h, \mathsf{sk}_h)$ and sends pk_h to $\mathcal{A}$.
1. $\mathcal{S}$ acts like an honest party during the seed generation and simulates $\mathcal{F}_{\mathsf{coin}}$ as stated in the definition. This way, $\mathcal{S}$ obtains public seed shares $\{\mathsf{seed}_j^{(2,i)}\}_{i\in[n]}$ and commitments $\{c_j^i\}_{i\in[n]}$ for $j \in [t]$. By simulating the honest party, $\mathcal{S}$ gets private seed shares $\mathsf{seed}_j^{(1,h)}$, commitments and openings $(c_j^h, d_j^h) \leftarrow \mathsf{Commit}(\mathsf{seed}_j^{(1,h)})$ for $j \in [t]$. Use the extractability property of the commitment scheme to obtain the private seed shares $\{\mathsf{seed}_j^{(1,i)}\}_{i\in[n]\setminus h}$. Finally, compute $\mathsf{seed}_j^i := \mathsf{seed}_j^{(1,i)} \oplus \mathsf{seed}_j^{(2,i)}$ for $j \in [t]$ and $i \in [n]$.
2. $\mathcal{S}$ runs the executions of Π_{SH} with $\mathcal{A}$ as follows and let trans_j denote the transcript and y_j^h denote the output of P_h in the j-th execution:
   - In the r-th execution, $\mathcal{S}$ invokes $\mathcal{S}_{\Pi}$ with randomness consistent with $\{\mathsf{seed}_r^i\}_{i\in[n]\setminus h}$ used by the corrupted parties in the r-th execution and outputs $\{y^i\}_{i\in[n]\setminus h}$ of the corrupted parties.
   - For all $j \in [t] \setminus r$, $\mathcal{S}$ acts like an honest party and derives randomness from seed_j^h.

   Then, $\mathcal{S}$ uses $\{\mathsf{seed}_j^i\}_{i\in[n]}$ to recompute the j-th semi-honest instance and to obtain transcript trans'_j for $j \in [t]$. For every $l \in [t]$ where $\mathsf{trans}'_l \neq \mathsf{trans}_l$, determine the first party P_m that has deviated from the protocol description at location loc, add (m, l, loc) to M_2 and l to L.
   If $r \in L$ continue in step 0'. Otherwise continue in 3.

0' Rewind $\mathcal{A}$ and run steps 1' - 2' below until[7] $M'_2 = M_2$ and $L' = L$.
  1' Run the seed generation with $\mathcal{A}$ as in step 1.
  2' For $j \in [t]$, run an execution of Π_{SH} with $\mathcal{A}$ where the randomness is derived from seed_j^h. Let trans_j denote the transcript and y_j^h denote the output of P_h in the j-th execution.
  $\mathcal{S}$ uses $\{\mathsf{seed}_j^i\}_{i\in[n]}$ to recompute the j-th semi-honest instance and to obtain transcript trans'_j for $j \in [t]$. For every $l \in [t]$ where $\mathsf{trans}'_l \neq \mathsf{trans}_l$,

determine the first party P_m that has deviated from the protocol description at location loc, add (m, l, loc) to M_2' and l to L'.

3. $\mathcal{S}$ receives message $\{(r^i, u^i, \{d^i_j\}_{j\in[t]})\}_{i\in[n]\setminus h}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{PG}}$ and executes the following steps.
   (a) If there is $l \in [t]$ and $m \in [n]$ where $\mathsf{Open}(c^m_l, d^m_l) = \mathsf{seed}^{(1,m)*}_l \neq \perp$ and $\mathsf{seed}^{(1,m)*}_l \neq \mathsf{seed}^{(1,m)}_l$, $\mathcal{S}$ outputs $\mathsf{ambiguous}$ and halts.
   (b) For every $l \in [t]$ and $m \in [n]$ where $\mathsf{Open}(c^m_l, d^m_l) = \perp$, add m to M_1 and l to L.
   (c) Compute $p \leftarrow \mathsf{TL.Generate}(pp, (r, \{d^i_j\}_{i\in[n], j\in[t]\setminus r}))$. $\mathcal{S}$ internally hands p to $\mathcal{A}$.
4. $\mathcal{S}$ creates signatures σ^h_j for $j \in [t]$ as an honest party would do and hands $\{\sigma^h_j\}_{j\in[t]}$ to $\mathcal{A}$. $\mathcal{S}$ receives $\{\sigma^i_j\}_{i\in[n]\setminus h, j\in[t]}$ that $\mathcal{A}$ sends to P_h.
5. If $\mathcal{A}$ has not sent valid messages in the first protocol steps in the expected communication round, send abort to $\mathcal{F}_{\mathsf{Cov}}$, hand abort to $\mathcal{A}$ and terminate. Otherwise, we distinguish three cases, where $m^* := \min_{i\in[n]}(i \in M_1)$ if $M_1 \neq \emptyset$ and $m^* := \min_{i\in[n]}((i, \cdot, \cdot) \in M_2)$ otherwise.
   – If $|L| \geq 2$ then send $\mathsf{corrupted}_{m^*}$ to $\mathcal{F}_{\mathsf{Cov}}$, set $r' = r$ and $\mathsf{flag} := \mathsf{corrupted}$, and continue in step 7.
   – If $|L| = 1$ then send cheat_{m^*} to $\mathcal{F}_{\mathsf{Cov}}$.
     • If $\mathcal{F}_{\mathsf{Cov}}$ returns $\mathsf{corrupted}_{m^*}$, set $r' \xleftarrow{\$} [t] \setminus L$, $\mathsf{flag} := \mathsf{corrupted}$, and continue below.
     • If $\mathcal{F}_{\mathsf{Cov}}$ returns $\mathsf{undetected}$, set $r' \in L$, $\mathsf{flag} := \mathsf{undetected}$, and continue below.
   – If $|L| = 0$ then set $r' = r$ and $\mathsf{flag} := \mathsf{honest}$ and continue below in step 7.

0* Rewind $\mathcal{A}$ and run steps 1* - 5* below until[7] $M_1' = M_1$, $M_2' = M_2$, and $L' = L$.
   1* Run the seed generation with $\mathcal{A}$ as in step 1.
   2* Run the semi-honest instances with $\mathcal{A}$ as in step 2'.
   3* $\mathcal{S}$ receives message $\{(r^i, u^i, \{d^i_j\}_{j\in[t]})\}_{i\in[n]\setminus h}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{PG}}$ and executes the following steps.
      (a*) If there is $l \in [t]$ and $m \in [n]$ where $\mathsf{Open}(c^m_l, d^m_l) = \mathsf{seed}^{(1,m)*}_l \neq \perp$ and $\mathsf{seed}^{(1,m)*}_l \neq \mathsf{seed}^{(1,m)}_l$, rewind to step 1*.
      (b*) For every $l \in [t]$ and $m \in [n]$ where $\mathsf{Open}(c^m_l, d^m_l) = \perp$, add m to M_1' and l to L'.
      (c*) Compute $p \leftarrow \mathsf{TL.Generate}(pp, (r', \{d^i_j\}_{i\in[n], j\in[t]\setminus r}))$. $\mathcal{S}$ internally hands p to $\mathcal{A}$.
   4* $\mathcal{S}$ creates signatures σ^h_j for $j \in [t]$ as an honest party would do and hands $\{\sigma^h_j\}_{j\in[t]}$ to $\mathcal{A}$. $\mathcal{S}$ receives $\{\sigma^i_j\}_{i\in[n]\setminus h, j\in[t]}$ that $\mathcal{A}$ sends to P_h.
   5* If $\mathcal{A}$ has not sent valid messages in the first protocol steps in the expected communication round, then return to step 1*.

7. $\mathcal{S}$ receives message $\{\mathsf{continue}_i\}_{i\in[n]\setminus h}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{PG}}$ and hands $(r', \{d^i_j\}_{j\in[t]\setminus r', i\in[n]})$ back to $\mathcal{A}$. If $\mathcal{A}$ responds with abort or $\mathcal{S}$ has not received $\mathsf{continue}_i$ for all $i \in [n] \setminus h$, continue in step 8 if $\mathsf{flag} = \mathsf{corrupted}$, or send abort to $\mathcal{F}_{\mathsf{Cov}}$ and continue in step 8 otherwise.
   If $\mathcal{A}$ does not respond with abort differentiate the following cases:
   – If $\mathsf{flag} = \mathsf{honest}$ then send $\mathsf{continue}$ to $\mathcal{F}_{\mathsf{Cov}}$ and halt.
   – If $\mathsf{flag} = \mathsf{undetected}$ send $y^h_{r'}$ to $\mathcal{F}_{\mathsf{Cov}}$ and halt.

  – If $\mathsf{flag} = \mathsf{corrupted}$ compute $(m^*, \mathsf{cert}) := \mathsf{Blame}(\mathsf{view}^h)$, where view^h is the view of the simulated party P_h. Internally send cert to $\mathcal{A}$ and halt.

8. If $\mathsf{flag} = \mathsf{corrupted}$ then $\mathcal{S}$ computes $((r', \{d^i_j\}_{j\in[t]\setminus r, i\in[n]}), \pi) := \mathsf{TL.Solve}(pp, p)$ and $(m^*, \mathsf{cert}) := \mathsf{Blame}(\mathsf{view}^h)$, where view^h is the view of the simulated party P_h, and internally sends cert to $\mathcal{A}$ and halts.
   If $\mathsf{flag} \in \{\mathsf{undetected}, \mathsf{honest}\}$ then hand abort to $\mathcal{A}$ and halt.

We show that the joint distribution of the output of the honest party P_h and the view of $\mathcal{A}$ in the ideal world is computationally indistinguishable from the output of the honest party P_h and the view of $\mathcal{A}$ in the real world. We prove this via a sequence of hybrids and show indistinguishability between each two subsequent hybrids. The proof is presented in appendix A.

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.

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 $\{d^i_j\}_{i\in[n], j\in[t]\setminus r}$. Either, all parties jointly trigger the puzzle generation functionality $\mathcal{F}_{\mathsf{PG}}$ 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 $\mathcal{F}_{\mathsf{PG}}$ 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, expect 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^m_l, d^m_l) for some $l \neq r$. The value c^m_l is signed directly by P_m and d^m_l 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.

[7] Ensuring that the simulation runs in expected polynomial time can be realized by standard techniques introduced by Goldreich and Kahan [GK96].

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 $\{\mathsf{seed}^{(2,i)}\}_{i\in[n]}$, 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 $\mathsf{Open}(c_j^{*h}, d_j^{*h}) = \bot$ 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 $\mathcal{F}_{\mathsf{PG}}$. 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^* or seed_j^h needs to be incorrect. Also, P_h does not create a signature for an invalid trans^*. Thus, trans^* is either correct or the adversary can forge signatures. The seed_j^h is calculated as $\mathsf{seed}_j^h := \mathsf{seed}_j^{(1,h)} \oplus \mathsf{seed}_j^{(2,h)}$. The public seed $\mathsf{seed}_j^{(2,h)}$ 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 $\mathsf{seed}_j^{(2,h)}$ 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 $(\mathsf{trans}_j^*, c_j^h, d_j^h, \mathsf{seed}_j^{(2,h)})$ 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 PVC Compiler for Input-Dependent Protocols

Damgård et al. [DOS20] have shown how to apply the cut-and-choose approach to publicly verifiable covert protocols in the input-dependent setting without sacrificing the privacy of the inputs. Their approach is based on the player virtualization paradigm introduced by [IPS08] and extended by [LOP11]. We show how to apply their technique to our compiler in order to extend it to the scope of input-dependent protocols.

