

Practical and Efficient MPC from FHE, without ZKPoKs

Kelong Cong¹ , Nigel P. Smart^{1,2} , Titouan Tanguy¹  and
Michael Walter¹ 

¹ Zama, Paris, France

² Computer Security and Industrial Cryptography, KU Leuven, Leuven, Belgium

Abstract

Fully Homomorphic Encryption (FHE) enables one to implement low-round and low-communication complexity Multi-Party Computation (MPC) which is secure in the static malicious corruption model. By expanding on an idea presented in the full version of the FHE-based MPC protocol of Smart (IMA, 2023), we show how to completely remove the need for Zero-Knowledge Proofs-of-Knowledge in that protocol. This simplification makes the final security proof of the MPC protocol much simpler, and more modular. The new protocol makes use of an FHE scheme with an encrypted PRF functionality, and in addition to our main result we formalize the security of such encrypted PRFs and prove the construction of Deo et al. (CCS 2025) secure in this model.

Keywords: Multi-Party Computation, Fully Homomorphic Encryption

1 Introduction

In [Sma23a] a Multi-Party Computation (MPC) protocol was presented which was fully actively secure, i.e. it was robust (i.e. it has Guaranteed Output Delivery), against static adversaries, which was based on threshold public-key Fully Homomorphic Encryption (FHE). The basic blueprint of this protocol followed a template established in Gentry’s thesis [Gen09]. Users enter data into the protocol using the public key encryption mechanism provided by the FHE, the desired functions are computed using the evaluation procedure of the FHE scheme, and then the outputs are obtained via threshold decryption. The underlying secret key of the FHE scheme is secret shared, in some initialization phase, between the entities which provide the threshold decryption service.

To obtain a robust protocol: Invalid user inputs are prevented by attaching a Zero-Knowledge Proof-of-Knowledge (ZKPoK) to every public key encryption. To prevent evaluation errors, a simple Verifiable Computation (VC) methodology is applied by replicating the entities which compute the FHE evaluation, and relying on the fact that FHE evaluation is a deterministic process. Finally, robustness of the threshold decryption is obtained by using a robust threshold decryption methodology, such as that given in [DDE⁺23].

The paper [Sma23a] frames the description of the underlying programming model for the MPC protocol in terms of a so-called F-Circuit, which is a specialization of the M-Circuit model introduced in [BDO⁺21]. The reason for introducing such a programming model is that it becomes easier to model exactly what operations are possible, and not

Emails: kelong.cong@zama.ai (Kelong Cong), nigel.smart@kuleuven.be, nigel@zama.ai (Nigel P. Smart), titouan.tanguy@zama.ai (Titouan Tanguy), michael.walter@zama.ai (Michael Walter)

possible within the system. It also makes defining the ideal functionality simpler, as the ideal functionality is essentially “emulate the programming model”.

The programming model in [Sma23a] was couched in terms of a generic FHE scheme, in which functions to homomorphically evaluate are (essentially) given as algebraic circuits over some ring or field. In addition, to enable more elaborate programs, the ability to store data in a transcribed form, or to have integer variables was also captured. This meant that programs were not just restricted to a circuit model of computation, but could proceed like in a conventional computer via a program counter, with stacks and indexed memory operations.

In an update to the eprint version [Sma23b] the programming model was extended to enable the homomorphic computation of a Pseudo-Random Function (PRF). This was in order to capture the, at that point, recent introduction of such a concept in [CPS18, DJL⁺25]. A simple application of this encrypted PRF was presented, which enabled the input of data into the protocol via the use of the encrypted PRF, thus bypassing the need for the public key encryptions and associated zero-knowledge proofs. However, the security proof, especially related to the addition of the encrypted PRF, was only sketched out in [Sma23b].

In this paper we significantly simplify the protocol of [Sma23a, Sma23b]. We re-present the protocol from [Sma23a] purely as one based solely on an FHE scheme which admits an encrypted PRF. This presents a number of optimizations and simplifications:

