Self-Guarding Arbitrary Cryptographic Primitives and 2PC Protocols*

Daniele Friolo[0000−0003−0836−1735], Andrea Reale[0009−0000−9321−4777], and Daniele Venturi[0000−0003−2379−8564]

Sapienza University of Rome, Italy
{friolo, venturi}@di.uniroma1.it
andrea.reale@uniroma1.it / papers@anri.dev

Abstract. In IEEE CSF '18, Fischlin and Mazaheri introduced the notion of self-guarding cryptographic protocols as a countermeasure to algorithm substitution attacks. After a trusted initialization phase, a Self-Guarder wraps the user's cryptographic algorithm implementation and sanitizes it in a way that (1) prevents that an adversary can exploit the subverted implementation to exfiltrate user's data and (2) maintain the correctness of the genuine implementation. Whilst the proposed solutions in CSF '18 support a bounded number of executions before requiring a re-initialization phase, we show a *universal* self-guarder supporting an *unbounded* number of executions from a single trusted setup. Our self-guarder can be applied to any cryptographic primitive and any two-party computation protocol in the stand-alone setting with the aid of a verifiable-computation-enabling compiler.

1 Introduction

Since the 1990s, research has been conducted about the idea of building algorithms that look correct, but secretly leak the user's secret data to other parties, typically through steganographic means; Young and Yung [33] named this notion "Kleptography", today it is known as "Algorithm Substitutions Attacks" [7], "Subversion" [6,8] or, in the security community, "backdoor". Practical attacks on cryptographic algorithms have carried out by nation states: the most known example of which is when the National Security Agency (NSA) of the United States of America devised a Pseudo-Random Number Generator based on elliptic-curves, named "Dual-EC DRBG" [10], where a backdoor was planted in the recommended parameters for the seed. The algorithm was later standardized by NIST from 2006 to 2014. Moreover, with the Snowden revelations of 2013, more light was shed on the NSA's internal Bullrun Program, which broad objective was to weaken encryption algorithms. While this incident may seem distant enough, in recent times there have been a few other occurrences of these attacks: in 2024, the compression utility "`xz`" gained fame due to a rogue contributor planting a backdoor inside the program, which ultimately allowed them to gain privileged access to any machine running a Secure SHell (SSH) server by sending carefully crafted RSA signatures. Even more recently, law enforcement agents around the European Union have been trying to lobby for ways to legally circumvent cryptography. Since 2022 a proposal named "Regulation to Prevent and Combat Child Sexual Abuse" (CSAR) has been repeatedly put forth by various EU members states in the commission. Due to the above, the debate on backdoor-prevention is still strong, even 15 years after the Snowden leaks. As alluded to, generally speaking we can note these two *types* of backdoors in the literature:

1. "Specious" algorithms, that typically (but not necessarily) use steganographic techniques [9] to embed hidden messages in the real messages of a scheme or protocol. These are *always* correct, but they additionally leak information.
2. "Arbitrary" algorithms, that may not be correct from the point of view of a protocol, but still endanger the user's secrets if a passive or active adversary receives a particular message.

* Full version of an accepted paper to IEEE CSF 2026.

Moreover, one may reasonably consider additional *properties* that give more or less power to the adversary; for example, the code for a backdoor may be "*input-triggered*" on specific messages sent by the adversary and pre-programmed in the backdoored algorithm. Or, the backdoor may decide to leak secrets which are not directly related to the algorithm being run, but are part of the user's machine in general.

We now recall the two main cryptographic countermeasures against algorithm substitution, namely Watchdogs and Reverse Firewalls.

1.1 Watchdogs

The idea is to *detect* if an algorithm deviates from a specification through a series of black-box tests, such is the work of "watchdogs" put forth by Russell *et al.* [31]. They might be divided into "offline watchdogs" (running tests before deployment) or "online watchdogs" (running continuous tests on the end-user machine). As an extension of the latter, the watchdog may also have access to the internal state of the watched algorithm ("omniscent watchdog"). The nicest property of watchdogs is that they can, potentially, test for the existence of arbitrary backdoors, since they have to verify that an algorithms correctly sticks to the given specification. However, watchdogs-based protocols also have the following downsides: first, they cannot necessarily detect input-triggered backdoors (like `xz`'s) since they are usually placed in a clever and undetectable fashion; second, they must be tailored to the algorithms at hand, because they must check them against a specification, which would make real-world implementations challenging.

1.2 Cryptographic Reverse Firewalls

As initially proposed by Mironov and Stephens-Davidowitz [29], one can install another device on a party's local network, the "Reverse Firewall", which sanitizes the communication between the party and the outside network. Indeed, this sanitization must be done in such a way the underlying protocol is not broken if no subliminal channel is present; in the case of specious backdoors the firewall may, for example, add randomness to the messages if the underlying messages can be re-randomized. Finally, such firewall must be modelled as separate party that may be even corrupted by the adversary. During the years, this technique has been expanded to IND-CCA-secure encryption by Dodis, Mironov and Stephens-Davidowitz [20] and full-on Universally-Composable MPC [14] by Li *et al.* [27] and Chakraborty *et al.* [18]. Focusing on the last work, the authors show a modified version of Canetti's UC framework [14] where each party is equipped with a firewall and then allow an explicit communication channel between the firewalls; additionally, they show how the Goldreich-Micali-Wigderson compiler [24] can be translated into their setting. Chakraborty *et al.* later expanded their work to build an authenticated channel from Password-Authenticated Key Exchange (PAKE) [17]. Finally, Arnold *et al.* [3] side-step the need to change the UC model and prove that one can build subversion-resilient protocols with firewalls in plain UC; this result is based on the idea of building firewalls that are *transparent*, that is, intuitively, the other parties do not need to have a firewall running to run the protocol. From this idea, the authors are able to show that such a firewall transforms a "specious party" into an honest one in the plain model.

The main downside of firewalls lies in the ergonomics and real-world practicality of having such a device. Secondly, in the last three mentioned works, the authors can only prevent specious backdoors; in particular, Arnold *et al.* [3] restrict themselves to the cases where the adversary can control the source of randomness (the adversary can substitute the `get_random()` function used by the algorithms, which suffices for establishing steganographic channels).

Paper structure. In the next two subsections, we introduce the concept of Self-Guarding and a technical overview of our main contributions. Then, in Sec. 2 we introduce the notation used, the cryptographic schemes and games formalisation, the 2PC model, and Verifiable Computation. In Sec. 3 we introduce our verifiable-computation-enabling compiler from arbitrary schemes and our self-guarder for any compiled scheme. Finally, Sec. 4, we discussed how to extend the same result to 2PC protocols in the stand-alone setting. In Sec. 5, we evaluate the efficiency of candidate self-guarder implementations and propose viable extensions of our results.

1.3 Self-Guarding

Let us say that we want to avoid black-box test deployment, nor maintain additional network devices: if we assume that an end-user machine is corrupted, it seems to be impossible to prevent any of these attacks, since we cannot trust the underlying code to be correct. Fischlin and Mazaheri [22] and Abdolmaleki *et al.* [1] overcame this issue by assuming trusted preprocessing and trusted simple programs or incorruptible atomic operations. In the case of Abdulmaleki *et al.* [1], they investigate subliminal steganographic channels, for protocol collusion, specifically in Zero-Knowledge Proofs (ZKPs).

Fischlin and Mazaheri [22], instead, assume that there exists a moment in time where the parties can trust their algorithms; they then generate a fixed amount of (possibly random) values, called "samples", from the genuine algorithms. Later, when the parties think that a backdoor has been planted (for example, due to a library update), they re-randomize or mask the inputs and outputs to/from the algorithms of the now-backdoored protocol using such samples. They call the set of algorithms that generates (and, later, uses) the samples a "Self-Guarder", which simply wraps the underlying scheme; they then show that they can construct it for various primitives such as Homomorphic Public-Key Encryption, Secret-Key Encryption, Digital Signatures and a variant of Key Agreement based on Physically Unclonable Functions (PUFs). The trusted setup lies in the assumption of having a genuine implementation at a particular moment in time.

This solution solves the problem without relying on external entities or black-box testing. Moreover, since they have control over the inputs that get passed to the underlying algorithm, if such an algorithm is malleable over both inputs and outputs, they can mask the inputs with random values and then "unmask" the output with the respective inverse operation. This would both avoid input-triggered backdoors, hide the input of the party *and* re-randomize the output in a single step.

However, this solution encounters some drawbacks. First, the authors limit it to *cryptographic schemes*, thereby ruling out any application to cryptographic protocols. Secondly, the code for the Self-Guarder (the trusted wrapper) should be simple enough to be completely written by the user, or stored in some trusted environment (such as an external cryptographic component or a secure enclave). This imposes a property called "simplicity", where a Self-Guarder should not reimplement the underlying algorithm.

Finally, one last issue with Fischlin and Mazaheri's solution is that if a party runs out of samples, the protocol cannot continue with a safe execution. This means that one has to know *beforehand* how many times they want to execute the protocol and choose *when* they think they might have a backdoored implementation; in fact, they support only a *bounded* number of executions.

In this paper we build upon the original techniques and ideas of Fischlin and Mazaheri [22] to address the following research question:

> *Can we construct a Self-Guarder that allows an unbounded number of executions of* any *protocol/scheme and guarantees security in the presence of malicious corruptions and arbitrary substitutions?*

We answer positively to this question by using two assumptions:

1. A trusted setup/pre-processing phase; and
2. A Verifiable Computation (VC) scheme.

In a nutshell, we are able to reduce subversion to a verifiable computation task: our solution is to require that a cryptographic scheme/two-party computation (2PC) protocol outputs a proof that the computation was carried out correctly with respect to the given inputs, then we define a generic Self-Guarder that verifies such proofs; if a scheme/protocol does not natively output such proofs, we construct a universal compiler that turns *genuine implementations* of such schemes/protocols in ones that are compatible with the above Self-Guarder.

1.4 Our contribution

We start by introducing our model working as a baseline for our definitional contribution.

Schemes, protocols, cores Schemes and protocols are usually decomposed into a set of algorithms. In general, we regard as a *scheme* any tuple of PPT algorithms $(\mathsf{F}_1, \ldots, \mathsf{F}_n, \mathsf{R})$, for a set of functions F_i and a randomness sampler R, where its security is proven with one or more cryptographic games with the challenger relying on such algorithms.

A protocol is an interactive run between two or more parties in a sequence of rounds, where in each round one of the two parties sends a message to the other. For a given round, a party P_i, given its input, the previous message, and the previous state, will compute its next-round message using a specific algorithm. The set of algorithms for the n-round execution identifies the party's core Ψ_i. In Two-Party Computation, security is defined by requiring indistinguishability between two worlds, the *ideal* world, populated by the adversary and a dummy party having access to an incorruptible trusted third party computing the required two-party function and then handling the outputs to parties, and the *real* world, where the real protocol resides. This indistinguishability notion is usually defined in terms of a simulator $\mathcal{S}$ that can simulate a real-world adversarial view via a dummy party execution in the ideal world.

Arbitrary substitutions for schemes Assume that a *scheme* $\Pi = (\mathsf{F}_1, \ldots, \mathsf{F}_N, \mathsf{R})$ is secure under a game $\mathbf{G}$. We can generally state that Π is secure under $\mathbf{G}$ if, for all PPT adversaries $\mathcal{A}$, the probability that $\mathcal{A}$ wins at $\mathbf{G}$ is negligibly close to some constant γ. A *subverted scheme* $\widetilde{\Pi} = \left(\widetilde{\mathsf{F}}_1, \ldots, \widetilde{\mathsf{F}}_N, \mathsf{R}\right)$ is a new adversarial tuple of algorithms, which possibly allows the adversary to break $\mathbf{G}$ with non-negligible probability.

Arbitrary substitutions for protocols We restrict to 2PC protocols in the stand-alone setting. Given a functionality $\mathcal{F}$, we have the *ideal* model where the parties send their inputs x_1 and x_2 to a trusted third party, which computes $\mathcal{F}(x_1, x_2) \to (y_1, y_2)$ and relays the respective outputs y_1 and y_2. In the *real* world, the mutually-distrustful parties run a protocol Π. An adversary, before the protocol execution, can *maliciously* (and statically) corrupt one of the parties.

We further augment the *real model* by allowing the adversary to additionally subvert the other party P_i by sending a new core Ψ_i to the real-world experiment; such a core is composed of new arbitrarily substituted next-message functions $\Psi_i = \left\{\widetilde{\mathsf{nmf}}^i_1, \ldots, \widetilde{\mathsf{nmf}}^i_n\right\}$. We call sREAL the joint distribution composed of the view of the adversary and the outputs.

Note that we leave the *ideal model* identical to the standard 2PC definition. Since the standard ideal model does not consider subversion, building a simulator from *our* real-world model with arbitrary substitutions to the standard 2PC ideal model implies that subversion of P_i has no effect on its security in the real world. Hence, the existence of such a simulator implies that we are basically converting a subverted party back into an honest one.

Furthermore, we additionally support the case of an unbounded number of executions with a fixed, reusable setup: a subverted party in the real-world can run an honest setup phase only once (before the substitution happens) and then make an unbounded number of executions using the self-guarder on the subverted $\widetilde{\mathsf{nmf}}$s.

Our Self-Guarders In both schemes and protocols, albeit with different syntax, we define a "Self-Guarder" as a set of PPT algorithms that, given a scheme/protocol, can be used to produce a new "self-guarded" scheme/protocol. In the case of a scheme $\Pi = (\mathsf{F}_1, \ldots, \mathsf{F}_N, \mathsf{R})$, a Self-Guarder $\mathcal{SG} = (\mathsf{Gen}, \mathsf{Guard}_1, \mathsf{Guard}_2)$ contains a setup algorithm Gen, an input encoding algorithm Guard_1 and an output decoding algorithm Guard_2. Both Guard_1 and Guard_2 take as input an index i corresponding to the algorithm F_i in Π, and the parameters generated by $\mathcal{SG}.\mathsf{Gen}$. Guard_1, in particular, also requires an input X for F_i; its job is to transform X into a new "guarded input" Γ_X that will be passed to F_i. Using this Γ_X, we can call F_i with Γ_X to obtain an output Γ_Y. The algorithm Guard_2, which also requires the auxiliary information z and the output Γ_Y, converts Γ_Y into the correct output Y for the scheme at hand. For security, we require that, after applying $\mathcal{SG}$ to a subverted $\widetilde{\Pi}$ to obtain a new scheme Π', the adversary wins at the game $\mathbf{G}$ (using the new Π') with a probability that is negligibly close to an associated constant $\gamma \in [0, 1)$. We also require a "simplicity"

property similar to Fischlin and Mazaheri's, which rules out the trivial self-guarder that simply reimplements Π, hence rendering $\mathcal{SG}$ as an efficient wrapper.

