Black-Box Transformations from Passive to Covert Security with Public Verifiability

Ivan Damgård, Claudio Orlandi, and Mark Simkin

Aarhus University, Aarhus, Denmark
{ivan, orlandi, simkin}@cs.au.dk

Abstract. In the context of secure computation, protocols with security against covert adversaries ensure that any misbehavior by malicious parties will be detected by the honest parties with some constant probability. As such, these protocols provide better security guarantees than passively secure protocols and, moreover, are easier to construct than protocols with full security against active adversaries. Protocols that, upon detecting a cheating attempt, allow the honest parties to compute a certificate that enables third parties to verify whether an accused party misbehaved or not are called publicly verifiable.
In this work, we present the first generic compilers for constructing two-party protocols with covert security and public verifiability from protocols with passive security. Our main compiler uses oblivious transfer to transform arbitrary passively secure protocols in a fully blackbox manner. It incurs only a constant multiplicative factor in terms of bandwidth overhead and a constant additive factor in terms of round complexity on top of the passively secure protocol it uses.
As a building block for our main result, we construct another compiler that only works for passively secure protocols that have no private inputs. Apart from being an important stepping stone towards our main result, this compiler may also be of independent interest, since it allows for compiling preprocessing protocols that are used to setup correlated randomness. In particular, we show how to use this compiler to construct the first secret-sharing based two-party protocol with covert security and public verifiability. Notably, the produced protocol achieves public verifiability essentially for free when compared with the best known previous solutions based on secret-sharing that did not provide public verifiability
Finally, we sketch how to extend our techniques to obtain multiparty computation protocols with covert security and public verifiability against arbitrary constant fractions of corruptions.

1 Introduction

In secure computation two or more parties want to compute a joint function of their private inputs, while revealing nothing beyond what is already revealed by the output itself. Privacy of the inputs and correctness of the output should be maintained, even if some of the parties are corrupted by an adversary. Historically, this adversary has mostly been assumed to be either passive or active. Passive adversaries observe corrupted parties, learn their private inputs, the random coins they use, and see all messages that are being sent or received by them. Active adversaries take full control of the corrupted parties, they can deviate from the protocol description in an arbitrary fashion, or may just stop sending messages altogether. Protocols that are secure against active adversaries provide very strong security guarantees. They ensure that any deviation from the protocol description by the corrupted parties is detected by the honest parties with an overwhelming probability. Unfortunately, such strong security guarantees do not come for free and actively secure protocols are typically much slower than their passively secure counterparts.

To provide a compromise between efficiency and security, Aumann and Lindell [AL07] introduced the notion of security against *covert* adversaries[1]. Loosely speaking, the adversary still has full control of the corrupted parties, but if any of them deviates from the protocol description, then this behavior will be

[1] In the remainder of the paper we will use the terms "covert security" and "security against covert adversaries" interchangeably to refer to the notion that was defined by Aumann and Lindell. We note, however, that the term "covert security" has also been used to denote a different flavor of secure computation as defined in [vHL05].

detected with some constant probability, say 1/2, by all honest parties. The main rationale for why such an adversarial model may be sensible in the real world, is that in certain scenarios the loss of reputation that comes from being caught cheating outweighs the gains that come from cheating successfully. Consider, for example, a large company that performs secure computations with its customers on a regular basis. It is reasonable to assume that the company's general reputation is more valuable than whatever it could possibly earn by tricking a few of its customers.

Asharov and Orlandi [AO12] observed that despite being well-motivated in practice, the original notion of security against covert adversaries may be a bit too weak. More concretely, the original notion ensures that the honest parties detect cheating with some constant probability, but it does not ensure the existence of a mechanism for convincing third parties that the adversary really cheated. For our hypothetical large company from before, this means that no cheated customer could convince the others of the company's misbehavior. Asharov and Orlandi therefore introduce the stronger notion of covert security *with public verifiability*, which, in case of detected cheating, ensures that the honest parties can compute a publicly verifiable certificate, which allows third parties to check that cheating by some accused party indeed happened.

Although covert security with and without public verifiability seems like a very natural security notion, comparatively few works focus on this security model. Goyal, Mohassel, and Smith [GMS08] present covertly secure two and multiparty protocols without public verifiability based on garbled circuits [Yao82] and its multiparty extension the BMR protocol [BMR90]. Subsequently, Damgård et al. [DKL$^+$13] present a preprocessing protocol with covert security without public verifiability for SPDZ [DPSZ12]. Asharov and Orlandi [AO12] present a two-party protocol with covert security and public verifiability based on garbled circuits and a flavor of oblivious transfer (OT), which they call signed-OT. Kolesnikov and Malozemoff [KM15] improve upon the construction of Asharov and Orlandi by constructing a signed-OT extension protocol based on the OT extension[2] protocol of Ishai et al. [IKNP03]. In a recent work by Hong et al. [HKK$^+$19], the authors present a new approach for constructing two-party computation with covert security and public verifiability based on garbled circuits from plain standard OT.

Apart from the concrete constructions above, one can also use so called protocol compilers that generically transform protocols with weaker security guarantees into protocols with stronger ones. The main advantage of compilers over concrete protocols is that they allow us to automatically obtain protocols with the stronger security guarantee from any future insight into protocols with the weaker one. In case of covert security with and without public verifiability, for example, most of the existing concrete constructions are based on garbled circuits. If at some point in the future a new methodology for constructing more efficient passively secure protocols is discovered, then we may end up in the situation that the techniques that we used to lift garbled circuits from passive to covert security may not be applicable. Compilers, on the other hand, will still be useful as long as the new protocols satisfy the requirements the compiler imposes on the protocols it transforms.

In terms of generic approaches for efficiently transforming arbitrary passively secure protocols into covertly secure ones very little is known. Damgård, Geisler, and Nielsen [DGN10] present a blackbox compiler that transforms passively secure protocols that are based on secret sharing into covertly secure protocols. Their compiler only works for the honest majority setting, i.e., it assumes that the honest parties form a strict majority, and therefore their compiler is not applicable to the two-party setting. Lindell, Oxman, and Pinkas [LOP11] present a compiler, based on the work of Ishai, Prabhakaran, and Sahai [IPS08], that transforms passively secure protocols in the dishonest majority setting into covertly secure ones. Their compiler makes blackbox use of a passively secure "inner" and non-blackbox use of an information-theoretic "outer" multiparty computation protocol with active security. The bandwidth overhead and round complexity of their compiler depends on the complexity of both the inner and the outer protocol. Unfortunately this means that the protocols this compiler produces are either not constant-round protocols or have a large bandwidth overhead. Using an information-theoretic outer protocol results in a protocol that is not constant-round, since constructing information-theoretic multiparty computation with a constant number of rounds is a long

[2] In general, OT requires the use of public key cryptography. OT extension protocols allow two parties to perform large numbers of OT protocol executions using only a small number of public key operations.

standing open problem.[3] Alternatively, the authors of [IPS08] show how to combine their compiler with a variant of a computationally secure protocol of Damgård and Ishai [DI05], which only makes blackbox use of pseudorandom generators, as the outer protocol. This approach, however, results in a protocol, which incurs a bandwidth overhead that is multiplicative in the security parameter and the circuit size on top of the communication costs of the underlying passively secure protocol.

None of the above compilers is publicly verifiable and, more generally, there is currently no better approach for constructing covertly secure protocols with public verifiability in a generic way than just taking a compiler that already produces actively secure protocols.[4] Even without public verifiability, there is currently no compiler for the dishonest majority setting that is fully blackbox in the sense that the code of the used secure computation protocols does not need to be known.

1.1 Our Contribution

In this work, we present the first generic compilers for transforming protocols with passive security into two-party protocols with covert security and public verifiability. Our compilers are conceptually simple and only make blackbox use of the underlying passively secure protocols and oblivious transfer.

Our main compiler is the first one that is simultaneously blackbox, increases the round complexity of the underlying protocol by only an additive constant factor, and incurs a constant multiplicative factor bandwidth overhead on top of the underlying passively secure protocol. It is the first compiler that generically produces protocols that have public verifiability.

As a building block for our main result, we introduce another blackbox compiler for protocols that have no private inputs. Apart from being an important stepping stone towards our main result, this compiler may also be of independent interest as it can be used to transformed preprocessing protocols, which are commonly used to setup correlated randomness among the parties. For example, one can combine this compiler with a suitable preprocessing protocol and the actively secure SPDZ online phase, similarly to [DKL$^+$13], to obtain the first protocol with covert security, public verifiability. The resulting protocol, somewhat surprisingly, achieves public verifiability essentially for free. That is, the efficiency of our resulting protocol is essentially the same as the efficiency of the best known secret-sharing based protocol for covert security without public verifiability [DKL$^+$13].

Lastly, we sketch how to extend our compilers to the multiparty setting. The resulting protocols are secure against an arbitrary constant fraction of corruptions.

1.2 Technical Overview

Before presenting the main ideas behind our compilers, let us first revisit the main ideas as well as the main technical challenges in existing protocols. Generally speaking, most covertly secure two-party protocols [AL07, GMS08, AO12, HKK$^+$19] follow the same blueprint. They all start from a passively secure protocol, which they run k times in parallel. They use $k-1$ randomly chosen executions to check the behavior of the participating parties and then use the last unopened protocol execution to actually compute the desired functionality on their private inputs. The intuition behind these approaches is that, if cheating happened, it will be detected with a probability of $1-1/k$, since the adversary would need to guess which execution will remain unopened. Importantly, this blueprint relies on the ability to open $k-1$ executions "late enough" to ensure that cheating in the unopened execution is not possible afterwards any more, while at the same time being able to open the checked executions "early enough" to ensure that no private inputs are leaked. Secure two- and multiparty computation protocols based on garbled circuits are a perfect match for the blueprint described above, since they consist of an input-independent garbling phase, and an actively secure evaluation phase. Checked executions are opened after circuit garbling, but before circuit evaluation.

[3] Note that the existing works on constant round MPC with information theoretic security [IK00, ABT18, ACGJ19] only apply to circuits that are in NC^1.

[4] Observe that in the two-party case active security implies covert security with public verifiability, since every attempt to cheat can simply be interpreted as an abort.

Unfortunately, however, it is not clear how to generalize this approach to arbitrary passively secure protocols, since it crucially relies on the concrete structure of garbled circuit based protocols.

Our first compiler focuses on a restricted class of two-party protocols, namely those that have no private inputs. Apart from being a good starting point for explaining some of the technical ideas behind our compiler for arbitrary protocols, this class also covers a large range of important protocols. Specifically, it includes so called preprocessing protocols that are used for setting up correlated randomness between the parties.

Not having to deal with private inputs, immediately suggests the following high-level approach for letting Alice and Bob compute some desired function with covert security: Both parties first jointly execute a given protocol Π with passive security k times in parallel. Once these executions are finished, Alice and Bob independently announce subsets $I^A \subset [k]$ and $I^B \subset [k]$ with $|I^A| = |I^B| < k/2$. Alice reveals the randomness used in each execution $i \in I^B$ and Bob does the same for all executions with an index in I^A. Knowing these random tapes each party can verify whether the other party behaved honestly in the checked executions. Since $I^A \cup I^B \subsetneq [k]$, the parties are guaranteed that there exists an execution that is not checked by either of the parties. If no cheating in any of the checked executions is detected, then both parties agree to accept the output of one of the unopened executions.