1. The ZKPoKs required to prove correct encryption are cumbersome in practice. In addition the most efficient schemes known, such as those in [Lib24], whilst providing post-quantum zero-knowledge, only provide pre-quantum soundness. By using the encrypted PRF for input of user data, we remove the need for the ZKPoKs, and so the protocol becomes entirely post-quantum (assuming the FHE scheme is post-quantum secure).
2. An issue avoided in [Sma23a] was the correctness definition for FHE. In the intervening years since the publication of [Sma23a] it has become clearer that more precise correctness notions are needed for FHE. In essence [Sma23a] assumes perfect correctness. Since all deployed schemes are not actually perfectly correct this is a problem. In particular the advent of the notions of IND-CPA-D and sIND-CPA-D have complicated the FHE landscape, see [LM21, CSBB24, CCP⁺24, BJSW25, SW26]. We address this issue in our paper.
3. The security proof in [Sma23a] was relatively complex, a complexity arising from the ZKPoKs. The proof of security in [Sma23a] was essentially the same proof of security for the offline phase of the SPDZ protocol from [DPSZ12]. The proof required knowledge extraction, and hence possible rewinding, in the ZKPoKs and thus, to ensure UC security, this rewinding needed to be done in the reduction and not the simulation, leading to a proof which was relatively opaque. By removing the need for ZKPoKs a simpler proof of security for the protocol can be given.
4. Another simplification in the proof is that we do not require the FHE scheme to be IND-KEY secure, but only IND-CPA secure. The notion of IND-KEY security is defined in [Sma23a], but was originally used (un-named) in the context of homomorphic encryption schemes in [DPSZ12]. In this notion, the key generation algorithm can either produce valid public keys or fake public keys. The real and fake public keys are *computationally indistinguishable*, but ciphertexts produced by the fake public key using the genuine encryption algorithm are *statistically hiding*. This notion has gone under various names in the literature for standard (non-homomorphic) encryption schemes: messy encryption [PVW08], meaningful/meaningless encryption [KN08], and lossy encryption [BHY09]. The problem is that for standard, efficient FHE schemes such as BGV, BFV or TFHE, the public key encryption algorithm only produced *computationally hiding* values, even for fake public keys. Thus use of

IND-KEY as an assumption in practice is problematic.

5. As our protocol is based on an FHE scheme admitting an encrypted PRF, i.e. those from [CPS18, DJL⁺25], we also present a security model for such constructions. Such a security model is missing from prior works. We also show that the encrypted PRF constructions from [CPS18, DJL⁺25] are secure in this new model.

Thus, in this paper we revisit the protocol from [Sma23a] and present it as based on an FHE protocol which admits an encrypted PRF. In the process we tighten up the underlying F-Circuit programming model, in some cases making it more complex (in order to more correctly model the encrypted PRF construction and the resulting input protocol) and in other cases making it less complex (we dispense with the ability to transcipher data for long term storage).

We also simplify the programming model to restrict the plaintext space to the ring $\mathcal{R} = \mathbb{Z}/(p^k)$; whereas in [Sma23a] one could take the ring $\mathcal{R}$ to be a ring consisting of many slots (as supported by schemes such as BGV or BFV [Bra12, BGV12, FV12]). Thus our presentation is focused on the TFHE algorithm [CGGI16, CGGI20, CLOT21, BdBB⁺25], which admits the efficient encrypted PRF from [DJL⁺25].

To make it clearer about who does threshold decryption and who does homomorphic evaluation, and the associated thresholds needed to ensure security in both cases, we also split up the roles of evaluators from decryptors.

In order to present a more direct security proof we also present a security model for an FHE scheme which admits an encrypted PRF. Such a suitable security model was missing from the works of [CPS18, DJL⁺25]. We also discuss the modified correctness definition, from [SW26], for FHE which is needed to support the usage of such encrypted PRFs in a protocol. Just as in [SW26] to achieve correctness, a randomization is required. But now, not only to inputs to function evaluations, but also to the outputs of the encrypted PRF evaluations. To achieve a deterministic evaluation procedure, these randomizations need to be de-randomized. We present the necessary changes needed, and outline the relevant changes to the proofs from [SW26].

The resulting modifications of the protocol from [Sma23a] has a number of a number of practical advantages. Firstly, it is inherently post-quantum secure, being based solely on post-quantum secure FHE. Secondly, from the client/user side the method to enter data into the protocol is now super lightweight (simple in-the-clear Pseudo-Random Generator (PRG) evaluations, and no FHE or zero-knowledge techniques needed). Thirdly, the method in this paper, for much the same reasons, requires less bandwidth. These advantages come with some disadvantages however. The method of this paper requires more homomorphic processing on the part of the evaluators, and the potential maintenance of a long term state to be held by the inputting and computational parties. To avoid this long term state we also present a variant which dispenses with the PRG, and uses simply a one-time-pad encryption using a fresh encrypted PRF key per input value.

We end this introduction by pointing out the following issue. It would appear, from a first reading of our protocol, that since the input of user data is done via a PRG, which has been keyed by the encrypted PRF, that the protocol does not require any form of public key encryption, and can solely rely on symmetric key encryption. However, this hides the fact that one requires public key encryption in order to obtain the re-randomization needed to obtain a strong (reactive) form of correctness, see [BJSW25, SW26]. The re-randomization (and associated de-randomized versions needed to obtain deterministic evaluation procedures) utilize the public key scheme to obtain new encryptions of zero.