Upon input x^h, party P_h splits its input into t shares using a t-out-of-t secret sharing. Each share is used as input of a single virtual party executed by P_h. It follows that each real party simulates t virtual parties. We therefore require a (tn)-party semi-honest protocol, Π^*_{SH}, realizing function f' defined as follows. Let $f(x^1, \ldots, x^n) = (y^1, \ldots, y^n)$ be the function all parties want to jointly compute with protocol Π_{SH}. The function f' takes as input a t-out-of-t secret sharing of each real party's input and computes a t-out-of-t secret sharing of each real party's output. More precisely, f' is defined as

$$(y_1^1, \ldots, y_t^1, \ldots, y_1^n, \ldots, y_t^n,) \leftarrow f'(x_1^1, \ldots, x_t^1, \ldots, x_1^n, \ldots, x_t^n,),$$

where

$$f\left(\bigoplus_{j\in[t]} x_j^1, \ldots, \bigoplus_{j\in[t]} x_j^n\right) = \left(\bigoplus_{j\in[t]} y_j^1, \ldots, \bigoplus_{j\in[t]} y_j^n\right).$$

As assumed by our compiler for input-independent protocols, we require each message to be sent to all real parties. Even if messages are sent from one virtual party to another virtual party which are both simulated by the same real parties.

Besides committing to the private seed shares in step 1, the parties need to commit to the input shares of the virtual parties in step 2. The corresponding opening values are inserted to the puzzle generation functionality as well. In the input-independent protocol parties validated a whole protocol instance at once by simulating the actions of all parties. In the input-dependent protocol, this is not possible because some of the virtual parties need to remain private and hence cannot be simulated. However, a virtual party can also be simulated individually as all of its actions are fully determined by its randomness, its input share and all of its in-going messages; all of this information is available once the virtual party has been opened. We stress that since the input is t-out-of-t secret shared and only $t-1$ input shares are opened, the input of a real party remains private.

Moreover, in contrast to the compiler for input-independent protocols, this compiler executes only one instance of the (tn)-party protocol Π^*_{SH} secure against $tn-1$ passive corruptions. The full specification of the input-dependent compiler is deferred to appendix B.

7 Short Certificates

For the sake of exposition, we have described our protocol and the corresponding algorithms in a straightforward way. In this section, we will present two additional improvements which reduce the size of the certificates and thus enable additional use-cases, e.g., uploading certificates to a public blockchain.

Reduce transcript size via a Merkle tree. The Judge-algorithm of the input-independent publicly verifiable covert secure protocol Π_{PVC} receives the whole transcript of one of the semi-honest protocol instances as part of the certificate of misbehavior. Based on the transcript, the algorithm verifies which party sent the first incorrect message. However, depending on the size of the semi-honest protocol, the transcript can become very large. By utilizing Merkle trees, we can replace the protocol transcript within a certificate by the root hash of a Merkle tree constructed over the transcript messages. In case of a deviation certificate, two messages and corresponding Merkle tree proofs are added.

In order to use the Merkle tree root as verifiable evidence, parties create a Merkle tree of each protocol transcript, i.e., all messages that have been sent during one semi-honest protocol instance, and include the Merkle tree root instead of the protocol transcript into the signatures. When checking for misbehavior, an honest party calculates the last message m_0 at position $k-1$ which has been valid and the first message m_1 at position k in which a party has deviated from the protocol description resp. its supposed randomness. These messages, the corresponding Merkle tree proofs and the position of m_1 within the transcript are inserted into the certificate. The judge then simulates the protocol until the k-th message and thus gets the messages m'_0 and m'_1 that have been supposed to be sent at position $k-1$ and k. Using the received information, the judge checks that (1) $m_0 = m'_0$, (2) $m_1 \neq m'_1$, (3) m_0 and m_1 are leaves at position $k-1$ resp. k of a Merkle tree, with a root that has been signed by the accused party, and (4) m_1 is the first incorrect message. For the latter, the judge checks that the Merkle proof of m_0 corresponds to the protocol messages (tree leaves) at position $< k$ that the judge has simulated itself.

This improvement is only applicable to the input-independent compiler, as this procedure requires to validate the actions of all parties in order to determine the first deviation from the expected transcript.

Separate signature for inconsistency certificates and deviation certificates. Currently, the parties create one signature which contains all the information that is needed in an inconsistency certificate and a deviation certificate. This means that data, which is not used by the Judge-algorithm for a certain type of certificate (e.g., the time-lock puzzle p in case of deviation certificates) still needs to be provided in order to enable the judge to verify the signatures. Such data depicts an unnecessary overhead in terms of certificate size. Fortunately, it is quite easy to get rid of this overhead. Instead of creating one signature per instance, parties create two signatures that contain the information only relevant for the inconsistency certificate respectively the deviation certificate. In particular, party P_h

signs the tuples

$$\begin{aligned}\mathsf{data}_j^{(1,h)} &:= (h, j, c_j^h, p)\\ \mathsf{data}_j^{(2,h)} &:= (h, j, \mathsf{seed}_j^{(2,h)}, c_j^h, \mathsf{trans}_j^h)\end{aligned}$$

for each $j \in [t]$. The former data is used for inconsistency certificates and the latter for deviation certificates. A similar improvement can be applied to the input-dependent compiler.

8 Evaluation

8.1 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 particular 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 $u = \Sigma\ u_i$, 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.

8.2 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 $\mathsf{Pr}[\mathsf{bad}]$ 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 $\mathsf{Pr}[\mathsf{bad}]$ and the expected number of protocol runs runs are calculated based on the inclusion-exclusion principle as follows:

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

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	2	1/2	-		2
	3	2/3	1/3		3
	10	9/10	4/10		10
3	2	1/2	-		4
	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.

A similar comparison can be made for the input-dependent setting. Note that both input-dependent compilers, ours and the one using the watchlist approach, execute one protocol instance between virtual parties instead of running multiple executions of the semi-honest base protocol. We refer to Section 6, Appendix B and [DOS20] for more detailed descriptions of the input-dependent compilers. In the following we will restrict the analysis to the setting in which an adversary corrupts $n-1$ of the n parties. The number of virtual parties per real party is denoted as t and the number of opened virtual parties per real party is denoted as s. Hence, the deterrence factor is $\epsilon = \frac{s}{t}$. Our compiler extension ensures that all parties open and check the same virtual parties. Hence, s can be chosen to be $t-1$ which enables us to achieve a high deterrence factor. In the watchlist approach, each party selects its own watchlist. In the input-dependent setting it, is not possible to tolerate that all virtual parties of one real party are checked as this leaks the real party's input and hence breaks privacy. Therefore, it needs to hold that $t > (n-1) \cdot s$. It follows that the deterrence factor is bound by $\epsilon < \frac{1}{(n-1)}$.

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 Zakarias. Multiparty computation from somewhat homomorphic encryption. *CRYPTO 2012*.

[GK96] Oded Goldreich and Ariel Kahan. How to construct constant-round zero-knowledge proof systems for NP. *Journal of Cryptology*, 1996.

[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 Venkitasubramaniam, 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.

[SSS21] Peter Scholl, Mark Simkin, and Luisa Siniscalchi. Multiparty computation with covert security and public verifiability. *IACR Cryptol. ePrint Arch.*, 2021.

[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*.

A Indistinguishability via Hybrids

In the following, we show that the joint distribution of the output of the honest party P_h and the view of $\mathcal{A}$ in the ideal world involving simulator $\mathcal{S}$ and ideal functionality $\mathcal{F}_{\mathsf{Cov}}$ is computationally indistinguishable from the output of the honest party P_h and the view of $\mathcal{A}$ in a real-world execution of the protocol Π_{PVC}. We will do so via a sequence of hybrids gradually reducing the usage of the simulator's *special* capabilities (e.g., rewinding) and by showing indistinguishability between each two subsequent hybrids.

Hybrid_0 :

In order to describe honest parties, simulator and ideal functionality in one experiment, we inline the actions of $\mathcal{S}$, $\mathcal{F}_{\mathsf{Cov}}$, and the honest party P_h to obtain the following experiment. This is the ideal-world execution with simulator $\mathcal{S}$ as described above in section 5.

0. Set $r \overset{\$}{\leftarrow} [t], M_1, M_2, L = \emptyset$, and compute $(y^1, \ldots, y^n) \leftarrow f(\bot)$. Then $\mathcal{S}$ generates keys $(\mathsf{pk}_h, \mathsf{sk}_h)$ and sends pk_h to $\mathcal{A}$.
1. $\mathcal{S}$ acts like an honest party during the seed generation and simulates $\mathcal{F}_{\mathsf{coin}}$ as stated in the definition. This way, $\mathcal{S}$ obtains public seed shares $\{\mathsf{seed}_j^{(2,i)}\}_{i\in[n]}$ and commitments $\{c_j^i\}_{i\in[n]}$ for $j \in [t]$. By simulating the honest party, $\mathcal{S}$ gets private seed shares $\mathsf{seed}_j^{(1,h)}$, commitments and openings $(c_j^h, d_j^h) \leftarrow \mathsf{Commit}(\mathsf{seed}_j^{(1,h)})$ for $j \in [t]$. Use the extractability property of the commitment scheme to obtain the private seed shares $\{\mathsf{seed}_j^{(1,i)}\}_{i\in[n]\setminus h}$. Finally, compute $\mathsf{seed}_j^i := \mathsf{seed}_j^{(1,i)} \oplus \mathsf{seed}_j^{(2,i)}$ for $j \in [t]$ and $i \in [n]$.
2. Run the executions of Π_{SH} with $\mathcal{A}$ as follows and let trans_j denote the transcript and y_j^h denote the output of P_h in the j-th execution:
   - In the r-th execution, $\mathcal{S}$ invokes $\mathcal{S}_\Pi$ with randomness consistent with $\{\mathsf{seed}_r^i\}_{i\in[n]\setminus h}$ used by the corrupted parties in the r-th execution and outputs $\{y^i\}_{i\in[n]\setminus h}$ of the corrupted parties.
   - For all $j \in [t] \setminus r$, $\mathcal{S}$ derives randomness from seed_j^h.

   Then, $\mathcal{S}$ uses $\{\mathsf{seed}_j^i\}_{i\in[n]}$ to recompute the j-th semi-honest instance and to obtain transcript trans'_j for $j \in [t]$. For every $l \in [t]$ where $\mathsf{trans}'_l \neq \mathsf{trans}_l$, determine the first party P_m that has deviated from the protocol description at location loc, add (m, l, loc) to M_2 and l to L.
   If $r \in L$ continue in step 0'. Otherwise continue in 3.

0' Rewind $\mathcal{A}$ and run steps 1' - 2' below until $M'_2 = M_2$ and $L' = L$.
   1' Run the seed generation with $\mathcal{A}$ as in step 1.
   2' For $j \in [t]$, run an execution of Π_{SH} with $\mathcal{A}$ where the randomness is derived from seed_j^h. Let trans_j denote the transcript and y_j^h denote the output of P_h in the j-th execution.
   $\mathcal{S}$ uses $\{\mathsf{seed}_j^i\}_{i\in[n]}$ to recompute the j-th semi-honest instance and to obtain transcript trans'_j for $j \in [t]$. For every $l \in [t]$ where $\mathsf{trans}'_l \neq \mathsf{trans}_l$, determine the first party P_m that has deviated from the protocol description at location loc, add (m, l, loc) to M'_2 and l to L'.

3. $\mathcal{S}$ receives message $\{(r^i, u^i, \{d_j^i\}_{j\in[t]})\}_{i\in[n]\setminus h}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{PG}}$ and executes the following steps.

(a) If there is $l \in [t]$ and $m \in [n]$ where $\mathsf{Open}(c_l^m, d_l^m) = \mathsf{seed}_l^{(1,m)*} \neq \perp$ and $\mathsf{seed}_l^{(1,m)*} \neq \mathsf{seed}_l^{(1,m)}$, $\mathcal{S}$ outputs $\mathsf{ambiguous}$ and halts.