This straightforward approach works for the plain version of covert security, but fails to achieve public verifiability, since neither of the parties has any way of convincing a third party of a detected cheating attempt. Even if each party signs each message it sends, a cheating party might simply stop responding if it does not like which executions are being checked. To achieve public verifiability, rather than asking for I^A and I^B in the clear, both parties use oblivious transfer to obtain the random tapes that correspond to the executions they would like to check. As before, we run k copies of the passively secure protocol Π, where Alice and Bob use the random tapes they input to the OT protocol. Both parties sign the complete transcript of messages exchanged during the executions of Π as well as the transcript of the OT. If, for instance, Alice detects cheating by Bob in some execution, then she can publish the signed transcripts of both the OT protocol and the protocol in which Bob cheated, together with the randomness she used in the OT protocol. Any third party can now use Alice's opened random tape in combination with the signed transcript of the OT protocol to recover Bob's random tape and use it to check whether Bob misbehaved or not in the protocol Π. The general idea of derandomizing the parties to achieve public verifiability has previously been used by Hong et al. [HKK+19]. However, as it turns out there are quite a few subtleties to take care of to make sure that the approach outlined above actually works and is secure. We will elaborate on these challenges in the later technical sections.

Using our compiler for protocols with no private inputs, we can obtain efficient protocols with covert security and public verifiability in the preprocessing model. In this model, protocols are split into a preprocessing protocol Π_{OFF} and an online phase Π_{ON}, where Π_{OFF} generates correlated randomness and Π_{ON} is a highly efficient protocol for computing a desired function using the correlated randomness and the parties' private inputs. We can apply our compiler to a passively secure version of the preprocessing protocol of SPDZ [DPSZ12] and combine it with an actively secure online protocol Π_{ON} to obtain an overall protocol with covert security and public verifiability.

Our main compiler for arbitrary protocols follows the player virtualization paradigm, which was first introduced by Bracha [Bra87] in the context of distributed computing and then first applied to secure computation protocols by Maurer and Hirth [HM00]. Very roughly speaking, the idea behind this paradigm is to let a set of real parties simulate a set of virtual parties, which execute some given protocol on behalf of the real parties. Despite the conceptual simplicity of this idea, it has led to many interesting results. Ishai, Prabhakaran, and Sahai [IPS08], for example, show how to use this paradigm in combination with OT to obtain actively secure protocols from passively secure ones in the dishonest majority setting. In another work, Ishai et al. [IKOS07] show how to transform secure multiparty computation protocols with passive security into zero-knowledge proofs. Cohen et al. [CDI+13] show how to transform three or four-party protocols that tolerate one active corruption into n-party protocols for arbitrary n that tolerate a constant fraction of active corruptions. Damgård, Orlandi, and Simkin [DOS18] show how to transform information-theoretically secure multiparty protocols that tolerate $\Omega(t)$ passive corruptions into information-theoretically secure ones that tolerate $\Omega(\sqrt{t})$ active corruptions.

In this work, we make use of this paradigm as follows: Assume Alice and Bob have inputs x^A and x^B, would like to compute some function f, and are given access to some $2m$-party protocol Π with security against $m + t$ passive corruptions, where m and t are parameters that will determine the probability with which cheating will be caught. Alice imagines m virtual parties $\mathbb{V}_1^A, \ldots, \mathbb{V}_m^A$ in her head and Bob imagines virtual parties $\mathbb{V}_1^B, \ldots, \mathbb{V}_m^B$ in his. Alice splits her input x^A into an m-out-of-m secret sharing with shares $x_1^A, \ldots, x_m^A$ and she will use share x_i^A as the private input of her virtual party $\mathbb{V}_i^A$. Bob does the same with his input. All $2m$ virtual parties jointly execute Π, which first reconstructs x^A and x^B from the given shares and then computes $f(x^A, x^B)$. During the protocol execution, Alice and Bob send messages on behalf of the virtual parties they simulate. If both parties perform their simulations honestly, then the protocol computes the desired result. If either of the real parties misbehaves, then it will necessarily misbehave in at least one of its virtual parties. Similarly to before, our idea here is to let Alice and Bob check subsets of each other's simulations. Assuming, for instance, Alice obtains the random tapes and inputs of t uniformly random virtual parties of Bob, then she can recompute all messages that those virtual parties should be sending. Since Bob does not know which of his virtual parties are checked, any attempt to cheat will be caught with a probability of t/m. Further, observe that as long as $t < m$, the inputs remain hidden, since the protocol tolerates $m + t$ corruptions and each input is m-out-of-m secret shared.

As in the case of the first compiler, the high-level idea of the simulation strategy above is reasonably simple, yet the details are in fact non-trivial and several subtle issues arise when trying to turn this idea into a working compiler.

What function to compute? The first question that needs to be addressed is that of which function exactly the virtual parties should compute. Using Π to reconstruct x^A and x^B and then directly compute $f(x^A, x^B)$ is good enough for the intuition above, but is actually not secure. The reason is, that security against covert adversaries requires that the adversary's decision to cheat is independent of the honest party's input and output. A passively secure protocol, however, may reveal the output bit-by-bit, which would allow the adversary to learn parts of the output before deciding on whether to cheat or not. Consider for example the case, where x^A and x^B are bit-strings and $f(x^A, x^B) = x^A \wedge x^B$. A passively secure protocol may simply compute and output the AND of each bit sequentially, which would enable an adversary to make its decision to cheat dependent on the output bits it has seen so far. To deal with this issue, we will use Π to secret share the output value among all virtual parties. This ensures that Π will not leak any information about any of the inputs or the output. At the same time all virtual parties can still reconstruct the output together by simply publishing their respective share.

How to check the behavior of virtual parties? The behavior of any virtual party is uniquely defined by its input, its random tape, and its current view of the protocol. Assuming, for example, Alice has access to input, random tape, and current view of some virtual party $\mathbb{V}_i^B$ of Bob, she can recompute the exact message this party should be sending. If the virtual party $\mathbb{V}_i^B$ deviates from that message, then Alice knows that a cheating attempt has happened. Assume for a second, that Alice somehow already obtained $\mathbb{V}_i^B$'s input and random tape. The question now is how she can keep track of that virtual party's view. If $\mathbb{V}_i^B$ is sending a protocol message to or receiving it from any $\mathbb{V}_j^A$, then Alice necessarily obtains that message and she can either check its validity or add it to $\mathbb{V}_i^B$'s view. However, what happens when $\mathbb{V}_i^B$ is receiving a message from another one of Bob's virtual parties $\mathbb{V}_j^B$? In this case things are a little trickier. If Alice is checking $\mathbb{V}_i^B$, she should be able to read the message, but if she is not checking this virtual party, then the sent message should remain hidden from her. To solve this problem, we establish private communication channels between all virtual parties of Bob and separately also between all the ones of Alice. More precisely, for $1 \leq i, j \leq m$, we will pick symmetric keys $k_{(i,j)}^B$, which will be used to encrypt the communication between $\mathbb{V}_i^B$ and $\mathbb{V}_j^B$. When Alice initially obtains input and random tape of $\mathbb{V}_i^B$, she will also obtain all keys that belong to communication channels that are connected to $\mathbb{V}_i^B$. Now if during the protocol execution $\mathbb{V}_i^B$ should send a message to $\mathbb{V}_j^B$, we let Bob encrypt the message with the corresponding symmetric key and send it to Alice, who can decrypt the message only if she is checking the receiver or sender.

In our description here we have assumed that quite a lot of correlated randomness magically fell from the sky. For instance, we assumed that all random tapes and all symmetric keys were chosen honestly and distributed correctly. To realize this setup, we will use our first compiler, which we will apply to an appropriate protocol with passive security.

2 Preliminaries

2.1 Secure Multiparty Computation

All of our security definitions follow the ideal/real simulation paradigm in the standalone model. In the real protocol execution, all parties jointly execute the protocol Π. Honest parties always follow the protocol description, whereas corrupted parties are controlled by an adversary $\mathcal{A}$. In the ideal execution all parties simply send their inputs to a trusted party $\mathcal{F}$, which computes the desired function and returns the output to the parties. Roughly speaking, we say that Π securely realizes $\mathcal{F}$, if for every real-world adversary $\mathcal{A}$, there exists an ideal-world adversary $\mathcal{S}$ such that the output distribution of the honest parties and $\mathcal{S}$ in the ideal execution is indistinguishable from the output distribution of the honest parties and $\mathcal{A}$ in the real execution. Different security notions, such as security against passive, covert, or malicious adversaries, differ in the capabilities the adversary has as well as the ideal functionalities they aim to implement. Throughout this paper we will consider synchronous protocols, static, rushing adversaries, and we assume the existence of secure authenticated point-to-point channels between the parties.

Let $\mathrm{P}_1, \ldots, \mathrm{P}_n$ be the involved parties and let $I \subset [n]$ be the set of indices of the corrupted parties that are controlled by the adversary $\mathcal{A}$. Let $\Pi : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ be an n-party protocol that takes one input from and returns one output to each party. Π internally may use an auxiliary ideal functionality $\mathcal{G}$. For the sake of simplicity, we assume that parties have inputs of the same length. Let $\bar{x} = (x_1, \ldots, x_n)$ be the vector of the parties' inputs and let z be an auxiliary input to $\mathcal{A}$. We define $\mathsf{REAL}_\lambda[\mathcal{A}(z), I, \Pi, \mathcal{G}, \bar{x}]$ as the output of the adversary $\mathcal{A}$ and the outputs of the honest parties in an execution of Π.

Passive adversaries. Security against passive adversaries is modelled by considering an environment $\mathcal{Z}$ that, in the real and ideal execution, picks the inputs of all parties. An adversary $\mathcal{A}$ gets access to views of the corrupted parties, but follows the protocol specification honestly. We consider the following ideal execution:

Inputs: Environment $\mathcal{Z}$ gets as input auxiliary information z and sends the vector of inputs $\bar{x} = (x_1, \ldots, x_n)$ to the ideal functionality $\mathcal{F}_{\mathsf{PASSIVE}}$.

Ideal functionality reveals inputs: If the ideal world adversary $\mathcal{S}$ sends $\mathsf{get\_inputs}$ to $\mathcal{F}_{\mathsf{PASSIVE}}$, then it gets back the inputs of all corrupted parties, i.e. all x_i, where $i \in I$.

Output generation: The ideal functionality computes $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$ and returns back y_i to each P_i. All honest parties output whatever they receive from $\mathcal{F}_{\mathsf{PASSIVE}}$. The ideal world adversary $\mathcal{S}$ outputs an arbitrary probabilistic polynomial-time computable function of the initial inputs of the corrupted parties, the auxiliary input z, and the messages received from the ideal functionality.

The joint distribution of the outputs of the honest parties and $\mathcal{S}$ in an ideal execution is denoted by $\mathsf{IDEAL}_\lambda[S(z), I, \mathcal{F}, \bar{x}]$.

Definition 1. *Protocol Π is said to securely compute $\mathcal{F}$ with security against passive adversaries in the $\mathcal{G}$-hybrid model if for every non-uniform probabilistic polynomial time adversary $\mathcal{A}$ in the real world, there exists a probabilistic polynomial time adversary $\mathcal{S}$ in the ideal world such that for all $\lambda \in \mathbb{N}$*

$$\left\{\mathsf{IDEAL}_\lambda[S(z), I, \mathcal{F}, \bar{x}]\right\}_{\bar{x}, z \in \{0,1\}^*} \equiv_c \left\{\mathsf{REAL}_\lambda[\mathcal{A}(z), I, \Pi, \mathcal{G}, \bar{x}]\right\}_{\bar{x}, z \in \{0,1\}^*}$$

Covert adversaries. We use the security definition of Aumann and Lindell [AL07] for defining security against covert adversaries. The security notion we consider here is the strongest one of several and is known as the Strong Explicit Cheat Formulation (SECF). Covert adversaries are modelled by considering active adversaries, but relaxing the ideal functionality we aim to implement. The relaxed ideal functionality $\mathcal{F}_{\mathsf{SECF}}$ allows the ideal-world adversary $\mathcal{S}$ to perform a limited amount of cheating. That is, the ideal-world adversary, can attempt to cheat by sending cheat to the ideal functionality, which flips an ϵ-biased coin to decide whether the attempt was successful or not. With probability ϵ, known as the *deterrence factor*, $\mathcal{F}_{\mathsf{SECF}}$ will send back $\mathsf{corrupted}$ and all parties will be informed of at least one corrupt party that attempted to cheat. With probability $1-\epsilon$, the simulator $\mathcal{S}$ will receive $\mathsf{undetected}$. In this case $\mathcal{S}$ learns all parties' inputs and can decide what the output of the ideal functionality is. The ideal execution proceeds as follows:

Inputs: Every honest party P_i sends its inputs x_i to $\mathcal{F}_{\mathsf{SECF}}$. The ideal world adversary $\mathcal{S}$ gets auxiliary input z and sends inputs on behalf of all corrupted parties. Let $\bar{x}=(x_1,\ldots,x_n)$ be the vector of inputs that the ideal functionality receives.

