Efficient Compiler to Covert Security with Public Verifiability for Honest Majority MPC

Thomas Attema[1,2,3], Vincent Dunning[1], Maarten Everts[4,5], and Peter Langenkamp[1]

[1] TNO, Cyber Security and Robustness, The Hague, The Netherlands
{vincent.dunning,thomas.attema,peter.langenkamp}@tno.nl
[2] CWI, Cryptology Group, Amsterdam, The Netherlands
[3] Leiden University, Mathematical Institute, Leiden, The Netherlands
[4] University of Twente, Services & Cyber Security, Enschede, The Netherlands
maarten.everts@utwente.nl
[5] Linksight, The Netherlands

Abstract. We present a novel compiler for transforming arbitrary, passively secure MPC protocols into efficient protocols with covert security and public verifiability in the honest majority setting. Our compiler works for protocols with any number of parties > 2 and treats the passively secure protocol in a black-box manner.

In multi-party computation (MPC), covert security provides an attractive trade-off between the security of actively secure protocols and the efficiency of passively secure protocols. In this security notion, honest parties are only required to detect an active attack with some constant probability, referred to as the *deterrence rate*. Extending covert security with *public verifiability* additionally ensures that any party, even an external one not participating in the protocol, is able to identify the cheaters if an active attack has been detected.

Recently, Faust et al. (EUROCRYPT 2021) and Scholl et al. (Pre-print 2021) introduced similar covert security compilers based on computationally expensive time-lock puzzles. At the cost of requiring an honest majority, our work avoids the use of time-lock puzzles completely. Instead, we adopt a much more efficient *publicly verifiable secret sharing* scheme to achieve a similar functionality. This obviates the need for a trusted setup and a general-purpose actively secure MPC protocol. We show that our computation and communication costs are orders of magnitude lower while achieving the same deterrence rate.

Keywords: Multi-Party Computation · Compiler · Covert Security · Honest Majority.

1 Introduction

Multi-party computation (MPC) is a subfield of cryptography allowing a set of mutually distrusting parties to jointly compute functions over their inputs without revealing anything but the outcome of the computation.

This way, nothing more can be deduced about the inputs of other parties than what could be deduced from the outcome of the computation alone. Traditionally, two types of adversaries have been considered in MPC; *passive* and *active* adversaries. Passive adversaries try to deduce as much private information as possible but follow the protocol honestly. Active adversaries are additionally allowed to arbitrarily deviate from the protocol, which might also compromise the correctness of the outcome. In general, passively secure protocols are fast but might not be considered secure in many realistic scenarios unless there is a good reason to assume that an untrusted party will not deviate from the protocol. Actively secure protocols are very secure in this regard, but active security comes at the cost of increasing the communication and computation complexity.

As a trade-off between the benefits of these two notions, *covert* security was introduced by Aumann and Lindell in 2007 [2]. Instead of safeguarding the protocol against an active attack, the idea of this notion is that it is sufficient to only detect the attack with a certain probability called the *deterrence rate* ϵ. Usually the deterrence rate can be chosen arbitrarily, thus providing a dynamic trade-off between the efficiency and security of passively and actively secure protocols, respectively. Goyal, Mohassel and Smith [13] presented a

covertly secure version of garbled-circuit based MPC protocols [4] and Damgard et al. [9] introduced a cheap cut-and-choose approach for an efficient and covertly secure offline phase for the SPDZ protocol [11], replacing costly zero-knowledge proofs required for active security. While this notion has led to promising results, in 2012 Asharov and Orlandi [1] observed that it might not be sufficient for practical applications. If a party detects a cheating attempt, there is in general no way of proving that another party has acted maliciously. Therefore they introduced the extended notion of *publicly verifiable* covert security. This property equips the parties with a mechanism to generate a *certificate* that proves a cheating attempt to *anyone*, including external parties not participating in the MPC protocol. Even though this notion looks promising for wider use in practice, relatively little research has been done in this area. The only concrete protocols in this security model have been presented in [1,16,15].

Another line of research is the trade-off between the number of corruptions a protocol can tolerate and efficiency, again giving up some security by tolerating less corruptions to achieve a more efficient protocol. A popular relaxation in literature is the assumption of an *honest majority*, meaning that more than half of the parties are guaranteed to behave honestly. Concrete protocols with active security and only sublinear overhead in the honest majority model have been presented in [14,6].

To ease the development of MPC with stronger security guarantees, *compilers* were introduced. Compilers allow for a modular approach to cryptographic protocol design; they provide a generic transformation from protocols with certain (security) properties to protocols with stronger properties. For instance, covert/active security compilers take as input a passively secure MPC protocol and output a protocol with covert/active security. The focus of this work will be on compiling passively secure protocols into efficient protocols with covert security and public verifiability for *any* passively secure protocol with an arbitrary number of parties $n > 2$.

Many MPC protocols proceed as follows. They first run an input independent pre-processing phase to set up some correlated randomness, e.g., Beaver triples [3]. Because this phase can be executed before the secret input values are available, it is also referred to as the offline phase. This pre-processing allows the actual computation, the online phase, to be executed very efficiently. Since actively secure online phases nowadays are quite efficient already, we specifically target our compiler towards the more expensive pre-processing protocols. As was proven in [10], combining a covertly secure pre-processing protocol with public verifiability and an actively secure online phase yields an overall protocol with covert security and public verifiability. Therefore, our compiler could for example be used to replace the actively secure pre-processing step of the SPDZ protocol with a covertly secure one from our compiler and combine it with the actively secure online phase of SPDZ [11] to improve the overall efficiency.

Typically, covert security is obtained by a cut-and-choose strategy where the passively secure protocol is simply executed multiple times after which some of these executions are "opened" to verify the behavior of the parties. An important predicament to overcome for public verifiability is the prevention of a *detection-dependent abort*. This means that an adversary should not be able to prevent the generation of a certificate once it sees its cheating attempt is going to be detected. The first covert security compiler *without* public verifiability was presented by Damgård, Geisler and Nielsen in 2010 [8]. Their approach is based on the assumption of an honest majority of participants. A covert security compiler *with* public verifiability, secure against *any* number of corruptions, was first presented by Damgård et al. in 2020 [10]. They presented two compilers in the 2-party case; one for input-independent protocols and one for input-dependent protocols. Furthermore, they sketch how to extend their approach to arbitrary numbers of parties. To prevent a detection-dependent abort, detecting active attacks is done by letting each party independently and obliviously choose which executions it wants to verify. To guarantee for a constant number of k executions that at least one execution remains closed, the number of executions that can be chosen by each party (and hence the deterrence rate) decreases for increasing numbers of parties. Concretely, each party can choose at most $\frac{k-1}{n}$ executions and thus obtains $\epsilon = \frac{k-1}{kn}$.

Constructions with a constant deterrence rate for any number of parties have been presented by Faust et al. [12] and concurrently by Scholl et al. [19]. Both works follow a *shared coin toss* (SCT) strategy. With this strategy, the parties together toss a coin to determine which executions will be verified by everyone guaranteeing maximal deterrence rates regardless of the amount of parties. To prevent a detection-dependent

abort, both [12] and [19] use time-lock puzzles (TLP) to lock the potential evidence before the coin toss such that the honest parties are guaranteed its availability in case the adversary aborts after seeing the outcome of the coin toss. A TLP hides a secret message and solving the TLP reveals this message. Moreover, solving a TLP is guaranteed to require a fixed amount of work. A TLP therefore guarantees that a message is hidden for a fixed amount of time and that it can be revealed after this fixed amount of time.

However, the time-locks introduce strict timing assumptions which introduce subtle issues in practice when used for this application. The entire security against a detection-dependent abort in these works relies on the assumption that the TLP is hidden for a few synchronous communication rounds. In theory, the synchronous communication model ensures that the parties communicate in fixed rounds through a global clock. In practice, this is typically realized by picking a certain *timeout* after which all messages for a round should have been received. With the TLP approach, if the amount of work required for solving the TLP is picked too low, an adversary has a higher probability of solving the TLP early and perform a detection-dependent abort. On the other hand, by picking a larger amount of work, the complexity for the honest parties to solve the TLP becomes undesirably high. The TLPs only need to be solved in case of misbehavior, so using an extremely complex puzzle could be acceptable to decrease the probability of the adversary solving the TLP too early. However, since we cannot make assumptions about the power of the adversary, it is still impossible to guarantee the security of the TLP and thus secrecy of the underlying message for a small number of communication rounds.

Furthermore, both TLP approaches require the availability of a general-purpose, actively secure MPC protocol to realize a trusted setup and implement an ideal functionality that constructs the TLP. This seems counterintuitive in a setting where the goal is to increase the security of a passively secure protocol through compilation. Furthermore, these functionalities prove to be very costly.

1.1 Contributions

In this work, we introduce a novel and efficient covert security compiler with public verifiability in the honest majority setting.

Our approach is based on the covert security compilers with public verifiability presented in [10,12,19]. We adapt their constructions and use a *publicly verifiable secret sharing scheme (PVSS)* to replace the costly time-lock puzzles (TLP). Compared to [10], our compilers yield much higher deterrence rates in the multi-party setting. This is achieved by following a *shared coin toss (SCT)* strategy, similar to the compilers of [12,19]. More precisely, for any number of executions of the passive protocol k, a deterrence rate of $1 - \frac{1}{k}$ can be achieved independent of the number of parties n. The public verifiability of the compilers of [12,19] is based on the use of TLPs to ensure availability of potential evidence after the coin toss. In contrast, we adopt a PVSS to distribute the evidence among all the parties. Due to the honest-majority assumption, the PVSS can be instantiated such that the adversary corrupting less than $n/2$ parties cannot reconstruct this secret evidence prematurely, while the honest parties are able to reconstruct. The prior works of [12,19] do however provide security against a dishonest majority.

With our adaptation, we remove the need for a trusted setup and an actively secure puzzle generation. We show that as a result, both the computation and communication complexity of our compiler decrease by multiple orders of magnitude. Moreover, an efficient and secure TLP instantiation for the purpose of achieving public verifiability, requires an accurate estimation of the adversary's computational resources. Therefore, in this application, it is inherently difficult to instantiate a TLP appropriately. For these reasons, our approach, avoiding TLPs altogether, provides security against a more realistic adversary model.

Our compiler makes black-box use of the passively secure protocol and can therefore enhance the security of *any* passively secure protocol, including future protocols. In [14] and [6], active security is obtained by adapting a specific secret-sharing based protocol and requires a stronger security notion than plain passive security. Therefore, these protocols are incomparable to our compiler.