(b) For every $l \in [t]$ and $m \in [n]$ where $\mathsf{Open}(c_l^m, d_l^m) = \perp$, add m to M_1 and l to L.

(c) Compute $p \leftarrow \mathsf{TL.Generate}(pp, (r, \{d_j^i\}_{i \in [n], j \in [t] \setminus r}))$. $\mathcal{S}$ internally hands p to $\mathcal{A}$.

4. $\mathcal{S}$ creates signatures σ_j^h for $j \in [t]$ as an honest party would do and hands $\{\sigma_j^h\}_{j \in [t]}$ to $\mathcal{A}$. $\mathcal{S}$ receives $\{\sigma_j^i\}_{i \in [n] \setminus h, j \in [t]}$ that $\mathcal{A}$ sends to P_h.

5. If $\mathcal{A}$ has not sent valid messages in the first protocol steps in the expected communication round, hand abort to $\mathcal{A}$, P_h outputs abort, and terminate the experiment.

   Otherwise, we distinguish three cases.
   - If $|L| \geq 2$ then set $r' = r$ and $\mathsf{flag} := \mathsf{corrupted}$, and continue in step 6.
   - If $|L| = 1$ then with probability ϵ set $r' \overset{\$}{\leftarrow} [t] \setminus L$ and $\mathsf{flag} := \mathsf{corrupted}$, and with remaining probability $(1 - \epsilon)$ set $r' \in L$ and $\mathsf{flag} := \mathsf{undetected}$. In any case, continue below.
   - If $|L| = 0$ then set $r' = r$ and $\mathsf{flag} := \mathsf{honest}$, and continue below in step 6.

0* Rewind $\mathcal{A}$ and run steps 1* - 5* below until $M_1' = M_1$, $M_2' = M_2$, and $L' = L$.

  1* Run the seed generation with $\mathcal{A}$ as in step 1.

  2* Run the semi-honest instances with $\mathcal{A}$ as in step 2'.

  3* $\mathcal{S}$ receives message $\{(r^i, u^i, \{d_j^i\}_{j \in [t]})\}_{i \in [n] \setminus h}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{PG}}$ and executes the following steps.

   (a*) If there is $l \in [t]$ and $m \in [n]$ where $\mathsf{Open}(c_l^m, d_l^m) = \mathsf{seed}_l^{(1,m)*} \neq \perp$ and $\mathsf{seed}_l^{(1,m)*} \neq \mathsf{seed}_l^{(1,m)}$, rewind to step 1*.

   (b*) For every $l \in [t]$ and $m \in [n]$ where $\mathsf{Open}(c_l^m, d_l^m) = \perp$, add m to M_1' and l to L'.

   (c*) Compute $p \leftarrow \mathsf{TL.Generate}(pp, (r', \{d_j^i\}_{i \in [n], j \in [t] \setminus r'}))$. $\mathcal{S}$ internally hands p to $\mathcal{A}$.

  4* $\mathcal{S}$ creates signatures σ_j^h for $j \in [t]$ as an honest party would do and hands $\{\sigma_j^h\}_{j \in [t]}$ to $\mathcal{A}$. $\mathcal{S}$ receives $\{\sigma_j^i\}_{i \in [n] \setminus h, j \in [t]}$ that $\mathcal{A}$ sends to P_h.

  5* If $\mathcal{A}$ has not sent valid messages in the first protocol steps in the expected communication round, then return to step 1*.

6. $\mathcal{S}$ receives message $\{\mathsf{continue}_i\}_{i \in [n] \setminus h}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{PG}}$ and hands $(r', \{d_j^i\}_{j \in [t] \setminus r', i \in [n]})$ back to $\mathcal{A}$. If $\mathcal{A}$ responds with abort or $\mathcal{S}$ has not received $\mathsf{continue}_i$ for all $i \in [n] \setminus h$, continue in step 7.

   If $\mathcal{A}$ does not respond with abort differentiate the following cases:
   - If $\mathsf{flag} = \mathsf{honest}$ then P_h outputs y^h and the experiment halts.
   - If $\mathsf{flag} = \mathsf{undetected}$ then P_h outputs $y_{r'}^h$ and the experiment halts.
   - If $\mathsf{flag} = \mathsf{corrupted}$ compute $(m, \mathsf{cert}) := \mathsf{Blame}(\mathsf{view}^h)$, where view^h is the view of the simulated party P_h. Internally send cert to $\mathcal{A}$, P_h outputs $\mathsf{corrupted}_m$ and the experiment halts.

7. If $\mathsf{flag} = \mathsf{corrupted}$ then $\mathcal{S}$ computes $((r', \{d_j^i\}_{j \in [t], i \in [n]}), \pi) := \mathsf{TL.Solve}(pp, p)$ and $(m, \mathsf{cert}) := \mathsf{Blame}(\mathsf{view}^h)$, where view^h is the view of the simulated party P_h, internally sends cert to $\mathcal{A}$, P_h outputs $\mathsf{corrupted}_m$, and the experiment halts. If $\mathsf{flag} \in \{\mathsf{undetected}, \mathsf{honest}\}$ then hand abort to $\mathcal{A}$, P_h outputs abort and the experiment halts.

$\underline{\mathsf{Hybrid}_1}$:
In Hybrid_1, we do no longer ask the ideal covert functionality to decide if non-blatant cheating should be detected but let the experiment decide. To this end, we set $r' \overset{\$}{\leftarrow} [t]$ in step 5 if $|L| = 1$ and set flag according to the result of the coin toss. We obtain:

5. – If $|L| = 1$ then set $r' \overset{\$}{\leftarrow} [t]$ and $\mathsf{flag} := \mathsf{corrupted}$ if $r' \notin L$ and set $\mathsf{flag} := \mathsf{undetected}$ if $r' \in L$. In any case continue below.

Since $r' \notin L$ with probability $1 - \frac{1}{t} = \epsilon$ and $r' \in L$ with probability $\frac{1}{t} = 1 - \epsilon$ if $|L| = 1$, the output distributions of Hybrid_0 and Hybrid_1 are identical.

$\underline{\mathsf{Hybrid}_2}$:
In Hybrid_2, we want to remove the utilization of the semi-honest simulator $\mathcal{S}_\Pi$. To this end, we change step 2 of the experiment. In the r-th instance, $\mathcal{S}$ selects a new seed seed^*_r and executes the semi-honest protocol with randomness derived from $\mathsf{seed}^*_r \oplus \mathsf{seed}^{(2,h)}_r$ instead of invoking $\mathcal{S}_\Pi$. In addition, the output of the r-th semi-honest instance is taken as the final output in case of honest behavior of $\mathcal{A}$. Thus, P_h outputs y^h_r in step 6 if $\mathsf{flag} = \mathsf{honest}$. The modified steps of Hybrid_2 are as follows:

0. Set $r \overset{\$}{\leftarrow} [t]$ and $M_1, M_2, L = \emptyset$. Then $\mathcal{S}$ generates keys $(\mathsf{pk}_h, \mathsf{sk}_h)$ and sends pk_h to $\mathcal{A}$.
2. Run the executions of Π_{SH} with $\mathcal{A}$ as follows and let trans_j denote the transcript and y^h_j denote the output of P_h in the j-th execution:
   – In the r-th execution, $\mathcal{S}$ samples a new κ-bit string seed^*_r and derives randomness from $\mathsf{seed}^*_r \oplus \mathsf{seed}^{(2,h)}_r$.
   – For all $j \in [t] \setminus r$, $\mathcal{S}$ derives randomness from seed^h_j.

   Then, $\mathcal{S}$ uses $\{\mathsf{seed}^i_j\}_{i\in[n]}$ to recompute the j-th semi-honest instance and to obtain transcript trans'_j for $j \in [t]$. For every $l \in [t]$ where $\mathsf{trans}'_l \neq \mathsf{trans}_l$, determine the first party P_m that has deviated from the protocol description at location loc, add (m, l, loc) to M_2 and l to L.
   If $r \in L$ continue in step 0'. Otherwise continue in 3.
6. – If $\mathsf{flag} = \mathsf{honest}$ then P_h outputs y^h_r and the experiment halts.

We first elaborate on the transcript generated in step 2 in Hybrid_1 and Hybrid_2. For all instances $j \in [t] \setminus r$, the transcript is generated by $\mathcal{S}$ acting like an honest party. In the r-th execution, $\mathcal{S}$ calls $\mathcal{S}_\Pi$ to generate a transcript in Hybrid_1 while $\mathcal{S}$ acts like an honest party in Hybrid_2. Notice that if $\mathcal{A}$ cheats in the r-th instance, i.e., if $r \in L$, the simulator rewinds and then generates the transcript by acting like an honest party in both experiments. Therefore, $\mathcal{S}$ simulates the transcript differently only if $\mathcal{A}$ behaves honestly.

Assuming an adversary acting like an honest party, the security of Π_{SH} against passive adversaries guarantees that the transcript generated by $\mathcal{S}_\Pi$ is indistinguishable from the transcript obtained during honest execution. Hence, $\mathcal{S}$ may act like an honest party instead of generating the transcript using $\mathcal{S}_\Pi$.

Moreover, the security notion fulfilled by Π_{SH} guarantees that the output of an honest execution is indistinguishable from $(y^1, \ldots, y^h) \leftarrow f(\perp)$.

It remains to show that the randomness used in the semi-honest protocol execution is uniformly distributed. Note that honest behavior of $\mathcal{A}$ implies using randomness derived from seeds $\{\mathsf{seed}_r^i\}_{i\in[n]\setminus h}$. Since $\mathsf{seed}_r^i = \mathsf{seed}_r^{(1,i)} \oplus \mathsf{seed}_r^{(2,i)}$, where $\mathsf{seed}_r^{(2,i)}$ is the uniform random output of $\mathcal{F}_{\mathsf{coin}}$, the seed seed_r^i is uniform as well.

It follows that Hybrid_1 and Hybrid_2 are indistinguishable.

$\underline{\mathsf{Hybrid}_3 :}$
In Hybrid_3, we want to align the randomness of the r-th semi-honest base protocol instance (previously executed via the semi-honest simulator $\mathcal{S}_\Pi$) with the randomness determined by the joined seed sampling procedure. To this end, we change step 2 of the experiment again. Instead of sampling a fresh seed seed_r^* for the r-th execution, $\mathcal{S}$ sets seed_r^* to be $\mathsf{seed}_r^{(1,h)}$.

We reduce the indistinguishably of Hybrid_2 and Hybrid_3 to the hiding property of the commitment scheme. Assume the existence of a distinguisher $\mathcal{D}$ which is able to distinguish Hybrid_2 and Hybrid_3. We use D to construct and adversary $\mathcal{A}_{\mathsf{Com}}$ that wins the commitment hiding game with non-negligible probability.

First, $\mathcal{A}_{\mathsf{Com}}$ samples two uniform seeds κ-bit strings s^0 and s^1 and sends them to the hiding game. The hiding game answers with a commitment c^*, which is the commitment of s^b, where $b \xleftarrow{\$} \{0,1\}$ is privately chosen by the security game. Secondly, $\mathcal{A}_{\mathsf{Com}}$ executes an instance of Hybrid_2, let us call it $\mathsf{Hybrid}_{2;3}$, with the following modifications:

- $\mathcal{A}_{\mathsf{Com}}$ advises $\mathcal{S}$ to use c^* instead of c_r^h.
- $\mathcal{A}_{\mathsf{Com}}$ advises $\mathcal{S}$ to set seed_r^* to be equal to s^1.

Note that if the security game selects b to be 1, the execution of $\mathsf{Hybrid}_{2;3}$ equals the execution of Hybrid_3, because the same seed is used for both, commitment and the execution of the r-th semi-honest instance. Otherwise, the execution of $\mathsf{Hybrid}_{2;3}$ equals the execution of Hybrid_2.