Abort options: If a corrupted party sends (abort, i) as its input to the $\mathcal{F}_{\mathsf{SECF}}$, then the ideal functionality sends (abort, i) to all honest parties and halts. If a corrupted party sends $(\mathsf{corrupted}, i)$ as its input, then the functionality sends $(\mathsf{corrupted}, i)$ to all honest parties and halts. If multiple corrupted parties send (abort, i), respectively $(\mathsf{corrupted}, i)$, then the ideal functionality only relates to one of them. If both $(\mathsf{corrupted}, i)$ and (abort, i) messages are sent, then the ideal functionality ignores the $(\mathsf{corrupted}, i)$ messages.

Attempted cheat: If $\mathcal{S}$ sends (cheat, i) as the input of a corrupted P_i, then $\mathcal{F}_{\mathsf{SECF}}$ decides via an ϵ-biased coin whether cheating was successful or not:
- $\mathsf{Detected}$: With probability ϵ, $\mathcal{F}_{\mathsf{SECF}}$ sends $(\mathsf{corrupted}, i)$ to the adversary and all honest parties.
- $\mathsf{Undetected}$: With probability $1-\epsilon$, $\mathcal{F}_{\mathsf{SECF}}$ sends $\mathsf{undetected}$ to the adversary. In this case $\mathcal{S}$ obtains the inputs of all honest parties from $\mathcal{F}_{\mathsf{SECF}}$. It specifies an output y_i for each honest P_i and $\mathcal{F}_{\mathsf{SECF}}$ outputs y_i to P_i.

If no (abort, i), $(\mathsf{corrupted}, i)$ or (cheat, i) commands have been sent, then the ideal execution continues as described below.

Ideal functionality answers adversary: The ideal functionality computes $(y_1,\ldots,y_n)=f(x_1,\ldots,x_n)$ and sends it to $\mathcal{S}$.

Ideal functionality answers honest parties: The adversary $\mathcal{S}$ either sends back $\mathsf{continue}$ or $(\mathsf{abort},\ \mathsf{i})$ for a corrupted P_i. If the adversary sends $\mathsf{continue}$, then the ideal functionality returns y_i to each honest parties P_i. If the adversary sends $(\mathsf{abort},\ \mathsf{i})$ for some i, then the ideal functionality sends back $(\mathsf{abort},\ \mathsf{i})$ to all honest parties.

Output generation: An honest party always outputs the message it obtained from $\mathcal{F}_{\mathsf{SECF}}$. The corrupted parties output nothing. The adversary outputs an arbitrary probabilistic polynomial-time computable function of the initial inputs of the corrupted parties, the auxiliary input z, and the messages received from the ideal functionality.

The outputs of the honest parties and $\mathcal{S}$ in an ideal execution is denoted by $\mathsf{IDEAL}^{\epsilon}_{\lambda}[S(z), I, \mathcal{F}_{\mathsf{SECF}}, \bar{x}]$. Note that the definition requires the adversary to *either* cheat or send the corrupted parties' inputs to the ideal functionality, but not both.

Definition 2. *Protocol Π is said to securely compute $\mathcal{F}$ with security against covert adversaries with ϵ-deterrent in the $\mathcal{G}$-hybrid model if for every non-uniform probabilistic polynomial time adversary $\mathcal{A}$ in the real world, there exists a probabilistic polynomial time adversary $\mathcal{S}$ in the ideal world such that for all $\lambda \in \mathbb{N}$*

$$\left\{\mathsf{IDEAL}^{\epsilon}_{\lambda}[S(z), I, \mathcal{F}_{\mathsf{SECF}}, \bar{x}]\right\}_{\bar{x},z\in\{0,1\}^*} \equiv_c \left\{\mathsf{REAL}_{\lambda}[\mathcal{A}(z), I, \Pi, \mathcal{G}, \bar{x}]\right\}_{\bar{x},z\in\{0,1\}^*}$$

Security against covert adversaries with public verifiability. This notion was first introduced by [AO12] and was later simplified by [HKK$^+$19]. In covert security with public verifiability, each protocol Π is extended with an additional algorithm Judge. We assume that whenever a party detects cheating during an execution of Π, it outputs a special message cert. The verification algorithm, Judge, takes as input a certificate cert and outputs the identity, which is defined by the corresponding public key, of the party to blame or $\perp$ in the case of an invalid certificate.

Definition 3 (Covert security with ϵ-deterrent and public verifiability). *Let $\mathsf{pk}^A, \mathsf{pk}^B$ be the keys of parties Alice and Bob and f be a public function. We say that (π, Judge) securely computes f in the presence of a covert adversary with ϵ-deterrent and public verifiability if the following conditions hold:*

> **Covert Security:** *The protocol Π (which now might output cert if an honest party detects cheating) is secure against a covert adversary according to the strong explicit cheat formulation above with ϵ-deterrent.*
> **Public Verifiability:** *If the honest party $P \in \{A, B\}$ outputs cert in an execution of the protocol, then $\mathsf{Judge}(\mathsf{pk}^A, \mathsf{pk}^B, f, \mathsf{cert}) = \mathsf{pk}^{\{A,B\}\setminus P}$ except with negligible probability.*
> **Defamation-Freeness:** *If party $P \in \{A, B\}$ is honest and runs the protocol with a corrupt party $\mathcal{A}$, then the probability that $\mathcal{A}$ outputs cert^* such that $\mathsf{Judge}(\mathsf{pk}^A, \mathsf{pk}^B, f, \mathsf{cert}^*) = \mathsf{pk}^P$ is negligible.*

Next-Message Functionality. From time to time it may be convenient to go through a n-party protocol Π step-by-step. For this purpose we define a next-message functionality $(v_1, \ldots, v_n) \leftarrow \Pi_{\mathsf{NEXT}}(i, x, r, T)$, which takes the party's index i, its input x, its random tape r and the transcript T of messages the party has seen so far as input and computes a vector $(v_1, \ldots, v_n)$ of messages, where party i should be sending message v_j to party j next. If $v_j = \perp$, then party j does not receive a message from party i in that round. The protocol ends when Π_{NEXT} outputs a special value (out, z), where z is then interpreted as the output of that party.

2.2 Helpful Building Blocks

Let us recall some basic building blocks that we will use in our constructions. We assume that ideal functionalities here satisfy active security with abort.[5]

Oblivious Transfer. In the k-out-of-n oblivious transfer functionality $\mathcal{F}_{\mathsf{OT}}$ (Figure 1) a sender has a message vector $(m_0, \ldots, m_n)$ and a receiver has index vector $(i_1, \ldots, i_k)$. The receiver learns $(m_{i_1}, \ldots, m_{i_k})$, but learns nothing about the other messages, whereas the sender learns nothing about the index vector.

$\mathcal{F}_{\mathsf{OT}}^{(n,k)}$

The functionality interacts with sender $\mathcal{S}$ and receiver $\mathcal{R}$.

---

1. On input $(\mathsf{rec}, \mathsf{sid}, (i_1, \ldots, i_k))$ from $\mathcal{R}$, if no message of the form $(\mathsf{rec}, \mathsf{sid}, *)$ was recorded in memory, then store $(\mathsf{rec}, \mathsf{sid}, (i_1, \ldots, i_k))$ and send $(\mathsf{rec}, \mathsf{sid})$ to $\mathcal{S}$.
2. On input $(\mathsf{snd}, \mathsf{sid}, (m_1, \ldots, m_n))$, if a message of the form $(\mathsf{rec}, \mathsf{sid}, *)$ is stored in memory, then send $(\mathsf{snd}, \mathsf{sid}, (m_{i_1}, \ldots, m_{i_k}))$ to $\mathcal{R}$.

Fig. 1. Ideal functionality for oblivious transfer.

[5] Active security with abort is defined like security against covert adversaries, but does not allow for sending cheat commands to the ideal functionality and only requires the honest parties to unanimously output abort, but does not require them to identify a malicious party.

Commitments. The commitment functionality $\mathcal{F}_{\mathsf{Com}}$ (Figure 2) allows a party to first commit to a message and then later open this commitment to another party. The commitment should be binding, i.e. the committing party should not be able to open the commitment to more than one message, and hiding, i.e. the commitment should not reveal any information about the committed messages before the commitment is opened.

Our compilers require commitments with a non-interactive opening phase. For the sake of simplicity, we will describe our protocols using commitments where the commitment phase is non-interactive as well, but our protocols could easily be extended to commitments with interactive commitment phases. To commit to a message m, we write: $(c, d) \leftarrow \mathsf{Com}(m; r)$ where c is the commitment and d is the opening information. The values d is then used to compute $m' \leftarrow \mathsf{Open}(c, d)$ with $m' = m$ or $\perp$ in case of incorrect opening.

$\mathcal{F}_{\mathsf{Com}}$

The functionality interacts with parties $\mathrm{P}_1, \ldots, \mathrm{P}_n$.

---

1. Upon receiving $(\mathsf{commit}, \mathsf{sid}, \mathrm{P}_i, \mathrm{P}_j, m)$ from party P_i, if no message of the form $(\mathsf{sid}, \mathrm{P}_i, \mathrm{P}_j, *)$ was recorded in memory, then store $(\mathsf{sid}, \mathrm{P}_i, \mathrm{P}_j, m)$ and send $(\mathsf{receipt}, \mathsf{sid}, \mathrm{P}_i, \mathrm{P}_j)$ to P_j.
2. Upon receiving $(\mathsf{open}, \mathsf{sid}, \mathrm{P}_i, \mathrm{P}_j)$ from P_j, if a message of the form $(\mathsf{sid}, \mathrm{P}_i, \mathrm{P}_j, m)$ is stored in memory, then send $(\mathsf{open}, \mathsf{sid}, \mathrm{P}_i, \mathrm{P}_j, m)$ to P_j.

Fig. 2. Ideal functionality for multiple commitments.

Correlated Randomness Functionality. A correlated randomness functionality $\mathcal{F}_{\mathsf{corr}}^{\mathcal{D}}$ (Figure 3) is parameterized by a distribution $\mathcal{D}$ and allows n parties to sample a vector of correlated randomness from $\mathcal{D}$ and distribute it among the n parties.

$\mathcal{F}_{\mathsf{corr}}^{\mathcal{D}}$

The functionality interacts with parties $\mathrm{P}_1, \ldots, \mathrm{P}_n$.

---

1. Upon receiving init from each party P_i for $i \in [n]$, the functionality samples $(r^1, \ldots, r^n) \leftarrow \mathcal{D}$ and sends r^i to party P_i for $i \in [n]$.

Fig. 3. Ideal functionality for generating correlated randomness.

One-Time MACs. A one-time message authentication code is composed of a pair of PPT algorithms $(\mathsf{MAC.Auth}, \mathsf{MAC.Verify})$. The authentication algorithm $\mathsf{MAC.Auth}$ takes a secret key $k \in \{0,1\}^\lambda$ and a message m as input and outputs a mac tag τ. The verification algorithm takes k, m, and τ as input and verifies whether the tag is valid or not.