In protocols, instead, our model is slightly different. In fact, whilst we associate schemes with self-guarded cryptographic games, protocols are associated with self-guarded two-party computation. For a protocol Π, we define a Self-Guarder $\mathcal{SG}^j = (\mathsf{Setup}, \mathsf{Guard}_1, \mathsf{Guard}_2)$ as a tuple of PPT algorithms that are tailored to P_j's core $\Psi_j = \left(\mathsf{nmf}_1^j, \ldots, \mathsf{nmf}_n^j\right)$. Setup generates public and private keys that may be used by Guard_1 and Guard_2. Guard_1 takes an index i (referring to nmf_i^j), the parameters generated from Setup, the input x for nmf_i^j, the current transcript of the protocol trans and the previous state st (if any). Guard_1's goal is to generate a "guarded" input encoding Γ_X of the triplet $(x, \mathsf{trans}, \mathsf{st})$ for nmf_i^j; it additionally outputs any useful auxiliary information z. Guard_2 takes the index i, the parameters from Setup, the input encoding Γ_X, auxiliary information z from Guard_1, an output encoding Γ_Y obtained from nmf_i^j, and returns an output y that contains the next message and the new state. Like in the scheme setting, one could use two $\left(\mathcal{SG}^1, \mathcal{SG}^2\right)$ to wrap both the underlying cores of Π to obtain a new protocol Π'. For security, it suffices to show the existence of a PPT simulator, for this new Π', such that $\mathsf{sREAL} \overset{c}{\approx} \mathsf{IDEAL}$.

This concludes the main definitional contributions.

Verifiable Computation In Verifiable Computation (VC) [21] a resource-constrained client wants to outsource its computation to a trustless server. VC allows the server to compute any arbitrary function dictated by the client, over an encoded input generated by a client. The server generates an encoding of the function's output, allowing the client to verify whether the output is correctly computed by the prescribed program. Security requires that the server cannot fool a client into accepting a wrong output. Moreover, VCs can satisfy an additional property called "input-privacy", where the server cannot distinguish between two encoded inputs. Looking ahead, the idea behind our self-guarders is to flip the problem in its head: treat the subverted scheme/core as VC's remote server and request that the core outputs VC output encodings with respect to the genuine implementations. By soundness of VC, the subverted algorithms cannot compute a different output than the genuine implementations. Furthermore, input privacy of VC prevents any sort of subliminal channel from selective aborts.

Since our universal Self-Guarder constructions assume a trusted VC implementation, we need to translate "normal" schemes/protocols to use it. In particular, we provide generic compilers that turn *genuine implementations* of a scheme/protocol into an equivalent VC-compatible one.

Universal Self-Guarding We can construct universal Self-Guarders for both schemes and protocols by combining all the previous definitions and techniques. First, we provide a compiler that converts any scheme/protocol into one that is VC-compatible, i.e., the outputs of its algorithms are encoded such that they can be verified by a trusted VC implementation. Our $\mathcal{SG}$ for schemes/protocols works as follows. In the trusted setup phase, we generate n sets of keys, one for each algorithm/next-message function, using the VC's key generation algorithm, which encodes the description of the algorithm/next-message function in a public key/secret key pair.

Then, during the execution of the subverted algorithms, the Self-Guarder, by using the probabilistic encoder of the VC with input the (function-specific) secret key and the input X, produces an encoded input Σ_X together with a verification key vk_X, executes the subverted VC-compatible algorithm/next-message function with input Σ_X and obtain an encoded output Σ_Y. By using the verification algorithm of the VC, the Self-Guarder either obtains the correct output (for soundness), or $\perp$, indicating that the computation was not correct. The short running time of the probabilistic encoder and the VC verifier ensures simplicity of our self-guarders.

1.5 Downsides and extensions

From a practical point of view, an end-user needs to compile their protocol using our VC compiler to enable the usage of our self-guarder. In case of a library update, the user should trust or verify that the updated

library indeed has no backdoor and then recompile the new code with our compiler, thus rendering the end-user indeed secure against algorithm substitution, but more prone to suffering from malicious library updates.

As we discuss in Sec. 5.1, the only known approach to instantiate our self-guarder at the time of writing is to realize the underlying input-private VC via Verifiable Fully Homomorphic Encryption (vFHE), typically constructed by cleverly combining FHE with Non-Interactive Zero-Knowledge proofs (NIZKs).

For ease of modeling, we designed a self-guarder for 2PC in the stand-alone model (instead of the UC framework [14]). However, as we discussed more in detail in Sec. 5.3 it could be easily extended to multiple parties by having each party self-guard the next-message functions and generate messages for each receiving party. Further, note that $\mathcal{VC}$'s soundness and input-privacy games do not explicitly require that the queries be made in sequential order. Since we also do not use any sort of rewinding, we conjecture, in Sec. 5.2, that it may be possible to modify the theorem to support the UC model and then change the reductions to send different types of queries in a concurrent fashion. We remark, however, that our main result lies in the stand-alone setting.

2 Preliminaries

2.1 Notation

Let $\mathbb{N}$ be the set of all natural numbers; for $n \in \mathbb{N}$, we write $[n]$ to denote the set $\{1, \ldots, n\}$. We use calligraphic letters to denote sets (such as $\mathcal{X}$). Throughout the paper, we use the abbreviation PPT to refer to probabilistic polynomial time. Given a PPT algorithm $\mathcal{A}$, we denote by $\mathcal{A}(x)$ the probability distribution of the output of $\mathcal{A}$ when run on input x.

When x is chosen uniformly from a set $\mathcal{X}$, we write $x \leftarrow\!\!\$\ \mathcal{X}$. If $\mathcal{A}$ is an algorithm, we write $y \leftarrow\!\!\$\ \mathcal{A}(x)$ to denote a run of $\mathcal{A}$ on input x with output y and random tape $r \leftarrow\!\!\$\ \{0,1\}^\lambda$. If an algorithm alg is deterministic, we use $x \leftarrow \mathsf{alg}(\cdot)$ for deterministic assignment.

We denote the security parameter by $\lambda \in \mathbb{N}$, and an arbitrary positive polynomial by $\mathsf{poly}(\cdot)$. Every algorithm takes the security parameter λ as input (in unary, i.e., 1^λ). When an algorithm has multiple inputs, 1^λ is typically omitted. A function $\nu : \mathbb{N} \to \mathbb{R}$ is called negligible if, for every positive polynomial $\mathsf{poly}(\cdot)$ and for all sufficiently large λ, it holds that $\nu(\lambda) \leq \frac{1}{\mathsf{poly}(\lambda)}$. We use $\mathsf{negl}(\lambda)$ to denote an unspecified negligible function.

2.2 Abstract Cryptographic Games

Following [4], we define a cryptographic game for *deterministic schemes*, with some slight modification.

Definition 1 (Deterministic Scheme). *Let $\mathcal{R}$ be a randomness space and $\mathcal{X}_i$ and $\mathcal{Y}_i$ for $i \in [N]$ be input and output spaces. A deterministic cryptographic scheme is a tuple of efficient algorithms $\Pi = (\mathsf{F}_1, \ldots, \mathsf{F}_N, \mathsf{R})$ where*

- *$\mathsf{F}_i : \mathcal{X}_i \to \mathcal{Y}_i$ for $i \in [N]$ are deterministic algorithms.*
- *$\mathsf{R}(1^\lambda)$ samples random values $r \in \mathcal{R}$.*

We assume that each F_i, other than the supposed input in $\mathcal{X}_i$, might take as inputs randomness generated by R or values generated by other algorithms between F_1 and F_N. We model security as a cryptographic game between the adversary and a challenger.

Definition 2 (Cryptographic Game). *A cryptographic game $\mathbf{G} = (\mathsf{C}, \gamma)$ is composed of a challenger C and a value $\gamma \in [0,1)$; the game is parametrized by a cryptographic scheme Π, an adversary $\mathcal{A}$ and the security parameter $\lambda \in \mathbb{N}$. During the game execution, the efficient challenger $\mathsf{C}(1^\lambda)$ interacts with the efficient adversary $\mathcal{A}(1^\lambda)$, finally outputting a decision $d \in \{0,1\}$. The output of the game is denoted as $d \leftarrow \langle \mathcal{A}(1^\lambda), \mathsf{C}^\Pi(1^\lambda) \rangle$ and $\mathsf{Game}^\Pi_{\mathcal{A},\mathsf{C}}$ as the random variable corresponding to the execution of the game $\mathbf{G}$ with scheme Π, adversary $\mathcal{A}$ and challenger C.*

We say that Π is (t,ϵ)-secure w.r.t to the game $\mathbf{G}$ if, for all PPT $\mathcal{A}$ running in time t, the following holds:

$$\left|\Pr\left[d = 1 \mid d \leftarrow\$ \langle \mathcal{A}(1^\lambda), \mathsf{C}^{\Pi}(1^\lambda)\rangle\right] - \gamma\right| \leq \epsilon,$$

where C^{Π} means that C has oracle access to the algorithms of Π. When for all $t \in \mathsf{poly}(\lambda)$ and there exists $\epsilon \in \mathsf{negl}(\lambda)$ such that Π is (t,ϵ)-secure w.r.t $\mathbf{G}$, we say that Π is secure w.r.t to $\mathbf{G}$.

We will assume that the adversary $\mathcal{A}$, when interacting with the challenger C^{Π}, can send a set of queries of type (j, x_i), where j is a function index. The challenger will reply with a message that depends on a function F_i, which takes as input some challenger secret information, and a function of the adversarial query input.

2.3 Stand-alone Two-Party Computation [28]

The definition of malicious security in the stand-alone setting compares the joint output distributions of the (real-world) simulator and the ideal execution of the protocol.

Let P_1, P_2 be two parties. The adversary $\mathcal{A}$ maliciously statically corrupts one of the two parties.

In the ideal model, there exists a trusted third party that allows the computation of $\mathcal{F} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$. We describe the protocol in the following phases:

1. **Inputs**: Each party P_j, with $j \in \{1,2\}$, starts with its input x_j. The adversary has auxiliary input z.
2. **Malicious corruption**: From the set of honest parties $\{P_1, P_2\}$, the adversary picks party P_i and corrupts it with a malicious corruption leaving P_j, $j \neq i$ honest.
3. **Send inputs to trusted party**: The party P_j, with $j \neq i$, sends the value x_j to the trusted party. The corrupted P_i sends an arbitrary value $x_i = x_i'$ picked by $\mathcal{A}$.
4. **Early abort**: The adversary may choose to send abort to the trusted party, via P_i, to abort the protocol earlier. The trusted party sends abort to the other party P_j, which halts execution. Otherwise, the adversary sends $\mathsf{continue}$ to the trusted party and the execution continues.
5. **Output to adversary**: The trusted party computes $\mathcal{F}(x_i, x_j)$ and sends the output y_i to P_i.
6. **Late abort**: As with the early abort, the adversary, via P_i, may wish to stop the computation depending on the output y_i; if so, it sends abort to the trusted party, if not it sends $\mathsf{continue}$. The trusted party then either aborts the execution and sends abort to P_j, or it continues.
7. **Output to honest parties**: The trusted party sends the output y_j to party P_j.
8. **Execution outputs**: P_j outputs y_j, while P_i outputs nothing. The adversary outputs the triplet (g, z, y_i), where g is any probabilistic poly-time computable function of x_j', z is the auxiliary input, and y_i is the output from the trusted party.

The ideal execution of $\mathcal{F}$ on inputs (x_1, x_2), auxiliary input z for $\mathcal{A}$ and security parameter λ, denoted as $\mathsf{IDEAL}_{\mathcal{F},\mathcal{A}(z),i}(\lambda, x_1, x_2)$, is the output pair of the honest party's outputs and the adversary's view.

In the real world, the adversary maliciously corrupts P_i, then it listens to the communication between P_i and P_j. We call the real execution of Π on inputs (x_1, x_2), auxiliary input z to $\mathcal{A}$ and security parameter λ, denoted as $\mathsf{REAL}_{\Pi,\mathcal{A}(z),i}(\lambda, x_1, x_2)$, the pair of P_j's output and the adversary's view from the real execution of Π. Let $2n$ be the number of protocol rounds. The odd messages $\{m_{2k+1}\}$ are generated by P_1 and sent to P_2, whereas even messages $\{m_{2k}\}$ are generated by party P_2 and sent to P_1. Here, the k-th next-message function nmf_k^j of a party j for computing the r-round message takes as inputs the j-th party input x_j[1], the previous protocol messages $\mathsf{trans}_{r-1} = (m_1, \ldots, m_{r-1})$, and the previous state st_{k-1}; it outputs the pair of the r-th round message and a new state (m_r, st_k). At the beginning of the protocol execution, the core Ψ_1 of P_1 runs $\mathsf{nmf}_1^1(x_1, \mathsf{st}_0^1 = \epsilon)$, obtaining (m_1, st_1). From now on, for $k \in [n]$, Ψ_1 (resp. Ψ_2), upon receiving a message m_{2k} (resp. m_{2k-1}) from Ψ_2 (resp. Ψ_1), obtains $m_{2k+1} \leftarrow \mathsf{nmf}_k^1(x_1, \mathsf{trans}_{2k}, \mathsf{st}_{k-1}^1)$ (resp. $m_{2k} \leftarrow \mathsf{nmf}_k^2(x_2, \mathsf{trans}_{2k-1}, \mathsf{st}_{k-1}^2)$) and sends m_{2k+1} (resp. m_{2k}) to Ψ_2 (resp Ψ_1). If the next-message function of any honest party outputs $\perp_f$, the party aborts the execution. We will implicitly assume that public parameters used by the underlying cryptographic primitives in the real execution are previously generated and distributed among parties in a trusted manner.

[1] We assume that the input further include the randomness R used for that nmf execution.