Additionally, note that if $\mathcal{A}$ does not cheat in any instance, then $\mathcal{S}$ does not need to use d_r^h because step 3* is never executed and step 3 does not use the opening d_r^h. If $\mathcal{A}$ cheats the simulator rewinds and generates a new value c_r^h and thus also obtains an opening d_r^h. This value is used in step 3*. Thus, $\mathcal{A}_{\mathsf{Com}}$ is able to produce valid executions for Hybrid_2 or Hybrid_3.

Finally, $\mathcal{A}_{\mathsf{Com}}$ gives the view generated by the instance $\mathsf{Hybrid}_{2;3}$ to $\mathcal{D}$ and outputs whatever $\mathcal{D}$ outputs. As the execution of $\mathsf{Hybrid}_{2;3}$ equals the execution of Hybrid_3 if $b = 1$ and the execution of Hybrid_2 otherwise and $\mathcal{D}$ is able to distinguish views generated by executing Hybrid_2 and Hybrid_3 with non-negligible probability, $\mathcal{A}_{\mathsf{Com}}$ has a non-negligible advantage in the security game.

Hence, the existence of $\mathcal{D}$ contradicts the hiding property of the commitment scheme. It follows that Hybrid_2 and Hybrid_3 are computationally indistinguishable.

$\underline{\mathsf{Hybrid}_4 :}$
In this hybrid, we get rid of the first rewinding steps by *collapsing* the first

rewound thread. Since steps 1 - 2 are identical to steps 1' - 2', the resulting experiment (Hybrid_4) is indistinguishable from Hybrid_3.

0. Set $r \xleftarrow{\$} [t]$ and $M_1, M_2, L = \emptyset$. Then $\mathcal{S}$ generates keys $(\mathsf{pk}_h, \mathsf{sk}_h)$ and sends pk_h to $\mathcal{A}$.
1. $\mathcal{S}$ acts like an honest party during the seed generation and simulates $\mathcal{F}_{\mathsf{coin}}$ as stated in the definition. This way, $\mathcal{S}$ obtains public seed shares $\{\mathsf{seed}_j^{(2,i)}\}_{i\in[n]}$ and commitments $\{c_j^i\}_{i\in[n]}$ for $j \in [t]$. By simulating the honest party, $\mathcal{S}$ gets private seed shares $\mathsf{seed}_j^{(1,h)}$, commitments and openings $(c_j^h, d_j^h) \leftarrow \mathsf{Commit}(\mathsf{seed}_j^{(1,h)})$ for $j \in [t]$. Use the extractability property of the commitment scheme to obtain the private seed shares $\{\mathsf{seed}_j^{(1,i)}\}_{i\in[n]\setminus h}$. Finally, compute $\mathsf{seed}_j^i := \mathsf{seed}_j^{(1,i)} \oplus \mathsf{seed}_j^{(2,i)}$ for $j \in [t]$ and $i \in [n]$.
2. For $j \in [t]$, run an execution of Π_{SH} with $\mathcal{A}$ where the randomness is derived from seed_j^h. Let trans_j denote the transcript and y_j^h denote the output of P_h in the j-th execution.
   $\mathcal{S}$ uses $\{\mathsf{seed}_j^i\}_{i\in[n]}$ to recompute the j-th semi-honest instance and to obtain transcript trans'_j for $j \in [t]$. For every $l \in [t]$ where $\mathsf{trans}'_l \neq \mathsf{trans}_l$, determine the first party P_m that has deviated from the protocol description at location loc, add (m, l, loc) to M_2 and l to L.
3. $\mathcal{S}$ receives message $\{(r^i, u^i, \{d_j^i\}_{j\in[t]})\}_{i\in[n]\setminus h}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{PG}}$ and executes the following steps.
   (a) If there is $l \in [t]$ and $m \in [n]$ where $\mathsf{Open}(c_l^m, d_l^m) = \mathsf{seed}_l^{(1,m)*} \neq \bot$ and $\mathsf{seed}_l^{(1,m)*} \neq \mathsf{seed}_l^{(1,m)}$, $\mathcal{S}$ outputs $\mathsf{ambiguous}$ and halts.
   (b) For every $l \in [t]$ and $m \in [n]$ where $\mathsf{Open}(c_l^m, d_l^m) = \bot$, add m to M_1 and l to L.
   (c) Compute $p \leftarrow \mathsf{TL.Generate}(pp, (r, \{d_j^i\}_{i\in[n], j\in[t]\setminus r}))$. $\mathcal{S}$ internally hands p to $\mathcal{A}$.
4. $\mathcal{S}$ creates signatures σ_j^h for $j \in [t]$ as an honest party would do and hands $\{\sigma_j^h\}_{j\in[t]}$ to $\mathcal{A}$. $\mathcal{S}$ receives $\{\sigma_j^i\}_{i\in[n]\setminus h, j\in[t]}$ that $\mathcal{A}$ sends to P_h.
5. If $\mathcal{A}$ has not sent valid messages in the first protocol steps in the expected communication round, hand abort to $\mathcal{A}$, P_h outputs abort, and terminate the experiment.
   Otherwise, we distinguish three cases.
   – If $|L| \geq 2$ then set $r' = r$ and $\mathsf{flag} := \mathsf{corrupted}$, and continue in step 6.
   – If $|L| = 1$ then set $r' \xleftarrow{\$} [t]$ and $\mathsf{flag} := \mathsf{corrupted}$ if $r' \notin L$ and set $\mathsf{flag} := \mathsf{undetected}$ if $r' \in L$. In any case continue below.
   – If $|L| = 0$ then set $r' = r$ and $\mathsf{flag} := \mathsf{honest}$, and continue below in step 6.

0* Rewind $\mathcal{A}$ and run steps 1* - 5* below until $M'_1 = M_1$, $M'_2 = M_2$, and $L' = L$.
   1* Run the seed generation with $\mathcal{A}$ as in step 1.
   2* Run the semi-honest instances with $\mathcal{A}$ as in step 2.
   3* $\mathcal{S}$ receives message $\{(r^i, u^i, \{d_j^i\}_{j\in[t]})\}_{i\in[n]\setminus h}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{PG}}$ and executes the following steps.
      (a*) If there is $l \in [t]$ and $m \in [n]$ where $\mathsf{Open}(c_l^m, d_l^m) = \mathsf{seed}_l^{(1,m)*} \neq \bot$ and $\mathsf{seed}_l^{(1,m)*} \neq \mathsf{seed}_l^{(1,m)}$, rewind to step 1*.
      (b*) For every $l \in [t]$ and $m \in [n]$ where $\mathsf{Open}(c_l^m, d_l^m) = \bot$, add m to M'_1 and l to L'.

(c*) Compute $p \leftarrow \mathsf{TL.Generate}(pp, (r', \{d_j^i\}_{i \in [n], j \in [t] \setminus r'}))$. $\mathcal{S}$ internally hands p to $\mathcal{A}$.

4* $\mathcal{S}$ creates signatures σ_j^h for $j \in [t]$ as an honest party would do and hands $\{\sigma_j^h\}_{j \in [t]}$ to $\mathcal{A}$. $\mathcal{S}$ receives $\{\sigma_j^i\}_{i \in [n] \setminus h, j \in [t]}$ that $\mathcal{A}$ sends to P_h.

5* If $\mathcal{A}$ has not sent valid messages in the first protocol steps in the expected communication round, then return to step 1*.

6. $\mathcal{S}$ receives message $\{\mathsf{continue}_i\}_{i \in [n] \setminus h}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{PG}}$ and hands $(r', \{d_j^i\}_{j \in [t] \setminus r', i \in [n]})$ back to $\mathcal{A}$. If $\mathcal{A}$ responds with abort or $\mathcal{S}$ has not received $\mathsf{continue}_i$ for all $i \in [n] \setminus h$, continue in step 7.
   If $\mathcal{A}$ does not respond with abort differentiate the following cases:
   - If $\mathsf{flag} = \mathsf{honest}$ then P_h outputs y_r^h and the experiment halts.
   - If $\mathsf{flag} = \mathsf{undetected}$ then P_h outputs $y_{r'}^h$ and the experiment halts.
   - If $\mathsf{flag} = \mathsf{corrupted}$ compute $(m, \mathsf{cert}) := \mathsf{Blame}(\mathsf{view}^h)$, where view^h is the view of the simulated party P_h. Internally send cert to $\mathcal{A}$, P_h outputs $\mathsf{corrupted}_m$ and the experiment halts.
7. If $\mathsf{flag} = \mathsf{corrupted}$ then $\mathcal{S}$ computes $((r', \{d_j^i\}_{j \in [t], i \in [n]}), \pi) := \mathsf{TL.Solve}(pp, p)$ and $(m, \mathsf{cert}) := \mathsf{Blame}(\mathsf{view}^h)$, where view^h is the view of the simulated party P_h, internally sends cert to $\mathcal{A}$, P_h outputs $\mathsf{corrupted}_m$, and the experiment halts. If $\mathsf{flag} \in \{\mathsf{undetected}, \mathsf{honest}\}$ then hand abort to $\mathcal{A}$, P_h outputs abort and the experiment halts.

Hybrid_5 :

In this hybrid, we align the execution of the main thread and the remaining rewinding thread in order to prepare the removal of the rewound thread. We will show computationally indistinguishability of Hybrid_4 and Hybrid_5 via a reduction to the security guarantees of the time-lock puzzle.

In Hybrid_5, we set $r' := r$ in step 0*:

0* Set $r' := r$, rewind $\mathcal{A}$ and run steps 1* - 5* below until $M_1' = M_1$, $M_2' = M_2$ and $L' = L$.

Unfortunately, we cannot directly reduce indistinguishability to the security of the TLP, as the puzzle distinguisher in the TLP security game is bound to run in time $\mathcal{T}$, while the distinguisher of the hybrids is allowed to run in arbitrary polynomial time.

Therefore, we will first examine the behavior of $\mathcal{A}$ and show that the two hybrids are computationally indistinguishable unless there exists a class of PPT adversaries $\mathbb{A}$ which are able to display a certain behavior at a specific time slot, and hence can be bound in time $\mathcal{T}$. In particular, adversaries that are able to display a specific behavior with different probability in the rewound threads of Hybrid_4 and Hybrid_5. Secondly, we show that the existence of any adversary $\mathcal{A}' \in \mathbb{A}$ contradicts the security of the time-lock puzzle.

Note that the main thread in the simulation is used to determine $\mathcal{A}$'s strategy and the rewound threads, if any, are executed until $\mathcal{A}$ repeats this strategy. As strategy, we understand the abort behavior and the cheating behavior of $\mathcal{A}$, i.e., the existence of an abort before step 5, the instances $\mathcal{A}$ cheats in (L), the location the adversary cheats in (loc) and the parties $\mathcal{A}$ uses to execute its attacks (M_1 and M_2). This corresponds directly to the behavior of aborting, blatant cheating ($|L| > 1$), strategic cheating ($|L| = 1$) and being honest ($|L| = 0$).