3 Compiler for Two-Party Protocols with no Inputs

In this section, we present our compiler for protocols with no inputs. Our compiler assumes the existence of a two-party protocol $\mathit{\Pi}$ with security against passive adversaries that realizes the two-party functionality

$\tilde{\mathcal{F}}^{\mathcal{D}}_{\mathsf{corr}}$ depicted in Figure 4. This functionality samples correlated randomness according to some distribution $\mathcal{D}$ and provides the parties with (authenticated) secret shares thereof.

$\tilde{\mathcal{F}}^{\mathcal{D}}_{\mathsf{corr}}$

The functionality interacts with Alice and Bob.

---

1. Upon receiving init from both Alice and Bob, the functionality samples $(r^A, r^B) \leftarrow \mathcal{D}$.
2. It picks $k^A, k^B \in \{0,1\}^\lambda$ and $r_1^A, r_2^A, r_1^B, r_2^B$ such that $r^A = r_1^A \oplus r_2^A$ and $r^B = r_1^B \oplus r_2^B$ uniformly at random.
3. It computes $\tau^A \leftarrow \mathsf{MAC.Auth}(k^A, r_2^A)$ and $\tau^B \leftarrow \mathsf{MAC.Auth}(k^B, r_2^B)$.
4. It sends $(k^A, r_1^A, r_2^B, \tau^B)$ to Alice and $(k^B, r_1^B, r_2^A, \tau^A)$ to Bob.

Fig. 4. Ideal functionality for generating secret shared correlated randomness.

The formal compiler is presented in Figure 5.

Alice holds $(\mathsf{sk}^A, \mathsf{pk}^A)$ and Bob holds $(\mathsf{sk}^B, \mathsf{pk}^B)$. Both parties have no inputs.

---

1. For $i \in [k] \cup \{\mathsf{R}\}$, Alice and Bob pick seeds s_i^A and s_i^B uniformly at random of appropriate length.
2. Party $P \in \{A, B\}$ derives vector $I^P \subset [k]$ from s_{R}^P, with $|I^A| = |I^B| = \lceil k/2 \rceil - 1$.
3. Alice and Bob run two invocations of $\Pi_{\mathsf{OT}}^{(k,k/2-1)}$, where Alice and Bob each once act as the receiver and once as the sender. The sender $S \in \{A, B\}$ uses $(s_1^S, \ldots, s_k^S)$ as its input to the OT. The receiver $R \in \{A, B\}$ uses input I^R and randomness derived from s_{R}^R. Alice receives all s_i^B with $i \in I^A$ and Bob receives all s_j^A with $j \in I^B$. Let T_{OT}^A and T_{OT}^B be the sets of transcripts of the OT executions, where Alice and Bob respectively were the receiver.
4. For all $i \in [k]$, party $P \in \{A, B\}$ derives (u_i^P, v_i^P, w_i^P), from seed s_i^P. Let $V^P = \{v_i^P\}_{i \in [k]}$. The parties exchange the V^P's.
5. Alice and Bob engage in k executions of Π, where in the i-th execution Alice uses random tape $r_i^A = u_i^A \oplus v_i^B$ and Bob uses random tape $r_i^B = u_i^B \oplus v_i^A$. Let T_Π be the set of transcripts of those executions.
6. Alice computes $\sigma^A \leftarrow \mathsf{Sign}(\mathsf{sk}^A, (T_{\mathsf{OT}}^B, V^B, T_\Pi))$ and sends σ^A to Bob. Bob computes $\sigma^B \leftarrow \mathsf{Sign}(\mathsf{sk}^B, T_{\mathsf{OT}}^A, V^A, T_\Pi))$ and sends σ^B to Alice. If σ^A or σ^B does not verify, then Bob or Alice respectively abort.
7. For $i \in I^A$, Alice checks, whether Bob's messages in step 4 and step 5 are consistent with the seed s_i^B she obtained in step 3. Let $J^A \subseteq I^A$ be the set of indices, where the seeds did not match the execution. Bob does the same to obtain J^B.
8. If $J^A \neq \emptyset$, Alice picks the smallest $j \in J^A$, outputs a certificate $\mathsf{cert} = \left(\sigma^B, T_{\mathsf{OT}}^A, s_{\mathsf{R}}^A, V^A, T_\Pi, j\right)$, and aborts the execution. Bob does the same with his set J^B.
9. Alice reveals w_i^B for $i \in I^A$ and Bob reveals w_i^A for $i \in I^B$, allowing the both parties to reconstruct the sets I^A and I^B.
10. Let $i = \mathsf{min}([k] \setminus (I^A \cup I^B))$. Let $(k^A, r_1^A, r_2^B, \tau^B)$ and $(k^B, r_1^B, r_2^A, \tau^A)$ be the output of Alice and Bob in the i-th execution.
11. Alice sends (r_2^B, τ^B) to Bob, who sends back (r_2^A, τ^A). Both parties verify the received share with the corresponding tag and, if valid, reconstruct and output the correlated randomness output.

Fig. 5. Compiler from security against passive to security with public verifiability against covert adversaries for two-party protocols with no inputs.

Judge has input $(\mathsf{pk}^A, \mathsf{pk}^B, \mathsf{cert})$

1. The judge parses $\mathsf{cert} = \left(\sigma^X, T_{\mathsf{OT}}^Y, s_{\mathsf{R}}^Y, V^Y, T_\Pi, j\right)$ with $(X, Y) \in \{(A, B), (B, A)\}$.
2. If $\mathsf{Verify}(\mathsf{pk}^X, \sigma^X, (T_{\mathsf{OT}}^Y, V^Y, T_\Pi)) = 0$, return $\perp$.
3. Verify the execution of Π_{OT}, where the receiver Y uses input I^Y and random coins derived from s_{R}^Y and the sender's messages are the ones from T_{OT}^Y. If the recomputed outgoing messages of Y are not consistent with the transcript T_{OT}^Y, then return $\perp$. Otherwise let s_i^X for $i \in I^Y$ be the output of that verified execution.
4. Use s_j^X and $v_j^Y \in V^Y$ to derive r_j^X as done in step 5 of the protocol.
5. Verify the messages from X to Y in the j-th execution of Π using random tapes r_j^X and the next message function Π_{NEXT}. If the transcript matches the j-th transcript in T_Π, then return $\perp$, otherwise return pk^X.

Fig. 6. Judge protocol for our compiler in Figure 5.

Theorem 1. *Let Π be a protocol that implements a two-party functionality $\tilde{\mathcal{F}}_{\mathsf{corr}}^{\mathcal{D}}$, which receives no private inputs, with security against passive adversaries. Let $\Pi_{\mathsf{OT}}^{(k,k/2-1)}$ be a protocol that implements $\mathcal{F}_{\mathsf{OT}}^{(k,k/2-1)}$ with active security and perfect correctness. Let* (MAC.Auth, MAC.Verify) *be a one-time MAC. Then the compiler illustrated in Figure 5 and 6 implements the two-party functionality $\mathcal{F}_{\mathsf{corr}}^{\mathcal{D}}$ with security and public verifiability against covert adversaries with deterrence factor $\epsilon = \frac{1}{2} - \frac{1}{k}$.*

Proof. Before providing the formal proof, we provide some proof intuition. Since the protocol is completely symmetric, we will only focus on the case that Alice is corrupt without loss of generality. In the first step of the protocol, both Alice and Bob pick random tapes s_i^A and s_i^B for all $i \in [k]$, which are in turn parsed as $s_i^P = (u_i^P, v_i^P, w_i^P)$ for $P \in \{A, B\}$. There are several reasons for this. Since Π is only secure against a passive adversary, we need to ensure that Alice uses a random tape that is indeed uniformly random and thus we choose them as $r_i^A = u_i^A \oplus v_i^B$ for $i \in [k]$. In Step 9, Alice should reveal to Bob which executions she chose to check in Step 3. For this purpose, we let Alice reveal the values w_i^B for $i \in I^A$. Importantly, it is not sufficient for her to reveal u_i^B, since those values may potentially be revealed to her during the i-th execution of Π.

In our proof we assume without loss of generality that any adversary $\mathcal{A}$ corrupting Alice, cheats in at most one point in a protocol execution. This is done for the sake of simplicity and our proofs can easily be extended to arbitrary adversaries, albeit at the cost of readability. The case of Bob being corrupted is symmetrical. Our simulator also uses a naive rewinding strategy, which may not terminate in polynomial time, but standard techniques [GK96] can be used to ensure that the simulation runs in expected polynomial time.

The simulator $\mathcal{S}$:

1. Generate $(\mathsf{sk}^B, \mathsf{pk}^B)$ and send pk^B to $\mathcal{A}$.
2. The simulator picks random seeds s_i^B, for $i \in [k] \cup \{\mathsf{R}\}$.
3. The simulator uses $\mathcal{S}_{\mathsf{OT}}$ to simulate both OT executions. In the one where it acts as a receiver it extracts $\mathcal{A}$'s input $s_1^A, \ldots, s_k^A$. In the one where it acts as the sender it extracts the vector I^A, and sends the corresponding seeds to the adversary. Let T_{OT}^A and T_{OT}^B be the sets of transcripts of the OT executions, where the adversary (on behalf of Alice) and the simulator (on behalf of Bob) respectively were the receiver.
4. For $i \in [k]$, derive (u_i^B, v_i^B, w_i^B) from seed s_i^B and send v_i^B to $\mathcal{A}$. Receive the v_i^A from $\mathcal{A}$.
5. The simulator $\mathcal{S}$ engages in k honest executions of Π with $\mathcal{A}$, where the simulator uses random tape $r_i^B = u_i^B \oplus v_i^A$ in execution i. Let T_Π be the set of transcripts of those executions.
6. The simulator computes $\sigma^B \leftarrow \mathsf{Sign}(\mathsf{sk}^B, (T_{\mathsf{OT}}^A, V^A, T_\Pi))$ and sends it to $\mathcal{A}$. The adversary sends back σ^A. If the adversary's does not send a signature or it does not verify, then the simulation aborts.

7. The simulator uses the seeds $s_1^A, \dots, s_k^A$ extracted in Step 3 to check, whether the adversary behaved honestly in steps 4 and 5. If $\mathcal{A}$ deviated from the protocol description in any of the steps, then the simulator sends (cheat, A) to the ideal functionality.
8. Now we will be in one of the following cases:
   - **Cheating was detected:** If $\mathcal{F}_{\mathsf{corr}}^{\mathcal{D}}$ responds with $\mathsf{corrupted}$, then the simulator rewinds $\mathcal{A}$ back to beginning of Step 2 and keeps running/rewinding the protocol completely honestly (including honest OT executions) until the adversary again cheats and is caught. The simulator computes the certificate on behalf of Bob and sends $\mathsf{corrupted}$ along with the certificate to $\mathcal{A}$. The simulator outputs whatever $\mathcal{A}$ outputs and terminates.
   - **Cheating was undetected:** If $\mathcal{F}_{\mathsf{corr}}^{\mathcal{D}}$ responds with $\mathsf{undetected}$, the simulator picks a random set $\tilde{I}^B \subset [k]$ of size $\lceil k/2 \rceil - 1$, such that the execution where cheating happened is not included and sends w_i^A for all $i \in \tilde{I}^B$ to $\mathcal{A}$ and receive the w_j^B for all $j \in I^A$. Let $i = \mathsf{min}([k] \setminus (I^A \cup \tilde{I}^B))$ and $(k^A, r_1^A, r_2^B, \tau^B)$ and $(k^B, r_1^B, r_2^A, \tau^A)$ be the output of that execution. The simulator sends (r_2^A, τ^A) to $\mathcal{A}$, who in turn outputs (r_2^B, τ^B). The simulator checks, whether the MAC tag authenticates. If it does, the simulator reconstructs its output and sends it to the ideal functionality to set the output of the honest party. If it does not, the simulator sends abort to the ideal functionality, which forwards it to the honest party. The simulator terminates outputting whatever $\mathcal{A}$ outputs.
   - **No cheating occurred:** The simulator reveals w_i^A for $i \in I^B$ to the adversary, who responds by sending w_i^B for $i \in I^A$. Let $i = \mathsf{min}([k] \setminus (I^A \cup I^B))$. Let $(k^A, r_1^A, r_2^B, \tau^B)$ and $(k^B, r_1^B, r_2^A, \tau^A)$ be the output of the adversary and the simulator in the i-th execution. The simulator requests output r^A from the ideal functionality. It computes share $\tilde{r}_2^A$ such that $r^A = r_1^A \oplus \tilde{r}_2^A$ and $\tilde{\tau}^A \leftarrow \mathsf{MAC.Auth}(k^A, \tilde{r}_2^A)$. The simulator sends $(\tilde{r}_2^A, \tilde{\tau}^A)$ to $\mathcal{A}$. If the adversary aborts, then the simulator sends (abort, A) to the ideal functionality and outputs whatever $\mathcal{A}$ outputs. Otherwise $\mathcal{A}$ outputs $(\tilde{r}_2^B, \tilde{\tau}^B)$. If the MAC tag does not verify, then send abort to $\mathcal{F}_{\mathsf{corr}}^{\mathcal{D}}$, otherwise send $\mathsf{continue}$ to $\mathcal{F}_{\mathsf{corr}}^{\mathcal{D}}$, output what $\mathcal{A}$ outputs and terminate.