Definition 3 (2PC Security). *Let Π be a two-party protocol for an ideal functionality $\mathcal{F}$. We say that Π securely realizes $\mathcal{F}$* in the presence of of static malicious corruptions *if, for all* PPT *adversaries, there exists a* PPT *simulator $\mathcal{S}$, such that, for all $i \in \{1,2\}$ the following holds:*

$$\left\{\mathsf{IDEAL}_{\mathcal{F},\mathcal{S}(z),i}(\lambda, x_1, x_2)\right\}_{\lambda,x_1,x_2,z} \overset{c}{\approx} \left\{\mathsf{REAL}_{\Pi,\mathcal{A}(z),i}(\lambda, x_1, x_2)\right\}_{\lambda,x_1,x_2,z}$$

where $x_1, x_2 \in \{0,1\}^*$ where $|x_1| = |x_2|$, $\lambda \in \mathbb{N}$, and $z \in \{0,1\}^*$

2.4 Verifiable Computation

A Verifiable Computation (VC) scheme [23,21,5] allows a party to delegate a computation to another party and later verify that the result was computed correctly. More formally, we define VC as follows:

Definition 4 (Verifiable Computation (VC) [21,5]). *Let f be a function. A VC scheme for f is a tuple of* PPT *algorithms $\mathcal{VC} = (\mathsf{KGen}, \mathsf{ProbGen}, \mathsf{Compute}, \mathsf{Vf})$ such that:*

- $\mathsf{KGen}(1^\lambda, f) \to (\mathsf{pk}, \mathsf{sk})$*: randomized key generation algorithm for f that returns a pair of public and secret keys. The "public key"* pk *includes a description of the function f (e.g., in circuit form), which is later used by* Compute*.*
- $\mathsf{ProbGen}(\mathsf{sk}, x) \to (\Sigma_x, \mathsf{vk}_x)$*: randomized algorithm that, given the input x for f, generates an encoding Σ_x of x and a verification key vk_x for such encoding. The encoding Σ_x is public, while the verifying key is kept private by the verifying party.*
- $\mathsf{Compute}(\mathsf{pk}, \Sigma_x) \to \Sigma_y$*: deterministic algorithm that, given the description of f (in* pk*) and the encoding of the input Σ_x for computing $f(x)$, returns the encoding Σ_y for the output $y = f(x)$.*
- $\mathsf{Vf}(\mathsf{sk}, \mathsf{vk}_x, \Sigma_y) \to y \mid \perp_{\mathcal{VC}}$*: deterministic algorithm that uses the secret key* sk *with the verify key* vk_x *to compute y.*

Definition 5 (Correctness for VC [5]). *Let $\mathcal{VC}$ be a VC scheme for some function f. We call $\mathcal{VC}$* correct *if for all values x, the following holds:*

$$\Pr\left[y = f(x) \;\middle|\; \begin{array}{c} (\mathsf{pk}, \mathsf{sk}) \leftarrow\!\$\ \mathsf{KGen}(1^\lambda, f) \\ (\Sigma_x, \mathsf{vk}_x) \leftarrow\!\$\ \mathsf{ProbGen}(\mathsf{pk}, x) \\ \Sigma_y \leftarrow \mathsf{Compute}(\mathsf{pk}, \Sigma_x) \\ y \leftarrow \mathsf{Vf}(\mathsf{sk}, \mathsf{vk}_x, \Sigma_y) \end{array}\right] = 1$$

For security, we at least require that the remote party, which does the computation, cannot cheat and fool the Verify into accepting a bad computation.

Definition 6 (Soundness for VC [5]). *Let $\mathcal{VC}$ be a VC scheme for some function f. Consider the following game between a* PPT *adversary $\mathcal{A}$ and a* PPT *challenger:*

1. *The challenger runs $(\mathsf{pk}, \mathsf{sk}) \leftarrow\!\$\ \mathsf{KGen}(1^\lambda, f)$, then sends pk to $\mathcal{A}$.*
2. *The adversary $\mathcal{A}$ makes (polynomially many) queries to the challenger, each in one of two ways:*
   (a) $\mathcal{A}$ specifies x_i; the challenger computes $(\mathsf{vk}_i, \Sigma_{x_i}) \leftarrow\!\$\ \mathsf{ProbGen}(\mathsf{sk}, x_i)$ and sends Σ_{x_i} to $\mathcal{A}$.
   (b) $\mathcal{A}$ specifies a verification query $(i, \hat{\Sigma})$; the challenger computes $y = \mathsf{Vf}(\mathsf{sk}, \mathsf{vk}_i, \hat{\Sigma})$ and returns 1 if $y \neq \perp_{\mathcal{VC}}$, otherwise it returns 0.
3. *After poly-many queries, the adversary eventually outputs a pair (i^*, Σ^*).*
   $\mathcal{A}$ wins at the game if and only if $y^* \neq f(x_{i^*})$ and $y^* \neq \perp_{\mathcal{VC}}$, such that $y^* = \mathsf{Vf}(\mathsf{sk}, \mathsf{vk}_{i^*}, \Sigma^*)$.

We call $\mathcal{VC}$ sound *if for all* PPT *adversaries $\mathcal{A}$:*

$$\Pr[\mathcal{A}\ \textit{wins}] \leq \mathsf{negl}(\lambda)$$

Moreover, we have to require a property of *input privacy*: given an output encoding, the adversary cannot distinguish which input encoding gives that output encoding; which, in turn, implies indistinguishability of outputs.

Definition 7 (Input-privacy for VC [5]). *Let $\mathcal{VC}$ be a VC scheme for some function f. Consider the following game between a* PPT *adversary $\mathcal{A}$ and a* PPT *challenger:*

1. *The challenger generates $(\mathsf{pk}, \mathsf{sk}) \leftarrow\!\!\$\ \mathsf{KGen}(1^\lambda, f)$ and picks a random bit $b \leftarrow\!\!\$\ \{0,1\}$; then it sends pk to $\mathcal{A}$.*
2. *$\mathcal{A}$ makes $q = \mathsf{poly}(\lambda)$ queries to the challenger of these two types:*
   (a) *(Encoding query) $\mathcal{A}$ sends an input x_i to the challenger. The latter computes $(\mathsf{vk}_i, \Sigma_{x,i}) \leftarrow\!\!\$\ \mathsf{ProbGen}(\mathsf{sk}, x_i)$ and sends $\Sigma_{x,i}$ to $\mathcal{A}$.*
   (b) *(Verification query) $\mathcal{A}$ sends an encoding of an output (i, Σ_y) to the challenger. The latter computes $y \leftarrow \mathsf{Vf}(\mathsf{sk}, \mathsf{vk}_i, \Sigma_y)$ and sends 1 to $\mathcal{A}$ if $y \neq \perp_{\mathcal{VC}}$, otherwise it sends 0.*
3. *$\mathcal{A}$ sends two inputs (x_0^*, x_1^*) to the challenger.*
4. *The challenger computes $\left(\mathsf{vk}^*, \Sigma_{x_b^*}\right) \leftarrow\!\!\$\ \mathsf{ProbGen}(\mathsf{sk}, x_b^*)$ and sends $\Sigma_{x_b^*}$ to $\mathcal{A}$ (the adversary is allowed make verification queries against $\Sigma_{x_b^*}$).*
5. *$\mathcal{A}$ can make other $\mathsf{poly}(\lambda)$ queries to the challenger as in Step 2.*
6. *$\mathcal{A}$ outputs a guess $b' \in \{0,1\}$. The adversary wins at the game if $b = b'$.*

We call $\mathcal{VC}$ input-private *if for all* PPT *adversaries $\mathcal{A}$, the following holds:*

$$\Pr[\mathcal{A}\ \textit{wins}] \leq \mathsf{negl}(\lambda)$$

Note that, in the game above, the adversary can query the challenger with x_0^* and x_1^* after receiving $\Sigma_{x_b^*}$; since $\mathsf{ProbGen}$ is randomized, the adversary should only be able to tell that the encodings come from the same distribution.

Finally, we need the notion of "outsourceability", that is computing the function by ourselves is more expensive than letting another party do it for us. More formally:

Definition 8 (Outsourceability for VC). *A VC scheme $\mathcal{VC}$ can be* outsourced *if it allows efficient generation and efficient verification: for any input x and output encoding Σ_y, the time (or circuit size) requires for $\mathsf{ProbGen}(\mathsf{sk}, x)$ plus the time (or circuit size) for $\mathsf{Vf}(\mathsf{sk}, \mathsf{vk}_x, \Sigma_y)$ is $o(T)$, where T is the time (or circuit size) required to compute $f(x)$.*

3 Self-Guarding for Cryptographic Schemes

Definition 9 (Self-Guarder). *A self-guarder $\mathcal{SG}$ for a cryptographic scheme $(\mathsf{F}_1, \ldots, \mathsf{F}_N, \mathsf{R})$ (Def. 1) is composed of a tuple of algorithm $(\mathsf{Gen}, \mathsf{Guard}_1, \mathsf{Guard}_2)$, where each algorithm behave as follows:*

- $\mathsf{Gen}(1^\lambda)$*: On input the security parameter 1^λ, this randomized algorithm outputs guarding parameters κ.*
- $\mathsf{Guard}_1(i, \kappa, X)$*: On input the function index, the guarding parameters κ and an input $X \in \mathcal{X}_i$ this deterministic algorithm outputs a guarded input Γ_X, and some auxiliary info z.*
- $\mathsf{Guard}_2(i, \kappa, z, \Gamma_Y)$*: On input the function index, this deterministic algorithm takes the guarding parameters κ, some auxiliary info z, an output Γ_Y, and outputs the guarded output $Y \in \mathcal{Y}_i$.*

Let $\widetilde{\Pi} = (\widetilde{\mathsf{F}}_1, \ldots, \widetilde{\mathsf{F}}_N, \mathsf{R})$ be a potentially subverted scheme, we denote $\mathcal{SG}(\kappa, \widetilde{\Pi})$ the self-guarded implementation $(\mathsf{F}_1^*, \ldots, \mathsf{F}_N^*, \mathsf{R})$, where $\mathsf{F}_i^*(X)$, works as follows:

- Compute $(\Gamma_X, z) \leftarrow \mathsf{Guard}_1(i, \kappa, X)$,
- Obtain $\Gamma_Y \leftarrow \widetilde{\mathsf{F}}_i(\mathsf{pk}_i, \Gamma_X)$, where pk_i is taken from $\kappa = \{\mathsf{pk}_j, \mathsf{sk}_j\}_{j \in [N]}$
- Compute $Y \leftarrow \mathsf{Guard}_2(i, \kappa, z, \Gamma_Y)$.

The security definition w.r.t subversion of Π in the self-guarding model w.r.t. respect to a cryptographic game $\mathbf{G} = (\mathsf{C}, \cdot)$ follows.

Definition 10 (Self-guarding against subversion). *Let Π be a cryptographic scheme, $\mathcal{SG}$ be a self-guarder for Π and $\mathbf{G} = (\mathsf{C}, \gamma)$ a security game for Π. We say that $\mathcal{SG}$ ensures security for Π with game $\mathbf{G}$ against arbitrary substitutions if, for all $\mathcal{A}$,*

$$\left|\Pr\left[\mathsf{Game}^{\mathcal{SG}(\kappa,\widetilde{\Pi})}_{\mathcal{A}(\mathsf{state}),\mathsf{C}} \;\middle|\; \begin{array}{c}(\mathsf{state},\widetilde{\Pi}) \leftarrow\!\$\ \mathcal{A}(1^\lambda);\\ \kappa \leftarrow\!\$\ \mathcal{SG}.\mathsf{Gen}(1^\lambda)\end{array}\right] - \gamma\right| \leq \mathsf{negl}(\lambda)$$

Definition 11 (Simplicity). *Let $\Pi = (\mathsf{R}, \mathsf{F}_1, \ldots, \mathsf{F}_N)$ be a cryptographic scheme and $\mathcal{SG}$ its self-guarder. $\mathcal{SG}$ is* simple *if, for any $\kappa \in \mathcal{SG}.\mathsf{Gen}(1^\lambda)$, for all $i \in [N]$, for any input x and unguarded output Γ_Y, the time (or circuit size) required for computing $\mathcal{SG}.\mathsf{Guard}_1(i, \kappa, X)$ plus the time (or circuit size) required for computing $\mathcal{SG}.\mathsf{Guard}_2(i, \kappa, z, \Gamma_Y)$ is $o(T)$, where T is the time (or circuit size) required to compute $F_i(X)$.*[2]

3.1 A Black-Box Self-Guarder for any Cryptographic Scheme

We construct a black-box self-guarder from a compiled (potentially insecure) compiled variant of the original cryptographic scheme from verifiable computation schemes. The description of the compiler follows.

Construction 1 (VC Compiler for Cryptographic Schemes) *Let $\Pi = (\mathsf{F}_1, \ldots, \mathsf{F}_N, \mathsf{R})$ be a Cryptographic Scheme (Def. 1) and let $\mathcal{VC}$ be a Verifiable Computation Scheme (Def. 4). The compiler $\mathcal{C}$, on input Π outputs a compiled scheme $\Pi' = (\mathsf{S}, \mathsf{F}'_1, \ldots, \mathsf{F}'_N, \mathsf{R})$, where:*

- *S, on input 1^λ, for each $i \in [N]$, computes $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow\!\$\ \mathcal{VC}_i.\mathsf{KGen}(\mathsf{F}_i)$ and outputs $\kappa = \{\mathsf{pk}_i, \mathsf{sk}_i\}_{i\in[N]}$*
- *$\mathsf{F}'_i(\mathsf{pk}_i, \Sigma_X)$, on input a VC public key pk_i and an input encoding Σ_X, where $(\Sigma_X, \cdot) \in \mathcal{VC}_i.\mathsf{ProbGen}(\mathsf{sk}_i, X)$, outputs an output encoding $\Sigma_y \leftarrow \mathcal{VC}.\mathsf{Compute}(\mathsf{pk}_i, \Sigma_X)$.*

We can now introduce a self-guarder $\mathcal{SG} = (\mathsf{Gen}, \mathsf{Guard}_1, \mathsf{Guard}_2)$ for the scheme Π' compiled against Π with the above construction.

Construction 2 (Self-Guarder for VC-Compiled Schemes) *Let $\Pi' = (\mathsf{S}, \mathsf{F}'_1, \ldots, \mathsf{F}'_N, \mathsf{R})$ be the compiled scheme against Π (Cons. 1) and let $\mathcal{VC}$ Verifiable Computation scheme (Def. 4), the self-guarder $\mathcal{SG} = (\mathsf{Gen}, \mathsf{Guard}_1, \mathsf{Guard}_2)$ for Π' behave as follows:*

- *$\mathsf{Gen}(1^\lambda)$: Output $\kappa \leftarrow\!\$\ \mathsf{S}(1^\lambda)$.*
- *$\mathsf{Guard}_1(i, \kappa, X)$: Parse κ as $\{(\mathsf{pk}_i, \mathsf{sk}_i)\}_{i\in[N]}$ and outputs $\Gamma_X = (\mathsf{pk}_i, \Sigma_X)$ and $z = \mathsf{vk}_X$, where $(\Sigma_X, \mathsf{vk}_X) \leftarrow\!\$\ \mathsf{ProbGen}(\mathsf{sk}_i, X)$.*
- *$\mathsf{Guard}_2(i, \kappa, z, \Gamma_Y)$: Parse κ as $\{(\mathsf{pk}_i, \mathsf{sk}_i)\}_{i\in[N]}$ and outputs $Y \leftarrow \mathsf{Vf}(\mathsf{sk}_i, z, \Gamma_Y)$.*

We are now ready to show that the self-guarder above ensure security w.r.t. the compiled cryptographic scheme against the game $\mathbf{G}$.