1.2 Technical Overview

Covert Security. Covert security is obtained in a similar fashion to related constructions, where active cheating is usually detected by some cut-and-choose mechanism. More precisely, the passively secure protocol is executed k times after which $t < k$ executions are opened to verify the behavior of the parties. Opening an execution is done by revealing the randomness used by each party during an execution of the protocol. Note that in this work we are specifically targeting input-independent protocols and hence the behavior of a party is completely determined by the (publicly known) protocol description and the randomness used. Given the randomness of the other parties, each party can replay the protocol execution and verify the behavior of the other parties during the actual protocol execution. If no deviations are detected, the result of one of the unopened executions can then be picked as the output of the protocol. However, this approach still allows a dishonest party to decide which randomness to reveal *after* learning which executions are to be opened, i.e., there is no guarantee that the revealed randomness was used during the executions. To prevent this, the parties are required to commit to their randomness before the protocol execution. This technique was introduced by Hong et al. [15] and is also referred to as *derandomization*. After the k parallel executions, the parties perform a joint coin toss outputting an integer $1 \leq i \leq k$ indicating the protocol execution that is to be used as output. The remaining $k - 1$ executions are opened and the parties verify each other's behavior.

Public Verifiability. Public verifiability is obtained by making each party accountable for its messages by letting them *sign* all the messages they send during the protocol executions. If it is later detected that a party has sent an incorrect message, anyone can verify that this party must have sent the malicious message. It is essential to prevent a so-called *detection-dependent abort*, meaning an adversary cannot prevent the generation of a certificate once it sees it is going to be detected. To prevent this, we "lock" the randomness used by sharing it among all parties using a PVSS *before* the coin toss. If an adversary aborts after the coin toss, the parties have enough shares to reconstruct the randomness and verify behavior anyways. Here the honest majority assumption is required to guarantee enough honest shares for reconstructing each randomness while the adversary cannot get hold of enough shares to reconstruct the randomness used in the output execution.

Note that we can not simply accuse a party who aborts at this stage as there is no publicly verifiable evidence of this (such as a signature of a party on a malicious message). An external party who wishes to verify the protocol execution can not distinguish between an actual, active attack or an accident such as a network failure. Therefore, this straightforward approach could lead to an honest party unjustly being punished or is deniable by an adversary who can claim that he was not at fault.

Public Verifiability from PVSS. By using a PVSS, the parties are guaranteed to be able to proof that an active attack occurred. To this end the parties can first use the PVSS to verify all the randomness shares distributed by each party before the coin toss. In case the verification of some share fails, anyone can verify that the adversary attempted to cheat by distributing inconsistent shares. If the verifications succeed, the shares are guaranteed to reconstruct to a well-defined value, namely the randomness of the distributor. The distributor is furthermore committed to this randomness by a proof of correct distribution. This verification can be performed without interaction with the distributor or any of the other parties. Therefore, verification can also be done by external parties, making it publicly verifiable.

In case an adversary aborts after the coin toss, the parties can combine their shares to reconstruct the randomness of the adversary. During this reconstruction phase, each party is required to also publish a proof of correct decryption. This way, the honest parties can combine only correct shares to reconstruct the value originally distributed. Furthermore, an adversary cannot 'incriminate' an honest party by publishing a different share than distributed by the honest party. As an additional benefit of the PVSS strategy, the protocol can still continue and succeed in case an otherwise honest party is not able to deliver this information in time.

2 Preliminaries

Our compiler uses several building blocks. As cryptographic building blocks, the compiler uses a commitment scheme ($\texttt{Com}, \texttt{Open}$) and a signature scheme ($\texttt{Gen}, \texttt{Sign}, \texttt{Verify}$). Throughout this work, the commitment scheme is assumed to be non-interactive, but our compiler could trivially be instantiated with an interactive commitment scheme as well. Committing to a message m with randomness r will be denoted by $(c, d) \leftarrow \texttt{Com}(m; r)$, where c is the resulting commitment and $d = (m, r)$ the opening information. Opening a commitment is then denoted with $m' \leftarrow \texttt{Open}(c, d)$. For a correct opening, we get that $m' = m$ and $m' = \perp$ otherwise. The commitment scheme should satisfy the *hiding* and *binding* properties.

The signature scheme should be *existentially unforgeable against chosen message attacks*. Before the protocol execution, all parties are expected to generate a public-private key pair (pk, sk) using $\texttt{Gen}$ and register their public key. Signing a message m using a private key sk is denoted as $\sigma \leftarrow \texttt{Sign}_{sk}(m)$. Verifying a signature using the corresponding public key is denoted as $\texttt{accept}, \perp \leftarrow \texttt{Verify}_{pk}(m, \sigma)$.

2.1 Multi-Party Computation

The goal of Multi-Party Computation (MPC) protocols is to allow a group of n participants $\mathcal{P} = P_1, \ldots, P_n$ to compute a shared function f over their private inputs $x_1, \ldots, x_n$ while keeping their inputs hidden from each other. This group of participants can be divided in two sets: *honest* participants and *corrupt* participants. The honest participants will strictly follow the protocol description while corrupt participants are assumed to be under the influence of a central adversary.

In general for an MPC protocol to be considered secure, it needs to satisfy two requirements: *privacy* and *correctness*. Privacy means that and adversary is not able to learn more than what it can deduce from its own inputs and the output of the protocol. Particularly the adversary should not be able to gain any additional information about the inputs of the honest parties. Correctness means that the outcome of the protocol received by the honest parties should be correct. To reason about the security of such a protocol in the presence of an adversary, we follow the standard real/ideal world paradigm to show that our protocol in the real world is indistinguishable from an ideal execution of the same functionality. Informally, this paradigm specifies an ideal Functionality $\mathcal{F}$ and proofs that the MPC protocol Π implements exactly this ideal execution and is thus as secure as the ideal world.

In this work we assume that the adversary $\mathcal{A}$ with auxiliary input z can statically corrupt a set $\mathbb{A} \subset \mathcal{P}$ of the parties with $|\mathbb{A}| < \frac{n}{2}$. Furthermore, let $\Pi : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ be the real-world protocol computing functionality f taking one input per party $\{x_1, x_2, \ldots, x_n\} = \overline{x}$ and returning one output to each party. We define the outputs of the honest parties and $\mathcal{A}$ in a real-world execution of Π as $\texttt{REAL}_\lambda[\mathcal{A}(z), \mathbb{A}, \Pi, \overline{x}]$, where λ is the security parameter.

In the passive security model, an ideal world adversary $\mathcal{S}$ is assumed to try to deduce as much information as possible while honestly participating in the protocol. On the other hand, active adversaries may arbitrarily deviate from the protocol in order to try to deduce more information or break the correctness of the outcome.

Covert Security The idea of covert security is to assume an adversary who *is* capable of performing an active attack, but a certain probability of being caught cheating is enough to refrain him from doing so. This probability of being caught is called the *deterrence rate* ϵ.

In this work, we follow the strongest definition for covert security originally defined by Aumann and Lindell [2] called *strong explicit cheat (SECF)*. The ideal functionality for calculating a function f in the presence of covert adversaries according to this definition will be called $\mathcal{F}_{\text{covert}}$. This functionality allows S to perform cheating like an active adversary. With a probability of ϵ, $\mathcal{F}_{\text{covert}}$ informs all the parties of a cheating attempt. With a probability of $1 - \epsilon$, a cheating attempt is successful in which case S learns the inputs of all the parties and may decide their outputs. For readability of our protocols, we slightly alter the original SECF definition to not require *identifiable abort*. Due to the honest majority assumption this can, however, easily be obtained by adding a byzantine agreement at the end of the protocol. The formal definition of this ideal functionality can be found in $\mathcal{F}_{\text{covert}}$.

$\mathcal{F}_{\text{covert}}$ Ideal functionality for MPC with covert security

1. **Inputs:** Every honest party P_i sends its input x_i to $\mathcal{F}_{\text{covert}}$. The ideal world adversary $\mathcal{S}$ sends inputs on behalf of all the corrupted parties.
2. **Abort Options:** A corrupt party may send $(\texttt{abort}, i)$ or $(\texttt{corrupted}, i)$ as input to $\mathcal{F}_{\text{covert}}$. If $(\texttt{abort}, i)$ was received, $\mathcal{F}_{\text{covert}}$ will respond by sending $(\texttt{abort})$ to all the honest parties and halt. In case $(\texttt{corrupted}, i)$ was received, $\mathcal{F}_{\text{covert}}$ sends $(\texttt{corrupted}, i)$ to all the honest parties and halts. If multiple parties send `corrupted` or `abort`, $\mathcal{F}_{\text{covert}}$ informs the honest parties of only one of these events and halt. Furthermore, $(\texttt{corrupted}, i)$ is ignored in case it receives a combination of both events.
3. **Attempted cheat:** $\mathcal{S}$ can send $(\texttt{cheat}, i)$ as input of a corrupted party P_i to $\mathcal{F}_{\text{covert}}$. Now, $\mathcal{F}_{\text{covert}}$ will respond with $(\texttt{corrupted}, i)$ with a probability of ϵ to all the parties, corresponding to a detected cheating attempt. With a probability of $1 - \epsilon$, $\mathcal{F}_{\text{covert}}$ responds with `undetected` to the adversary. In case the cheating attempt was undetected, $\mathcal{S}$ gets all the inputs x_i of the honest parties P_i and specifies an output y_i for each of them which $\mathcal{F}_{\text{covert}}$ will output to P_i.

This is normally the end of the ideal execution. However, if no corrupted party sent $(\texttt{abort}, i), (\texttt{corrupted}, i)$ or $(\texttt{cheat}, i)$, the ideal execution continues with:

4. **Answer adversary:** The ideal functionality computes $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$ and sends it to $\mathcal{S}$.
5. **Answer honest parties:** $\mathcal{S}$ can now decide to either `continue` or $(\texttt{abort}, i)$ for a corrupted P_i. In case the adversary continues, the ideal functionality returns y_i to each honest P_i. In the case of $(\texttt{abort}, i)$, the ideal functionality relays this to all honest parties.
6. **Output:** Honest parties always output the message they receive from $\mathcal{F}_{\text{covert}}$ where the corrupted parties output nothing. The adversary outputs an arbitrary function of the initial inputs of the corrupted parties and the outputs received from $\mathcal{F}_{\text{covert}}$.

The joint distribution of the outputs of the honest parties and the ideal-world adversary $\mathcal{S}$ (with auxiliary input z) is denoted as $\texttt{IDEAL}^{\epsilon}_{\lambda}[\mathcal{S}(z), \mathbb{A}, \mathcal{F}_{\text{covert}}, \bar{x}]$. Covert security can now be defined as follows, where $\overset{\text{c}}{\equiv}$ denotes computationally indistinguishable:

Definition 1 (Covert security with deterrence rate ϵ). *A protocol Π securely computes $\mathcal{F}_{covert}$ with deterrence rate ϵ if for every real-world adversary $\mathcal{A}$, we can find an ideal-world adversary $\mathcal{S}$ such that for all security parameters $\lambda \in \mathbb{N}$:*

$$\{\mathit{IDEAL}^{\epsilon}_{\lambda}[\mathcal{S}(z), \mathbb{A}, \mathcal{F}_{covert}, \overline{x}]\}_{\overline{x}, k \in \{0,1\}^*} \overset{\text{c}}{\equiv} \{\mathit{REAL}_{\lambda}[\mathcal{A}(c), \mathbb{A}, \Pi, \overline{x}]\}_{\overline{x}, k \in \{0,1\}^*} \ .$$

Public Verifiability As an extension to covert security, the notion of *publicly verifiable* covert security (PVC) was proposed by Asharov and Orlandi in 2012 [1]. This form of security provides the parties with a mechanism to generate a publicly verifiable *certificate* in case cheating is detected. This certificate proves to *anyone* that a certain party attempted to cheat during the protocol.