Consider the following sequence of hybrids.

Hybrid$_1$: Let this be the ideal-world execution, where the simulator $\mathcal{S}$ additionally has full access to the internal state of the ideal functionality $\mathcal{F}$. This is just a semantic change that does not affect the output distribution.

Hybrid$_2$: If no malicious behavior is detected in Step 7, then the simulator uses the output that was computed in execution $i = \mathsf{min}([k] \setminus (I^A \cup I^B))$ to complete the simulation as before, instead of using the one that was provided by the ideal functionality. The honest parties get the corresponding outputs through the functionality also in control of $\mathcal{S}$. The computational indistinguishability of the adversary's view in this hybrid from the previous one follows from the security guarantees of Π. The distribution of outputs of the honest parties in this hybrid is statistically indistinguishable from their distribution in the previous hybrid due to the information-theoretic MACs. We note that at this point, in the cases of "cheating was undetected" and "no cheating occurred", the simulator behaves identically, apart from the choice of I^B.

Hybrid$_3$: The simulator chooses a random subset I^B of size $\lceil k/2 \rceil - 1$ at the very beginning of the simulation. In Step 7, we do not query the ideal functionality any longer, but instead decide, whether cheating was detected or not based on $j^* \in I^B$, where j^* is the execution in which the adversary misbehaved. This hybrid is distributed identically to the previous one.

Hybrid$_4$: The simulator runs the oblivious transfer in Step 3 honestly like Bob would do in the real world. Indistinguishability of this hybrid from the previous one follows from the security of the OT.

Hybrid$_5$: The simulator does not rewind in Step 8 any more. This hybrid is statistically indistinguishable from the previous one.

This concludes the hybrids, since the last hybrid corresponds to a real-world execution.

Public Verifiability. If Alices attempted to cheat, then she must have misbehaved in one of the protocol steps, i.e. a message she sent must be inconsistent with the random tape she was supposed to be using. If Bob published a certificate, then it must have obtained her random tape for that execution along with a signature by Alice. Importantly, Alice had to produce the signature before knowing, whether cheating will be successful or not.

Defamation-Freeness. It can be easily seen that our construction is defamation-free. For a judge to blame Bob, it must have a valid signature under Bob's key for a transcript along with a random tape that is inconsistent. Since Bob is honest, it will not sign a transcript that is different from the honest one. Thus the random tape in the certificate must be the wrong one. Recall, that Bob's random tapes are $r_i^B = u_i^B \oplus v_i^A$ for $i \in [k]$, where the v_i^A are part of the signature. Thus, assuming the unforgeability of the singnature scheme, the u_i^B value must be wrong. However, this value is the output of a perfectly correct OT. Given the signed transcript of the OT, there is exactly one valid output for Alice that is consistent with that transcript, no matter Alice's inputs and randomness. It follows that the only way Alice can produce a false certificate, is to forge a signature under Bob's key.

In Theorem 1 we have assumed that each party checks less than half of the executions, which guarantees that there exists an execution that is not checked by either of the parties, but limits our deterrence factor ϵ to be less than $1/2$. We can modify our protocol to allow each party to check δk executions for any constant $1/2 \leq \delta < 1$, which allows us to obtain deterrence factors larger than $1/2$. The main observation here is that, as long as each party leaves a a constant fraction of the executions unchecked, we have a constant probability of ending up with an execution that is not checked by either of the parties. In our modified protocol, we run the protocol from Figure 5 as before up to including Step 9. If there exists no execution that is not checked by either party, i.e., $[k] \setminus (I^A \cup I^B) = \emptyset$, then the parties simply start over the whole protocol run until the condition is satisfied.

Lemma 1. *The modified protocol described above runs in expected polynomial time.*

Proof. To clarify terminology, we will call one execution of the overall protocol from Figure 5 an *outer execution* and the executions of Π within will be called *inner executions*. Observe that one outer execution runs in polynomial time. Let good be the event that the outer execution has an unchecked inner execution and let bad be the opposite event. If we can show that $\Pr[\mathsf{good}] \geq c$, where c is a constant, then we are done, since this means that, in expectation, we need to run the outer protocol $1/c$ times until the event good happens.

Now observe that for any δk inner executions chosen by Alice, Bob would need to choose the remaining $k - \delta k$ with his δk choices to trigger the bad event. We can loosely upper bound the probability of this event by just considering the Bernoulli trial, where we only ask Bob to pick $k - \delta k$ inner executions (with repetitions) that were not chosen by Alice.

$$\begin{aligned}\Pr[\mathsf{bad}] \leq & \binom{\delta k}{k-\delta k}\left(\frac{k-\delta k}{k}\right)^{k-\delta k}\left(1-\frac{k-\delta k}{k}\right)^{\delta k-(k-\delta k)} \\ = & \binom{\delta k}{k-\delta k}(1-\delta)^{k(1-\delta)}\,\delta^{k(2\delta-1)}\end{aligned}$$

Now since both k and δ are constants, this whole expression is also a constant. Since the probability for bad is upper bounded by a constant, the event good is lower bounded by a constant too.

□

The calculation above is very loose and just aims to show that the expected number of repetitions is constant. To get a better feeling of how often we have to rerun the protocol for some given deterrence factor, consider for instance $k = 3$ with $\epsilon = 2/3$. The probability of the event good is the probability of Bob picking the same two executions that Alice picked, that is, the probability is $2/3 \cdot 1/2 = 1/3$, meaning that we need to repeat the protocol three times in expectation.

4 Efficient Two-Party Computation in the preprocessing model

In this section, we consider actively secure two-party protocols in the preprocessing model. Such protocols are composed of an input-independent preprocessing protocol Π_{OFF} for generating correlated randomness and separate protocol Π_{ON} for the online phase, which uses the preprocessed correlated randomness to compute some desired function on some given private inputs. The main advantage of such protocols is that the computational "heavy lifting" can be done in the preprocessing, such that the online phase can be executed very efficiently, in particular, much faster than what can be done by a standalone protocol that computes the same functionality without correlated randomness from scratch. One of the most well known protocols in the preprocessing model is the SPDZ [DPSZ12] protocol for computing arbitrary arithmetic circuits.

A natural question to ask is, whether we can use our results from Section 3 to construct more efficient protocols in the preprocessing model by relaxing the security guarantees from active to covert. The main idea here is to replace slow preprocessing protocols with active security by faster preprocessing protocols with passive security, which are then used in combination with our compiler from Theorem 1 to generate the correlated randomness. Note, that we do not need to apply our compiler to the online phase, which is already actively secure. That is, we can combine a preprocessing protocol with covert security with an actively secure online phase to obtain an overall protocol which is secure against covert adversaries as we show in the following Lemma.

Lemma 2. *Let* $(\Pi_{\mathsf{OFF}}, \Pi_{\mathsf{ON}})$ *be a protocol implementing* $\mathcal{F}_f$ *with active security, where the preprocessing protocol* Π_{OFF} *implements* $\mathcal{F}_{\mathsf{OFF}}$ *and* Π_{ON} *implements* $\mathcal{F}_{\mathsf{ON}}$ *with active security respectively. If* $\widetilde{\Pi}_{\mathsf{OFF}}$ *implements* $\mathcal{F}_{\mathsf{OFF}}$ *with security against covert adversaries (and public verifiability), then* $(\widetilde{\Pi}_{\mathsf{OFF}}, \Pi_{\mathsf{ON}})$ *implements* $\mathcal{F}_f$ *with security against covert adversaries (and public verifiability) with the same deterrence factor as* $\widetilde{\Pi}_{\mathsf{OFF}}$*.*

Proof. To prove the statement above, we need to construct a simulator $\mathcal{S}$ that interacts with the (covert version of) ideal functionality $\mathcal{F}_f$. By assumption, there exists simulators $(\mathcal{S}_{\mathsf{OFF}}, \mathcal{S}_{\mathsf{ON}})$ for $(\Pi_{\mathsf{OFF}}, \Pi_{\mathsf{ON}})$ and simulator $\widetilde{\mathcal{S}}_{\mathsf{OFF}}$ for $\widetilde{\Pi}_{\mathsf{OFF}}$. We will use $(\widetilde{\mathcal{S}}_{\mathsf{OFF}}, \mathcal{S}_{\mathsf{ON}})$ to simulate the view of the adversary $\mathcal{A}$. If $\mathcal{A}$ attempts to cheat *after* we finished the preprocessing simulation, then we can simply consider each attempt as an abort, since the online phase is actively secure. If during the preprocessing simulation $\mathcal{A}$ attempts to cheat and consequently $\widetilde{\mathcal{S}}_{\mathsf{OFF}}$ outputs a cheat command, we will forward it to $\mathcal{F}_f$. If cheating is undetected, then the ideal functionality returns all inputs and finishing the simulation is trivial. If cheating is detected, then we inform $\widetilde{\mathcal{S}}_{\mathsf{OFF}}$ and finish the simulation accordingly.

Observe that if cheating was detected in the preprocessing, then the simulated view is indistinguishable from a real one by assumption on the security of $\widetilde{\mathcal{S}}_{\mathsf{OFF}}$. If no cheating happened, then the actively secure and covertly secure versions of $\mathcal{F}_f$ have identical input/output behaviors and thus the simulated views are also indistinguishable by assumption.

The resulting protocol is also publicly verifiable, since the preprocessing protocol is publicly verifiable, and in the (actively secure) online phase only "blatant cheating" can be performed.

□

We now consider the concrete case of SPDZ for two-party computation. In the SPDZ preprocessing, Alice and Bob generate correlated randomness in the form of secret shared multiplication triples over some field $\mathbb{F}$, i.e. Alice should obtain a set $\{(a_i^A, b_i^A, c_i^A)\}_{i \in [\ell]}$ and Bob should obtain $\{(a_i^B, b_i^B, c_i^B)\}_{i \in [\ell]}$ such that for all $i \in [\ell]$ it holds that

$$(a_i^A + a_i^B) \cdot (b_i^A + b_i^B) = (c_i^A + c_i^B).$$

Current actively secure preprocessing protocols for generating such randomness use a combination of checks based on message authentication codes and zero-knowledge proofs. To obtain a passively secure counterpart that we can then plug into our compiler, we simply take one of those existing protocols and remove those checks.