Theorem 1. *Let $\Pi = (\mathsf{F}_1, \ldots, \mathsf{F}_N, \mathsf{R})$ be a cryptographic scheme, $\mathbf{G}$ be a game for Π and $\Pi' = (\mathsf{F}'_1, \ldots, \mathsf{F}'_N, \mathsf{R})$ be the compiled scheme of Cons. 1 from Π with a correct (Def. 5), sound (Def. 6), input-private (Def. 7) and outsourceable (Def. 8) VC scheme (Def. 4), then the self-guarder $\mathcal{SG}$ of Cons. 2 is simple (Def. 11) and ensures security of Π' against arbitrary substitution w.r.t. $\mathbf{G}$ (Def. 10).*

Proof. We need to show that

$$\left|\Pr\left[\mathsf{Game}^{\mathcal{SG}(\kappa,\widetilde{\Pi'})}_{\mathcal{A}(\mathsf{state}),\mathsf{C}} \;\middle|\; \begin{array}{c}(\mathsf{state},\widetilde{\Pi'}) \leftarrow\!\$\ \mathcal{A}(1^\lambda);\\ \kappa \leftarrow\!\$\ \mathcal{SG}.\mathsf{Gen}(1^\lambda)\end{array}\right] - \gamma\right| \leq \mathsf{negl}(\lambda)$$

Let $\mathcal{SG}(\kappa, \widetilde{\Pi'}) = (\mathsf{F}^*_1, \ldots, \mathsf{F}^*_N, \mathsf{R})$. Let us consider the following hybrids:

[2] Since we are in the cryptographic circuit model, we consider the description of the functions as the circuits themselves.

- $\mathsf{Hyb}^0_i(\lambda)$: As in $\mathsf{Game}^{\mathcal{SG}(\kappa,\widetilde{\Pi'})}_{\mathcal{A}(\mathsf{state}),\mathsf{C}}$, except that each F^*_j, $0 < j \leq i$ is replaced with an algorithm $\hat{\mathsf{F}}_j$, working as follows:
  - If $\mathsf{F}^*_j(X)$ outputs $\perp_{\mathcal{VC}}$, then output $\perp_{\mathcal{VC}}$,
  - Otherwise output $\mathsf{F}_j(X)$.
- $\mathsf{Hyb}^1_{i,j}(\lambda)$: As in $\mathsf{Hyb}^0_N(\lambda)$, except that, $\hat{\mathsf{F}}_k$ for $k < i$ works as follows:
  - Sample $X_\$ \leftarrow\$ \mathcal{X}_k$ and compute $y \leftarrow \mathsf{F}^*_k(X_\$)$.
  - If $y = \perp_{\mathcal{VC}}$ output $\perp_{\mathcal{VC}}$. Otherwise, output $\mathsf{F}_k(X)$.

  Moreover, C, when receiving for the queries up to the j-th for the i-th function from $\mathcal{A}$, works as follows:
  - Sample a random $X_\$$ and compute $Y \leftarrow \mathsf{F}^*_i(X_\$)$.
  - If $Y = \perp_{\mathcal{VC}}$ output $\perp_{\mathcal{VC}}$. Otherwise, generate the challenger response using $\mathsf{F}_i(X)$, where X is the input extracted from the adversarial query.

  C responds to the rest of the queries as in $\mathsf{Hyb}^0_N(\lambda)$, i.e. if $F^*_i(X) = \perp_{\mathcal{VC}}$, output $\perp_{\mathcal{VC}}$. Otherwise, use $\mathsf{F}_i(X)$, where X is the input extracted from the adversarial query.
- $\mathsf{Hyb}^1_N(\lambda)$ is exactly $\mathsf{Hyb}^1_{N-1,q}(\lambda)$, i.e. $\hat{\mathsf{F}}_k$ for $k \in [N]$ works as follows:
  - Sample $X_\$ \leftarrow\$ \mathcal{X}_k$ and compute $y \leftarrow \mathsf{F}^*_k(X_\$)$.
  - If $y = \perp_{\mathcal{VC}}$ output $\perp_{\mathcal{VC}}$. Otherwise, output $\mathsf{F}_k(X)$.

Our proof strategy relies on the soundness and input privacy of the VC scheme. First, to argue indistinguishability between $\mathsf{Hyb}^1_i(\lambda)$ and $\mathsf{Hyb}^1_{i+1}(\lambda)$, we rely on soundness. Whenever Guard_2, which invokes the verification algorithm $\mathcal{VC}.\mathsf{Vf}$, does not return the abort symbol $\perp_{\mathcal{VC}}$ (the two hybrids have the same distribution in case of $y = \perp_{\mathcal{VC}}$), we can safely replace the $i+1$-th function output with the correct evaluation $F_i(X)$. Then, we rely on input privacy to argue that the abort behavior is independent of the actual input. To achieve this, we transition from $\mathsf{Hyb}^1_{i,j-1}(\lambda)$ to $\mathsf{Hyb}^1_{i,j}(\lambda)$, where $\perp_{\mathcal{VC}}$ is returned only if Guard_2 would abort when the subverted function for the j-th query on the i-th function is evaluated on a randomly chosen input, rather than the real one.

Lemma 1. $\mathsf{Hyb}^1_i(\lambda) \stackrel{c}{\approx} \mathsf{Hyb}^1_{i+1}(\lambda)$ *for* $i \in \{0, \ldots, N-1\}$.

Proof. Let us call $\mathbf{BAD}_i$ the event that the challenger answers to an adversarial query (i, x) with a message $m' \neq m$, where m is the output of $\mathsf{C}^{\mathcal{SG}_\Pi}$ for the same query and the same challenger inputs (such as randomness or public keys). Since $\mathbf{BAD}_i$ is the only event differentiating the two hybrids, we can state that, assuming that the adversary makes $\mathsf{poly}(\lambda)$ queries to the challenger for the i-th function,

$$\Pr\big[\mathsf{Hyb}^1_i(\lambda)\big] - \Pr\big[\mathsf{Hyb}^1_{i+1}(\lambda)\big] \leq \Pr[\mathbf{BAD}_i] \cdot \mathsf{poly}(\lambda).$$

To trigger the bad event $\mathbf{BAD}_i$, F^*_i shall produce an output $y' \neq y$, where $y = \mathsf{F}_i(X)$ (for the correctness of $\mathcal{VC}$). We have that by Soundness of $\mathcal{VC}$, given $(\Sigma_X, \mathsf{vk}_X) \leftarrow \mathsf{ProbGen}(\mathsf{pk}_i, X)$ for $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{KGen}(\mathsf{F}_i)$ and $y' \leftarrow \mathsf{Vf}(\mathsf{sk}_i, \mathsf{vk}_X, \Sigma'_y)$ computed by F^*_i and Σ'_y obtained from the invocation of the subverted algorithm $\widetilde{\mathsf{F}}_i(\mathsf{pk}_i, \Sigma_X)$, the probability that $y' \neq y$, where $y = \mathsf{F}_i(X)$ is negligible.

Lemma 2. *Let* $q \in \mathsf{poly}(\lambda)$ *be the number of queries that* $\mathcal{A}$ *will trigger to* C. $\mathsf{Hyb}^1_{i,j-1}(\lambda) \stackrel{c}{\approx} \mathsf{Hyb}^1_{i,j}(\lambda)$ *for* $i \in [N]$, $j \in [q]$.

Proof. Let D be a distinguisher between $\mathsf{Hyb}^1_{i,j}(\lambda)$ and $\mathsf{Hyb}^1_{i,j+1}(\lambda)$. We can construct an adversary $\mathcal{A}_{priv}$ breaking input privacy of $\mathcal{VC}$. $\mathcal{A}_{priv}$ behaves as follows:

- Wait to receive $(\mathsf{state}, \widetilde{\Pi})$ from D and pk from the VC challenger.
- Initialize a counter $c \leftarrow 0$.
- When receiving a query (i, x) from D, simulate the challenger C by invoking the VC challenger on X, where X is the input extracted from x supposed to be sent by C to $\widetilde{\mathsf{F}}_i$, and:
  - Set $c \leftarrow c + 1$.
  - if $c < j$, send $X_\$ \leftarrow\$ \mathcal{X}_i$ to the challenger and wait to receive back Σ_X. Then, obtain $\Sigma_Y \leftarrow \widetilde{\mathsf{F}}_i(\mathsf{pk}_i, \Sigma_X)$ and query the VC challenger with (c, Σ_Y) to obtain b. If $b = 0$, return $\perp$ to $\mathcal{A}$. Else, return $F_i(X)$ to $\mathcal{A}$.

- if $c = j$, send $(X_{\$} \leftarrow\$ \mathcal{X}_i, X)$ to the challenger and wait to receive back Σ_{X^*}. Then, obtain $\Sigma_{Y^*} \leftarrow \widetilde{\mathsf{F}}_i(\mathsf{pk}_i, \Sigma_{X^*})$ and finally and query the VC challenger with (c, Σ_{Y^*}) to obtain back a bit b. If $b = 0$, return $\perp$ to $\mathcal{A}$. Else, return $F_i(X)$ to $\mathcal{A}$.
- if $c > j$, send X to the challenger and wait to receive back Σ_X. Then, obtain $\Sigma_Y \leftarrow \widetilde{\mathsf{F}}_i(\mathsf{pk}_i, \Sigma_X)$ and query the VC challenger with (c, Σ_Y) to obtain b. If $b = 0$, return $\perp$ to $\mathcal{A}$. Else, return $F_i(X)$ to $\mathcal{A}$.

– Simulate all the queries of the other functions with index $k \neq i$ as in $\mathsf{Hyb}^1_{i,j}(\lambda)$.

$\mathcal{A}_{priv}$ perfectly simulates $\mathsf{Hyb}^1_{i,j-1}(\lambda)$ when the VC challenger computes Σ_X, whereas perfectly simulates $\mathsf{Hyb}^1_{i,j}(\lambda)$ when the VC challenger computes $\Sigma_{X_{\$}}$.

Lemma 3. $\left|\Pr\left[\mathsf{Hyb}^1_N(\lambda)\right] - \gamma\right| \leq \mathsf{negl}(\lambda)$.

Proof. Since $\mathsf{Hyb}^1_N(\lambda)$, in the case that $\hat{\mathsf{F}}_i$ does not output $\perp$, uses the genuine F_i, and since the aborting cases are all independent because of the random inputs, the probability, for the adversary, to win $\mathsf{Hyb}^1_N(\lambda)$ is bounded by the probability of winning $\mathsf{Game}^{\Pi}_{\mathcal{A}(1^\lambda),\mathsf{C}}(\lambda)$.

Lemma 4. *$\mathcal{SG}$ is simple.*

Proof. Simplicity follows straghtforwadly from Outsourceability of the underlying VC (Def. 8).

The theorem follows by combining Lemma 1 to Lemma 4.

4 Self-Guarding for 2PC

Let P_i be the i-th party of a protocol Π and Ψ_i be their "core" (or "code"). By core we mean the actual algorithm running on P_i's machine to execute Π.

Definition 12 (Core). *Given a functionality $\mathcal{F}$ and a protocol Π that realizes it, the set of algorithms run by party P_i is the "core" Ψ_i.*

In any standard security setting, an honest party P has their core Ψ matching the protocol Π to the extent that they are practically the same. In the setting of subversion, the core Ψ may not be *genuine* but, instead, it may be replaced with a *subverted* $\widetilde{\Psi}$ provided by an adversary $\mathcal{A}$, to leak a secret. In practice $\widetilde{\Psi}$ implements another protocol $\widetilde{\Pi}$ different from Π yet unbeknownst to P_i.

To enable self-guarding, we augment the standard 2PC model of [28]. We start by introducing the self-guarder for cores.

Definition 13 (Self-Guarder). *A self-guarder $\mathcal{SG}$ for a core $(\mathsf{nmf}_1, \ldots, \mathsf{nmf}_n)$ (Def. 12) is composed of a tuple of algorithm $(\mathsf{Setup}, \mathsf{Guard}_1, \mathsf{Guard}_2)$, where each algorithm behave as follows:*

– $\mathsf{Setup}(1^\lambda)$*: On input the security parameter 1^λ, this randomized algorithm outputs guarding outputs pairs $\{(\mathsf{pk}_i, \mathsf{sk}_i)\}_{i\in[n]}$.*

– $\mathsf{Guard}_1(i, \mathsf{pk}_i, \mathsf{sk}_i, x, \mathsf{trans}, \mathsf{st})$*: On input the function index, the pair $(\mathsf{pk}, \mathsf{sk})$ the party's input x, a transcript trans and the previous state st_{i-1}, this deterministic algorithm outputs a guarded input Γ_X together with some auxiliary info z.*

– $\mathsf{Guard}_2(i, \mathsf{pk}_i, \mathsf{sk}_i, z, \Gamma_y)$*: On input the function index, the guarding parameters $\mathsf{pk}_i, \mathsf{sk}_i$, the guarded input Γ_X, auxiliary info z and an output encoding Γ_Y, this deterministic algorithm outputs the guarded output Y in the support of nmf_i.*

model of [20] with the notion of self-guarding. We start by

Like Fischlin and Mazaheri [22], we forbid the trivial case where the party re-implements the protocol as a Self-Guarding algorithm by modeling the concept of "simplicity": the Self-Guarding algorithm should *not* be as complex (computational complexity) as the underlying protocol.