Formally, a single execution of the hybrid experiment Hybrid_i with security parameter $\kappa \in \mathbb{N}$ involving an PPT adversary $\mathcal{A}$ with auxiliary input $z_0 \in \{0,1\}^*$ and random tape $z_1 \in \{0,1\}^*$ corrupting a subset of parties I with $|I| = n-1$ is defined as $\mathsf{Hybrid}^i_{\mathcal{A}(z_0,z_1),I}(1^\kappa)$. Further, the joint distribution of the output of honest parties and adversary $\mathcal{A}$ in the experiment $\mathsf{Hybrid}^i_{\mathcal{A}(z_0,z_1),I}(1^\kappa)$ is defined as $\{\mathsf{Hybrid}^i_{\mathcal{A}(z_0,z_1),I}(1^\kappa)\}_{z_0,z_1,\kappa}$ where the distribution is taken over all possible choices for z_0, z_1, and κ. In the following, we will simplify the notation and denote $\mathsf{Hybrid}^i_{\mathcal{A}(z_0,z_1),I}(1^\kappa)$ as $\mathsf{Hybrid}^{\mathcal{A}}_i$ and $\{\mathsf{Hybrid}^i_{\mathcal{A}(z_0,z_1),I}(1^\kappa)\}_{z_0,z_1,\kappa,\mathcal{A}}$ as $\{\mathsf{Hybrid}^{\mathcal{A}}_i\}$. Finally, we denote the subset of $\{\mathsf{Hybrid}^{\mathcal{A}}_i\}$ including executions of $\mathsf{Hybrid}^{\mathcal{A}}_i$ that involve a specific event E as $\{\mathsf{Hybrid}^{\mathcal{A}}_i \mid E\}$.

We make the following observations. First, the two hybrids are exactly the same up to the end of step 5. The decision if the experiment needs to rewind or not is made in step 5. Hence, it follows that the probability of rewinding is exactly the same in both hybrids.

$$\Pr[\text{rewind in } \mathsf{Hybrid}^{\mathcal{A}}_4] = \Pr[\text{rewind in } \mathsf{Hybrid}^{\mathcal{A}}_5] \tag{1}$$

Secondly, in any instance without rewinding, the modified line, step 0*, is not executed in both of the hybrids. Hence, it follows that conditioned on the event that there is no rewinding the distributions of the two hybrids are the same.

$$\begin{aligned} &\{\mathsf{Hybrid}^{\mathcal{A}}_4 \mid \mathsf{Hybrid}^{\mathcal{A}}_4 \text{ does not rewind}\} \\ \equiv\ &\{\mathsf{Hybrid}^{\mathcal{A}}_5 \mid \mathsf{Hybrid}^{\mathcal{A}}_5 \text{ does not rewind}\} \end{aligned} \tag{2}$$

Finally, we need to show that the distributions of the two hybrids conditioned on the event that there is rewinding are at least computationally indistinguishable:

$$\begin{aligned} &\{\mathsf{Hybrid}^{\mathcal{A}}_4 \mid \mathsf{Hybrid}^{\mathcal{A}}_4 \text{ does rewind}\} \\ \equiv_C\ &\{\mathsf{Hybrid}^{\mathcal{A}}_5 \mid \mathsf{Hybrid}^{\mathcal{A}}_5 \text{ does rewind}\} \end{aligned} \tag{3}$$

To show indistinguishability of the distributions conditioned on the rewinding event, we traverse through the hybrids and show that the views generated by the two hybrids until the particular step are indistinguishable. Again, note that both hybrids are the same up to step 5 (inclusively) and hence the views after executing step 5 have the same distribution. The difference between the two hybrids is that r' is sampled uniformly random in $[t]$ in Hybrid_4 and set to equal r in Hybrid_5. However, r is a uniform random element in $[t]$, no state of the main thread (including r) is used during the execution of the rewound thread, the adversary cannot know anything about the main thread after being rewound and the actions taken in the rewound thread of both hybrids are the same. Hence, the view of the adversary up to step 5* is distributed equally in both hybrids.

After step 5* the adversary is rewound until it displays the same strategy as in the main thread. At this point there can emerge a discrepancy in the view of the adversary in the two hybrids.

Intuitively speaking, in Hybrid_4, it can happen that the strategy the adversary would *prefer* in the rewound thread (based on coin r') is different from the one determined by the main thread (with coin r) that is enforced on the adversary via rewinding. Hence, the view distribution in Hybrid_4 can contain more final views that involve a coin r' and a strategy B_1 than expected from an experiment without rewinding, because r' is typically associated with a strategy B_2. Looking ahead, such a view distribution is not possible in Hybrid_5. Still intuitively speaking, we will show via a reduction to the security guarantees of the time-lock puzzle that an adversary is not able to base its decision on the coin r' and hence cannot utilize its decision to distinguish the two hybrids.

More formally, we define a class of PPT adversaries $\mathbb{A}$, for which there exist a strategy B which is followed in the main thread of both hybrids with non-negligible probability, in the rewound thread of Hybrid_4 with probability $\mathsf{Pr}[\mathsf{Hybrid}_4, B]$ and in the rewound thread of Hybrid_5 with $\mathsf{Pr}[\mathsf{Hybrid}_5, B]$, such that $|\mathsf{Pr}[\mathsf{Hybrid}_4, B] - \mathsf{Pr}[\mathsf{Hybrid}_5, B]| > \mathsf{negl}$. It follows that all adversaries $A \notin \mathbb{A}$ follow every strategy B that is followed in the main thread with non-negligible probability with negligible close probability in the rewound threads of Hybrid_4 and Hybrid_5.

We continue with showing that statement 3 holds for all $A \notin \mathbb{A}$. After jumping from the rewound loop back to the main thread the view of the adversary is equal to the one after step 5* but involves a strategy which is enforced by the simulator. The strategy enforced from outside looks arbitrary to the adversary as the adversary has no information about the main thread. Still, the enforced strategy has the same distribution in both hybrids as the main thread determining the enforced strategy is the same in both hybrids. Additionally, each $A \notin \mathbb{A}$ follows each strategy with the same probability in the rewound thread of Hybrid_4 and Hybrid_5 and hence independent of the coin r'. Therefore, A cannot use the enforced strategy to distinguish Hybrid_4 and Hybrid_5 and the view of A right before step 6 os computationally indistinguishable. As the two hybrids are the same in the succeeding steps, the view of each $A \notin \mathbb{A}$ in both hybrids is computationally indistinguishable.

Finally, we will show that $\mathbb{A} = \emptyset$ via a reduction to the security guarantees of the time-lock puzzle and hence show that statement 3 holds for all PPT adversaries. An observation, we need for the reduction to the time-lock puzzle is that the execution of $\mathcal{S}$ for the protocol in steps 3* to 5*, denoted as $\mathcal{S}_{3*-5*}$, takes time [8] $T(\mathcal{S}_{3*-5*}, \mathcal{S}_{2*}^{\mathsf{state}})$, where $\mathcal{S}_{2*}^{\mathsf{state}}$ is the state of the simulator right after step 2*. Note that adversary $\mathcal{A}$ is internally executed by $\mathcal{S}$. Hence, $\mathcal{S}_{3*-5*}$ also includes the execution of $\mathcal{A}$ for the steps 3* to 5* and thus the decision of $\mathcal{A}$ to continue or abort. For the same reason, $\mathcal{S}_{2*}^{\mathsf{state}}$ includes the state of $\mathcal{A}$ right before step 3*.

[8] See section 3.1 for the notion of time used in this work and the definition of the time-cost function $T(\cdot, \cdot)$.

If $\mathcal{A}$ follows a different strategy than in the main thread in steps 3* or 5*, $\mathcal{S}$ interprets this different strategy as abort. Each of these steps takes maximally one communication round (of time T_c). Thus, in case of an aborting $\mathcal{A}$, $\mathcal{S}_{3*-5*}$ takes maximally time $2 \cdot T_c < \mathcal{T}$ Finally, if $\mathcal{A}$ wants to continue, it needs to sent valid messages in step 3* or 5*. Otherwise, the behavior is interpreted as abort. As the messages need to be sent within the correct communication round, continuing also takes time less than $\mathcal{T}$. Hence, it holds that $T(\mathcal{S}_{3*-5*}, \mathcal{S}_{2*}^{\mathsf{state}}) \leq \mathcal{T}$.

Next, we will show that the existence of any attacker $\mathcal{A}' \in \mathbb{A}$ contradicts the security guarantees of the time-lock puzzle and thus show that $\mathbb{A} = \emptyset$. We construct a polynomial-time adversary $(\mathcal{A}_1^{\mathsf{TLP}}, \mathcal{A}_2^{\mathsf{TLP}})$ against the security of the time-lock puzzle, where the running time of $\mathcal{A}_2^{\mathsf{TLP}}$ is bounded from above by $\mathcal{T}$. This adversary $(\mathcal{A}_1^{\mathsf{TLP}}, \mathcal{A}_2^{\mathsf{TLP}})$ utilizes an arbitrary $\mathcal{A}' \in \mathbb{A}$ and acts as $\mathcal{S}$, when interacting with $\mathcal{A}'$.

Without loss of generality assume $\mathcal{A}'$ follows a strategy B (which leads to rewinding) in the main thread with non-negligible probability $\mathsf{Pr}[\mathsf{main}, B]$, in the rewound threads of Hybrid_4 with probability $\mathsf{Pr}[\mathsf{Hybrid}_4, B]$ and in the rewound threads of Hybrid_5 with probability $\mathsf{Pr}[\mathsf{Hybrid}_5, B]$ such that $\mathsf{Pr}[\mathsf{Hybrid}_4, B] > \mathsf{Pr}[\mathsf{Hybrid}_5, B] + \mathsf{negl}$.

In the time-lock puzzle security game, $\mathcal{A}_1^{\mathsf{TLP}}$ executes one instance of $\mathcal{S}$ in Hybrid_4 up to right after protocol step 5. If $\mathcal{A}'$ does not follow strategy B or aborts before step 5, $\mathcal{A}_1^{\mathsf{TLP}}$ sends arbitrary values to the security game and $\mathcal{A}_2^{\mathsf{TLP}}$ outputs 1. Otherwise, $\mathcal{A}_1^{\mathsf{TLP}}$ continues with the simulation until step 2* and extracts the decommitments $\{d_j^i\}_{i\in[n],j\in[t]}$, the coin r, which has been used in the main thread, and the coin r', which has not been used yet, from the current state of the simulator, $\mathcal{S}_{2*}$. As $\mathcal{S}_{2*}^{\mathsf{state}}$ corresponds to the execution of Hybrid_4, it is possible that $r \neq r'$.

Then, $\mathcal{A}_1^{\mathsf{TLP}}$ sends two secrets s_0 and s_1 to the security game, where

$$s_0 := (r', \{d_j^i\}_{i\in[n],j\in[t]})$$

and

$$s_1 := (r, \{d_j^i\}_{i\in[n],j\in[t]}).$$

Upon receiving a puzzle Z_b, $\mathcal{A}_2^{\mathsf{TLP}}$ continues with the rewound thread $\mathcal{S}_{2*}$ by executing $\mathcal{S}_{3*-5*}$ but using Z as time-lock puzzle p. If the adversary repeats strategy B output 0, otherwise output 1.

Note that depending on whether the game returns s_0 or s_1, the rewound threads executed by $\mathcal{A}_2^{\mathsf{TLP}}$ correspond exactly to the rewound thread executed by Hybrid_4 or the rewound thread executed by Hybrid_5. Since $\mathcal{S}_{3*-5*}$ runs in time $T < \mathcal{T}$, as shown above, $\mathcal{A}_2^{\mathsf{TLP}}$ fulfills the requirement of being bounded from above by $\mathcal{T}$.