For completeness, we sketch how one such preprocessing protocol could look like. Assume we are given access to a somewhat homomorphic encryption scheme, i.e., a scheme that allows us to compute additions $\mathsf{Enc}(\mathsf{pk}, a) + \mathsf{Enc}(\mathsf{pk}, b) = \mathsf{Enc}(\mathsf{pk}, a + b)$ and a limited number of multiplications $\mathsf{Enc}(\mathsf{pk}, a) \cdot \mathsf{Enc}(\mathsf{pk}, b) =$

$\mathsf{Enc}(\mathsf{pk}, a \cdot b)$. Furthermore, assume that the decryption key sk is shared between Alice and Bob. For $P \in \{A, B\}$, each party P separately picks random values $\{a_i^P, b_i^P, r_i^P\}_{i \in [\ell]}$ and sends $\{\mathsf{Enc}(\mathsf{pk}, a_i^P), \mathsf{Enc}(\mathsf{pk}, b_i^P), \mathsf{Enc}(\mathsf{pk}, r_i^P)\}_{i \in [\ell]}$ to the other party. Each party $P \in \{A, B\}$ uses the homomorphic properties of the encryption scheme to compute $\{\mathsf{Enc}(\mathsf{pk}, r_i)\}_{i \in [\ell]}$, where $r_i = r_i^A + r_i^B$, and $\{\mathsf{Enc}(c_i)\}_{i \in [\ell]}$, where $c_i = \left(a_i^A + a_i^B\right) \cdot \left(b_i^A + b_i^B\right)$. To obtain the secret shared multiplication triple, both can jointly decrypt $\{\mathsf{Enc}(\mathsf{pk}, c_i - r_i)\}$. Since our preprocessing requires those values to be secret shared and authenticated, we need to share and compute a standard one-time MAC homomorphically on top of that.

One more detail that we need to consider here is that actively secure preprocessing protocols output correlated randomness along with authentication values. Our preprocessing protocol from above can easily be modified output these authentication values as well. Without going into too much detail, these authentication values are essentially the product of the secret shared multiplication triples and some encrypted key. These multiplications can be performed as above by using the homomorphic properties of the encryption scheme.

Combining the passively secure preprocessing protocol outlined above with our compiler from Theorem 1 results in a covertly secure preprocessing protocol with public verifiability that, for a deterrence factor of 1/3, is roughly 3 times faster than the best known actively secure protocol. Since the total running time of SPDZ is mostly dominated by the running time of the preprocessing protocol, we also obtain an overall improvement in the total running time of SPDZ of roughly the same factor 3.

5 General Compiler for Two-Party Protocols

The first ingredient we need in this section is a two-party protocol $\widetilde{\Pi}_{\mathsf{SETUP}}$ that realizes $\mathcal{F}_{\mathsf{SETUP}}^{(m,t)}$ (Figure 7) with passive security. Let Π_{SETUP} be the protocol one obtains by applying our compiler from Theorem 1 to the protocol $\widetilde{\Pi}_{\mathsf{SETUP}}$.

$\mathcal{F}_{\mathsf{SETUP}}^{(m,t)}$

The functionality interacts with Alice and Bob.

Upon receiving $(\mathsf{init}, \mathsf{sid})$ from both Alice and Bob, the functionality does the following for $P \in \{A, B\}$:

1. For $i \in [m]$, pick s_i^P uniformly at random.
2. For $i, j \in [m]$, pick $k_{(i,j)}^P$ uniformly at random with $k_{(i,j)}^P = k_{(j,i)}^P$.
3. For $i \in [m]$,
   (a) define $\mathbb{V}_i^P = \left(s_i^P, \{k_{(i,j)}^P\}_{j \in [m]}\right)$.
   (b) compute $(c_i^P, d_i^P) \leftarrow \mathsf{Com}(\mathbb{V}_i^P)$.
4. Pick $I^P \subset [m]$, uniformly at random with $|I^P| = t$.
5. Output $\left(\{c_i^B, c_i^A, d_i^A\}_{i \in [m]}, I^A, \{d_j^B\}_{j \in I^A}\right)$ and $\left(\{c_i^A, c_i^B, d_i^B\}_{i \in [m]}, I^B, \{d_j^A\}_{j \in I^B}\right)$ to A and B respectively.

Fig. 7. Ideal functionality for our correlated randomness setup.

Given the protocol Π_{SETUP}, we are now ready to present our main compiler in Figure 8. In the protocol description, we overload notation and identify the i-th virtual party $\mathbb{V}_i^P$ belonging to real party P with the set containing the random tapes and symmetric encryption keys used by this party in the protocol. This set, together with the public ciphertext containing the party's input encrypted with its own symmetric encryption key, completely determines the behaviour of that virtual party in the protocol. Let $f\left(x^A, x^B\right)$ be the function Alice and Bob would like to compute on their respective inputs x^A and x^B. Let Π be a $2m$-party protocol that implements a related functionality

$$\mathcal{F}_g(x_1^A, \ldots, x_m^A, x_1^B, \ldots, x_m^B) := \left(z_1^A, \ldots, z_m^A, z_1^B, \ldots, z_m^B\right),$$

where

$$f\left(\bigoplus_{i\in[m]} x_i^A, \bigoplus_{i\in[m]} x_i^B\right) = \bigoplus_{i\in[m],P\in\{A,B\}} z_i^P \; .$$

with security against $m+t$ passive corruptions. That is, the ideal functionality $\mathcal{F}_g$ takes as input an m-out-of-m secret sharing of x^A and x^B and computes a $2m$-out-of-$2m$ secret sharing of $f(x^A, x^B)$.

Alice has input x^A and key-pair $(\mathsf{sk}^A, \mathsf{pk}^A)$. Bob has input x^B and key-pair $(\mathsf{sk}^B, \mathsf{pk}^B)$.

---

0. For each $P \in \{A, B\}$: abort in any case of blatant cheating e.g., if P receives an invalid signature or a wrong commitment opening in step 2, 4, or 6.
1. Alice and Bob run Π_{SETUP} to obtain y^A and y^B respectively, where
$$y^A = \left(\{c_i^B, c_i^A, d_i^A\}_{i\in[m]}, I^A, \{d_j^B\}_{j\in I^A}\right)$$
and
$$y^B = \left(\{c_i^A, c_i^B, d_i^B\}_{i\in[m]}, I^B, \{d_j^A\}_{j\in I^B}\right) .$$
2. Alice and Bob verify that the outputs they received from Π_{SETUP} are well-formed, and if so, each $P \in \{A, B\}$ parses $\mathsf{Open}(c_i^P, d_i^P)$ as $\mathbb{V}_i^P = \left(s_i^P = (r_i^P, u_i^P), \{k_{(i,j)}^P\}_{j\in[m]}\right)$, then computes and sends $\sigma_c^P \leftarrow \mathsf{Sign}\left(\mathsf{sk}^P, \{c_i^P\}_{i\in[m]}\right)$.
3. Each $P \in \{A, B\}$ secret shares its input x^P as $x^P = \bigoplus_{i=1}^m x_i^P$.
4. For each $(S, R) \in \{(A, B), (B, A)\}$, for $i \in [m]$, the sender S computes $e_i^S \leftarrow \mathsf{Enc}(k_{(i,i)}^S, x_i^S)$ and $\sigma_i^S \leftarrow \mathsf{Sign}(sk^S, e_i^S)$ and sends (e_i^S, σ_i^S) to the receiver R.
5. For each $R \in \{A, B\}$, for $i \in I^R$, party R decrypts $x_i^S \leftarrow \mathsf{Dec}_{k_{(i,i)}}(e_i^S)$.
6. Alice and Bob jointly run the $2m$-party protocol Π, where each $\mathbb{V}_i^P$ uses input x_i^P and randomness r_i^P. In the following, let T be the transcript of all exchanged messages so far and $(v, j) \leftarrow \Pi_{\mathsf{next}}(i, x_i^S, r_i^S, T)$ be the next message computed by virtual party $\mathbb{V}_i^S$ belonging to real party S given the current protocol transcript T. Then we have three cases:
   (a) **$\mathbb{V}_i^S$ sends v to $\mathbb{V}_j^R$** (virtual parties belonging to different real parties): Party S computes $\sigma_{T\|v} \leftarrow \mathsf{Sign}(\mathsf{sk}^S, T\|v)$ and sends $(v, \sigma_{T\|v})$ to R. If $i \in I^R$, then R checks whether the sent message was honestly generated. If not, then R outputs $\mathsf{cert} = (\mathtt{type}_1, i, j, (T, v, \sigma_{T\|v}), (\{c_k^S\}_{k\in[m]}, \sigma_c^S, d_i^S))$ and aborts.
   (b) **$\mathbb{V}_i^S$ sends v to $\mathbb{V}_j^S$** (virtual parties belonging to same real party): Party S sends $(e, \sigma_{T\|e})$ to R, where $e \leftarrow \mathsf{Enc}(k_{(i,j)}^S, v)$ and $\sigma_{T\|e} \leftarrow \mathsf{Sign}(\mathsf{sk}^S, T\|e)$. If $j \in I^R$, then R decrypts and adds the message to its local view of $\mathbb{V}_j^S$. If $i \in I^R$, then R decrypts and checks, whether the message is generated correctly. If not, then R outputs $\mathsf{cert} = (\mathtt{type}_2, i, j, (T, e, \sigma_{T\|e}), (\{c_k^S\}_{k\in[m]}, \sigma_c^S, d_i^S))$
   (c) **Output phase:** For $S \in \{A, B\}$, $i \in [m]$, $\mathbb{V}_i^S$ obtains z_i^S. Party $\mathbb{V}_i^S$ sends $(\gamma_i^S, \sigma_{T\|\gamma}^S)$ to R, where $(\gamma_i^S, \delta_i^S) \leftarrow \mathsf{Com}(z_i^S; u_i^S)$ and $\sigma_i^S \leftarrow \mathsf{Sign}(\mathsf{sk}^S, T\|\gamma_i^S)$. If $i \in I^R$ then R recomputes z_i^S and the commitment γ_i^S. If cheating is detected, then R outputs $\mathsf{cert} = (\mathtt{type}_3, i, 0, (T, \gamma, \sigma_{T\|\gamma}), (\{c_k^S\}_{k\in[m]}, \sigma_c^S, d_i^S))$
7. Alice and Bob exchange all decommitments δ_i^P, open the output shares z_i^P and reconstruct the output $f(x^A, x^B)$.

Fig. 8. Compiler from security against passive to security with public verifiability against covert adversaries for arbitrary two-party protocols.