Definition 14 (Simplicity for Cores). *Let $\Psi = (\mathsf{nmf}_1, \ldots, \mathsf{nmf}_n)$ be a core of a cryptographic protocol and $\mathcal{SG}$ the corresponding self-guarder. $\mathcal{SG}$ is* simple *if, for any $\{(\mathsf{pk}_i, \mathsf{sk}_i)\}_{i\in[n]} \in \mathcal{SG}.\mathsf{Setup}(1^\lambda)$, for all $i \in [n]$, for any input x, transcript trans, state st and output Γ_Y, the time (or circuit size) required for computing $(\Gamma_X, z) \leftarrow\$ \mathcal{SG}.\mathsf{Guard}_1(i, \mathsf{pk}_i, \mathsf{sk}_i, x, \mathsf{trans}, \mathsf{st})$ plus the time (or circuit size) required for computing $\mathcal{SG}.\mathsf{Guard}_2(i, \mathsf{pk}_i, \mathsf{sk}_i, z, \Gamma_Y)$ is $o(T)$, where T is the time (or circuit size) required to compute $\mathsf{nmf}_i(x, \mathsf{trans}, \mathsf{st})$.*

4.1 Our Self-Guarding Model for 2PC

The ideal world experiment is identical to the standard ideal world in stand-alone 2PC (Sec. 2.3).

We now augment the real world with a self-guarder $\mathcal{SG}^j = (\mathsf{Setup}^j, \mathsf{Guard}_1^j, \mathsf{Guard}_2^j)$ for each of the cores Ψ_j, $j \in \{1,2\}$.

Let $2n$ be the number of protocol rounds. The odd messages $\{m_{2k+1}\}$ are generated by P_1 and sent to P_2, whereas even messages $\{m_{2k}\}$ are generated by party P_2 and sent to P_1.

The real execution is divided into the following phases:

1) **Setup phase**: During the setup phase, for each $j \in [2]$ runs $\{(\mathsf{pk}_i^j, \mathsf{sk}_i^j)\} \leftarrow\!\$\ \mathsf{Setup}^j(1^\lambda)$ and send $\{(\mathsf{pk}_i^j, \mathsf{sk}_i^j)\}_{i\in[n]}$ to P_j.
2) **Malicious corruption**: The following steps are performed:
   (a) The adversary maliciously corrupt a party P_i.
   (b) **(substitution)** The adversary substitutes the core Ψ_j (with $j \in [2]$ and $j \neq i$) of P_j with $\widetilde{\Psi}_j = (\widetilde{\mathsf{nmf}}_1^j, \ldots, \widetilde{\mathsf{nmf}}_n^j)$ — i.e., their next-message functions.
3) **Protocol Run**: For each $k \in [n]$, let $\widetilde{\mathsf{nmf}}_k^j$ be the substituted next-message function of Ψ_j for computing the corresponding message m_{2k} if $j = 2$ and m_{2k+1} if $j = 1$. Party P_j will compute the guarded next-message as follows:
   – Compute $(\Gamma_X, z) \leftarrow \mathsf{Guard}_1(k, \mathsf{pk}_k^j, \mathsf{sk}_k^j, x, \mathsf{trans}_{2k-1}, \mathsf{st}_{k-1}^j)$,
   – Obtain $(\Gamma_Y, \cdot) \leftarrow \widetilde{\mathsf{nmf}}_k^j(\Gamma_X, \mathsf{trans}_{2k-1}, \mathsf{st}_{k-1}^j)$ (trans_{2k} if $j = 1$),
   – Compute $(m_{2k}, \mathsf{st}_k^j) \leftarrow \mathsf{Guard}_2(k, \mathsf{pk}_k^j, \mathsf{sk}_k^j, z, \Gamma_Y)$ (m_{2k+1} if $j = 1$). If it outputs $(\perp, \cdot)$, abort the protocol execution.

The ideal execution of $\mathcal{F}$ on inputs (x_1, x_2), auxiliary input z for $\mathcal{A}$, and security parameter λ, denoted as $\mathsf{IDEAL}_{\mathcal{F},\mathcal{A}(z),i}(x_1, x_2, \lambda)$, is the output pair of the honest party's outputs and the adversary's output.

We call the real execution of Π on inputs (x_1, x_2), auxiliary input z to $\mathcal{A}$ security parameter λ, denoted as $\mathsf{sREAL}^{\mathcal{SG}}_{\Pi,\mathcal{A}(z),i}(x_1, x_2, \lambda)$, the pair of honest party's output and the adversary's output from the real execution of Π. $\mathcal{A}$ maliciously corrupts P_i and substitutes the core Ψ_j of P_j (with $j \in [2]$, $j \neq i$).

Definition 15 (Self-guarding 2PC Security). *Let Π be a two-party protocol and $\mathcal{SG}^i = (\mathsf{Setup}^i, \mathsf{Guard}_1^i, \mathsf{Guard}_2^i)$ be simple (Def. 14) Self-Guarders for the cores (Ψ_1, Ψ_2) of Π.*

We say that Π securely realizes $\mathcal{F}$ in the presence of static malicious corruptions and arbitrary substitution *with self-guarders $\mathcal{SG} = (\mathcal{SG}^1, \mathcal{SG}^2)$ if, for all $i \in [2]$, for all* PPT *adversary $\mathcal{A}$, there exists a* PPT *simulator $\mathcal{S}$, the following holds:*

$$\left\{\mathsf{IDEAL}_{\mathcal{F},\mathcal{S}(z),i}(\lambda, x_1, x_2)\right\}_{\lambda,x_1,x_2,z} \overset{c}{\approx} \left\{\mathsf{sREAL}^{\mathcal{SG}}_{\Pi,\mathcal{A}(z),i}(\lambda, x_1, x_2)\right\}_{\lambda,x_1,x_2,z}$$

where $x_1, x_2 \in \{0,1\}^*$ with $|x_1| = |x_2|$, $j \neq i$, $\lambda \in \mathbb{N}$, and $z \in \{0,1\}^*$

4.2 Composing multiple protocols

Definition 16 ($\mathcal{G}$-hybrid model). *Let Π_g be a protocol that realizes an ideal functionality $\mathcal{G}$ under malicious corruptions and arbitrary substitutions.*

Let Π_f be a protocol that, for all PPT *adversaries $\mathcal{A}$, realizes an ideal functionality $\mathcal{F}$ under malicious corruptions and arbitrary substitutions, but internally uses Π_g. Then, for all parties $i \in [2]$, inputs $x_1, x_2 \in \{0,1\}^*$ such that $|x_1| = |x_2|$, and auxiliary input $z \in \{0,1\}^*$, define:*

$$\mathsf{HYBRID}^{\mathcal{G}}_{\Pi_f,\mathcal{A}(z),i}(\lambda, x_1, x_2)$$

As the joint distribution of the outputs of Π_f as in $\mathsf{sREAL}_{\Pi_f,\mathcal{A}(z),i}(\lambda, x_1, x_2)$, but where each usage of Π_g is substituted with ideal calls to the functionality $\mathcal{G}$.

Additionally, denote this model as the $\mathcal{G}$-hybrid *model.*

Theorem 2 (Sequential Composition Theorem). *Let Π_f and Π_g be two protocols that securely realize, respectively, the functionalities $\mathcal{F}$ and $\mathcal{G}$ under malicious corruptions and arbitrary substitutions, such that Π_f is in the $\mathcal{G}$-hybrid model. Then, for all* PPT *adversaries $\mathcal{A}$, for all parties, and for all inputs $x_1, x_2 \in \{0,1\}^*$ such that $|x_1| = |x_2|$, for all auxiliary inputs $z \in \{0,1\}^*$ and for all $i \in [2]$, the following holds:*

$$\mathsf{sREAL}^{\mathcal{SG}}_{\Pi_f,\mathcal{A}(z),i}(\lambda, x_1, x_2) \overset{c}{\approx} \mathsf{HYBRID}^{\mathcal{G}}_{\Pi_f,\mathcal{A}(z),i}(\lambda, x_1, x_2)$$

Proof. We have to show that we can build a PPT simulator $\mathcal{S}$ that can simulate the execution in sREAL from the one in the $\mathcal{G}$-hybrid model. However, since Π_g realizes $\mathcal{G}$ and Π_f realizes $\mathcal{F}$, we have access to the underlying simulators $\mathcal{S}_g$ and $\mathcal{S}_f$; we can directly use $\mathcal{S}_f$, but use $\mathcal{S}_g$ when running the parts of Π_g included in Π_f. Hence, the proof follows the same strategy of [13].

4.3 A Black-box 2PC Self-Guarder from Verifiable-Computation

We propose a universal way to prevent substitution attacks by assuming a trusted implementation of *Verifiable Computation* (VC), introduced in Sec. 2.4.

In our setting, we exploit the following observation: a Self-Guarder $\mathcal{SG}$ for a protocol Π is "outsourcing" each computation to each nmf of the protocol; if we take the assumption on VC and require that each nmf outputs a VC proof, the Self-Guarder can simply verify the proofs in each Guard algorithm. This technique enjoys two interesting properties: first, we do not have any restrictions on the number of executions of the protocol, since the Self-Guarder can always verify the proofs; secondly, by the soundness of VC we can prevent arbitrary substitutions.

We first build a compiler for any protocol Π that wraps it with a VC scheme, to allow the $\mathcal{SG}$ to later verify it.

Our Black-box compiler. Given a protocol Π, we wish to use a VC scheme $\mathcal{VC}$ to build a new $\Pi_{\mathcal{VC}}$ that simply executes each $\Pi.\mathsf{nmf}$ under VC.

Our compiler, described as follows, will convert a standard protocol into a Verifiable-Computation-enabled one.

Construction 3 (VC compiler) *Let Π be a protocol realizing a functionality $\mathcal{F}$, with cores $\Psi_1 = \left(\mathsf{nmf}_1^1, \ldots, \mathsf{nmf}_n^1\right)$ and $\Psi_2 = \left(\mathsf{nmf}_1^2, \ldots, \mathsf{nmf}_n^2\right)$, and let $\mathcal{VC} = (\mathsf{KGen}, \mathsf{ProbGen}, \mathsf{Compute}, \mathsf{Vf})$ a VC scheme. Our compiler outputs, for each $j \in [2]$, compiled core $\Psi'_j = \left(\mathsf{Setup}^j, \mathsf{nmf}_1'^j, \ldots, \mathsf{nmf}_n'^j\right)$ as follows:*

- $\mathsf{Setup}^j(1^\lambda)$*:*
  On input the security parameter, the setup algorithm works as follows. For each $i \in [n]$, generate $(\mathsf{pk}_i^j, \mathsf{sk}_i^j) \leftarrow\!\!\$\ \mathcal{VC}.\mathsf{KGen}(\mathsf{nmf}_i^j)$ and output $\{(\mathsf{pk}_i^j, \mathsf{sk}_i^j)\}_{i\in[n]}$
- $\mathsf{nmf}_i'^j((\mathsf{pk}_i^j, \Sigma_X), \bot, \bot)$ *outputs $(\Sigma_Y, \bot)$, where $\Sigma_Y \leftarrow \mathcal{VC}.\mathsf{Compute}\left(\mathsf{pk}_i^j, \Sigma_X\right)$ for some $(\Sigma_X, \cdot) \in \mathsf{ProbGen}(\mathsf{pk}_i^j, (x_j, \mathsf{trans}_{r-1}, \mathsf{st}_{i-1}^j))$ with trans_{r-1} some previous transcript, st_{i-1}^j, a previous state with $r = 2i-1$ if $j = 1$ and $r = 2i$ if $j = 2$.*

We now build a self-guarder for any protocol with the transformation in Cons. 3.

Construction 4 (Self-Guarder from VC) *Let $\mathcal{VC} = (\mathsf{KGen}, \mathsf{ProbGen}, \mathsf{Compute}, \mathsf{Vf})$ be a VC scheme. We now construct a self-guarder $\mathcal{SG} = (\mathsf{Setup}, \mathsf{Guard}_1, \mathsf{Guard}_2)$ for a core $\Psi' = (\mathsf{Setup}, \mathsf{nmf}_1, \ldots, \mathsf{nmf}_n)$ obtained from a compiled protocol $\Pi_{\mathcal{VC}}$.*

- $\mathsf{Setup}(1^\lambda)$*: Output $\{(\mathsf{pk}_i, \mathsf{sk}_i)\}_{i\in[n]} \leftarrow\!\!\$\ \Psi'.\mathsf{Setup}(1^\lambda)$*
- $\mathsf{Guard}_1(i, \mathsf{pk}_i, \mathsf{sk}_i, x, \mathsf{trans}, \mathsf{st})$*: Output $((\mathsf{pk}_i, \Sigma_X), \mathsf{vk}_i) \leftarrow\!\!\$\ \mathcal{VC}.\mathsf{ProbGen}(\mathsf{sk}_i, (x, \mathsf{trans}, \mathsf{st}))$.*
- $\mathsf{Guard}_2(i, \mathsf{pk}_i, \mathsf{sk}_i, z, \Gamma_Y)$*: Let $z = \mathsf{vk}_x$. If $Y \leftarrow \mathcal{VC}.\mathsf{Vf}(\mathsf{sk}_i, \mathsf{vk}_x, \Gamma_Y)$ returns $\bot_f$ or $\bot_{\mathcal{VC}}$, then output $\bot$. Otherwise, output Y.*

To support re-usable setup, we now consider a new functionality wrapping the underlying ideal functionality for an unbounded number of executions.

Definition 17 (Multi-stage Functionality). *Let $\mathcal{F}$ be an ideal functionality, define $\hat{\mathcal{F}}_{\mathcal{SG}}(\mathcal{F})$ as an ideal functionality that wraps $\mathcal{F}$ such that it handles the following messages:*

1. $(\mathtt{sid}, \mathtt{corrupt}, P_j)$ *from* $\mathcal{A}$.
   When the adversary sends this message for a non-existent session `sid`*, record such session as fresh new and assume party P_j is maliciously corrupted. Accept this message only once for* `sid`.
   The functionality can now accept messages for `sid`.
2. $(\mathtt{sid}, \mathtt{input}, P_j, x)$ *from* P_j.
   Ignore the message if any of these conditions are true:
   (a) *The session* `sid` *does not exist (it was not started).*
   (b) *P_j already sent a* $(\mathtt{sid}, \mathtt{input}, P_j, \cdot)$ *for the same session* `sid`.
   Otherwise, store P_j's input x_j, send it to the underlying functionality $\mathcal{F}$ and relay its messages; if the reply is an `abort`*, send* `abort` *to every party (adversary included), mark this session* `sid` *as* `expired` *and ignore further messages for* `sid`.
3. $(\mathtt{sid}, \mathtt{output}, P_j)$ *from* P_j.
   Ignore the message if any of these conditions are true:
   (a) *The session* `sid` *does not exist (was never started).*
   (b) *The session* `sid` *is already marked as* `expired`.
   (c) *P_j already sent a* $(\mathtt{sid}, \mathtt{output}, P_j, \cdot)$ *for the same session* `sid`.
   Otherwise, ask $\mathcal{F}$ for P_j's output and relay it accordingly. When every party obtains its output (or an abort occurs), mark the session `sid` *as expired and ignore any further messages related to* `sid`.