We use the approach of [15] where a `Judge` algorithm is added to a real-world protocol Π. If, in the execution of Π, cheating is detected, the protocol outputs a certificate `cert`. The `Judge` algorithm verifies this certificate and outputs the public key (the "identity") of the cheater if it is valid. The vector of public keys is defined as $\bar{pk} = (pk^1, \ldots, pk^n)$, corresponding to the P_is. Furthermore, we have extracted the verification procedure of the protocol to a separate `Blame` algorithm. `Blame` takes the view of a party P_i, returns a certificate `cert` and outputs $\texttt{corrupted}_j$ in case party P_j is found to be cheating. Formally, we define covert security with public verifiability as:

Definition 2 (Covert security with deterrence rate ϵ and public verifiability). *A protocol $(\Pi, \mathit{Blame}, \mathit{Judge})$ securely computes $\mathcal{F}_{covert}$ with a deterrence rate of ϵ and public verifiability if the following three conditions hold:*

- ***Covert security:*** *Π is secure against a covert adversary according to Definition 1 for covert security with deterrence rate ϵ. Additionally, Π might now output* `cert` *in case cheating is detected.*
- ***Public Verifiability:*** *If an honest party P_i detects cheating by another party P_j and outputs* `cert` *in an execution of Π, then* $\texttt{Judge}(\bar{pk}, \mathcal{F}, \texttt{cert}) = pk^j$ *except with negligible probability.*
- ***Defamation-Freeness:*** *If party P_i is honest and executes Π in the presence of an adversary $\mathcal{A}$, then the probability that $\mathcal{A}$ creates* $\texttt{cert}^*$ *such that* $\texttt{Judge}(\bar{pk}, \mathcal{F}, \texttt{cert}^*) = pk^i$ *is negligible.*

2.2 Publicly Verifiable Secret Sharing

Verifiable secret sharing (VSS) [7] is an extension of regular secret-sharing that provides additional security against active attacks. VSS protects honest parties against malicious participants by equipping the secret sharing scheme with mechanisms to (i) verify that they received consistent shares from an untrusted dealer and (ii) verify that they received the correct shares from the other parties during reconstruction. With *publicly* verifiable secret sharing (PVSS) [21,18], properties (i) and (ii) can be verified by *anyone*, also parties outside the secret sharing protocol, without any interaction. In general, a PVSS can be instantiated from any secret sharing scheme with an arbitrary access structure $\mathcal{A}$. For this work, a threshold access structure such as realized with Shamir's secret sharing scheme [20] is sufficient. In this work, we require the PVSS to satisfy the definition first presented by Schoenmakers [18], which adds an additional proof of correct decryption:

Definition 3 (PVSS Scheme). *A PVSS scheme with a set of players $\mathcal{P}$ and access structure $\mathcal{A} \subseteq \mathcal{P}$ consists of the following three algorithms:*

- $(E_i(s_i)_{i\in\mathcal{P}}, \texttt{dproof}) \leftarrow \texttt{Distribute}(s)$*: The distribution algorithm takes as input a secret s and publishes a set of encrypted shares $E_i(s_i)_{i\in\mathcal{P}}$ and some public distribution proof* `dproof`*.*
- `true` *or* $\perp \leftarrow \texttt{Verify}(\texttt{dproof}, E_i(s_i))$*: The verification algorithm takes as input a distribution proof and an encrypted share $E_i(s_i)$ and outputs* `true` *if $E_i(s_i)$ encrypts a valid share s_i of s according to* `dproof`*.*
- $s' \leftarrow \texttt{Reconstruct}(\{\texttt{rproof}_i, s_i\}_{i\in A})$*: The reconstruction algorithm takes a set of decrypted shares $s_i, i \in A$ and corresponding decryption proofs of some subset $A \subseteq \mathcal{P}$ and outputs the reconstructed value s'. In case $A \in \mathcal{A}$, we call A a qualified subset and as a result, $s' = s$ if the verifications of the encrypted shares succeeded according to the proofs.*

Here, it is assumed that we already have a registered public key of all the participants. Instead of generating and distributing the secrets directly, a dealer publishes encrypted shares $E_j(s_j)$ with the known public keys of each party P_j. Furthermore, the dealer publishes a string `dproof` which shows that each E_j encrypts a consistent share s_j. This proof also commits the dealer to the value of the secret s and guarantees that no one can wrongly claim to have received a wrong share since anyone can verify this. In this work we will abuse notation and let $\texttt{Verify}(\texttt{dproof}, E_i(s_i)_{\mathrm{P}\in\mathcal{P}})$ denote the verification of all shares destined for the parties in P of the same secret s. Now, `true` is interpreted as all verifications succeeding while $\perp$ means at least one verification failed. If the reconstruction succeeds, we are guaranteed that this is the original secret s. During the reconstruction phase, the parties decrypt and publish their shares s_j from $E_j(s_j)$ along with a string $\texttt{rproof}_j$ which shows that they performed the decryption correctly. Using these, the other parties can now exclude the shares of participants who failed to decrypt correctly. If enough decryptions $(t+1)$ pass the verification, the parties can reconstruct the original secret successfully.

We require the PVSS to satisfy the *correctness*, *soundness* and *privacy* security guarantees.

Definition 4 (Correctness). *If a dealer honestly follows the* `Distribute` *algorithm to publish the encrypted shares $E_i(s_i)_{i\in\mathcal{P}}$ and a public proof* `dproof`*, then the outcome* $\texttt{Verify}(\texttt{dproof}, E_i(s_i))$ *is guaranteed to be* `true`*. Furthermore, if during reconstruction a party P_i honestly decrypts $E_i(s_i)$, publishes its share s_i and honestly generates the proof* $\texttt{rproof}_i$*, then another honest party receiving the decrypted share s_i and* $\texttt{rproof}_i$ *accepts this share. Finally, a qualified subset $\mathcal{A} \subseteq \mathcal{P}$ is guaranteed to reconstruct the original secret s if the dealer and the parties in $\mathcal{A}$ honestly follow the* `Distribute` *and* `Reconstruct` *protocols.*

Definition 5 (Soundness). *If* $\texttt{Verify}(\texttt{dproof}, E_i(s_i)) == \texttt{true}$*, then for all qualified subsets* $\mathcal{A}_1, \mathcal{A}_2 \subset \mathcal{P}$*, the following holds:*

$$\texttt{Reconstruct}(\{\texttt{rproof}_i, s_i\}_{i\in\mathcal{A}_1}) == \texttt{Reconstruct}(\{\texttt{rproof}_i, s_i\}_{i\in\mathcal{A}_2}).$$

Furthermore, if a malicious party submits a fake share during reconstruction, verification of this share fails with an overwhelming probability.

Definition 6 (Privacy). *An adversary corrupting a set of participants* $\mathcal{A}$ *such that* $|A| < t$ *should not be able to learn anything about the secret* s *from the shares* s_i *with* $i \in \mathcal{A}$*.*

3 Building Blocks

In this section, we will introduce the basic building blocks of our PVC compiler. The PVC compiler uses a public bulletin board and a public coin tossing functionality. Furthermore, our compiler slightly modifies the passively secure protocol execution.

Public Bulletin Board. For public communication required by the PVSS, we model an ideal functionality $\mathcal{F}_{\text{bb}}$, which represents a public bulletin board.

$\mathcal{F}_{\text{bb}}$ Ideal bulletin board functionality

- Consider a number of parties $P_1, P_2, \ldots, P_n$.
- If $\mathcal{F}_{\text{bb}}$ receives a message $(\texttt{send}, m, i)$ from P_i, it sends (m, i) to each party P_i for $1 \leq i \leq n$.

The public bulletin board functionality guarantees that the honest parties agree on all the messages that have been sent. In practice, this functionality could be realized using the *echo broadcast* protocol of [17].

$\mathcal{F}_{\text{coin}}$ Ideal coin-tossing functionality

- Consider a number of parties $P_1, P_2, \ldots, P_n$
- If $\mathcal{F}_{\text{coin}}$ receives a message $(\texttt{flip})$ from P_i, it stores $(\texttt{flip}, P_i)$ in memory if it is not stored in memory yet.
- Once $\mathcal{F}_{\text{coin}}$ has stored all the messages $(\texttt{flip}, P_i)$ for $i \in [n]$, $\mathcal{F}_{\text{coin}}$ picks a random value $r \in_R \{0,1\}^\lambda$ and sends $(\texttt{flip}, r)$ to all the parties.

Coin Tossing. An ideal functionality $\mathcal{F}_{\text{coin}}$ receives $\texttt{ok}_i$ from each party $P_i, i \in [n]$ and outputs a random λ-bit string r to all the parties. The adversary should not be able to influence the outcome of the coin-tossing protocol. Therefore, we require a coin-tossing protocol with security against an active adversary $\mathcal{A}$.

Passively Secure Protocol. The compiler presented in this work is designed to compile an arbitrary input-independent protocol Π_{pass} with passive security. Furthermore, we require the parties to agree on a public transcript that is the same in case of an honest execution, to compare to expected executions later on. To obtain such transcripts, we assume a fixed ordering in the messages and that every party can see each message sent during an execution of the protocol. In case Π_{pass} is secure against $n-1$ corruptions, we can simply broadcast every message since the adversary was allowed to see each message anyways. Otherwise, we need to keep the messages hidden by broadcasting symmetric-key encrypted messages instead, as presented

$\Pi_{\mathtt{seed}}$ Seed generation procedure

This protocol works with an arbitrary number n of parties $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$. To generate uniformly random seeds for every party, the parties execute the following steps:

1. Party P_i samples uniformly random a private seed $\mathtt{seed}^i_{\text{priv}}$, generates $(c^i, d^i) \leftarrow \mathtt{Com}(\mathtt{seed}^i_{\text{priv}})$ and sends c^i to the other parties.
2. For each $j \in [n]$, P_i samples a uniformly random public seed share $\mathtt{seed}^{(i,j)}_{(\text{pub})}$ and sends $\mathtt{seed}^{(i,j)}_{(\text{pub})}$ to all the other parties.
3. Each party calculates the public seeds $\mathtt{seed}^i_{\text{pub}}$ for each P_i as $\bigoplus_{j=0}^{n} \mathtt{seed}^{(i,j)}_{\text{pub}}$.
4. If the parties have not received all the expected messages before some predefined timeout, the parties send `abort` to all the other parties and output `abort`. Otherwise, P_i outputs $(\mathtt{seed}^i_{\text{priv}}, d^i, \{\mathtt{seed}^j_{\text{pub}}, c^j\}\}_{j\in[n]})$.

in [10]. To ease notation, we will assume Π_{pass} to be secure against $n-1$ corruptions but adding symmetric-key encryption for an arbitrary number of corruptions could be realized in a straightforward way by simply opening the keys in the execution opening protocol as well.