The win probability is calculated as follows. If the security game has chosen s_0, the probability that $(\mathcal{A}_1^{\mathsf{TLP}}, \mathcal{A}_2^{\mathsf{TLP}})$ wins equals the probability that B is followed in the main thread and in the rewound thread. As the security game has chosen s_0, the rewound thread corresponds to Hybrid_4:

$$\mathsf{Pr}[\mathsf{win}|b = 0] = \mathsf{Pr}[\mathsf{main}, B] \cdot \mathsf{Pr}[\mathsf{Hybrid}_4, B]$$

If the security game has chosen s_1, the probability that $(\mathcal{A}_1^{\mathsf{TLP}}, \mathcal{A}_2^{\mathsf{TLP}})$ wins equals the probability that B is not chosen in the main thread or B is chosen in the main thread but not in the rewound thread. As the security game has chosen s_1, the rewound thread corresponds to Hybrid_5:

$$\mathsf{Pr}[\mathsf{win}|b=1] = (1 - \mathsf{Pr}[\mathsf{main}, B]) + \mathsf{Pr}[\mathsf{main}, B] \cdot (1 - \mathsf{Pr}[\mathsf{Hybrid}_5, B])$$

Overall, $(\mathcal{A}_1^{\mathsf{TLP}}, \mathcal{A}_2^{\mathsf{TLP}})$ wins the security game with probability:

$$\begin{aligned}\mathsf{Pr}[\mathsf{win}] &= \frac{1}{2} \cdot (\mathsf{Pr}[\mathsf{main}, B] \cdot \mathsf{Pr}[\mathsf{Hybrid}_4, B]) \\ &+ \frac{1}{2} \cdot ((1 - \mathsf{Pr}[\mathsf{main}, B]) + \mathsf{Pr}[\mathsf{main}, B] \cdot (1 - \mathsf{Pr}[\mathsf{Hybrid}_5, B])) \\ &= \frac{1}{2} + \frac{1}{2} \cdot (\mathsf{Pr}[\mathsf{main}, B] \cdot \mathsf{Pr}[\mathsf{Hybrid}_4, B] - \mathsf{Pr}[\mathsf{main}, B] \cdot \mathsf{Pr}[\mathsf{Hybrid}_5, B])\end{aligned}$$

As $\mathsf{Pr}[\mathsf{main}, B] > \mathsf{negl}$ and $\mathsf{Pr}[\mathsf{Hybrid}_4, B] > \mathsf{Pr}[\mathsf{Hybrid}_5, B] + \mathsf{negl}$, $(\mathcal{A}_1^{\mathsf{TLP}}, \mathcal{A}_2^{\mathsf{TLP}})$ wins the security game with non-negligible advantage. This contradicts the security of the time-lock puzzle. It follows that $\mathbb{A} = \emptyset$.

Hence, statement 3 holds for all PPT adversaries $\mathcal{A}$ and Hybrid_4 and Hybrid_5 are computationally indistinguishable.

$\underline{\mathsf{Hybrid}_6:}$

In Hybrid_6, we simplify the specification of the experiment in order to ease the explanations in the following hybrid. In particular, we replace all occasions of r' with r as they are always identical (r' is always defined to equal r before being used). Additionally, observe that in step 6 the experiment does exactly the same if $\mathsf{flag} = \mathsf{undetected}$ or $\mathsf{flag} = \mathsf{honest}$. Therefore, we merge these two cases into one. As the changes made in Hybrid_6 are just semantic and do not influence the output distribution of Hybrid_6, Hybrid_5 and Hybrid_6 are identical.

$\underline{\mathsf{Hybrid}_7:}$

In this hybrid, we remove the remaining rewinding steps by *collapsing* the rewound thread. As the steps 1* - 5* are identical to steps 1 - 5, the resulting experiment (Hybrid_7) is indistinguishable from Hybrid_6.

0. Set $r \xleftarrow{\$} [t]$ and $M_1, M_2, L = \emptyset$. Then $\mathcal{S}$ generates keys $(\mathsf{pk}_h, \mathsf{sk}_h)$ and sends pk_h to $\mathcal{A}$.
1. $\mathcal{S}$ acts like an honest party during the seed generation and simulates $\mathcal{F}_{\mathsf{coin}}$ as stated in the definition. This way, $\mathcal{S}$ obtains public seed shares $\{\mathsf{seed}_j^{(2,i)}\}_{i\in[n]}$ and commitments $\{c_j^i\}_{i\in[n]}$ for $j \in [t]$. By simulating the honest party, $\mathcal{S}$ gets private seed shares $\mathsf{seed}_j^{(1,h)}$, commitments and openings $(c_j^h, d_j^h) \leftarrow \mathsf{Commit}(\mathsf{seed}_j^{(1,h)})$ for $j \in [t]$. Use the extractability property of the commitment scheme to obtain the private seed shares $\{\mathsf{seed}_j^{(1,i)}\}_{i\in[n]\setminus h}$. Finally, compute $\mathsf{seed}_j^i := \mathsf{seed}_j^{(1,i)} \oplus \mathsf{seed}_j^{(2,i)}$ for $j \in [t]$ and $i \in [n]$.
2. For $j \in [t]$, run an execution of Π_{SH} with $\mathcal{A}$ where the randomness is derived from seed_j^h. Let trans_j denote the transcript and y_j^h denote the output of P_h in

the j-th execution.

$\mathcal{S}$ uses $\{\mathsf{seed}^i_j\}_{i\in[n]}$ to recompute the j-th semi-honest instance and to obtain transcript trans'_j for $j \in [t]$. For every $l \in [t]$ where $\mathsf{trans}'_l \neq \mathsf{trans}_l$, determine the first party P_m that has deviated from the protocol description at location loc, add (m, l, loc) to M_2 and l to L.

3. $\mathcal{S}$ receives message $\{(r^i, u^i, \{d^i_j\}_{j\in[t]})\}_{i\in[n]\setminus h}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{PG}}$ and executes the following steps.
   (a) If there is $l \in [t]$ and $m \in [n]$ where $\mathsf{Open}(c^m_l, d^m_l) = \mathsf{seed}^{(1,m)*}_l \neq \perp$ and $\mathsf{seed}^{(1,m)*}_l \neq \mathsf{seed}^{(1,m)}_l$, $\mathcal{S}$ outputs $\mathsf{ambiguous}$ and halts.
   (b) For every $l \in [t]$ and $m \in [n]$ where $\mathsf{Open}(c^m_l, d^m_l) = \perp$, add m to M_1 and l to L.
   (c) Compute $p \leftarrow \mathsf{TL.Generate}(pp, (r, \{d^i_j\}_{i\in[n],j\in[t]\setminus r}))$. $\mathcal{S}$ internally hands p to $\mathcal{A}$.
4. $\mathcal{S}$ creates signatures σ^h_j for $j \in [t]$ as an honest party would do and hands $\{\sigma^h_j\}_{j\in[t]}$ to $\mathcal{A}$. $\mathcal{S}$ receives $\{\sigma^i_j\}_{i\in[n]\setminus h, j\in[t]}$ that $\mathcal{A}$ sends to P_h.
5. If $\mathcal{A}$ has not sent valid messages in the first protocol steps in the expected communication round, hand abort to $\mathcal{A}$, P_h outputs abort, and terminate the experiment.
   Otherwise, we distinguish three cases.
   – If $|L| \geq 2$ then set $\mathsf{flag} := \mathsf{corrupted}$, and continue below.
   – If $|L| = 1$ then set $\mathsf{flag} := \mathsf{corrupted}$ if $r \notin L$ and set $\mathsf{flag} := \mathsf{undetected}$ if $r \in L$. In any case continue below.
   – If $|L| = 0$ then set $\mathsf{flag} := \mathsf{honest}$, and continue below.
6. $\mathcal{S}$ receives message $\{\mathsf{continue}_i\}_{i\in[n]\setminus h}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{PG}}$ and hands $(r, \{d^i_j\}_{j\in[t]\setminus r, i\in[n]})$ back to $\mathcal{A}$. If $\mathcal{A}$ responds with abort or $\mathcal{S}$ has not received $\mathsf{continue}_i$ for all $i \in [n] \setminus h$, continue in step 7.
   If $\mathcal{A}$ does not respond with abort differentiate the following cases:
   – If $\mathsf{flag} \in \{\mathsf{honest}, \mathsf{undetected}\}$ then P_h outputs y^h_r and the experiment halts.
   – If $\mathsf{flag} = \mathsf{corrupted}$ compute $(m, \mathsf{cert}) := \mathsf{Blame}(\mathsf{view}^h)$, where view^h is the view of the simulated party P_h. Internally send cert to $\mathcal{A}$, P_h outputs $\mathsf{corrupted}_m$ and the experiment halts.
7. If $\mathsf{flag} = \mathsf{corrupted}$ then $\mathcal{S}$ computes $((r, \{d^i_j\}_{j\in[t],i\in[n]}), \pi) := \mathsf{TL.Solve}(pp, p)$ and $(m, \mathsf{cert}) := \mathsf{Blame}(\mathsf{view}^h)$, where view^h is the view of the simulated party P_h, internally sends cert to $\mathcal{A}$, P_h outputs $\mathsf{corrupted}_m$, and the experiment halts.
   If $\mathsf{flag} \in \{\mathsf{undetected}, \mathsf{honest}\}$ then hand abort to $\mathcal{A}$, P_h outputs abort and the experiment halts.

<u>Hybrid_8 :</u>

Up to Hybrid_8, $\mathcal{S}$ utilizes extracted variables that should remain private during the whole protocol to assess the adversary's behavior, i.e., the decommitment d^m_r. In Hybrid_8, we avoid the utilization of this *special* power and assess the behavior solely based on variables that will eventually be published unless the adversaries chooses to abort. However, the variables might not public at the time they are used. To this end, we change the computation of the sets M_1, M_2 and L. Instead of checking misbehavior of the adversary in all instances $j \in [t]$, we restrict the simulator to $j \in [t] \setminus r$. To emphasize the difference, we define the sets $\hat{M}_1$, $\hat{M}_2$ and $\hat{L}$ in step 0. In step 2 and 3 of Hybrid_8, $\mathcal{S}$ construct the sets $\hat{M}_1$, $\hat{M}_2$ and $\hat{L}$ only based on instances $l \in [t] \setminus r$. We further change step 5:

0. Set $r \xleftarrow{\$} [t]$ and $\hat{M}_1, \hat{M}_2, \hat{L} = \emptyset$. Then $\mathcal{S}$ generates keys $(\mathsf{pk}_h, \mathsf{sk}_h)$ and sends pk_h to $\mathcal{A}$.
2. For $j \in [t]$, run an execution of Π_{SH} with $\mathcal{A}$ where the randomness is derived from seed_j^h. Let trans_j denote the transcript and y_j^h denote the output of P_h in the j-th execution.
   $\mathcal{S}$ uses $\{\mathsf{seed}_j^i\}_{i\in[n]}$ to recompute the j-th semi-honest instance and to obtain transcript $\widetilde{\mathsf{trans}}_j$ for $j \in [t] \setminus r$. For every $l \in [t] \setminus r$ where $\widetilde{\mathsf{trans}}_l \neq \mathsf{trans}_l$, determine the first party P_m that has deviated from the protocol description at location loc, add (m, l, loc) to $\hat{M}_2$ and l to $\hat{L}$.
3. $\mathcal{S}$ receives message $\{(r^i, u^i, \{d_j^i\}_{j\in[t]})\}_{i\in[n]\setminus h}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{PG}}$ and executes the following steps.
   (a) If there is $l \in [t]$ and $m \in [n]$ where $\mathsf{Open}(c_l^m, d_l^m) = \mathsf{seed}_l^{(1,m)*} \neq \perp$ and $\mathsf{seed}_l^{(1,m)*} \neq \mathsf{seed}_l^{(1,m)}$, $\mathcal{S}$ outputs $\mathsf{ambiguous}$ and halts.
   (b) For every $l \in [t] \setminus r$ and $m \in [n]$ where $\mathsf{Open}(c_l^m, d_l^m) = \perp$, add m to $\hat{M}_1$ and l to $\hat{L}$.
   (c) Compute $p \leftarrow \mathsf{TL.Generate}(pp, (r, \{d_j^i\}_{i\in[n], j\in[t]\setminus r}))$. $\mathcal{S}$ internally hands p to $\mathcal{A}$.
5. If $\mathcal{A}$ has not sent valid messages in the first protocol steps in the expected communication round, hand abort to $\mathcal{A}$, P_h outputs abort, and terminate the experiment.
   Otherwise, we distinguish two cases.
   – If $\hat{L} \neq \emptyset$ then set $\mathsf{flag} := \mathsf{corrupted}$ and continue below.
   – If $\hat{L} = \emptyset$ then set $\mathsf{flag} := \mathsf{undetected}$ and continue below.