We now prove that the self-guarder in Cons. 4 is secure w.r.t. the above functionality.

Theorem 3. *Let Π be a protocol that realizes functionality $\mathcal{F}$ in the plain ideal model under malicious corruptions and let Π', having cores (Ψ'_1, Ψ'_2) be the protocol obtained by applying compiler of Cons. 3 to Π.*
for $j \in [2]$, let $\mathcal{SG}^j$ be the Self-Guarder of Cons. 4 for a verifiable computation scheme $\mathcal{VC}$ (Def. 4) and core Ψ'_j. If $\mathcal{VC}$ is correct (Def. 5), sound (Def. 6), and input-private (Def. 7), then, Π' realizes $\hat{\mathcal{F}}(\mathcal{F})$ in the presence of static malicious corruptions and arbitrary substitutions with self-guarders $(\mathcal{SG}^1, \mathcal{SG}^2)$ (Def. 15).

Proof. The full proof is done by hybrid argument on the number of nmfs; that is, we would start with the completely arbitrarily-substituted protocol (all $\widetilde{\mathsf{nmf}}$s provided by the adversary), and then in each hybrid we first switch these $\widetilde{\mathsf{nmf}}$s into genuine nmfs one by one by using the simulator for the underlying $\mathcal{S}$, except for the aborting conditions that are still caused by the subverted algorithms. The reduction follows from the soundness of the VC. When all nmfs are genuine, then the party behaves as an honest one (except for the aborting conditions). Reduction follows straightforwardly to the underlying simulator. To show that aborts due to the incorrect computation caused by the subverted algorithms do not undermine the protocol security, we rely on the input privacy of $\mathcal{VC}$ to switch to a new series of hybrids where aborts are not generated by the $\widetilde{\mathsf{nmf}}$s on the party's input, but from random independent inputs.

For the aim of simplicity, we will consider an adversary who corrupts a party P_i and subverts the P_j in all executions. However, the proof can be easily extended to the case of alternating corruptions using similar arguments (and assuming that each party acts, in fact, as a fresh party with an empty state). Denote as $\ell = \mathsf{poly}(\lambda)$ the maximum number of executions made by the parties.

– $\underline{\mathsf{sREAL}_{\Pi,\mathcal{A},i}(\lambda, x_1, x_2)}$: This is the original real-world scenario with P_i corrupted and P_j subverted. Let us consider $\left\{\widetilde{\mathsf{nmf}}_k^j\right\}_{k\in[n]}$ from $\mathcal{A}$ the subverted next-message function used by P_j for all the executions $l \in [\ell]$.

– $\underline{\mathsf{HybSound}_i^j(\lambda, x_1, x_2)}$: We still have the original party P_j with its input x_j that behaves as in the real execution; except that for all $k \in [i]$, it checks if the output of $\widetilde{\mathsf{nmf}}_k^j$ was correctly computed using $\mathcal{VC}.\mathsf{Vf}$. If $\mathcal{VC}.\mathsf{Vf}$ did not output $\perp_{\mathcal{VC}}$, sends to the adversary the output of the genuine nmf_k^j. More formally, for all $l \in [\ell]$ executions, in the generic k-th out of the first i nmf^js, with $r = 2k-1$ being the round number:

1. Let $X = \left(x_j, \mathsf{trans}_{r-1}, \mathsf{st}_{k-1}^j\right)$. Compute the encoded input $\left(\Sigma_{X_k}, \mathsf{vk}_k^j\right) \leftarrow\!\$\ \mathsf{Guard}_1(k, \mathsf{pk}_k^j, \mathsf{sk}_k^j, X)$.
2. Call $(\Sigma_y, \cdot) \leftarrow\!\$\ \widetilde{\mathsf{nmf}}_k^j((\mathsf{pk}_k^j, \Sigma_{X_k}), \perp, \perp)$.
3. Call $y \leftarrow \mathsf{Guard}_2^j(k, \mathsf{pk}_k^j, \mathsf{sk}_k^j, \mathsf{vk}_k^j, \Sigma_y)$ (that is $\mathcal{VC}.\mathsf{Vf}(\mathsf{sk}_k^j, \mathsf{vk}_k^j, \Sigma_y)$) and if $y \stackrel{?}{=} \perp$, abort the execution.
4. Call $\left(m_r, \mathsf{st}_k^j\right) \leftarrow\!\$\ \mathsf{nmf}_k^j(x_j, \mathsf{trans}_{r-1}, \mathsf{st}_{k-1}^j)$ and send m_r to $\mathcal{A}$.
5. If $r \neq 2n+1$, receive m_{r+1} from $\mathcal{A}$.

At the end, the party outputs the transcript of the overall ℓ-stage execution.

– $\underline{\mathsf{HybSim}^j(\lambda, x_1, x_2)}$:
The party P_j behaves as in the last $\mathsf{HybSound}_n^j$, i.e., *all* its replies to the adversary go through the genuine nmfs. However, now it has access to the underlying simulator $\mathcal{S}_{\Pi}$ for Π, so it switches all its replies to $\mathcal{A}$ to use messages generated by $\mathcal{S}_{\Pi}$.
For all executions $l \in [\ell]$, for each $i \in [n]$, letting $r = 2i-1$, it does the following:

1. Denote $X = \left(x_j, \mathsf{trans}_{r-1}, \mathsf{st}_{i-1}^j\right)$, then compute the encoded input $\left((\mathsf{pk}_i^j, \Sigma_{X_i}), \mathsf{vk}_i^j\right) \leftarrow\!\$\ \mathsf{Guard}_1^j(i, \mathsf{pk}_i^j, \mathsf{sk}_i^j, X)$.
2. Call $(\Sigma_y, \cdot) \leftarrow\!\$\ \widetilde{\mathsf{nmf}}_i^j((\mathsf{pk}_i^j, \Sigma_{X_i}), \perp, \perp)$.
3. Call $y \leftarrow \mathsf{Guard}_2^j(i, \mathsf{pk}_i^j, \mathsf{sk}_i^j, \mathsf{vk}_i^j, \Sigma_y)$, and, if $y \stackrel{?}{=} \perp$, abort.

Output the output of the genuine nmf^js for all the executions, except that in any execution $l \in [\ell]$ that was aborted as above, $\perp$ is put in the place of l-th transcript.

– $\underline{\mathsf{HybInputPrivacy}_{i,l}^j(\lambda, x_1, x_2)}$: This is the same as HybSim^j, except that P_j, for the first i out of n nmfs and the first l executions, additionally picks a random input in the support of nmf_j^i. P_j will use a dummy input $X_{i,l,\$}^j$ to check the aborting case through the VC, while for the remaining $\ell - l$ executions of the i-th nmf and all the executions of the remaining $n-i$ nmfs, it will use its input x_j^l:

1. P_j runs the setup phase, i.e., stores $\left\{(\mathsf{pk}_i^j, \mathsf{sk}_i^j) \mid (\mathsf{pk}_i^j, \mathsf{sk}_i^j) \leftarrow\!\$\ \mathcal{VC}.\mathsf{KGen}(1^\lambda, \mathsf{nmf}_i^j))\right\}_{i \in [n]}$
2. P_j receives $\left\{\widetilde{\mathsf{nmf}}_i^j\right\}_{i \in [n]}$ from $\mathcal{A}$.
3. For all first $t \in [l]$ executions P_j behave as follows:
   (a) $K = \begin{cases} n & \text{if } t < l \\ i & \text{if } t = l \end{cases}$
   (b) For $k \in [K]$, with $r = 2k-1$ being the round number:
      i. Pick a random input $X_{\$}$ in the support of nmf_k^j.
      ii. Compute the encoded random input $\left((\mathsf{pk}_k^j, \Sigma_{X_{\$}}), \mathsf{vk}_{k,\$}^j\right) \leftarrow\!\$\ \mathsf{Guard}_1^j(k, \mathsf{pk}_k^j, \mathsf{sk}_k^j, X_{\$})$.
      iii. Call $(\Sigma_{y_{\$}}, \cdot) \leftarrow\!\$\ \widetilde{\mathsf{nmf}}_k^j((\mathsf{pk}_k^j, \Sigma_{X_{\$}}), \perp, \perp)$ and $y_{\$} \leftarrow \mathsf{Guard}_2(k, \mathsf{pk}_k^j, \mathsf{sk}_k^j, \mathsf{vk}_{k,\$}^j, \Sigma_{y_{\$}})$.
      iv. If $y_{\$} \stackrel{?}{=} \perp$, then abort the execution.

   The remaining $n-i$ nmfs in execution l and the next $\ell - l$ executions of the $i-th$ nmf are as in HybSim^j.

Output the output of the simulator for all the executions, except that in any execution $l \in [\ell]$ that were aborted as above, $\perp$ is put in the place of l-th transcript.

– $\underline{\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},i}(\lambda, x_1, x_2)}$: We completely substituted the party's input to the nmf with a dummy input; here we finally introduce a PPT simulator $\mathcal{S}$ that talks to $\hat{\mathcal{F}}_{\mathcal{SG}}(\mathcal{F})$, internally runs the adversary $\mathcal{A}$ and has access to the underlying simulator $\mathcal{S}_{\Pi}$ for $\mathcal{F}$. It works as follows:

1. Runs the setup phase to generate, $\forall i \in [n]$,
   $\left\{ \left(\mathsf{pk}_i^j, \mathsf{sk}_i^j\right) \;\middle|\; \left(\mathsf{pk}_i^j, \mathsf{sk}_i^j\right) \leftarrow\!\$\; \mathcal{VC}.\mathsf{KGen}(1^\lambda, \mathsf{nmf}_i) \right\}$.
2. Runs $\mathcal{A}$ and receives $\left\{\widetilde{\mathsf{nmf}}_i^j\right\}_{i\in[n]}$.
3. For all executions $l \in [\ell]$:
   (a) Sends $(\texttt{sid} = l, \texttt{corrupt}, P_{3-j})$ to the ideal functionality.
   (b) For $k \in [n]$, with $r = 2k-1$ being the current round number:
      i. Pick a random $X_\$$ in the support of nmf_k^j.
      ii. Encode the k-th input for $\widetilde{\mathsf{nmf}}_k^j$:
      $\left((\mathsf{pk}_k^j \Sigma_{X_\$}), \mathsf{vk}_{k,\$}^j\right) \leftarrow\!\$\; \mathsf{Guard}_1^j(k, \mathsf{pk}_k^j, \mathsf{sk}_k^j, X_\$)$
      iii. Call $(\Sigma_y, \cdot) \leftarrow\!\$\; \widetilde{\mathsf{nmf}}_k^j((\Sigma_X, \mathsf{pk}_k^j)), \bot, \bot)$.
      iv. If $\mathsf{Guard}_2^j(k, \mathsf{pk}_k^j, \mathsf{sk}_k^j, \mathsf{vk}_{k,\$}^j, \Sigma_y)$ outputs $\bot$, then abort the execution.
   (c) Invoke $\mathcal{S}$, and, when receiving the call to $\mathcal{F}$ including $\mathcal{A}$'s extracted inputs and the dummy input, relay the inputs to $\hat{\mathcal{F}}$ for $\texttt{sid} = l$ with $\texttt{input}$ command, receiving back y, which is, in turn, forwarded to the underlying simulator $\mathcal{S}_\Pi$, which will finally output the simulated transcript.

   Output the output of the simulator for all the executions, except that in any execution $l \in [\ell]$ that were aborted as above, $\bot$ is put in the place of l-th transcript.

Lemma 5. $\forall i \in [n],\ \mathsf{HybSound}_{i-1}^j(\lambda, x_1, x_2) \stackrel{c}{\approx} \mathsf{HybSound}_i^j(\lambda, x_1, x_2)$

Proof. The difference between the two hybrids is that we switch the i-th nmf from $\widetilde{\mathsf{nmf}}_i^j$ (provided by the adversary) to nmf_i^j (of the original Π). Since the adversary sees the abort generated by $\widetilde{\mathsf{nmf}}_i^j$ in both hybrids, we have to concern ourselves only with the output of y coming from $\mathcal{VC}.\mathsf{Vf}$. The probability that $\mathcal{A}$ distinguishes the two hybrids is dependent on the probability that the output of the real nmf_i^j diverges from the (non-$\bot$) result of $\mathcal{VC}$ on the output encoding computed by a subverted $\widetilde{\mathsf{nmf}}_i^j$. Let $r = 2i-1$ be the round number, let Σ_X be the encoding for $X = \left(x_j, \mathsf{trans}_{r-1}, \mathsf{st}_{i-1}^j\right)$ given as input to $\widetilde{\mathsf{nmf}}_i^j$, call **BAD** the following event:

$$
\begin{aligned}
&\left(m_r, \mathsf{st}_i^j\right) \leftarrow \mathsf{nmf}_i^j(x_j, \mathsf{trans}_{r-1}, \mathsf{st}_{i-1}^j) \\
\wedge\ &\left(\Sigma_X, \mathsf{vk}_i^j\right) \leftarrow\!\$\; \mathcal{VC}.\mathsf{ProbGen}(\mathsf{sk}_i^j, X) \\
\wedge\ &(\Sigma_y, \cdot) \leftarrow \widetilde{\mathsf{nmf}}_i^j((\Sigma_X, \mathsf{pk}_i^j), \bot, \bot) \\
\wedge\ &y \leftarrow \mathcal{VC}.\mathsf{Vf}(\mathsf{sk}_i^j, \mathsf{vk}_i^j, \Sigma_y) \\
\wedge\ &y \neq \bot_{\mathcal{VC}} \ \wedge\ y \neq \left(m_r, \mathsf{st}_i^j\right)
\end{aligned}
$$

Note that the output of $\mathsf{ProbGen}$ and Vf are exactly the outputs Guard_1 and Guard_2. The adversary has to notice a difference in *any* one of the ℓ executions, so we have to account for a $\mathsf{poly}(\ell)$ slack. The probability that **BAD** happens is exactly the same as the probability of the adversary breaking soundness: the adversary is tasked to provide a $\widetilde{\mathsf{nmf}}_i^j$ such that its output differs from the original nmf_i^j, which describes both **BAD** and exactly the soundness game. However, by hypothesis we assumed that $\mathcal{VC}$ is sound; so $\Pr[\mathbf{BAD}] \leq \mathsf{negl}(\lambda)$, and thus the probability that $\mathcal{A}$ distinguishes after ℓ executions is $\Pr[\mathcal{A}\text{ wins}] \leq \mathsf{poly}(\ell) \cdot \Pr[\mathbf{BAD}] \leq \mathsf{poly}(\ell) \cdot \mathsf{negl}(\lambda) \approx \mathsf{negl}(\lambda)$

Lemma 6. $\mathsf{HybSound}_n^j(\lambda, x_1, x_2) \stackrel{c}{\approx} \mathsf{HybSim}^j(\lambda, x_1, x_2)$

Proof. The difference between the two hybrids is that we are switching the real messages coming from each genuine nmf, to the simulated messages coming from the simulator $\mathcal{S}_\Pi$. The distributions are indistinguishable

from the point of view of the adversary, by the assumption on Π securely realizing $\mathcal{F}$, thus by the existence of $\mathcal{S}_\Pi$.