Remark 1. Note that an execution could also have been opened by letting the parties open an execution by revealing their entire view of the protocol, which consists of their randomness, all received messages and output. This way, we would remove the requirement of broadcasting every message and instead open all of this after the execution. However, in this work we chose to reduce the complexity of opening and verifying executions (possibly by outside parties) by having to open only the randomness used in the execution at the cost of a more complex protocol execution phase.

4 PVC Compiler

In this section we will present the main compiler Π_{comp} for transforming an arbitrary n-party MPC protocol Π_{pass} with passive security and no private inputs into an n-party MPC protocol with covert security and public verifiability. This compiler uses a commitment scheme, a signature scheme, a publicly-verifiable secret sharing scheme (PVSS) and an actively secure coin tossing protocol. We assume that every party already has registered a public key at the start of the protocol. Roughly speaking, Π_{comp} works in four separate phases: *seed generation*, *protocol execution*, *evidence creation* and *execution opening and verification*. In the seed generation phase, the parties set up k seeds from which they derive their randomness during the k executions of Π_{pass}. In the protocol execution phase, Π_{pass} is executed k times. In the evidence creation phase, the parties use the PVSS to secret share their seed openings to all the other parties and sign the information so that they can be held accountable later on. Finally, in the execution opening and verification phase, the parties toss a coin to select $k-1$ executions, open the randomness seeds for these $k-1$ executions and verify the behavior of the other parties. If no cheating is detected, the parties output their output in the unopened execution. Otherwise, they output the obtained certificate. Next, we will go over the four phases and finally present the complete compiler $\Pi_{\mathtt{comp}}$ as well as the additonal algorithms `Blame` and `Judge`.

Seed Generation. In order to guarantee covert security for *any* passively secure protocol Π_{pass}, we need to be guaranteed that the used randomness is picked uniformly at random. To achieve this, we run an actively secure seed generation procedure Π_{seed} for each of the executions of Π_{pass}. A formal description of this procedure can be found in Π_{seed}.

In the seed generation procedure, each party P_i picks a private seed $\mathtt{seed}^i_{\text{priv}}$ for itself and publicly commits to this seed. Together all the parties generate a public seed for each party P_i by first picking a public seed *share* and defining the public seed $\mathtt{seed}^i_{\text{pub}}$ for P_i as the sum of the shares of all the parties. During the executions of Π_{pass}, the parties derive randomness from a seed that is the XOR of the private and public seed, and is thus uniformly random.

Remark 2. In case Π_{pass} satisfies perfect correctness, the adversary cannot force the protocol to fail by picking randomness in a malicious way and thus the seed generation procedure can be ignored.

Protocol Execution In the protocol execution phase, the parties run the passively secure protocol k times in parallel and obtain an output y_j^i and transcript $\texttt{trans}_j^i$ for each of the executions. In these executions, every party sends each message to every other party and *signs* each message to hold them accountable.

Evidence Creation In the evidence creation phase, the parties are required to generate publicly verifiable, encrypted shares for the opening information of *all* of the k randomness seeds used:

$$(\{E_h(d_h)^{(i,j)}\}_{h\in[n]}, \texttt{dproof}_j^i) \leftarrow \text{PVSS.}\texttt{Distribute}(d_j^i).$$

and publicly broadcast these using $\mathcal{F}_{\text{bb}}$. This ensures availability of all the used randomness seeds after the coin toss for verification by the honest parties. In case the adversary aborts after seeing the coin toss, the honest parties can reconstruct its seeds using these shares, which are guaranteed to be correct if the PVSS verifications succeed. Furthermore, the parties sign the tuple:

$$\texttt{evidence}_j = (i, j, \{c_j^l, \texttt{dproof}_j^l\}_{l\in[n]}, \texttt{trans}_j)$$

With which they can be held accountable later on.

In the next sections, we will explain the subprotocols Π_{open}, $\Pi_{\text{reconstruct}}$ and `Blame` executed in the *execution opening and verification* phase.

4.1 Execution Opening

After executing k parallel instantiations of the passively secure MPC protocol, the parties will run Π_{open} to open the seeds used in $k-1$ of these executions. Before Π_{open} is executed, the parties have already published encrypted shares of the opening information of all of their seeds. In Π_{open}, the parties then verify these encrypted shares using the PVSS. If a verification fails, the parties generate a certificate and abort. If all verifications succeed, the parties jointly toss a coin to select the executions to open. At this point, it is too late for an adversary to abort since its seed openings have already been correctly distributed. Now, either the parties simply open all the seeds used in these executions (the optimistic case) or engage in $\Pi_{\text{reconstruct}}$ to reconstruct missing seeds (the pessimistic case). Note that we cannot simply indicate parties who fail to open their seeds as malicious since we are not able to generate a publicly verifiable certificate of this as an external judge is not able to distinguish between an active attack or an accidental abort. As an additional benefit, the PVSS strategy gives us a form of fault-tolerance. By being able to verify the executions in case of an abort, the protocol can still continue and succeed in case an otherwise honest party was accidentally not able to deliver this information in time.

4.2 Seed Reconstruction

If reconstructions are required, the parties engage in an execution of $\Pi_{\text{reconstruct}}$. This protocol starts by the parties announcing to everyone which seeds they are missing. For every missing message received, the parties decrypt their own share of the published share encryptions of the corresponding seed opening. This share together with a publicly verifiable proof of correct decryption is then published on the bulletin board. Using these proofs, these parties can then pool together $t+1$ shares of which the proofs are valid to reconstruct the correct seed opening. Due to the honest majority, we have that $t < \frac{n}{2}$, which guarantees that the honest parties can reconstruct missing seed openings in case an adversary refuses to distribute them. Finally the parties output a complete set of all seed openings $\mathcal{D}_i$. Note that in an honest execution of the PVC protocol, the parties already have a complete set after Π_{open} and thus $\Pi_{\text{reconstruct}}$ can be skipped. Using all the seeds, the parties can now verify the behavior of all the other parties in the opened executions. This procedure has been extracted to a separate `Blame` algorithm.

Π_{open} Protocol for opening a set of executions

At the start of the protocol, all the parties know the encrypted seed shares $\{E_h(d_h)^{(i,j)}\}$ of every party $P_i, i \in [n]$ in every execution $j \in [k]$ for every party $P_h, h \in [n]$ as well as the corresponding proofs $\texttt{dproof}_j^i$. Furthermore, the parties have the signatures σ_j^i together with corresponding evidence tuples $\texttt{evidence}_j$. Finally, each party P_i holds a set of private seed openings $\{d_1^i, d_2^i, \ldots, d_k^i\}$, a set of outputs $\{y_1^i, y_2^i, \ldots, y_k^i\}$ and a set of transcripts $\{\texttt{trans}_1^i, \texttt{trans}_2^i, \ldots, \texttt{trans}_k^i\}$. To open $k-1$ protocol executions, do the following:

Share verification:

1. First, the parties use the `Verify` algorithm of PVSS to check the validity of all the shares to generate the set:

$$M = \left\{(l,m) \in ([n],[k]) : \text{PVSS.}\texttt{Verify}(\texttt{dproof}_m^l, E_h(d_h)_{h\in[n]}^{(l,m)}) = \perp\right\}.$$

   If any of the parties obtain $M \neq \varnothing$, choose the tuple $(l,m) \in M$ with minimal l and m, calculate the certificate $\texttt{cert}_{\text{invs}} = \left(pk_l, \texttt{evidence}_m, E_j(d_j)_{j\in[n]}^{(l,m)}, \sigma_m^l\right)$ and output $\texttt{corrupted}_l$.

Joint coin tossing phase:

2. If all the verifications succeed, each party P_i sends $(\texttt{flip})$ to $\mathcal{F}_{\text{coin}}$, receives $(\texttt{flip}, r)$ and calculates the joint coin toss as $\texttt{coin} = r \mod k$.
3. Now, the parties exchange the set of seeds they have used in the $k-1$ executions according to the coin toss such that each party P_i obtains:

$$\mathcal{D}_i = \{d_j^h : h \in [n], j \in [k] \setminus \texttt{coin}\}$$

   Optimistic case: Each party P_i generates $\phi_j^i \leftarrow \texttt{Sign}(d_j^i)$ for all of its seed openings $\{d_j^i\}_{j\in[k]\setminus\texttt{coin}}$ and sends (ϕ_j^i, d_j^i) to all the other parties. Each party P_i verifies the signatures and constructs $\mathcal{D}_i$.
   Pessimistic case: If a number of parties P_j fails to publish their seed shares and/or valid signatures within a given amount of time, the parties engage in an execution of $\Pi_{\text{reconstruct}}$ to obtain $\mathcal{D}_i$.

Output:

4. Finally, each party outputs $(\mathcal{D}_i, \texttt{coin})$.

Protocol $\Pi_{\text{reconstruct}}$

At the start of the protocol, the encrypted seed openings, shares $\{E_h(d_h)^{(i,j)}\}$ of every party $P_i, i \in [n]$ in every execution $j \in [k]$ meant for every party $P_h, h \in [n]$ as well as the corresponding proof strings $\texttt{dproof}_j^i$ are publicly known. The parties recover the seed openings they are missing in the following way:

Missing seeds announcement:

1. Each party P_i starts with a (non-complete) set of seed openings $\mathcal{D}_i$. Assume P_i did not receive the seed openings d_m^l of some party P_l in some execution m. Call the set of tuples (l,m) of missing seed openings $\mathcal{E}_i$.
2. For every tuple $(l,m) \in \mathcal{E}_i$, P_i sends a message $\texttt{missing}_{(l,m)}^i$ to all the other parties.

Missing seed reconstruction:

3. For every $\texttt{missing}_{(l,m)}^j$ message received by P_i, P_i performs the following steps:
   - If $m == \texttt{coin}$, skip this message.
   - Otherwise, P_i decrypts its corresponding share d_i from $E_i(d_i)^{(l,m)}$, computes the string $\texttt{rproof}_{(l,m)}^i$ and sends $(\texttt{send}, (d_i, \texttt{rproof}_{(l,m)}^i), i)$ to $\mathcal{F}_{\text{bb}}$.
4. For every tuple $(l,m) \in \mathcal{E}_i$, P_i does the following:
   - For every message received from $\mathcal{F}_{\text{bb}}$ of the form $((d_j, \texttt{rproof}_{(l,m)}^j), j)$, P_i verifies the $\texttt{rproof}_{(l,m)}^j$.
   - Once $t+1$ of the received proofs are successfully verified, P_i reconstructs the seed opening d_m^l from the $t+1$ shares and adds this to $\mathcal{D}_i$.