Let L be defined as in Hybrid_7.
It holds that

$$|L| \geq 2 \text{ or } |L| = 1; r \notin L \Leftrightarrow \hat{L} \neq \emptyset$$

and

$$|L| = 1; r \in L \text{ or } |L| = 0 \Leftrightarrow \hat{L} = \emptyset.$$

Moreover, $\mathcal{S}$ acts identically in steps 6 and 7 if $\mathsf{flag} = \mathsf{honest}$ or $\mathsf{flag} = \mathsf{undetected}$. Therefore, we can merge both cases, honest behavior and undetected cheating, to the case where $\mathsf{flag} := \mathsf{undetected}$

It follows that Hybrid_7 and Hybrid_8 are distributed identically.

Hybrid_9 :
In Hybrid_9, we go one step further and avoid $\mathcal{S}$'s utilization of variables before they become public. To this end, we change steps 6 and 7 by differentiating the behavior of the adversary based on the output of the Blame-algorithm instead of the value of flag. Since this means that flag is never used by the simulator, we may change step 5, where flag is defined. Moreover, the sets $\hat{M}_1$, $\hat{M}_2$, and $\hat{L}$ are only required for defining flag in step 5, and thus we can remove the computation of these sets from steps 2 and 3 as well. The changed steps of Hybrid_9 are as follows:

0. Set $r \xleftarrow{\$} [t]$. Then $\mathcal{S}$ generates keys $(\mathsf{pk}_h, \mathsf{sk}_h)$ and sends pk_h to $\mathcal{A}$.

2. For $j \in [t]$, run an execution of Π_{SH} with $\mathcal{A}$ where the randomness is derived from seed_j^h. Let trans_j denote the transcript and y_j^h denote the output of P_h in the j-th execution.
3. $\mathcal{S}$ receives message $\{(r^i, u^i, \{d_j^i\}_{j\in[t]})\}_{i\in[n]\setminus h}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{PG}}$ and executes the following steps.
   If there is $l \in [t]$ and $m \in [n]$ where $\mathsf{Open}(c_l^m, d_l^m) = \mathsf{seed}_l^{(1,m)*} \neq \bot$ and $\mathsf{seed}_l^{(1,m)*} \neq \mathsf{seed}_l^{(1,m)}$, $\mathcal{S}$ outputs $\mathsf{ambiguous}$ and halts.
   Otherwise, compute $p \leftarrow \mathsf{TL.Generate}(pp, (r, \{d_j^i\}_{i\in[n], j\in[t]\setminus r}))$. $\mathcal{S}$ internally hands p to $\mathcal{A}$.
5. If $\mathcal{A}$ has not sent valid messages in the first protocol steps in the expected communication round, hand abort to $\mathcal{A}$, P_h outputs abort, and terminate the experiment.
6. $\mathcal{S}$ receives message $\{\mathsf{continue}_i\}_{i\in[n]\setminus h}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{PG}}$ and hands $(r, \{d_j^i\}_{j\in[t]\setminus r, i\in[n]})$ back to $\mathcal{A}$. If $\mathcal{A}$ responds with abort or $\mathcal{S}$ has not received $\mathsf{continue}_i$ for all $i \in [n] \setminus h$, continue in step 7.
   If $\mathcal{A}$ does not respond with abort then compute $(m, \mathsf{cert}) := \mathsf{Blame}(\mathsf{view}^h)$, where view^h is the view of the simulated party P_h. If $\mathsf{cert} \neq \bot$, internally send cert to $\mathcal{A}$, P_h outputs $\mathsf{corrupted}_m$ and the experiment halts. Otherwise, if $\mathsf{cert} = \bot$, P_h outputs y_r^h and the experiment halts.
7. $\mathcal{S}$ computes $((r, \{d_j^i\}_{j\in[t], i\in[n]}), \pi) := \mathsf{TL.Solve}(pp, p)$ and $(m, \mathsf{cert}) := \mathsf{Blame}(\mathsf{view}^h)$, where view^h is the view of the simulated party P_h. If $\mathsf{cert} \neq \bot$, internally send cert to $\mathcal{A}$, P_h outputs $\mathsf{corrupted}_m$ and the experiment halts.
   Otherwise, if $\mathsf{cert} = \bot$, then hand abort to $\mathcal{A}$, P_h outputs abort and the experiment halts.

We first observe that if the opening values d_j^i provided by the adversary in step 3 open to some valid seed $\mathsf{seed}_j^{(1,i)*} \neq \mathsf{seed}_j^{(1,i)}$, $\mathcal{S}$ outputs $\mathsf{ambiguous}$ and halts in both Hybrid_8 and Hybrid_9.

In case the adversary provides an incorrect opening value d_j^i in step 3 of Hybrid_8 such that $\mathsf{Open}(c_j^i, d_j^i) = \bot$ for any $i \in [n] \setminus h$ and $j \in [t] \setminus r$, $\mathcal{S}$ adds j to set $\hat{L}$ and defines $\mathsf{flag} := \mathsf{corrupted}$ in step 5. If $\mathcal{A}$ executes the same behavior in Hybrid_9, the Blame-algorithm detects that $\mathsf{Open}(c_j^i, d_j^i) = \bot$ and outputs a valid certificate.

In case the adversary provides correct opening values, such that $\mathsf{Open}(c_j^i, d_j^i) = \mathsf{seed}_j^{(1,i)}$, for all $i \in [n] \setminus h$ and $j \in [t] \setminus r$, the opened values used in the Blame-algorithm equal the values extracted by $\mathcal{S}$ in step 1 of Hybrid_8. Therefore, the recomputation of the semi-honest instances are exactly the same and any misbehavior is detected in both hybrids.

Since the Blame-algorithm analyzes the behavior of all parties for instances $j \in [t] \setminus r$, the Blame algorithm performs the same checks as computed explicitly by $\mathcal{S}$ in step 2, 3 and 5 in Hybrid_8 and, hence, captures the identical behavior. It follows that for $(m, \mathsf{cert}) := \mathsf{Blame}(\mathsf{view}^h)$ it holds

$$\mathsf{cert} \neq \bot \Leftrightarrow \mathsf{flag} = \mathsf{corrupted}$$

and

$$\mathsf{cert} = \bot \Leftrightarrow \mathsf{flag} = \mathsf{undetected}.$$

It follows that Hybrid_8 and Hybrid_9 are distributed identically.

$\underline{\mathsf{Hybrid}_{10}:}$
In Hybrid_{10}, we show that $\mathcal{S}$ outputs ambiguous in the experiment only with negligible probability via reduction to the binding property of the commitment scheme. This allows us to remove $\mathcal{S}$'s extra power of detecting ambiguous opening values while obtaining a computational indistinguishable Hybrid_{10}. Step 3 is as follows.

3. $\mathcal{S}$ receives message $\{(r^i, u^i, \{d^i_j\}_{j\in[t]})\}_{i\in[n]\setminus h}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{PG}}$, computes $p \leftarrow \mathsf{TL.Generate}(pp, (r, \{d^i_j\}_{i\in[n], j\in[t]\setminus r}))$ and internally hands p to $\mathcal{A}$.

In order to show computational indistinguishability between Hybrid_9 and Hybrid_{10}, we need to show that $\mathcal{S}$ output ambiguous in Hybrid_9 only with negligible probability. To this end, we construct an adversary $\mathcal{A}_{\mathsf{Bin}}$ against the binding property of the commitment scheme that utilizes adversary $\mathcal{A}$ of Hybrid_9. We show that if $\mathcal{A}$ force $\mathcal{S}$ in Hybrid_9 to output ambiguous with non-negligible probability, our constructed adversary $\mathcal{A}_{\mathsf{Bin}}$ breaks the binding property.

Adversary $\mathcal{A}_{\mathsf{Bin}}$ acts exactly like $\mathcal{S}$ in Hybrid_9. In particular, $\mathcal{A}_{\mathsf{Bin}}$ uses the extractability property of the commitment scheme to obtain the value $\mathsf{seed}^{(1,i)}_j$ and the opening values d^i_j for $i \in [n] \setminus h$ and $j \in [t]$ in step 1 and $\mathcal{A}_{\mathsf{Bin}}$ checks if adversary $\mathcal{A}$ provides a value d^{m*}_l in step 3 where $\mathsf{Open}(c^m_l, d^{m*}_l) = \mathsf{seed}^{(1,m)*}_l \neq \perp$ and $\mathsf{seed}^{(1,m)*}_l \neq \mathsf{seed}^{(1,m)}_l$. If this is the case, $\mathcal{A}_{\mathsf{Bin}}$ found a tuple (c^m_l, d^m_l, d^{m*}_l) which breaks the binding property.

The success probability of $\mathcal{A}_{\mathsf{Bin}}$ is exactly the probability that the simulator $\mathcal{S}$ in Hybrid_9 outputs ambiguous. By contradiction, it follows that $\mathcal{S}$ outputs ambiguous only with negligible probability if the binding property holds.

$\underline{\mathsf{Hybrid}_{11}:}$
Finally, we want $\mathcal{S}$ to execute $\mathcal{F}_{\mathsf{PG}}$, $\mathcal{F}_{\mathsf{coin}}$ and the commitment scheme as intended, without injecting forged variables or extracting secret information. To this end, we replace choosing $r \xleftarrow{\$} [t]$ at the outset of the experiment with sampling $r^h \xleftarrow{\$} [t]$ and setting $r := \sum_{i=1}^{n} r^i \mod t$ in step 3. Moreover, we remove the extraction of the corrupted parties' seed shares in step 1. By applying these changes, the steps presented below changed accordingly. This results in the real world protocol execution.

0. $\mathcal{S}$ generates keys $(\mathsf{pk}_h, \mathsf{sk}_h)$ and sends pk_h to $\mathcal{A}$.
1. $\mathcal{S}$ acts like an honest party during the seed generation and simulates $\mathcal{F}_{\mathsf{coin}}$ as stated in the definition. This way, $\mathcal{S}$ obtains public seed shares $\{\mathsf{seed}^{(2,i)}_j\}_{i\in[n]}$ and commitments $\{c^i_j\}_{i\in[n]}$ for $j \in [t]$. By simulating the honest party, $\mathcal{S}$ gets private seed shares $\mathsf{seed}^{(1,h)}_j$, commitments and openings $(c^h_j, d^h_j) \leftarrow \mathsf{Commit}(\mathsf{seed}^{(1,h)}_j)$ for $j \in [t]$. Finally, compute $\mathsf{seed}^h_j := \mathsf{seed}^{(1,h)}_j \oplus \mathsf{seed}^{(2,h)}_j$ for $j \in [t]$.
2. $\mathcal{S}$ receives message $\{(r^i, u^i, \{d^i_j\}_{j\in[t]})\}_{i\in[n]\setminus h}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{PG}}$ and executes the following steps.

Sample $r^h \overset{\$}{\leftarrow} [t]$ and set $r := \sum_{i=1}^{n} r^i \bmod t$. Compute $p \leftarrow \mathsf{TL.Generate}(pp, (r, \{d_j^i\}_{i\in[n],j\in[t]\setminus r}))$. Internally hand p to $\mathcal{A}$.

Since r^h is chosen uniform at random from $[t]$ the value of r is distributed uniform randomly as well. Thus, r is distributed identically in Hybrid_{10} and Hybrid_{11}.

Moreover, the seeds of the corrupted parties seed_j^i for $j \in [t]\setminus r$ and $i \in [n]\setminus h$ are only required in the computation of the Blame-algorithm in steps 6 and 7. Since the values can be computed within Blame after the openings of the private seed shares d_j^i are either revealed in 6 or by solving the time-lock puzzle in 7, there is no need to use the extractability property of the commitment scheme in step 1. Hybrid_{10} and Hybrid_{11} are distributed identically.