Lemma 7. $\forall l \in [\ell]$, $\forall i \in [0,n]$, $\mathsf{HybInputPrivacy}^j_{i,l-1}(\lambda, x_1, x_2) \overset{c}{\approx} \mathsf{HybInputPrivacy}^j_{i,l}(\lambda, x_1, x_2)$

Proof. In the second hybrid, we make use of a dummy random input when calling the first i $\widetilde{\mathsf{nmf}}^j$s. It may happen that, for a dummy input, any of the first i $\widetilde{\mathsf{nmf}}^j$s aborts, allowing the adversary to distinguish between the two hybrids due to the difference in abort behavior. We show that, due to the hypothesis on $\mathcal{VC}$ being input-private, this cannot happen.

Assume by contradiction that there exists a PPT adversary $\mathcal{A}$ that can distinguish between the two hybrids, then we can build the following PPT reduction $\mathcal{B}$ for input-privacy of $\mathcal{VC}$:

1. Generate $\left\{ \left(\mathsf{pk}^j_k, \mathsf{sk}^j_k\right) \;\middle|\; \left(\mathsf{pk}^j_k, \mathsf{sk}^j_k\right) \leftarrow\$ \mathcal{VC}.\mathsf{KGen}(1^\lambda, \mathsf{nmf}^j_k)_{k\in[n]\setminus\{i\}} \right\}$
2. Receive pk^j_i from the input-privacy challenger.
3. Receive the set of nmfs $\left\{\widetilde{\mathsf{nmf}}^j_k\right\}_{k\in[n]}$ by $\mathcal{A}$.
4. For all executions $t \in [\ell]$, For $k \in [n]$, with $r = 2k-1$ as round number:
   - For $s \in [n]$, let $X^j_{s,t} = \left(x^t_j, \mathsf{trans}^t_{r-1}, \mathsf{st}^j_{k-1,t}\right)$ with $r = 2s-1$ if $j = 1$ and $r = 2s$ if $j = 2$ where trans^t_{s-1} and $\mathsf{st}^j_{r-1,t}$ are the output of the execution up to round $r-1$ of the t-th execution of $\mathsf{HybSound}^j_n$, and $X_{s,t,\$}$ be a random value in the support of the j-th nmf for round j $\mathsf{nmf}^j_{s,t}$.
   - if $\mathsf{nmf}^j_s(X^j_{s,t}) \overset{?}{=} \perp_f$, abort the execution.
   - if $k < i$: Generate $\left(\Sigma_{X_{k,t,\$}}, \mathsf{vk}^j_k\right) \leftarrow\$ \mathcal{VC}.\mathsf{ProbGen}(\mathsf{pk}^j_k, \mathsf{sk}_k, X_{k,t,\$})$, then call $\Sigma_y \leftarrow \widetilde{\mathsf{nmf}}^j_k((\Sigma_{X_{k,t,\$}}, \mathsf{pk}^j_k), \perp, \perp)$, and abort if $\mathcal{VC}.\mathsf{Vf}(\mathsf{sk}^j_k, \mathsf{vk}^j_k, \Sigma_y)$ outputs $\perp_{\mathcal{VC}}$.
   - if $k > i$: Generate $\left(\Sigma_{X_{k,t}}, \mathsf{vk}^j_k\right) \leftarrow\$ \mathcal{VC}.\mathsf{ProbGen}(\mathsf{pk}^j_k, \mathsf{sk}_k, X_{k,t})$, then call $\Sigma_y \leftarrow \widetilde{\mathsf{nmf}}^j_k((\Sigma_{X_{k,t}}, \mathsf{pk}^j_k), \perp, \perp)$, and abort if $\mathcal{VC}.\mathsf{Vf}(\mathsf{sk}^j_k, \mathsf{vk}^j_k, \Sigma_y)$ outputs $\perp_{\mathcal{VC}}$.
   - if $k \neq i \;\wedge\; t > l$: Generate $\left(\Sigma_{X_{k,t}}, \mathsf{vk}^j_k\right) \leftarrow\$ \mathcal{VC}.\mathsf{ProbGen}(\mathsf{pk}^j_k, \mathsf{sk}^j_k, X_{k,t})$, then call $\Sigma_y \leftarrow \widetilde{\mathsf{nmf}}^j_{k,t}((\Sigma_{X_{k,t}}, \mathsf{pk}^j_k), \perp, \perp)$, and abort if $\mathcal{VC}.\mathsf{Vf}(\mathsf{sk}^j_k, \mathsf{vk}^j_k, \Sigma_y)$ outputs $\perp_{\mathcal{VC}}$.
   - if $k = i \;\wedge\; t < l$:
     - Send $X_{i,t,\$}$ as an input-encoding query to the challenger and receive back $\Sigma_{X_{i,t,\$}}$.
     - Compute $\Sigma_{y_\$} \leftarrow \widetilde{\mathsf{nmf}}^j_{i,t}((\Sigma_{X_{i,t,\$}}, \mathsf{pk}^j_i), \perp, \perp)$
     - Send $\Sigma_{y_\$}$ as a verification query to the challenger, abort the execution if it replies with 0.
   - if $k = i \wedge\; t = l$:
     - Send $(X_{i,l}, X_{i,l,\$})$ to the input-privacy challenger, receiving back an encoding Σ_{X^*}.
     - Compute the output $\Sigma_{y^*} \leftarrow\$ \widetilde{\mathsf{nmf}}^j_i((\mathsf{pk}^j_i, \Sigma_{X^*}), \perp, \perp)$ and make a verification query by sending Σ_{y^*} to the input-privacy challenger, receiving back a bit b. If $b \overset{?}{=} 0$, abort the execution.
   - if $k = i \;\wedge\; t > l$:
     - Send $X_{i,t}$ as an input-encoding query to the challenger and receive back $\Sigma_{X_{i,t}}$.
     - Compute $\Sigma_y \leftarrow \widetilde{\mathsf{nmf}}^j_i((\Sigma_{X_{i,t}}, \mathsf{pk}^j_i), \perp, \perp)$
     - Send Σ_y as a verification query to the challenger, abort if it replies with 0.

   Output the output of the simulator for all the executions, except that in any execution $l \in [\ell]$ that was aborted as above, $\perp$ is put in the place of l-th transcript.

Notice that $\mathcal{B}$ perfectly simulates $\mathsf{HybInputPrivacy}^j_{i,l-1}$ when the input-privacy challenger selects the input $X_{i,l}$, whereas it perfectly simulates $\mathsf{HybInputPrivacy}^j_{i,l}$ when the input-privacy challenger selects the input $X_{i,l,\$}$.

Hence, $\mathcal{B}$ breaks $\mathcal{VC}$'s input-privacy with the same probability of $\mathcal{A}$ distinguishing between the two distributions.

Lemma 8. $\mathsf{HybInputPrivacy}^{j}_{n,\ell}(\lambda, x_1, x_2) \equiv \mathsf{IDEAL}^{\mathcal{SG}}_{\mathcal{F},\mathcal{S},i}(\lambda, x_1, x_2)$

Proof. Finally, we have the actual simulator, which makes calls to the ideal functionality and tries to extract $\mathcal{A}$'s input. From the point of view of the adversary, the messages they receive in both hybrids are identically distributed since they are computed exactly as in $\mathsf{HybInputPrivacy}^{j}_{n}$.

Lemma 9. *The Self-Guarders $(\mathcal{SG}^1, \mathcal{SG}^2)$ in Cons. 4 are simple.*

Proof. The proof follows directly from the outsourceability property of $\mathcal{VC}$. In fact, Guard^{j}_{1} performs input encodings via $\mathcal{VC}.\mathsf{ProbGen}$, and Guard^{j}_{2} performs encoding verifications via $\mathcal{VC}.\mathsf{Vfs}$. Since $\mathcal{SG}$ uses the outputs of the $\widetilde{\mathsf{nmfs}}$ and due to $\mathcal{VC}$'s outsourceability, the cost of the $\mathcal{VC}.\mathsf{ProbGen}$ and $\mathcal{VC}.\mathsf{Vf}$ algorithms is $o(T)$ where T is the running time of $\widetilde{\mathsf{nmf}}^{j}_{i}$ for each $i \in [n]$ and $j \in [2]$.

Proof. By hybrid argument, using the previous lemmas, we have that:

$$
\begin{aligned}
\mathsf{sREAL} &\equiv \mathsf{HybSound}^{j}_{0} \\
&\overset{c}{\approx} \mathsf{HybSound}^{j}_{1} \ldots \overset{c}{\approx} \mathsf{HybSound}^{j}_{n} \\
&\overset{c}{\approx} \mathsf{HybSim}^{j} \\
&\equiv \mathsf{HybInputPrivacy}^{j}_{0,0} \\
&\overset{c}{\approx} \mathsf{HybInputPrivacy}^{j}_{0,1} \ldots \overset{c}{\approx} \mathsf{HybInputPrivacy}^{j}_{0,\ell} \\
&\equiv \mathsf{HybInputPrivacy}^{j}_{1,0} \ldots \overset{c}{\approx} \mathsf{HybInputPrivacy}^{j}_{n,\ell} \\
&\equiv \mathsf{IDEAL}
\end{aligned}
$$

The above argument, together with Lemma 9, concludes the proof of the theorem.

5 Efficiency Estimates and Extensions

5.1 Efficiency

To practically instantiate both the black-box VC compiler and the SG construction, the only known method, at the time of writing, is by relying on Verifiable Fully-Homomorphic Encryption (vFHE) schemes, of which we highlight the following results: Viand at al. [26] discuss the primitive at length and provide instantiations from BGV [11] and different zero-knowledge succinct proof systems; while the current state of the art is, to our knowledge, Cascudo et al. result [15,16] from 2025, based on CKKS [19] with Brakedown [25] and sum-check [12]. In all cases, the algorithms for the Self-Guarder, Guard_1 and Guard_2, would amount to the following:

1. Guard_1 encrypts the input for FHE.
2. Each (VC-compiled) next-message function runs the vFHE algorithm which evaluates the underlying algorithm and simultaneously proves the correctness of the circuit's evaluation with some (succinct) proof system.
3. Guard_2 verifies the proof and decrypts the ciphertext.

$\mathcal{VC}.\mathsf{Compute}$ dominates the runtime of the overall compiled protocols and is the main bottleneck, while the encryption and SNARK/SNARG verification procedures are overall negligible.

For specific primitives such as the ones based on simple matrix multiplication, like Ajtai's hash function [2] or Regev's encryption scheme [30], we can rely on the result from Thibault at al. [32] to efficiently run proofs on FHE-encrypted hashes/ciphertexts. Let $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^N + 1)$. On a 16-thread AMD Ryzen 7 PRO 8840U laptop, by massaging the benchmarks' code of [32], for $N = 1024$ coefficients, $n = 256$, $m = 8196$ ($t = 2^3 \implies m = t \cdot N$), we can encrypt a value $\boldsymbol{m} \leftarrow\!\!\$\ \mathcal{R}_q^{8196}$, and evaluate, prove and verify

the multiplication of a matrix $A \leftarrow\!\!\$\ \mathcal{R}_q^{256\times 8196}$ with $\boldsymbol{m}$. Evaluation and proof generation combined (the $\mathcal{VC}.\mathsf{Compute}$) take an average of *1.2596s*, whereas encryption (our Guard_1) take on average *349.9µs*, while verification and decryption (our Guard_2 combined) take on average *8.9769ms* and *5.4891ms* respectively. This result is useful for the simple Ajtai's hash function, and we expect Regev's cryptosystem to yield slightly worse but comparable prover times.

However, for full-on arbitrary functions, we need to resort to the generic constructions in [26,15] which, for similar circuits, incurs in considerable prover time ($\mathcal{VC}.\mathsf{Compute}$) in the order of at most 25 seconds [15, Sec. 6.3] for a multiplication gate, including operations such as flooding and mod-switching. Nonetheless, the literature on vFHE is slowly improving, and we are positive that future *generic* constructions for vFHE may follow a similar path of non-verifiable FHE in terms of efficiency enhancements.

5.2 Universal Composability

Universal Composability (UC) is an MPC security framework introduced by Ran Canetti in its seminal FOCS 2001 paper [14]. The problem with stand-alone security, considered in this paper, lies on the fact that it does not state anything about security when executing a protocol multiple times and/or when interleaved with multiple executions of itself or other protocols; the goal of UC is to bridge that gap by including a PPT Environment $\mathcal{Z}$ that watches over all interactions in all protocols.

An equivalent UC proof for Cons. 4 would follow a similar strategy of the proof of Thm. 3, except that now the protocol may be run concurrently multiple times and overlapped with other unrelated protocol executions due to the existence of the environment $\mathcal{Z}$. We note the following observations:

1. The Setup phase is executed *exactly once*, after which we reuse the generated keys in all executions.
2. VC's soundness and input-hiding games allow an adaptive number of challenger queries, that can be interleaved arbitrarily.
3. The simulator runs the underlying straightline-extractable simulator never rewinds the adversary, so the simulation inherits the underlying straight-line extractability.

Since $\hat{\mathcal{F}}_{\mathcal{SG}}(\mathcal{F})$ already supports concurrent executions (multiple sessions can be started at once), it should suffice to change the definitions of the hybrids such that the executions are not done sequentially but in parallel. The simulator would start multiple `sid`s in parallel and react accordingly.

5.3 Multiple parties

When extending to Multi-Party Computation (MPC), one should change the description of a core (Def. 12) and Self-Guarder (Def. 13) such that a nmf additionally specifies one or more recipients for the resulting messages. For example, let N be the number of parties and R be the number of rounds, then party P_i may have as its core $\Psi_i = \{\mathsf{nmf}_j^{i\to\{k\}_{k\neq i}}\}_{j\in[R]}$, which means that P_i will send messages m_k^j to party P_k for each $k \in [N] \setminus \{i\}$ and round $j \in [R]$.