Output:

5. Finally, P_i outputs the set of seed openings $\mathcal{D}_i$.

Algorithm `Blame(view)`

The `Blame` algorithm takes as input the view `view` of a party, which consists of:

- Public coin `coin`
- All the seed commitments and openings $\{c_j^i, d_j^i\}_{i\in[n],j\in[k]\setminus\texttt{coin}}$
- Encrypted seed shares $\{E_h(d_h)^{(i,j)}\}_{h,i\in[n],j\in[k]}$
- The set $\mathcal{E}$ of tuples of seed openings obtained via reconstruction
- PVSS proofs for distribution $\{\texttt{dproof}_j^i\}_{i\in[n],j\in[k]}$ and reconstruction $\{\texttt{rproof}_{(l,m)}^j\}_{j\in[n],(l,m)\in\mathcal{E}}$
- Public keys $\{pk_j\}_{j\in[n]}$, signatures $\{\sigma_j^i\}_{i\in[n],j\in[k]}$ and $\{\phi_j^i\}_{i\in[n],j\in[k]}$
- Additional information $\{\texttt{evidence}_j\}_{j\in[k]}$

To verify the behavior of the parties, do:

1. Open the private seeds of all the parties $P_i, i \in [n]$ in each execution $j \in [k] \setminus \texttt{coin}$ as $\texttt{seed}_{(j,\text{priv})}^i \leftarrow \texttt{Open}(c_j^i, d_j^i)$.
2. Construct the set $S = \{(l,m) \in ([n],[k]\setminus\texttt{coin}) : \texttt{seed}_{(l,\text{priv})}^i == \perp\}$. If S is not empty, pick the tuple (l,m) with the lowest l, m and produce an invalid opening certificate:
   - **If** $(l,m) \in \mathcal{E}$: set
     $\texttt{cert}_{\text{invo1}} = (pk_l, \texttt{evidence}_m, \{d_j, \texttt{rproof}_{(l,m)}^j\}_{j\in[n]}, \{E_j(d_j)^{(l,m)}\}_{j\in[n]}, \sigma_m^l)$.
   - **Otherwise**: set $\texttt{cert}_{\text{invo2}} = (pk_l, \texttt{evidence}_m, d_m^l, \phi_m^l, \sigma_m^l)$

   And output $(l, \texttt{cert}_{\text{invo}(1/2)})$.
3. If all the verifications succeeded, set $\texttt{seed}_j^i = \texttt{seed}_{(j,\text{priv})}^i \oplus \texttt{seed}_{(j,\text{pub})}^i$. as the randomness seed of each party P_i in each execution $j \in [k]$.
4. Re-run each execution j of Π_{pass} for $j \in [k] \setminus \texttt{coin}$ by simulating party P_i using random seed $\texttt{seed}_j^i$ to obtain each transcript $\texttt{trans}_j'$.
5. Using $\texttt{evidence}_j$, construct the set $S = \{m : \texttt{trans}_m \neq \texttt{trans}_m'\}$. If S is not empty, pick the lowest m and find the party P_l that sends the first message in $\texttt{trans}_m$ which is inconsistent with the expected message from $\texttt{trans}_m'$ and construct a protocol deviation certificate

$$\texttt{cert}_{\text{dev}} = (\text{pk}_l, \texttt{evidence}_m, \{d_m^i\}_{i\in[n]}, \sigma_m^l),$$

   and output $(l, \texttt{cert}_{\text{dev}})$. Otherwise, output $(\cdot, \perp)$.

4.3 Blame Algorithm

In the `Blame` algorithm, the behavior of the parties is verified and a certificate is generated in case cheating was detected. This `Blame` algorithm takes the view of a party as input. First, the `Blame` algorithm verifies the seed openings of all the parties. If the seed opening was obtained via reconstruction, an *invalid opening (1)* certificate is returned. In case the seed opening was given directly by the adversary, and *invalid opening (2)* certificate is generated. To ensure the parties agree on which party cheated, the one with the lowest party- and execution id is picked. If all the seeds can be opened correctly, the `Blame` algorithm simulates the executions using the randomness seeds obtained in the previous step, resulting in *expected* transcripts. If for any execution the actual transcript does not match with the expected transcript, the first party deviating from the protocol is identified and a *deviation certificate* is generated.

4.4 Complete Compiler

4.5 Judge Algorithm

The `Judge` algorithm takes a certificate and verifies it to confirm that the accused party actually cheated. If the verification succeeds, the public key of the cheater is output and otherwise $\perp$ is outputted. This algorithm does not require any communication with the parties and can thus be run by third parties as well. We assume the judge has access to the messages publicly stored via $\mathcal{F}_{\text{bb}}$. The judge performs a number of steps depending on the certificate type. If the certificate does not match any of the four templates, $\perp$ is returned. Regardless of which certificate type it receives, it first verifies the signature of the accused party on the `evidence`. If this signature is invalid, we can never be sure that the information was communicated by the accused party and thus $\perp$ is returned.

$\Pi_{\mathbf{comp}}$ Full compiler

Before the protocol execution, we assume the parties have agreed on the amount of executions k, protocol description Π_{pass} and the public keys of all the parties $\{pk_i\}_{i\in[n]}$. Furthermore, each party P_i knows its own secret key sk_i. Finally, the compiler assumes a publicly verifiable secret sharing scheme PVSS is available. Now, the passively secure protocol Π_{pass} is compiled into a protocol Π_{PVC} with covert security and public verifiability in the following way:

Seed generation:

1. For each $j \in [k]$, party P_i and all the other parties engage in an execution of $\Pi_{\texttt{seed}}$ to obtain:

$$\left(\texttt{seed}^i_{(j,\text{priv})}, d^i_j, \{\texttt{seed}^l_{(j,\text{pub})}, c^l_j\}\}_{l\in[n]}\right)$$

   And P_i computes its seeds for each execution j as $\texttt{seed}^i_j = \texttt{seed}^i_{(j,\text{priv})} \oplus \texttt{seed}^i_{(j,\text{pub})}$.

Protocol execution:

2. Next, all the parties engage in k executions of Π_{pass} where P_i uses the random seed $\texttt{seed}^i_j$ and obtains an output y^i_j and transcript $\texttt{trans}^i_j$ in each execution $j \in [k]$.

Evidence creation:

3. For each $d^i_j, j \in [k]$, P_i generates and publishes using $\mathcal{F}_{\text{bb}}$:

$$\left(\left\{E_h(d_h)^{(i,j)}\right\}_{h\in[n]}, \texttt{dproof}^i_j\right) \leftarrow \text{PVSS.}\texttt{Distribute}(d^i_j).$$

4. For each $j \in [k]$, party P_i creates a signature $\sigma^i_j \leftarrow \texttt{Sign}_{\text{sk}_i}(\texttt{evidence}_j)$ where $\texttt{evidence}_j$ is defined as:

$$\texttt{evidence}_j = \left(i, j, \{\texttt{seed}^l_{(j,\text{pub})}, c^l_j, \texttt{dproof}^l_j\}_{l\in[n]}, \texttt{trans}^i_j\right)$$

   P_i broadcasts all the σ^i_j's and verifies the received signatures.

Execution opening & verification:

5. Next, all the parties engage in an execution of the execution opening protocol Π_{open} such that each P_i obtains: $(\texttt{resp}, \texttt{coin}) \leftarrow \Pi_{\text{open}}$.
6. If $\texttt{resp} == \texttt{corrupted}_j$, P_i outputs $\texttt{corrupted}_j$.
7. Otherwise, P_i calculates $(l, \texttt{cert}) = \texttt{Blame}(\texttt{view}^i)$.
8. If $\texttt{cert} \neq \perp$, P_i broadcasts $\texttt{cert}$ and outputs $\texttt{corrupted}_l$. Otherwise, output $y^i_{\texttt{coin}}$.

4.6 Security

To prove that the compiler presented above satisfies Definition 2 for covert security with public verifiability, we first state the guarantees in Theorem 1 and then prove that our compiler satisfies the requirements of *covert security (with deterrence rate ϵ)*, *public verifiability* and *defamation-freeness* separately.

Theorem 1. *Suppose the PVSS (`Distribute`, `Verify`, `Reconstruct`) satisfies the privacy, correctness and soundness properties with a threshold $t < n/2$. Furthermore, assume the commitment scheme (`Com`, `Open`, `Verify`) is binding and hiding. Let the signature scheme (`Gen` , `Sign`, `Verify`) be existentially unforgeable under chosen plaintext attacks. Finally, assume Π_{coin} implements $\mathcal{F}_{coin}$ with active security. If Π_{pass} is passively secure, the compiler $\texttt{COMP}_{PVC} = (\Pi_{comp}, \Pi_{open}, \Pi_{reconstruct})$ with the additional algorithms `Blame` and `Judge` is covertly secure with public verifiability against $t < \frac{n}{2}$ corruptions with deterrence rate $\epsilon = 1 - \frac{1}{k}$.*

Intuitively, an adversary can try to cheat in a number of ways in the resulting protocol Π_{PVC}. First, it can do so by causing the seed openings of its own seeds to fail. This could be achieved by either (i) distributing

Algorithm $\texttt{Judge}(\texttt{cert})$

We assume the judge knows the function Π_{pass} to be computed. To check a certificate, do:

- If $\texttt{Verify}(\texttt{evidence}_m, \sigma_m^l) = \perp$, output $\perp$.
- Else, interpret $\texttt{evidence}_m$ as $\big(i, m, \{\texttt{seed}^l_{(m,\text{pub})}, c^l_m, \texttt{dproof}^l_m\}_{l\in[n]}, \texttt{trans}^i_m\big)$.

Depending on the type of certificate, do:

invs:
- $\texttt{cert}_{\text{invs}} = (pk_l, \texttt{evidence}_m, E_j(d_j)^{(l,m)}_{j\in[n]}, \sigma^l_m)$.
- If PVSS.$\texttt{Verify}(\texttt{dproof}^l_m, E_j(d_j)^{(l,m)}_{j\in[n]}) = \perp$, output pk_l. Otherwise, output $\perp$.

invo1:
- $\texttt{cert}_{\text{invo1}} = (pk_l, \texttt{evidence}_m, \{d_j, \texttt{rproof}^j_{(l,m)}\}_{j\in[n]}, \{E_j(d_j)^{(l,m)}\}_{j\in[n]}, \sigma^l_m)$.
- If PVSS.$\texttt{Verify}(\texttt{dproof}^l_m, E_j(d_j)^{(l,m)}_{j\in[n]}) = \perp$, output $\perp$.
- Verify $t+1$ of the $\texttt{rproof}^j_{(l,m)}$'s and use the corresponding d_j's to reconstruct d^l_m. If no $t+1$ valid shares are available, output $\perp$.
- If $\texttt{Open}(c^l_m, d^l_m) \neq \perp$, output $\perp$. Otherwise, output pk_l.

invo2:
- $\texttt{cert}_{\text{invo2}} = (pk_l, \texttt{evidence}_m, d^l_m, \phi^l_m, \sigma^l_m)$.
- If $\texttt{Verify}_{\text{pk}_l}(d^l_m, \phi^l_m) = \perp$, output $\perp$.
- If $\texttt{Open}(c^l_m, d^l_m) \neq \perp$, output $\perp$. Otherwise, output pk_l.

dev:
- $\texttt{cert}_{\text{dev}} = (\text{pk}_l, \texttt{evidence}_m, \{d^i_j\}_{i\in[n], j\in[k]\setminus\texttt{coin}}, \sigma^l_m)$.
- For every party P_i and execution m, open $\texttt{seed}^i_{(m,\text{priv})} \leftarrow \texttt{Open}(c^i_m, d^i_m)$ and calculate $\texttt{seed}^i_m = \texttt{seed}^i_{(m,\text{priv})} \oplus \texttt{seed}^i_{(m,\text{pub})}$.
- Re-run execution m of Π_{pass} by simulating each party P_i using random seed $\texttt{seed}^i_m$ to obtain transcript $\texttt{trans}'_m$.
- If $\texttt{trans}'_m == \texttt{trans}_m$, output $\perp$.
- If the first party that sends an incorrect message in $\texttt{trans}'_m$ is indeed P_l, output pk_l. Otherwise, output $\perp$.