Since Hybrid_{11} equals the real-world execution, this concludes the proof and shows that the ideal-world execution is computationally indistinguishable from the real-world execution.

B Input-Dependent Protocol

In this section, we specify the input-dependent compiler described in Section 6. As the compiler is similar to the input-independent one, we emphasize the differences to the input-independent compiler by highlighting them in red.

The protocol of the input-dependent compiler. The main difference between the input-dependent publicly verifiable covert protocol ($\Pi_{\mathsf{PVC}}^{\mathsf{id}}$) and the input-independent one (Π_{PVC}) is that in $\Pi_{\mathsf{PVC}}^{\mathsf{id}}$ parties do not run t instances of a semi-honest n-party protocol but one instance of a semi-honest $(t \cdot n)$-party protocol, in which each real party plays the role of t virtual parties. Real parties secret share their input between each of their virtual parties and simulate the actions of the virtual parties in the protocol Π_{SH}^{*} (see Ssection 6 for the specification of Π_{SH}^{*}). In the verification step, parties open all but one of their virtual parties by revealing the input shares and random tapes used by the virtual parties.

Protocol $\Pi_{\mathsf{PVC}}^{\mathsf{id}}$

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 and secret input x^h.

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

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

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

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

2. Party P_h creates t secret shares x_j^h of input x^h such that $x^h = \bigoplus_{j\in[t]} x_j^h$, calculates $(c_j^h, d_j^h) := \mathsf{Commit}(x_j^h)$ for all $j \in [t]$ and sends $\{c_j^h\}_{j\in[t]}$ to each other party.

Semi-honest protocol execution:

3. Party P_h engages in one instance of the $(t \cdot n)$-parties protocol Π^*_{SH} with all other parties. Party P_h plays the role of the virtual parties V_h^j, which has the input x_h^j, uses randomness derived from seed_j^h and receives output y_j^h, for $j \in [t]$. Additionally, P_h receives the protocol transcript trans. Denote by trans_j^i the ordered set of all messages that have been sent or received by the virtual party V_j^i. As all messages are sent to all parties, each party can construct each set trans_j^i.

Create publicly verifiable evidence:

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

$$\mathsf{data}_j^h := (h, j, \mathsf{seed}_j^{(2,i)}, c_j^{(1,i)}, c_j^i, p, \mathsf{trans}_j^i).$$

P_h broadcasts its signatures and verifies the received signatures.

Optimistic case:

6. If any of the following cases happens
   – P_h has not received valid messages in the first protocol steps in the expected communication round.
   – $\mathcal{F}_{\mathsf{PG}}$ returned abort, or
   – any other party has sent abort

   party P_h broadcasts and outputs abort.
7. Otherwise, P_h sends $\mathsf{continue}_h$ to $\mathcal{F}_{\mathsf{PG}}^{\mathsf{id}}$, receives $(r, \{(d_j^{*(1,i)}, d_j^{*i})\}_{i\in[n],j\in[t]\setminus r})$ as response and calculates

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

where view^h is the view of P_h.
If $\mathsf{cert} \neq \perp$, broadcast cert and output $\mathsf{corrupted}_m$. Otherwise, P_h outputs $\bigoplus_{j\in[t]} y_j^h$.

Pessimistic case:

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

$$((r, \{(d_j^{*(1,i)}, d_j^{*i})\}_{i\in[n],j\in[t]\setminus r}), \pi) := \mathsf{TL.Solve}(pp, p)$$

and calculates

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

where view^h is the view of P_h.
If $\mathsf{cert} \neq \perp$, broadcast cert and output $\mathsf{corrupted}_m$. Otherwise, output abort.

Puzzle generation functionality of the input-dependent compiler. The maliciously secure puzzle generation functionality, $\mathcal{F}_{\mathsf{PG}}^{\mathsf{id}}$, which can be instantiated using a maliciously secure general purpose multi-party computation protocol, takes as additional input the openings (d_j^i) of each virtual party's input share and includes them into the puzzle and output according to the coin toss.

Functionality $\mathcal{F}_{\mathsf{PG}}^{\mathsf{id}}$

Inputs: Each party P_i with $i \in [n]$ inputs $(r^i, u^i, \{(d_j^{(1,i)}, d_j^i)\}_{j\in[t]})$, where $r^i \in [t]$, $u^i \in \{0,1\}^\kappa$, and $(d_j^{(1,i)}, d_j^i) \in \{0,1\}^{2\cdot\kappa}$.

- Compute $r := \sum_{i=1}^n r^i \mod t$ and $u := \bigoplus_{i=1}^n u^i$.
- Generate puzzle $p \leftarrow \mathsf{TL.Generate}(pp, (r, \{(d_j^{(1,i)}, d_j^i)\}_{i\in[n], j\in[t]\setminus r}))$ 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.
- Upon receiving $\mathsf{continue}$ from each party, send $(r, \{(d_j^{(1,i)}, d_j^i)\}_{i\in[n], j\in[t]\setminus r})$ to $\mathcal{A}$.
  - If $\mathcal{A}$ returns abort or some party does not send $\mathsf{continue}$, send abort to all honest parties and stop.
  - Otherwise, send $(r, \{(d_j^{(1,i)}, d_j^i)\}_{i\in[n], j\in[t]\setminus r})$ to all honest parties.

Blame algorithm of the input-dependent compiler. The blame algorithm of the input-dependent protocol, $\mathsf{Blame}^{\mathsf{id}}$, checks that the openings of the input shares and seed shares are correct and that the semi-honest protocol between the virtual parties has been executed correctly. For the latter, the algorithm simulates the protocol execution between the virtual parties. The opened virtual parties are simulated based on their randomness and input share and the unopened ones by re-using their messages of the original transcript.

Algorithm $\mathsf{Blame}^{\mathsf{id}}$

On input the view view of a party which contains:

- public parameters (n, t)
- public seed shares $\{\mathsf{seed}_j^{(2,i)}\}_{i\in[n]}$
- shared coin r
- private seed share commitments and decommitments $\{c_j^{(1,i)}, d_j^{(1,i)}\}_{i\in[n], j\in[t]\setminus r}$
- input share commitments and decommitments $\{c_j^i, d_j^i\}_{i\in[n], j\in[t]\setminus r}$
- additional certificate information $(\{\mathsf{pk}_i\}_{i\in[n]}, \pi, \{(\mathsf{data}_j^i, \sigma_j^i)\}_{i\in[n], j\in[t]})$

do:

1. Calculate $\mathsf{seed}_j^{(1,i)} := \mathsf{Open}(c_j^{(1,i)}, d_j^{(1,i)})$ and $x_j^i := \mathsf{Open}(c_j^i, d_j^i)$ for each $i \in [n], j \in [t] \setminus r$.

2. Let $M_1 := \{(i,j) \in ([n],[t] \setminus r) : \mathsf{seed}_j^{(1,i)} = \perp \text{ or } x_j^i = \perp\}$. If $M_1 \neq \emptyset$, choose the tuple $(m,l) \in M_1$ with minimal m and l, prioritized by m, compute $(\cdot,\pi) := \mathsf{TL.Solve}(pp,p)$, if $\pi = \perp$, set $\mathsf{cert} := (\mathsf{pk}_m, \mathsf{data}, \pi, r, \{(d_j^{(1,i)}, d_j^i)\}_{i\in[n],j\in[t]\setminus r}, \sigma_l^m)$ and output (m, cert).
3. Set $\mathsf{seed}_j^i := \mathsf{seed}_j^{(1,i)} \oplus \mathsf{seed}_j^{(2,i)}$ for all $i \in [n]$ and $j \in [t] \setminus r$.
4. Simulate Π_{SH}^* by playing the role of all virtual parties. The virtual parties V_j^i for $i \in [n]$ and $j \in [t] \setminus r$ are executed as defined by Π_{SH}^* and with randomness seed_j^i and input x_j^i. The virtual parties V_r^i are assumed to behave correctly and are simulated by making them send exactly the same messages as in trans. Denote the transcript of the simulated protocol as trans'.
5. If $\mathsf{trans} \neq \mathsf{trans}'$, determine the virtual party V_l^m that has sent the first message which is different in trans and trans'. Set $\mathsf{cert} := (\mathsf{pk}_m, \mathsf{data}, d_l^{(1,m)}, d_l^m, \sigma_l^m)$ and output (m, cert).
6. Output $(0, \perp)$.

Judge algorithm of the input-dependent compiler. In the judge algorithm of the input-dependent compiler, $\mathsf{Judge}^{\mathsf{id}}$, parties accuse virtual parties instead of real party. If the judge detects misbehavior of a virtual party it announces the responsible real party to be corrupted. In case of inconsistency certificates, the judge behaves as the judge in the input-independent protocol but additionally receives and checks the commitment and opening of the virtual party's input share. In case of deviation certificates, the judge cannot simulate the whole protocol, as done in the input-independent compiler, as the judge cannot simulate the unopened virtual parties. Instead, the judge receives the ordered set of all messages received or sent by the accused virtual party. Then, the judge checks if the sent messages are correct based on the input share, the randomness and the received messages of the virtual party.

Algorithm $\mathsf{Judge}^{\mathsf{id}}(\mathsf{cert})$

Inconsistency certificate:
On input $\mathsf{cert} = (\mathsf{pk}_m, \mathsf{data}, \pi, r, \{(d_j^{(1,i)}, d_j^i)\}_{i\in[n],j\in[t]\setminus r}, \sigma_l^m)$ do:

- If $\mathsf{Verify}_{\mathsf{pk}_m}(\mathsf{data}; \sigma_l^m) = \perp$, output $\perp$.
- Parse data to $(m, l, \cdot, (c_l^{(1,m)}, c_l^m), p, \cdot)$.
- If $\mathsf{TL.Verify}(pp, p, (r, \{(d_j^{(1,i)}, d_j^i)\}_{i,j}), \pi) = 0$ output $\perp$.
- If $r = l$, output $\perp$.
- If $\mathsf{Open}(c_l^{(1,m)}, d_l^{(1,m)}) \neq \perp$ and $\mathsf{Open}(c_l^m, d_l^m) \neq \perp$, output $\perp$. Else output pk_m.

Deviation certificate:
On input $\mathsf{cert} = (\mathsf{pk}_m, \mathsf{data}, d_l^{(1,m)}, d_l^m, \sigma_l^m)$.

- If $\mathsf{Verify}_{\mathsf{pk}_m}(\mathsf{data}; \sigma_l^m) = \perp$, output $\perp$.

- Parse data to $(m, l, \mathsf{seed}_l^{(2,m)}, c_l^{(1,m)}, c_l^m, \cdot, \mathsf{trans}_l^m)$.
- Set $\mathsf{seed}_l^{(1,m)} \leftarrow \mathsf{Open}(c_l^{(1,m)}, d_l^{(1,m)})$ and $x_l^m \leftarrow \mathsf{Open}(c_l^m, d_l^m)$. If $\mathsf{seed}_l^{(1,m)} = \perp$ or $x_l^m = \perp$, output $\perp$.
- Set $\mathsf{seed}_l^m := \mathsf{seed}_l^{(1,m)} \oplus \mathsf{seed}_l^{(2,m)}$.
- Simulate the behavior of the virtual party $V_l^{'m}$ in Π_{SH}^*, by using x_l^m as input of V_l^m, seed_l^m as randomness of V_l^m, the messages in trans_l^m that have been received by V_l^m as incoming messages of V_l^m and by defining the outgoing messages of V_l^m according to Π_{SH}^*. Denote the resulting transcript of party V_l^m as trans'.
- If $\mathsf{trans}' = \mathsf{trans}_l^m$, output $\perp$. Otherwise, output pk_m.

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