Theorem 2. *Let f be an arbitrary two-party functionality and let g be the related functionality as defined above. Let Π be a $2m$-party protocol that implements the ideal functionality $\mathcal{F}_g$ with security against $m+t$ passive corruptions. Let Π_{SETUP} be the protocol from Figure 5 realizing ideal functionality $\mathcal{F}_{\mathsf{SETUP}}$ from Figure 7 with deterrence factor $\frac{t}{m}$. Let* Com *be a commitment scheme that realizes $\mathcal{F}_{\mathsf{Com}}$. Then the compiler illustrated in Figure 8 and 9 implements the two-party ideal functionality $\mathcal{F}_f$ with security and public verifiability against covert adversaries with deterrence factor $\epsilon = t/m$.*

Judge has input $\left(\mathsf{pk}^A, \mathsf{pk}^B, \mathsf{cert}\right)$

1. The judge parses $\mathsf{cert} = (\mathtt{type_b}, i, j, (T, v, \sigma_{T\|v}), (\{c_k^X\}_{k\in[m]}, \sigma_c^X, d_i^X))$
2. If $\mathsf{Verify}\left(\mathsf{pk}^X, \sigma_c^X, \{c_k^X\}_{k\in[m]}\right) = 0$, return $\bot$.
3. If $\mathsf{Verify}\left(\mathsf{pk}^X, \sigma_{T\|v}, T\|v\right) = 0$, return $\bot$.
4. Let $\mathbb{V}_i^X = \mathsf{Open}(c_i^X, d_i^X)$, parse $\mathbb{V}_i^X = \left((r_i^X, u_i^X), \{k_{(i,j)}^X\}_{j\in[m]}\right)$, unless $\mathbb{V}_i^X = \bot$ in which case return $\bot$.
5. Compute $x_i^X \leftarrow \mathsf{Dec}(k_{(i,i)}^X, e_i^X)$ from e_i^X in T.
6. Compute $v^* = \Pi_{\mathsf{NEXT}}\left(i, x_i^X, r_i^X, T\right)$.
7. Depending on b do the following:
   (a) $b = 1$**:** If $v_j^* \neq v$, then return pk^X.
   (b) $b = 2$**:** If $v_j^* \neq \mathsf{Dec}(k_{(i,j)}, v)$, then return pk^X.
   (c) $b = 3$**:** If $v^* = (\mathsf{out}, z^*)$ and $v \neq \mathsf{Com}(z^*; u_i^X)$, then return pk^X.
8. Return 0.

Fig. 9. Judge protocol for our two-party protocol in Figure 8.

Proof. We will proceed by proving security, public verifiability, and defamation-freeness separately. Without loss of generality again we assume that Alice is corrupted by the adversary $\mathcal{A}$. The case for Bob being corrupted is completely symmetrical. As before, we assume that Alice misbehaves in at most one location for the sake or readability. For proving security, we construct a simulator $\mathcal{S}$ playing the role of Alice in the ideal world and using $\mathcal{A}$ as a subroutine. Let $\mathcal{S}_{\mathsf{SETUP}}$ be the simulator corresponding to the ideal functionality $\mathcal{F}_{\mathsf{SETUP}}$.

Simulation:

0. Generate $(\mathsf{sk}^B, \mathsf{pk}^B)$ and send pk^B to $\mathcal{A}$.
1. Simulate the functionality $\mathcal{F}_{\mathsf{SETUP}}$.
   (a) In case $\mathcal{A}$ inputs cheat, the simulator forwards cheat to $\mathcal{F}_f$. If $\mathcal{F}_f$ outputs $\mathsf{corrupted}$, the simulation sets $\mathsf{flag} = \mathsf{corrupted}$ and stops.
   (b) If $\mathcal{F}_f$ outputs $\mathsf{undetected}$, it also provides the simulator with the input x^B of the honest party. The simulator then sets $\mathsf{flag} = \mathsf{undetected}$ and allows $\mathcal{A}$ to pick the output y^B of B in $\mathcal{F}_{\mathsf{SETUP}}$ and runs the protocol honestly, finally it has to provide $\mathcal{F}_f$ with the output for B in the ideal world, and he does so by running the protocol as an honest party would do with $\mathcal{A}$, and using the output z^B obtained in this execution.
   (c) Finally, if $\mathcal{A}$ does not attempt to cheat in $\mathcal{F}_{\mathsf{SETUP}}$, the simulator picks the output of the corrupt party y^A in the following way: Pick a random set I^A, and run the simulator $\mathcal{S}_\Pi$ of the $2m$-party protocol Π, to produce the random tapes of all the virtual parties belonging to A for all $i \in [m]$ and of the checked virtual parties belonging to B for $i \in I^A$.
   The simulator also picks random keys $k_{(i,j)}^A$ for $i, j \in [m]$ and $k_{(i,j)}^B$ for all $i, j \in I^A$ at random. Now the simulator can honestly produce commitments $(c_i^A, d_i^A) \leftarrow \mathsf{Com}(\mathbb{V}_i^A)$ for all $i \in [m]$ and $(c_i^B, d_i^B) \leftarrow \mathsf{Com}(\mathbb{V}_i^B)$ for $i \in I^A$. Finally, use the simulator $\mathcal{S}_{\mathsf{Com}}$ to simulate all commitments c_i^B with $i \notin I^A$.
2. As in the protocol, send the signature σ_c^B to $\mathcal{A}$. Receive σ_c^A and abort if invalid.
3. Receive e_i^A, σ_i^A for $i \in [m]$. Compute $e_i^B = \mathsf{Enc}(k_{(i,i)}^B, \tilde{x}_i^B)$ where $\tilde{x}_i^B$ is uniformly random if $i \in I^A$ or 0 otherwise.
4. Decrypt $x_i^A \leftarrow \mathsf{Dec}(k_{(i,j)}^A, e_i^A)$ for all $i \in [m]$ and reconstruct x^A.
5. Simulate the execution of the $2m$-party protocol Π in the following way: Use the simulator $\mathcal{S}_\Pi$ to simulate the execution of the unchecked virtual parties belonging to B, i.e. the virtual parties $\mathbb{V}_i^B$ for $i \notin I^A$, and execute the checked parties belonging to B as an honest party would do e.g., consistent with the random tapes and keys from step 1c, and inputs $\tilde{x}_i^B$ as defined in step 3.

In particular, this means that all messages that are sent from the virtual parties belonging to B to the virtual parties belonging to A or to the checked virtual parties belonging to B are generated using the simulator. The former kind are simply sent to $\mathcal{A}$ while the latter kind are first encrypted using the appropriate keys. To simulate the message exchanged between pairs of unchecked parties belonging to B, send encryptions of 0 to $\mathcal{A}$. Moreover, if the simulator gets to step 6c, it produces commitments γ_i^B using the simulator for the commitment $\mathcal{S}_{\mathsf{Com}}$ for $i \in [m] \setminus I^A$, and commits honestly to the shares obtained by the virtual parties for all $i \in I^A$.
This concludes the decription of how the simulator simulates *outgoing* messages from $\mathcal{S}$ to $\mathcal{A}$. The simulator reacts to *incoming* messages from $\mathcal{A}$ to $\mathcal{S}$ as follows: the simulator performs the checks in steps 6a–6c of the protocol for all virtual parties belonging to A i.e., $\forall i \in [m]$. If $\mathcal{A}$ deviated in any of its virtual parties i^*, then the simulator sends (cheat, A) to the ideal functionality $\mathcal{F}_f$, which responds with $\mathsf{resp} = \{\mathsf{corrupted}, \mathsf{undetected}\}$ and we will be in one of the following cases:

- **Cheating was detected:** The simulator produces a certificate as an honest party would do (since the simulator knows the output y^A of the $\mathcal{F}_{\mathsf{SETUP}}$, the simulator knows the decommitting information for all $i \in [m]$).
- **Cheating was undetected:** The simulator receives the input of the honest party x^B. Now the simulator rewinds $\mathcal{A}$ to step 3, where it now generates the $\tilde{x}_i^B$'s as a secret sharing of x^B. The simulator keeps rewinding $\mathcal{A}$ until it again successfully cheats and then completes the full execution. The simulator reconstructs the output z^* from this execution and provides it to the ideal functionality $\mathcal{F}_f$.
- **No cheating happened:** The simulator computes $\mathcal{A}$'s shares of the output z_i^A for all $i \in [m]$ by executing the virtual parties belonging to $\mathcal{A}$ using their random tapes and inputs which have been already extracted in steps 1 and 3 of the simulation. Then the simulator inputs x^A (which was reconstructed in step 3 of the simulation) to $\mathcal{F}_f$ and receives the output z. Now the simulator decommits honestly to the commitments γ_i^B for $i \in I^A$ but uses the simulator $\mathcal{S}_{\mathsf{Com}}$ to produce decommitting information δ_i^B for the commitments γ_i^B with $i \notin I^A$ such that the opened values z_i^B are uniformly random under the constraint that $\oplus_{i=1}^m z_i^B = z \oplus_{i=1}^m z_i^A$. This concludes the simulation.

To conclude the proof, we show that the real and the simulated view of the adversary is indistinguishable.

Hybrid$_1$: Let this be the ideal-world execution, where the simulator $\mathcal{S}$ additionally has full access to the internal state of the ideal functionality $\mathcal{F}_f$. This is just a semantic change that does not affect the output distribution.

Hybrid$_2$: At the start of the simulation, the simulator picks a uniformly random set $I^B \subset [m]$ of size t. In Step 5, the simulator does not query $\mathcal{F}_f$ with the cheat command. Instead, if Alice's j-th virtual party $\mathbb{V}_j^A$ misbehaves, then we set $\mathsf{flag} = \mathsf{corrupted}$, if $j \in I^B$ and we set $\mathsf{flag} = \mathsf{undetected}$, if $j \notin I^B$. This hybrid is perfectly indistinguishable from the previous one.

Hybrid$_3$: In Step 1c, the simulator picks all random tapes honestly, like the real ideal functionality $\mathcal{F}_{\mathsf{SETUP}}$ would do (instead of using the simulator for Π). Each messages that was simulated in Step 5, will now be computed honestly using the honest parties' inputs (note that our simulator has access to the internal state of $\mathcal{F}_f$ and thus has access to the honest parties' inputs, so it can run the real protocol).

If the adversary does not cheat, then the indistinguishability of this hybrid from the previous one follows immediately from the security of Π. Next, we observe that the adversary only gets to see the prefix of a transcript of an honest protocol execution (before rewinding), if cheating happened, irrespective of whether cheating was detected or not. We observe that this hybrid remains indistinguishable, even if the adversary does misbehave. Thus this hybrid is indistinguishable from the last one.

Hybrid$_4$: Since we now run on the real honest parties' inputs from the start, we can stop rewinding in Step 5. This hybrid is statistically indistinguishable from the previous one.

Hybrid$_5$: In Step 5, instead of encrypting 0, we now encrypt the real messages between virtual parties of B that are not checked by the adversary. Indistinguishability of this hybrid from the previous one follows from the security of the encryption scheme.

Hybrid$_6$: In Step 5, in the case of $\mathsf{flag} = \mathsf{all\_good}$, we can now honestly decommit the correct value, since we are running the real execution from the start. Indistinguishability follows form the security of the commitment scheme.

At this point we observe that our simulator is not making an explicit case distinction in Step 5 or Step 5 based on the value of flag any longer, but in fact behaves like the honest parties would do in an honest execution.

Hybrid$_7$: In the last hybrid, we run $\mathcal{F}_{\mathsf{SETUP}}$ honestly instead of simulating it, since our simulation does not need access to the adversary's internal coins any longer. At this point, the adversary is running in a real-world execution, where the simulator acts on behalf of the honest parties, which concludes the hybrids.

Public Verifiability. Without loss of generality assume that Alice is corrupt. Since the setup phase is already publicly-verifiable, we only need to worry about public verifiability in the rest of the protocol. If an honest Bob outputs a certificate it must be because Alice has cheated during step 6 of the protocol i.e., one of the messages is not consistent with the input and random tape for that virtual party. Since Alice signs every outgoing message together with the entire transcript of the protocol execution, as well as her commitments in step 2 of the protocol, the judge can verify whether the message is correct or not.