Otherwise:
- If the certificate does not match any of the four formats, output $\perp$.

inconsistent shares in step 3 of Π_{comp} or (ii) sending an incorrect opening in step 3 of $\Pi_{\texttt{open}}$. Cheating strategy (i) is easily detected by the verification algorithm of the PVSS scheme, which anyone can verify. Furthermore, the proofs of correct decryption ensure that the adversary cannot announce a wrong share and the honest parties will always obtain the correct seed openings. Cheating strategy (ii) is noticed when any of the seed openings fail. In this case, the adversary has already published a signature on the commitment *and* on the opening which means anyone can see that the opening fails and the adversary must have sent this.

Furthermore, an adversary can attempt to cheat by deviating from the protocol description in any of the protocol executions. Since the protocol is run without private inputs, deviating means sending a message that is inconsistent with the protocol description and the committed randomness. If all of the seed openings succeeded, the parties can detect this when simulating the protocol executions later on. Since everyone knows the commitment and the opening, everyone knows the randomness that should have been used. Furthermore, the commitments to the seeds have been signed and thus an adversary cannot deny that he has sent an inconsistent message.

We have defined an ideal functionality for the coin flipping functionality. As can be seen in the description of $\mathcal{F}_{\text{coin}}$, all the parties receive the output from the ideal functionality simultaneously. This means that we require the real-world protocol with which this functionality is built to guarantee *fairness*. It has been proven in [5] that obtaining fairness is possible in the case of an honest majority. Therefore we will assume the existence of such a protocol Π_{coin} which securely implements $\mathcal{F}_{\text{coin}}$ with fairness in the presence of an honest majority.

The security guarantees given by our compiler have been formalized in Theorem 1.

Proof (Proof of Theorem 1).

Covert Security. To show that our compiler meets the definition for covert security with deterrence rate $\epsilon = 1 - \frac{1}{k}$, we will construct a simulator $\mathcal{S}$ in the ideal world, talking to the trusted party $\mathcal{F}_{\text{covert}}$ and the real-world adversary $\mathcal{A}$. After that, we will argue that the joint distribution of $\mathcal{S}$ and the output of $\mathcal{F}_{\text{covert}}$ is indistinguishable from the views of all the parties in the real-world execution of $\texttt{COMP}_{\text{PVC}}$. This proves that the `view` of the adversary in the real world can also be generated in the ideal world and thus the adversary is not able to learn more about the outputs of the honest parties than what is allowed.

Let the adversary $\mathcal{A}$ corrupt all the parties in some set $\mathbb{A}$ with $|\mathbb{A}| < \frac{n}{2}$. Furthermore, let $\mathbb{P} = [n] \setminus \mathbb{A}$ be the set of honest parties. The Simulator $\mathcal{S}$, simulating the honest parties, proceeds as follows:

0 For $P_i \in \mathbb{P}$, $\mathcal{S}$ generates a random pair (sk^i, pk^i) and sends all pk^i to $\mathcal{A}$ for $i \in \mathbb{P}$. $\mathcal{S}$ receives all pk^i for $i \in \mathbb{A}$ from $\mathcal{A}$.

1 For each $P_i \in \mathbb{P}$, $\mathcal{S}$ honestly engages in the k executions of Π_{pass}, receives $\texttt{seed}^i_j$ for $j \in [k]$ and obtains $\left(\texttt{seed}^i_{(j,\text{priv})}, d^i_j, \{\texttt{seed}^l_{(j,\text{pub})}, c^l_j\}\right)_{l\in[n]}$. If any of the expected messages from the adversary are missing, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\text{covert}}$ and $\mathcal{A}$ and halts.

2 $\mathcal{S}$ engages in k executions of Π_{pass} with $\mathcal{A}$ where for $i \in \mathbb{P}$, $\mathcal{S}$ uses randomness derived from $\texttt{seed}^i_j = \texttt{seed}^i_{(j,\text{priv})} \oplus \texttt{seed}^i_{(j,\text{pub})}$. Let $\texttt{trans}_j$ be the transcript obtained by $\mathcal{S}$ for execution $j \in [k]$. Let y^i_j be the output obtained by P_i in execution j.

3 Each party P_i distributes its seed openings d^i_j for execution $j \in [k]$ as:

$$(\{E_h(d_h)^{(i,j)}\}_{h\in[n]}, \texttt{dproof}^i_j) \leftarrow \text{PVSS.}\texttt{Distribute}(d^i_j)$$

. $\mathcal{S}$ does this honestly for $P_i \in \mathbb{P}$ while $\mathcal{A}$ does this for $P_i \in \mathbb{A}$.

4 $\mathcal{S}$ computes signatures σ^i_j for each $P_i \in \mathbb{P}$ and execution $j \in [k]$ as an honest party and sends these to $\mathcal{A}$. For each $i \in \mathbb{A}$, $\mathcal{S}$ receives σ^i_j from $\mathcal{A}$.

5 If any of the received signatures are invalid or $\mathcal{S}$ has not received the (expected) messages from some party P_i in any of the communication rounds, S sends $(\texttt{abort}, i)$ to $\mathcal{F}_{\text{covert}}$ and $\mathcal{A}$ and halts.

6 For each set $(\{E_h(d_h)^{(i,j)}\}_{h\in[n]}, \texttt{dproof}^i_j)$ with $i \in \mathbb{A}$, $\mathcal{S}$ uses PVSS.`Verify` to check whether the distributed shares are valid and if not:
- Send $(\texttt{corrupted}, i^*)$ to $\mathcal{F}_{\text{covert}}$ where i^* is the first party to distribute invalid shares.
- Compute an invalid sharing certificate like an honest party would do and send this to $\mathcal{A}$.
- Output whatever $\mathcal{A}$ outputs and stop the simulation.

7 $\mathcal{S}$ checks whether $\mathcal{A}$ has cheated in any of the execution in step 2. Let P_l be the first party to cheat in some execution m. Add all tuples (l, m) to a set M.

8 $\mathcal{S}$ decrypts all the shares $\{E_i(d_i)^{(l,j)}_{i\in\mathbb{P}}\}$ to reconstruct d^i_j for each party $P_i \in \mathbb{A}$ for each execution $j \in [k]$. For each of the pairs (c^i_j, d^i_j), if $\texttt{Open}(c^i_j, d^i_j) = \perp$ then add (i, j) to the set M as well if it was not in there yet.

9 With M as the set of all executions in which $\mathcal{A}$ cheated in some way, we distinguish three distinct cases:
- $|M| > 1$: In this case, cheating is guaranteed to be detected and $\mathcal{S}$ sets $\texttt{flag} == \texttt{detected}$.
- $|M| = 1$: In this case, $\mathcal{S}$ sends $(\texttt{cheat}, l)$ to $\mathcal{F}_{\text{covert}}$ and receives either `detected` or `undetected`.
  - In case `detected` was received, set $\texttt{flag} = \texttt{detected}$.
  - In case `undetected` was received, set $\texttt{flag} = \texttt{undetected}$.
- $|M| = 0$: In this case, set $\texttt{flag} = \texttt{all\_honest}$.

10 Depending on `flag`, do the following:

(a) If $\texttt{flag} == \texttt{detected}$, repeat the following steps:

1*. All parties send $(\texttt{flip})$ to $\mathcal{F}_{\text{coin}}$ and receive $(\texttt{flip}, r)$.

2*. All parties calculate $\texttt{coin}^* = r \mod k$.

3*. If $|M \setminus \texttt{coin}^*| > 0$, set $\texttt{coin} = \texttt{coin}^*$ and continue. Otherwise, rewind $\mathcal{A}$ to before step 1^* and try again.

(b) If $\texttt{flag} == \texttt{undetected}$, repeat the following steps:

1**. All parties send (flip) to $\mathcal{F}_{\text{coin}}$ and receive (flip, r).
2**. All parties calculate $\texttt{coin}^* = r \mod k$.
3**. If $\texttt{coin}^* \in M$, set $\texttt{coin} = \texttt{coin}^*$ and continue. Otherwise, rewind $\mathcal{A}$ to before step 1** and try again.

(c) If flag == all_honest, all parties send (flip) to $\mathcal{F}_{\text{coin}}$, receive (flip, r) and calculate $\texttt{coin} = r \mod k$.

11 For each execution $j \in [k] \setminus \texttt{coin}$, $\mathcal{S}$ computes $\phi_j^i \leftarrow \texttt{Sign}(d_j^i)$ for each $P_i \in \mathbb{P}$ and sends (ϕ_j^i, d_j^i) to $\mathcal{A}$. $\mathcal{S}$ receives (ϕ_j^i, d_j^i) for $P_i \in \mathbb{A}$ from $\mathcal{A}$.
- For every *valid* pair (ϕ_m^l, d_m^l) received, if $\texttt{Open}(c_m^l, d_m^l) \neq \perp$ and (l, m) is in M only because it was detected in step 8, remove (l, m) from M. If now $M = \varnothing$, set $\texttt{flag} = \texttt{all\_honest}$.
- For every *invalid* pair (ϕ_m^l, d_m^l) received for some honest party $P_i \in \mathbb{P}$, it sends $\texttt{missing}_{(l,m)}^i$ to $\mathcal{A}$.

If $\mathcal{S}$ receives a message $\texttt{missing}_{(l,m)}^i$ from $\mathcal{A}$ and $m \neq \texttt{coin}$, $\mathcal{S}$ sends decryptions of all the shares $\{E_i(d_i)_{i\in\mathbb{P}}^{(l,m)}\}$ and the corresponding proofs to $\mathcal{A}$.

12 Finally depending on flag, $\mathcal{S}$ does the following:

(a) If flag == detected:
- Send (corrupted, l) to $\mathcal{F}_{\text{covert}}$.
- Compute a certificate like an honest party would do and send this to $\mathcal{A}$.
- Output whatever $\mathcal{A}$ outputs and stop the simulation.

(b) If flag == undetected: Send $y_{\texttt{coin}}^i$ as the output of each P_i to $\mathcal{F}_{\text{covert}}$, outputs whatever $\mathcal{A}$ outputs and stop the simulation.