Regarding security, assuming that an N-party protocol Π allows up to $T < N$ corruptions, the worst possible case would be that T parties are maliciously corrupted and the remaining $N-T$ suffer from arbitrary substitutions. Proving an equivalent multi-party formulation of Def. 15, in this setting, implies that the Self-Guarder converts the $N-T$ arbitrarily-substituted cores (living in sREAL), to $N-T$ honest cores (in the *plain* IDEAL world), which falls exactly in the security guarantee of the underlying protocol Π.

Regarding the VC-based construction given in this paper, the communication complexity in the MPC case remains the same as in the base protocol: since the Self-Guarder does not inject new messages, but only checks the proofs from the nmfs. For any other SG construction one may wish to build, communication overhead of SG has to be independently analyzed.

5.4 Handling library updates

It may be possible to show a stronger variant of Thm. 3, where the adversary sends new substituted $\widetilde{\mathsf{nmfs}}$ in each execution and not just once. Allowing this implies that the $\mathcal{SG}$ in Cons. 4 can protect the user in place of code updates that may potentially have backdoors. However, note that this is a double-edge sword: improvements in the new code will be rejected by $\mathcal{VC}.\mathsf{Vf}$ and thus the party will abort. This may or may not be an acceptable trade-off.

6 Acknowledgements

We would like to thank both the shepherd and the reviewers for their thoughtful comments. Moreover, Andrea Reale thanks Rahul Satish for the insightful conversations.

Daniele Venturi and Daniele Friolo are members of the Gruppo Nazionale Calcolo Scientifico Istituto Nazionale di Alta Matematica (GNCS-INdAM) and their research was supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union-NextGenerationEU. Daniele Venturi's research was also supported by project PARTHENON (B53D23013000006), under the MUR National Recovery and Resilience Plan funded by the European Union-NextGenerationEU, and by project BEAT, funded by Sapienza University of Rome.

References

1. Abdolmaleki, B., Fleischhacker, N., Goyal, V., Jain, A., Malavolta, G.: Steganography-free zero-knowledge. In: Kiltz, E., Vaikuntanathan, V. (eds.) TCC 2022: 20th Theory of Cryptography Conference, Part I. Lecture Notes in Computer Science, vol. 13747, pp. 143–172. Springer, Cham, Switzerland, Chicago, IL, USA (Nov 7–10, 2022). https://doi.org/10.1007/978-3-031-22318-1_6
2. Ajtai, M.: Generating hard instances of lattice problems (extended abstract). In: Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing. p. 99–108. STOC '96, Association for Computing Machinery, New York, NY, USA (1996). https://doi.org/10.1145/237814.237838, https://doi.org/10.1145/237814.237838
3. Arnold, P., Berndt, S., Müller-Quade, J., Ottenhues, A.: Protection against subversion corruptions via reverse firewalls in the plain universal composability framework. In: Fischlin, M., Moonsamy, V. (eds.) ACNS 2025: 23rd International Conference on Applied Cryptography and Network Security, Part II. Lecture Notes in Computer Science, vol. 15826, pp. 510–539. Springer, Cham, Switzerland, Munich, Germany (Jun 23–26, 2025). https://doi.org/10.1007/978-3-031-95764-2_20
4. Ateniese, G., Francati, D., Magri, B., Venturi, D.: Public immunization against complete subversion without random oracles. In: Deng, R.H., Gauthier-Umaña, V., Ochoa, M., Yung, M. (eds.) ACNS 2019: 17th International Conference on Applied Cryptography and Network Security. Lecture Notes in Computer Science, vol. 11464, pp. 465–485. Springer, Cham, Switzerland, Bogota, Colombia (Jun 5–7, 2019). https://doi.org/10.1007/978-3-030-21568-2_23
5. Ateniese, G., Kiayias, A., Magri, B., Tselekounis, Y., Venturi, D.: Secure outsourcing of cryptographic circuits manufacturing. In: Baek, J., Susilo, W., Kim, J. (eds.) ProvSec 2018: 12th International Conference on Provable Security. Lecture Notes in Computer Science, vol. 11192, pp. 75–93. Springer, Cham, Switzerland, Jeju, South Korea (Oct 25–28, 2018). https://doi.org/10.1007/978-3-030-01446-9_5
6. Ateniese, G., Magri, B., Venturi, D.: Subversion-resilient signature schemes. In: Ray, I., Li, N., Kruegel, C. (eds.) ACM CCS 2015: 22nd Conference on Computer and Communications Security. pp. 364–375. ACM Press, Denver, CO, USA (Oct 12–16, 2015). https://doi.org/10.1145/2810103.2813635
7. Bellare, M., Paterson, K.G., Rogaway, P.: Security of symmetric encryption against mass surveillance. In: Garay, J.A., Gennaro, R. (eds.) Advances in Cryptology – CRYPTO 2014, Part I. Lecture Notes in Computer Science, vol. 8616, pp. 1–19. Springer Berlin Heidelberg, Germany, Santa Barbara, CA, USA (Aug 17–21, 2014). https://doi.org/10.1007/978-3-662-44371-2_1
8. Bemmann, P., Berndt, S., Chen, R.: Subversion-resilient signatures without random oracles. In: Pöpper, C., Batina, L. (eds.) ACNS 2024: 22nd International Conference on Applied Cryptography and Network Security, Part I. Lecture Notes in Computer Science, vol. 14583, pp. 351–375. Springer, Cham, Switzerland, Abu Dhabi, UAE (Mar 5–8, 2024). https://doi.org/10.1007/978-3-031-54770-6_14

9. Berndt, S., Liskiewicz, M.: Algorithm substitution attacks from a steganographic perspective. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017: 24th Conference on Computer and Communications Security. pp. 1649–1660. ACM Press, Dallas, TX, USA (Oct 31 – Nov 2, 2017). https://doi.org/10.1145/3133956.3133981
10. Bernstein, D.J., Lange, T., Niederhagen, R.: Dual EC: A standardized back door. Cryptology ePrint Archive, Report 2015/767 (2015), https://eprint.iacr.org/2015/767
11. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference. p. 309–325. ITCS '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2090236.2090262, https://doi.org/10.1145/2090236.2090262
12. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference. p. 309–325. ITCS '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2090236.2090262, https://doi.org/10.1145/2090236.2090262
13. Canetti, R.: Security and composition of multiparty cryptographic protocols. Journal of Cryptology **13**(1), 143–202 (Jan 2000). https://doi.org/10.1007/s001459910006
14. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd Annual Symposium on Foundations of Computer Science. pp. 136–145. IEEE Computer Society Press, Las Vegas, NV, USA (Oct 14–17, 2001). https://doi.org/10.1109/SFCS.2001.959888
15. Cascudo, I., Costache, A., Cozzo, D., Fiore, D., Guimarães, A., Soria-Vazquez, E.: Verifiable computation for approximate homomorphic encryption schemes. In: Kalai, Y.T., Kamara, S.F. (eds.) Advances in Cryptology – CRYPTO 2025, Part VII. Lecture Notes in Computer Science, vol. 16006, pp. 643–677. Springer, Cham, Switzerland, Santa Barbara, CA, USA (Aug 17–21, 2025). https://doi.org/10.1007/978-3-032-01907-3_21
16. Cascudo, I., Costache, A., Cozzo, D., Fiore, D., Guimarães, A., Soria-Vazquez, E.: Verifiable computation for approximate homomorphic encryption schemes. Cryptology ePrint Archive, Report 2025/286 (2025), https://eprint.iacr.org/2025/286
17. Chakraborty, S., Magliocco, L., Magri, B., Venturi, D.: Key exchange in the post-snowden era: Universally composable subversion-resilient PAKE. In: Chung, K.M., Sasaki, Y. (eds.) Advances in Cryptology – ASIACRYPT 2024, Part V. Lecture Notes in Computer Science, vol. 15488, pp. 101–133. Springer, Singapore, Singapore, Kolkata, India (Dec 9–13, 2024). https://doi.org/10.1007/978-981-96-0935-2_4
18. Chakraborty, S., Magri, B., Nielsen, J.B., Venturi, D.: Universally composable subversion-resilient cryptography. In: Dunkelman, O., Dziembowski, S. (eds.) Advances in Cryptology – EUROCRYPT 2022, Part I. Lecture Notes in Computer Science, vol. 13275, pp. 272–302. Springer, Cham, Switzerland, Trondheim, Norway (May 30 – Jun 3, 2022). https://doi.org/10.1007/978-3-031-06944-4_10
19. Cheon, J.H., Kim, A., Kim, M., Song, Y.S.: Homomorphic encryption for arithmetic of approximate numbers. In: Takagi, T., Peyrin, T. (eds.) Advances in Cryptology – ASIACRYPT 2017, Part I. Lecture Notes in Computer Science, vol. 10624, pp. 409–437. Springer, Cham, Switzerland, Hong Kong, China (Dec 3–7, 2017). https://doi.org/10.1007/978-3-319-70694-8_15
20. Dodis, Y., Mironov, I., Stephens-Davidowitz, N.: Message transmission with reverse firewalls—secure communication on corrupted machines. In: Robshaw, M., Katz, J. (eds.) Advances in Cryptology – CRYPTO 2016, Part I. Lecture Notes in Computer Science, vol. 9814, pp. 341–372. Springer Berlin Heidelberg, Germany, Santa Barbara, CA, USA (Aug 14–18, 2016). https://doi.org/10.1007/978-3-662-53018-4_13
21. Fiore, D., Gennaro, R., Pastro, V.: Efficiently verifiable computation on encrypted data. In: Ahn, G.J., Yung, M., Li, N. (eds.) ACM CCS 2014: 21st Conference on Computer and Communications Security. pp. 844–855. ACM Press, Scottsdale, AZ, USA (Nov 3–7, 2014). https://doi.org/10.1145/2660267.2660366
22. Fischlin, M., Mazaheri, S.: Self-guarding cryptographic protocols against algorithm substitution attacks. In: Chong, S., Delaune, S. (eds.) CSF 2018: IEEE 31st Computer Security Foundations Symposium. pp. 76–90. IEEE Computer Society Press, Oxford, UK (Jul 9–12, 2018). https://doi.org/10.1109/CSF.2018.00013
23. Gennaro, R., Gentry, C., Parno, B.: Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In: Rabin, T. (ed.) Advances in Cryptology – CRYPTO 2010. Lecture Notes in Computer Science, vol. 6223, pp. 465–482. Springer Berlin Heidelberg, Germany, Santa Barbara, CA, USA (Aug 15–19, 2010). https://doi.org/10.1007/978-3-642-14623-7_25
24. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or A completeness theorem for protocols with honest majority. In: Aho, A. (ed.) 19th Annual ACM Symposium on Theory of Computing. pp. 218–229. ACM Press, New York City, NY, USA (May 25–27, 1987). https://doi.org/10.1145/28395.28420
25. Golovnev, A., Lee, J., Setty, S.T.V., Thaler, J., Wahby, R.S.: Brakedown: Linear-time and field-agnostic SNARKs for R1CS. In: Handschuh, H., Lysyanskaya, A. (eds.) Advances in Cryptology – CRYPTO 2023, Part II. Lecture

Notes in Computer Science, vol. 14082, pp. 193–226. Springer, Cham, Switzerland, Santa Barbara, CA, USA (Aug 20–24, 2023). https://doi.org/10.1007/978-3-031-38545-2_7
26. Knabenhans, C., Viand, A., Merino-Gallardo, A., Hithnawi, A.: vfhe: Verifiable fully homomorphic encryption. In: Proceedings of the 12th Workshop on Encrypted Computing & Applied Homomorphic Cryptography. p. 11–22. WAHC '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3689945.3694806, https://doi.org/10.1145/3689945.3694806
27. Li, G., Liu, J., Zhang, Z., Zhang, Y.: Uc-secure cryptographic reverse firewall–guarding corrupted systems with the minimum trusted module. In: Information Security and Cryptology: 17th International Conference, Inscrypt 2021, Virtual Event, August 12–14, 2021, Revised Selected Papers. p. 85–110. Springer-Verlag, Berlin, Heidelberg (2021). https://doi.org/10.1007/978-3-030-88323-2_5, https://doi.org/10.1007/978-3-030-88323-2_5
28. Lindell, Y.: How to simulate it - A tutorial on the simulation proof technique. In: Lindell, Y. (ed.) Tutorials on the Foundations of Cryptography, pp. 277–346. Springer International Publishing (2017). https://doi.org/10.1007/978-3-319-57048-8_6, https://doi.org/10.1007/978-3-319-57048-8_6
29. Mironov, I., Stephens-Davidowitz, N.: Cryptographic reverse firewalls. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology – EUROCRYPT 2015, Part II. Lecture Notes in Computer Science, vol. 9057, pp. 657–686. Springer Berlin Heidelberg, Germany, Sofia, Bulgaria (Apr 26–30, 2015). https://doi.org/10.1007/978-3-662-46803-6_22
30. Regev, O.: Lattice-based cryptography (invited talk). In: Dwork, C. (ed.) Advances in Cryptology – CRYPTO 2006. Lecture Notes in Computer Science, vol. 4117, pp. 131–141. Springer Berlin Heidelberg, Germany, Santa Barbara, CA, USA (Aug 20–24, 2006). https://doi.org/10.1007/11818175_8
31. Russell, A., Tang, Q., Yung, M., Zhou, H.S.: Cliptography: Clipping the power of kleptographic attacks. In: Cheon, J.H., Takagi, T. (eds.) Advances in Cryptology – ASIACRYPT 2016, Part II. Lecture Notes in Computer Science, vol. 10032, pp. 34–64. Springer Berlin Heidelberg, Germany, Hanoi, Vietnam (Dec 4–8, 2016). https://doi.org/10.1007/978-3-662-53890-6_2
32. Thibault, L.T., Walter, M., Zhang, J.: Practical SNARGs for matrix multiplications over encrypted data. Cryptology ePrint Archive, Paper 2026/027 (2026), https://eprint.iacr.org/2026/027
33. Young, A., Yung, M.: Kleptography: Using cryptography against cryptography. In: Fumy, W. (ed.) Advances in Cryptology – EUROCRYPT'97. Lecture Notes in Computer Science, vol. 1233, pp. 62–74. Springer Berlin Heidelberg, Germany, Konstanz, Germany (May 11–15, 1997). https://doi.org/10.1007/3-540-69053-0_6