Defamation-Freeness. Assume that $\mathcal{A}$ acting as Alice manages to break the defamation-freeness of the protocol and blames an honest B. We argue that this leads to a forgery to the underlying signature scheme, thus reaching a contradiction with the assumption of the theorem. In particular, for a Judge to blame Bob it must hold that in step 7 the judge finds a message from B to A which is not consistent with the next-message function of Π in execution j^*. Consider e.g., step 7a. Since Bob is honest, if the judge blames Bob it must be the case that that either the transcript of the protocol (T^*, v^*) included in the certificate, the random tape r^* or the input x^* used by the judge to verify B in this step are not the ones that Bob used in the protocol (this of course is true for all $j \in [m]$ and therefore independent of which execution j^* is claimed by $\mathcal{A}$). Since Bob is honest he will not sign a protocol transcript $(T^*, v^*) \neq (T, v)$, e.g., a transcript different than the one in the protocol executed by the honest Bob. Thus, x^* or r^* must be the wrong input or random tape. From step 5 we know that x^* is computed by decrypting some ciphertext e^* included in T^* using key k^*. Again, since Bob is honest the signed e^* must be the correct one from the protocol execution, so the fault must be in k^*. This is derived in step 4 as the opening of some commitment c^* with decomitting information d^*. As the commitment c^* was signed this must be the right commitment $c^B_{j^*}$ that Bob received from the setup functionality. And, since the commitment is binding, the decommitting information d^* cannot open c^* to anything else but the view of the virtual party used by Bob in the honest execution of the protocol. The judge can also blame Bob in steps 7b or 7c. The argument for why this is not possible for a PPT $\mathcal{A}$ is very similar to the one for the case 7a with few differences: in case 7b we also need to argue that $\mathcal{A}$ cannot blame Bob making the judge use a wrong decryption key k^*. This is however taken care of since the key is part of the view of the virtual party, which is committed (and the committment is signed by Bob). Similarly, in case 7c, $\mathcal{A}$ could also blame an innocent Bob by making the judge use the wrong randomness u^* but, this is also taken care of since the randomness is part of the view of the virtual party, which is committed (and the committment is signed by Bob).

6 Compiler for Multiparty Computation

It is natural to ask, whether the compilers described in the previous sections of the paper extend to the multiparty setting. In this section we provide some intuition for how the compiler from Section 5 can be extended the multiparty setting with a constant fraction of corruptions. More concretely, we argue that any $(cn^2 + n)$-party protocol Π with passive security against $cn^2 + cn$ corruptions, for some $1/2 < c < 1$, can be compiled into a n-party protocol with covert security and public verifiability against cn corruptions.[6]

Assume we have n parties $\mathrm{P}_1, \dots, \mathrm{P}_n$ with private inputs $x^1, \dots, x^n$, who would like to compute $f(x^1, \dots, x^n)$, and we would like to tolerate cn corruptions, for some $1/2 < c < 1$. Assume that Π implements the following $(cn^2 + n)$-party functionality, where $m = cn + 1$

[6] $c < 1/2$ is possible, but here we are specifically interested in the dishonest majority setting

$$\mathcal{F}_g \begin{pmatrix} x_1^1, \ldots, x_m^1, \\ \vdots \\ x_1^n, \ldots, x_m^n \end{pmatrix} := \begin{pmatrix} \{z_i^1\}_{i \in [m]} \\ \vdots \\ \{z_i^n\}_{i \in [m]} \end{pmatrix}$$

where

$$\bigoplus_{i \in [m], j \in [n]} z_i^j = f\left(\bigoplus_{i \in [m]} x_i^1, \ldots, \bigoplus_{i \in [m]} x_i^n \right)$$

The compiler for the multiparty setting is very similar to the compiler for the two-party setting. Each real party P_i will simulate virtual parties $\mathbb{V}_j^i$ for $1 \leq j \leq cn+1$. All virtual parties will jointly run a passively secure (cn^2+n)-party protocol. For every pair of real parties $(\mathrm{P}_i, \mathrm{P}_j)$ with $i \neq j$, we let P_i check *one* virtual party of P_j, chosen uniformly at random. Observe that for any honest P_j, the corrupted parties learn at most cn out of the $m = cn+1$ corresponding virtual parties' views, thus the adversary will not learn enough shares to reconstruct the input of the honest party P_i. In total, the adversary learn the views of

$$\overbrace{cn(cn+1)}^{\text{corrupted by } \mathcal{A}} + \overbrace{(1-c)n \cdot cn}^{\text{checked by } \mathcal{A}} = cn^2 + cn$$

virtual parties which is exactly the number of passive corruptions that we have assumed are tolerated by the protocol $\mathit{\Pi}$.

To instantiate the compiler we need a generalized multiparty version of our two-party setup functionality $\mathcal{F}_{\mathsf{SETUP}}$ from Figure 7. This generalized functionality, called $\mathcal{F}_{\mathsf{SETUP}}^{(n,cn+1)}$, can be found in Figure 10.

$\mathcal{F}_{\mathsf{SETUP}}^{(n,m)}$

The functionality interacts with parties $\mathrm{P}_1, \ldots, \mathrm{P}_n$.

Upon receiving $(\mathsf{init}, \mathsf{sid})$ from each P_i, the functionality performs the following actions for each $\ell \in [n]$:

1. The functionality picks $r_1^\ell, \ldots, r_m^\ell$ uniformly at random.
2. For $i, j \in [m]$ and any $\ell' \in [n]$, it picks $k_{(\ell',j)}^{(\ell,i)}$ uniformly at random such that $k_{(\ell',j)}^{(\ell,i)} = k_{(\ell,i)}^{(\ell',j)}$.
3. For $i \in [m]$
   (a) define $\mathbb{V}_i^\ell = \left(r_i^\ell, \{k_{(\ell',j)}^{(\ell,i)}\}_{j \in [m], \ell' \in [n]} \right)$.
   (b) compute $c_i^\ell \leftarrow \mathsf{Com}(\mathbb{V}_i^\ell)$ and $d_i^\ell \leftarrow \mathsf{Open}(c_i^\ell)$
4. For $i \in [n]$, pick $I_i^\ell \in [m]$ uniformly at random.
5. Return $\left(\{c_i^{\ell'}\}_{\ell' \in [n], i \in [m]}, \{d_i^\ell\}_{i \in [m]}, \{I_{\ell'}^\ell, \{d_i^{\ell'}\}_{i \in I_{\ell'}^\ell}\}_{\ell' \in [n]} \right)$ to P_ℓ.

Fig. 10. Ideal functionality for our correlated randomness setup in the multiparty setting.

Observe that the multiparty functionality generates symmetric keys for *all* communication channels, including the ones between virtual parties belonging to different real parties. The reason is that we could have the case, where a virtual party of P_1 sends a message to a virtual party of P_2, which should be checked by P_3. Thus we need to ensure that P_3 can decrypt the message if he is indeed checking the corresponding virtual party. Each real party P_i secret shares its input x^i into shares $x_1^i, \ldots, x_{cn+1}^i$, encrypts every share x_j^i under key $k_{(i,j)}^{(i,j)}$, and broadcasts the encrypted shares to all real parties. The virtual parties jointly execute the passively secure protocol Π. Whenever a virtual party sends a message to any other virtual party, it encrypts the message with the corresponding communication channel key and broadcasts the message among all real parties. Once the computation using Π is complete, each party commits to its output share. Finally all parties open their commitments and compute the desired output.

Security of this construction can be argued in a very similar fashion to the compiler from Section 5. The main technical difference between the two proofs lies in the calculation of the deterrence factor. Assume that the adversary misbehaves in some corrupt party P_i^*. This means that the adversary misbehaves in at least one of the $cn+1$ corresponding virtual parties. Let $\mathbb{V}_j^i$ be that party. Out of those $cn+1$ virtual parties, the $(1-c)n$ honest parties each check one uniformly random one. The probability that none of the honest parties check $\mathbb{V}_j^i$ is

$$\left(1 - \frac{1}{cn+1}\right)^{(1-c)n} \approx 1 - \frac{(1-c)n}{cn+1}$$

by binomial approximation. Thus we get a deterrence factor $\epsilon \approx \frac{1-c}{c}$.

Acknowledgements

This work was supported by the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme under grant agreement No 669255 (MPCPRO), grant agreement No 803096 (SPEC), Danish Independent Research Council under Grant-ID DFF-6108-00169 (FoCC), and the Concordium Blockhain Research Center.

References

ABT18. Benny Applebaum, Zvika Brakerski, and Rotem Tsabary. Perfect secure computation in two rounds. In *TCC 2018*, LNCS, pages 152–174, 2018.

ACGJ19. Prabhanjan Ananth, Arka Rai Choudhuri, Aarushi Goel, and Abhishek Jain. Two round information-theoretic MPC with malicious security. In *EUROCRYPT 2019*, LNCS, pages 532–561, 2019.

AL07. Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In *TCC 2007*, LNCS, pages 137–156, 2007.

AO12. Gilad Asharov and Claudio Orlandi. Calling out cheaters: Covert security with public verifiability. In *ASIACRYPT 2012*, LNCS, pages 681–698, 2012.

BMR90. Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd ACM STOC*, pages 503–513, 1990.

Bra87. Gabriel Bracha. An o(log n) expected rounds randomized byzantine generals protocol. *J. ACM*, 34(4):910–920, 1987.

CDI+13. Gil Cohen, Ivan Bjerre Damgård, Yuval Ishai, Jonas Kölker, Peter Bro Miltersen, Ran Raz, and Ron D. Rothblum. Efficient multiparty protocols via log-depth threshold formulae - (extended abstract). In *CRYPTO 2013*, LNCS, pages 185–202, 2013.

DGN10. Ivan Damgård, Martin Geisler, and Jesper Buus Nielsen. From passive to covert security at low cost. In *TCC 2010*, LNCS, pages 128–145, 2010.

DI05. Ivan Damgård and Yuval Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In *CRYPTO 2005*, LNCS, pages 378–394, 2005.

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. In *ESORICS 2013*, LNCS, pages 1–18, 2013.

DOS18. Ivan Damgård, Claudio Orlandi, and Mark Simkin. Yet another compiler for active security or: Efficient MPC over arbitrary rings. In *CRYPTO 2018*, LNCS, pages 799–829, 2018.

DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO 2012*, LNCS, pages 643–662, 2012.

GK96. Oded Goldreich and Ariel Kahan. How to construct constant-round zero-knowledge proof systems for NP. *Journal of Cryptology*, 9(3):167–190, June 1996.

GMS08. Vipul Goyal, Payman Mohassel, and Adam Smith. Efficient two party and multi party computation against covert adversaries. In *EUROCRYPT 2008*, LNCS, pages 289–306, 2008.

HKK^{+}19. Cheng Hong, Jonathan Katz, Vladimir Kolesnikov, Wen-jie Lu, and Xiao Wang. Covert security with public verifiability: Faster, leaner, and simpler. In *EUROCRYPT 2019*, LNCS, pages 97–121, 2019.

HL10. Carmit Hazay and Yehuda Lindell. *Efficient Secure Two-Party Protocols - Techniques and Constructions*. ISC. Springer, Heidelberg, 2010.

HM00. Martin Hirt and Ueli M. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *Journal of Cryptology*, 13(1):31–60, January 2000.

IK00. Yuval Ishai and Eyal Kushilevitz. Randomizing polynomials: A new representation with applications to round-efficient secure computation. In *41st FOCS*, pages 294–304, 2000.

IKNP03. Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *CRYPTO 2003*, LNCS, pages 145–161, 2003.

IKOS07. Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *39th ACM STOC*, pages 21–30, 2007.

IPS08. Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In *CRYPTO 2008*, LNCS, pages 572–591, 2008.

KM15. Vladimir Kolesnikov and Alex J. Malozemoff. Public verifiability in the covert model (almost) for free. In *ASIACRYPT 2015*, LNCS, pages 210–235, 2015.

Lin16. Yehuda Lindell. How to simulate it - A tutorial on the simulation proof technique. Cryptology ePrint Archive, Report 2016/046, 2016. http://eprint.iacr.org/2016/046.

LOP11. Yehuda Lindell, Eli Oxman, and Benny Pinkas. The IPS compiler: Optimizations, variants and concrete efficiency. In *CRYPTO 2011*, LNCS, pages 259–276, 2011.

vHL05. Luis von Ahn, Nicholas J. Hopper, and John Langford. Covert two-party computation. In *37th ACM STOC*, pages 513–522, 2005.

Yao82. Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pages 160–164, 1982.