(c) If flag == all_honest: $\mathcal{S}$ receives outputs $(y_1, y_2, \ldots y_n)$ from $\mathcal{F}_{\text{covert}}$ for each party $P_i, i \in [n]$. Now, $\mathcal{S}$ rewinds $\mathcal{A}$ back to before step 2 and executes Step 13 until it terminates.

13 S picks a random $\texttt{coin}^* \in [k]$ and engages in k honest executions of Π_{pass} with $\mathcal{A}$. For each $P_i, i \in \mathbb{P}$ and each $j \in [k] \setminus \texttt{coin}^*$, $\mathcal{S}$ uses randomness derived from $seed_j^i$. For execution $\texttt{coin}^*$, $\mathcal{S}$ uses different randomness, namely randomness consistent with randomness seeds $\{\texttt{seed}_{\texttt{coin}^*}^i\}_{i\in\mathbb{A}}$ and outputs $(y_1, y_2, \ldots, y_n)$. $\mathcal{S}$ executes the other steps of the simulator as before, with the exception that in Step 10(c), $\mathcal{S}$ keeps rewinding $\mathcal{A}$ until the received $\texttt{coin} == \texttt{coin}^*$. Furthermore, if $\mathcal{A}$ aborts or misbehaves, $\mathcal{S}$ rewinds $\mathcal{A}$ back to the start of this step. Finally, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs and terminates.

We will now argument, through a series of hybrids, that the joint distribution of $\mathcal{A}$ and the honest parties in this ideal world is computationally indistinguishable from a real execution of the protocol in the $\mathcal{F}_{\text{coin}}$-hybrid model.

Hybrid 1 In this hybrid, we give $\mathcal{S}$ full access to the internal state of $\mathcal{F}_{\text{covert}}$ in order to decide the outputs of the honest parties. Furthermore, instead of sending (corrupted, i) to $\mathcal{F}_{\text{covert}}$, $\mathcal{S}$ now gets to send this directly to the honest parties. From the viewpoint of the honest parties, $\mathcal{S}$ behaves exactly like $\mathcal{F}_{\text{covert}}$ and thus the two hybrids are indistinguishable.

Hybrid 2 In Step 10, we no longer use $\mathcal{F}_{\text{coin}}$ but instead let $\mathcal{S}$ sample a uniformly random λ-bit string r itself and send this to $\mathcal{A}$. Indistinguishability follows from the uniformly randomness of $\mathcal{F}_{\text{coin}}$.

Hybrid 3 In this hybrid we combine Steps 9 and 10. Now, instead of asking $\mathcal{F}_{\text{covert}}$ to decide whether to set $\texttt{flag} == \texttt{detected}$ or $\texttt{flag} == \texttt{undetected}$ when $|M| = 1$, $\mathcal{S}$ sets the flag based on the outcome $\texttt{coin} = r \mod k$:

---

9 $\mathcal{S}$ receives (flip) from all parties samples $r \in_R \{0,1\}^\lambda$ and sends this to $\mathcal{A}$. All parties calculate $\texttt{coin} = r \mod k$. If $|M\ \texttt{coin}| > 0$, $\mathcal{S}$ sets $\texttt{flag} = \texttt{detected}$. Otherwise, set $\texttt{flag} = \texttt{all\_honest}$.

Let j^* be the single execution in which $\mathcal{A}$ cheated. Due to the uniform randomness of r, we get that $\texttt{coin} == j^*$ with probability $\frac{1}{k} = \epsilon$ and $\texttt{coin} \neq j^*$ with probability $1 - \frac{1}{k}$ and thus this hybrid is indistinguishable from Hybrid 2.

Hybrid 4 In this hybrid, we change Step 13 to no longer let $\mathcal{S}$ use different randomness than what was committed to in execution $\texttt{coin}^*$, but honestly use the committed randomness in each execution j. Due to the hiding property of the commitment scheme, this hybrid is indistinguishable from the previous hybrid.

Hybrid 5 In this hybrid, we change Step 13 to no longer rewind $\mathcal{A}$ to align `coin` and `coin`* but simply let $\mathcal{S}$ (which is now in control of $\mathcal{F}_{\text{covert}}$) give output y^i_{coin} as output of each $P_i, i \in [n]$ similar to the case of `flag` == `undetected`:

12(b) If `flag` == `undetected` or `flag` == `all_honest`: Send $y^i_{\texttt{coin}}$ as the output of the protocol to each $P_i, i \in \mathbb{P}$, output whatever $\mathcal{A}$ outputs and stop the simulation.

Indistinguishability from the previous hybrid follows from the passive security of Π_{pass}.

In Hybrid 5, $\mathcal{S}$ does not need rewinding nor access to the ideal functionality $\mathcal{F}_{\text{covert}}$ anymore and thus this hybrid corresponds to a real-world execution.

Public Verifiability. First, we prove that $\texttt{COMP}_{\text{PVC}}$ prevents a selective abort and after that we prove that the generated certificates will be accepted by the `Judge` with an overwhelming probability.

Note that an adversary *is* able to abort the protocol and hence the generation of a certificate *before* the coin-toss, but should be unable to prevent the generation of a certificate *after* it has seen the outcome of the coin-toss. To this end, observe that every party is asked to distribute the opening information for all of its seeds in Step 3 of Π_{comp} and sign the `data` that is required to create a certificate for every execution in $[k]$ in Step 4. Both of these happen before the coin-toss is even performed. At this point the adversary can thus not base its decision to cheat on the outcome of the coin toss. Cheating at this point means the adversary either distributes incorrect shares for (some of) its seed openings or distributes an incorrect signature for (some of) the `data`s. If any of these two happens, the other parties can detect it by the verifiability of the PVSS (Step 1 of Π_{open} and the signature scheme (Step 4 of Π_{comp}) before the coin toss. If a party distributes incorrect shares, this can be seen by *anyone* and thus public verifiability of this cheating attempt is easily obtained. If any of the signature verifications fail, the parties simply abort, which is accepted as explained earlier.

On the other hand, if verifications for all the shares for all of the seed openings succeeded, and valid signatures for all of the `data`s have been received by all the parties, the honest parties are guaranteed to be able to generate a certificate if they detect cheating. To see this, observe that Π_{coin} gives all the parties the outcome of the coin toss at the same time. After that, the parties either obtain the correct seed openings from all the other parties (i) directly (optimistic case) or (ii) can reconstruct the secret from the earlier distributed shares (pessimistic case). To see why (ii) holds, we look at the properties of the PVSS. Since the PVSS satisfies the *correctness* property, we are guaranteed that since the verification of the distributed shares succeeded, any reconstructed seed will always be the one that was originally distributed, except with negligible probability. Furthermore, this means that if a party correctly decrypts its share and publishes the corresponding proof, any honest party will accept this share. Due to the *soundness* property, we are also guaranteed that a subset of $t+1$ of valid shares is guaranteed be able to reconstruct the original secret. Since we assume an honest majority, we are guaranteed that at least $t+1$ parties successfully decrypt and publish a proof in Step (3) of $\Pi_{\text{reconstruct}}$. After that, we are guaranteed that a party missing a seed opening obtains at least $t+1$ valid shares and is thus able to reconstruct the original seed opening successfully. On the other hand, the adversary is unable to reconstruct the seed openings belonging to `coin` since this requires at least 1 share of an honest participant. A seed opening d^l_m and signed data σ^l_m are enough to create a certificate if party $P_l, l \in [n]$ is detected to have cheated in execution $m \in [k]$.

Now, it remains to show that if an honest party P_i outputs `cert` when it detects cheating by another party P_j, then the `Judge` outputs pk^j except with negligible probability. To prove this, we show it for the four types of certificate separately:

invs If an honest party outputs an invalid sharing certificate, this is because the PVSS.`Verify` method failed for some share for some seed opening of a party P_l for some execution m. Since the corresponding proof $\texttt{dproof}^l_m$ has been signed directly in σ^l_m and the PVSS is publicly verifiable, the `Judge` can check the validity of the signature and whether the verification indeed fails.

invo1 If an honest party outputs an invalid opening (1) certificate, this is because for some party P_l and some execution m, the opening information d^l_m received indirectly via the shares $\{E_j(d_j)^{(\text{l,m})}\}_{j\in[n]}$

is inconsistent with the commitment c_m^l. This c_m^l has been signed directly in σ_m^l while d_m^l has been obtained via the PVSS. Using the publicly verifiable $\texttt{dproof}_m^l$, the `Judge` can verify that the distribution was done correctly and due to the signature, P_l cannot claim to have distributed a different seed. Using the $\texttt{rproof}_{(\mathrm{l,m})}^j$s and the corresponding d_j for $j \in [n]$, the `Judge` can find at least $t+1$ valid shares, which are guaranteed to reconstruct the originally distributed d_m^l. Now, the `Judge` can simply verify that the opening information d_m^l is indeed inconsistent with c_m^l.

invo2 If an honest party outputs an invalid opening (2) certificate, this is because for some party P_l and some execution m, the opening information d_m^l received directly from P_l is inconsistent with the commitment c_m^l. This c_m^l has been signed directly in σ_m^l while d_m^l has been signed in ϕ_m^l. The `Judge` can simply check the validity of the signatures and whether c_m^l and d_m^l are indeed inconsistent.

dev Finally, if an honest party outputs a deviation certificate, this means that for some execution $m \in [k] \backslash \texttt{coin}$ of Π_{pass}, the honest party has detected cheating. Let P_l be the first party to send an inconsistent message in $\texttt{trans}_j$. The `Judge` can open all the private seed shares for this execution as $\texttt{seed}_{(m,\mathrm{priv})}^j \leftarrow \texttt{Open}(c_m^j, d_m^j)$ for $j \in [n]$ and compute the seeds that should have been used by all the parties as an honest party does. After that, the `Judge` can simulate the execution as well to obtain $\texttt{trans}'_m$. Now, $\texttt{trans}_j$ has been signed directly in σ_m^l and d_m^l either directly via ϕ_m^l or indirectly via σ_m^l (optimistic or pessimistic respectively). Therefore, if $\texttt{trans}_m == \texttt{trans}'_m$ *and* P_m is also the first party to send an inconsistent message in $\texttt{trans}'_m$, the `Judge` knows P_m must have cheated. Note that cheating in the input-independent setting simply means a party used randomness that was inconsistent with its randomness seed. Since in σ_m^l, party P_m signed the commitment to its own seed opening c_m^l, there is no way for P_m to somehow claim to have used a different randomness seed.

Defamation-freeness. Recall that in order for a protocol to have *defamation-freeness*, it should be impossible for an adversary to craft a certificate that incriminates an honest party successfully (i.e., such that the `Judge` accepts it), except with negligible probability. To prove that our compiler has this, we will show that *if* an adversary would be able to craft such a certificate incriminating an honest party P_i, this contradicts the security of either the commitment scheme, the PVSS or the signature scheme. We do this for the four types of certificates separately.

invs If the `Judge` accepts an invalid sharing certificate this means that for some seed opening d_m^i of an honest party P_i in execution $m \in [k]$,

$$\text{PVSS.}\texttt{Verify}(\texttt{dproof}_m^i, E_h(d_h)_{h\in[n]}^{(l,m)}) = \perp$$

. Since P_i would only honestly publish the proof as well as the encrypted shares, this means that the *correctness* of the PVSS should be broken which contradicts the security assumptions from theorem 1.

invo1 If the `Judge` accepts an invalid opening (1) certificate, this means that for some seed commitment c'^i_m sent by an honest P_i and opening d'^i_m reconstructed via the PVSS, $\texttt{Open}(c'^i_m, d'^i_m) = \perp$. Note that c'^i_m has been signed in σ'^i_m while d'^i_m has been obtained via the PVSS. An honest party P_i would only distribute the correct seed opening with the PVSS and a successful reconstruction should therefore always lead to the correct seed opening unless the *soundness* of the PVSS is broken. Furthermore, an honest party P_i only signs the commitment and openings that he received from the seed generation protocol. Therefore, either c'^i_m and d'^i_m must be correct or an adversary must be able to break the *existential unforgeability* of the signature scheme. Both of these contradict the security assumptions from theorem 1.

invo2 If the `Judge` accepts an invalid opening (2) certificate, this means that for some seed commitment c'^i_m and opening d'^i_m sent by an honest P_i, $\texttt{Open}(c'^i_m, d'^i_m) = \perp$. Note that c'^i_m has been signed in σ'^i_m while d'^i_m has been signed in ϕ'^i_m. Now, an honest party P_i only signs the commitment and openings that he received from the seed generation protocol. Therefore, in order for an adversary to make the opening fail, it must be able to break the *existential unforgeability* of the signature scheme, contradicting the security assumptions from theorem 1. Otherwise, c'^i_m and d'^i_m must be correct.

dev Finally if the `Judge` accepts a deviation certificate, this means that the certificate must contain a transcript $\texttt{trans}'_m$ for an execution m signed by P_i in σ_m^i where P_i is the first to send a message that

Step	Comp. Complexity
Distribution	$\frac{n^2+3n+4}{2} \cdot k$
Verification	$(\frac{n^2}{2} + 4n) \cdot (kn - k)$
Decryption	$3 \cdot m$
Reconstruction	$(4 \cdot n + \frac{n}{2}) \cdot e$

Table 1: Computation complexity as number of modular exponentiations.

Step	Comm. Complexity
Distribution	$k \cdot (\frac{n}{2} + 2n + 1)$
Opening	$2 \cdot k$
Reconstruction	$2 \cdot m$

Table 2: Communication complexity as number of field elements communicated per party.

is inconsistent with its randomness seed $\texttt{seed}^i_m$. However, since P_i is honest, he will follow the protocol honestly, this means that he will behave honestly in execution m and only sign the transcript honestly. If the signature σ^i_m on $\texttt{trans}'_m$ is valid but the transcript does blame P_i, this means that the adversary must be able to break the *existential unforgeability* of the signature scheme, contradicting the security assumptions. On the other hand, it could be that the transcript and the signature are valid, but that somehow the adversary can convince the `Judge` that another randomness seed $\texttt{seed}'^i_m \neq \texttt{seed}^i_m$ should have been used by P_i. The seed is obtained via the pair (c^i_m, d^i_m). Since these have also been signed in σ^i_m, this again means that the adversary must be able to forge a signature. Finally, the adversary could find another d'^i_m such that $\texttt{Open}(c^i_m, d'^i_m)$ is valid, but in that case the adversary has found two messages m and m' such that $\texttt{Com}(m) == \texttt{Com}(m')$, which means it must be able to break the *binding* property of the commitment scheme. All of this contradicts the security assumptions from theorem 1.

5 Computation and Communication Complexity

In this section, we analyze the computation and communication complexity of our compiler. For concreteness, we assume that the PVSS used for our compiler is the scheme presented by Schoenmakers [18], but stress that our compiler will work with any PVSS satisfying Definition 3. As our compiler simply executes the passively secure protocol k times while signing the messages, the computational complexity of the protocol execution phase is roughly k times the passively secure protocol. Note that the k executions are independent of each other and can therefore fully be executed in parallel, preserving the round complexity of the passively secure protocol. In terms of communication, each party needs to be able to see each message sent during the protocol execution. Therefore, the communication complexity of the compiler increases with a factor of $n-1$. Note that this is inherent to all currently known constructions for compilers in our setting [10,12,19].

The main difference in terms of complexity between our work and previous works lies in the execution opening and verification phase, where the goal is to open $k-1$ executions while preventing a detection-dependent abort. The total number of exponentiations required to distribute the seed openings of k executions with n parties and verify the distributed seeds of all the parties is given in Table 1. Furthermore, the number of exponentiations required for decryption and reconstruction in case of an aborting adversary is given as well. Here, m is the amount of missing messages in total while e is the amount of missing seeds of a single party. The total number of group elements communicated via $\mathcal{F}_{\text{bb}}$ in the execution opening phase of our protocol is given in Table 2. In an honest execution, every party uses the PVSS to distribute its seeds and then simply opens its seeds. If a party refuses to do this, for m distinct seeds missing, the parties need to publish their decrypted shares together with a proof of correct decryption.

5.1 Comparison with Prior Work

In contrast to our approach, the deterrence rate ϵ of [10] is inversely proportional to the number of parties n. For this reason, we focus on comparing our construction with the TLP approach of Faust et al.[12]. More specifically, we focus on comparing the execution opening and verification phase. In our case, this is realized by Π_{open} and possibly $\Pi_{\text{reconstruct}}$ while the work of [12] uses a maliciously secure TLP generation functionality for this.

Note that their puzzle generation does not include the solving of a TLP. The puzzle generation always has to be executed but the parties only need to solve a TLP in case of an abort. They presents an estimation for the total number of AND gates for the circuit of this puzzle generation functionality. This circuit has a linear complexity in the number of parties, while our seed distribution introduces a cubic computational complexity. However, the complexity of their functionality is dependent on the length of the RSA modulus N in the terms: $192|N|^3 + 112|N|^2 + 22|N|$. To illustrate the effects of both complexities, we present a concrete example. Take an honest execution of the protocol with $n = 5$, $t = 2$, $k = 2$ and thus $\epsilon = \frac{1}{2}$. With a security parameter of 128 bits, our approach costs approximately 10^8 bit operations while the circuit of [12] requires in the order of 10^{12} AND gates to be maliciously evaluated for an RSA modulus of 2048 bits.

In terms of communication complexity, our solution is linearly dependent on the number of parties and in the above scenario, the opening phase would require around 31 group elements to be communicated via $\mathcal{F}_{\text{bb}}$. Assuming $\mathcal{F}_{\text{bb}}$ is naively implemented using an echo-broadcast protocol, this would require each party to send $(n-1)^2 + 3n + 3$ messages per group element. In the above example, this would mean each party has to communicate around 8000 bytes with 64-bit messages. Instantiating [12] with the actively secure protocol of Yang et al. [22] requires 193 bytes per party per multiplication triple. This would thus require in the order of 10^{14} bytes to be communicated. Altogether, we expect our construction to outperform the earlier works in practical scenarios.

Acknowledgements The research activities that led to this result were funded by ABN AMRO, CWI, De Volksbank, Rabobank, TMNL, PPS-surcharge for Research and Innovation of the Dutch Ministry of Economic Affairs and Climate Policy, TNO's Appl.AI programme and the Vraaggestuurd Programma Cyber Security & Resilience, part of the Dutch Top Sector High Tech Systems and Materials program.

References

1. Asharov, G., Orlandi, C.: Calling out cheaters: Covert security with public verifiability. In: ASIACRYPT 2012. pp. 681–698 (2012)
2. Aumann, Y., Lindell, Y.: Security against covert adversaries: Efficient protocols for realistic adversaries. In: TCC 2007. pp. 137–156 (2007)
3. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: CRYPTO'91. pp. 420–432 (1992)
4. Beaver, D., Micali, S., Rogaway, P.: The round complexity of secure protocols (extended abstract). In: 22nd ACM STOC. pp. 503–513 (1990)
5. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In: 20th ACM STOC. pp. 1–10 (1988)
6. Boyle, E., Gilboa, N., Ishai, Y., Nof, A.: Efficient fully secure computation via distributed zero-knowledge proofs. In: ASIACRYPT 2020, Part III. pp. 244–276 (2020)
7. Chor, B., Goldwasser, S., Micali, S., Awerbuch, B.: Verifiable secret sharing and achieving simultaneity in the presence of faults (extended abstract). In: 26th FOCS. pp. 383–395 (1985)
8. Damgård, I., Geisler, M., Nielsen, J.B.: From passive to covert security at low cost. In: TCC 2010. pp. 128–145 (2010)
9. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In: ESORICS 2013. pp. 1–18 (2013)
10. Damgård, I., Orlandi, C., Simkin, M.: Black-box transformations from passive to covert security with public verifiability. In: CRYPTO 2020, Part II. pp. 647–676 (2020)
11. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: CRYPTO 2012. pp. 643–662 (2012)
12. Faust, S., Hazay, C., Kretzler, D., Schlosser, B.: Generic compiler for publicly verifiable covert multi-party computation. In: EUROCRYPT 2021, Part II. pp. 782–811 (2021)
13. Goyal, V., Mohassel, P., Smith, A.: Efficient two party and multi party computation against covert adversaries. In: EUROCRYPT 2008. pp. 289–306 (2008)
14. Goyal, V., Song, Y., Zhu, C.: Guaranteed output delivery comes free in honest majority MPC. In: CRYPTO 2020, Part II. pp. 618–646 (2020)
15. Hong, C., Katz, J., Kolesnikov, V., Lu, W., Wang, X.: Covert security with public verifiability: Faster, leaner, and simpler. In: EUROCRYPT 2019, Part III. pp. 97–121 (2019)

16. Kolesnikov, V., Malozemoff, A.J.: Public verifiability in the covert model (almost) for free. In: ASIACRYPT 2015, Part II. pp. 210–235 (2015)
17. Reiter, M.K.: Secure agreement protocols: Reliable and atomic group multicast in rampart. In: ACM CCS 94. pp. 68–80 (1994)
18. Schoenmakers, B.: A simple publicly verifiable secret sharing scheme and its application to electronic. In: CRYPTO'99. pp. 148–164 (1999)
19. Scholl, P., Simkin, M., Siniscalchi, L.: Multiparty computation with covert security and public verifiability. IACR Cryptol. ePrint Arch. p. 366 (2021)
20. Shamir, A.: How to share a secret. Communications of the Association for Computing Machinery pp. 612–613 (1979)
21. Stadler, M.: Publicly verifiable secret sharing. In: EUROCRYPT'96. pp. 190–199 (1996)
22. Yang, K., Wang, X., Zhang, J.: More efficient MPC from improved triple generation and authenticated garbling. In: ACM CCS 2020. pp. 1627–1646 (2020)