One may question as to whether the full power of a public key scheme is needed to obtain such re-randomization of ciphertexts. However, it is easily seen that any homomorphic encryption scheme which has a re-randomization procedure can be turned into a public key scheme: One simply takes any ciphertext $\mathbf{ct}$, re-randomizes it to obtain a new ciphertext $\mathbf{ct}'$ encrypting the same message, then the ciphertext $\mathbf{ct} - \mathbf{ct}'$ is an encryption of zero, and

so can be added onto any message in order to obtain an encryption of the said message. The above methodology then becomes the public key encryption mechanism. Thus we still use a public key FHE scheme, even though we never directly encrypt user data with the scheme.

2 Preliminaries

In this section we outline the basic cryptographic building blocks we will be using, and the associated notation.

2.1 Notation:

We let $x \leftarrow X$ denote the uniformly random assignment to the variable x from the set X , assuming a uniform distribution over X . We also write $x \leftarrow y$ as shorthand for $x \leftarrow \{y\}$. If $\mathcal{D}$ is a probability distribution over a set X , then we let $x \leftarrow \mathcal{D}$ denote sampling from X with respect to the distribution $\mathcal{D}$. If A is a (probabilistic) algorithm then we denote by $a \leftarrow A$ the assignment of the output of A , where the probability distribution is over the random tape of A .

2.2 System Overview

We consider an MPC framework in which participants are divided into four distinct roles: input providers $\mathbb{I} = (\mathcal{I}_1, \dots, \mathcal{I}_I)$, output recipients $\mathbb{O} = (\mathcal{O}_1, \dots, \mathcal{O}_O)$, computing parties responsible for performing the homomorphic evaluation $\mathbb{C} = (\mathcal{C}_1, \dots, \mathcal{C}_C)$, and parties involved in the threshold decryption protocol $\mathbb{T} = (\mathcal{T}_1, \dots, \mathcal{T}_T)$. The explicit separation of roles among parties, in particular input, output and computing parties, comes from the paper [DI05]. This division was applied to various practical MPC systems in [BNP08, BK11, BKL⁺14]. We extend this division by dividing the computing parties into ones performing homomorphic evaluation and ones performing threshold decryption.

We assume a single (monolithic) adversary capable of static corruption. This adversary may corrupt any number of input parties, any number of output parties, and up to a bounded fraction of the computing and decryption parties. Specifically, at most $t_{\mathbb{C}}$ computing parties may be corrupted, where we require $t_{\mathbb{C}} < C/2$ to guarantee robust execution of the homomorphic computations. Since security is meaningless if all inputs are compromised, we assume that at least one input party remains honest.

In [Sma23a], a simplifying assumption was made that $\mathbb{C} = \mathbb{T}$, whilst in this paper, we propose a protocol in a hybrid model for the threshold decryption functionality. Therefore, the parties in $\mathbb{T}$ are abstracted away by this functionality and are not explicitly involved in our hybrid protocol execution. However, when implementing the threshold decryption functionality via a concrete protocol one needs to consider the number of parties $t_{\mathbb{T}}$ who can be corrupted in $\mathbb{T}$. Assuming $t_{\mathbb{T}} < T/3$ ensures robustness of the distributed decryption protocol using the techniques from [DDE⁺23]³.

Our model is described in the synchronous communication setting. However, it can be extended to the asynchronous setting with minor adjustments, such as employing an asynchronous implementation of the Declassify operation below (which corresponds to a threshold decryption operation for the underlying FHE scheme), and introducing a standard synchronization mechanism for collecting inputs. We assume that all parties communicate over authenticated channels, except in cases where private channels are explicitly required.

³In principle, this bound can be relaxed to $t_{\mathbb{T}} < T/2$, but doing so leads to significantly less efficient protocols with current techniques, since error correction would no longer rely on information theoretic constructions.

2.3 Fully Homomorphic Encryption

We consider a public key FHE scheme with plaintext space a ring $\mathcal{R} = \mathbb{Z}/(p^k)$. A public key FHE scheme, with plaintext space a ring $\mathcal{R}$, is a tuple of algorithms $(\text{Gen}, \text{KeyGen}, \text{Enc}, \text{Dec}, \text{Eval})$ and a space $\mathcal{E}$ of “valid” encryptions. The set $\mathcal{E}$ is a subset of a larger set $\mathbb{E}$. These algorithms have the following signatures:

- $\text{Gen}(1^{\text{sec}})$: On input of a security parameter sec this outputs a secure set of parameters params for the underlying FHE scheme.
- $\text{KeyGen}(\text{params})$: A probabilistic algorithm which on input of a set of secure parameters will produce a pair of keys (pk, sk) ; the key pk is the public key (which we assume contains the evaluation key), and sk is the private key.
- $\text{Enc}(m, \text{pk}; r)$: On input of $m \in \mathcal{R}$, the public key pk , and randomness r from a space of random coins Coins , this produces a ciphertext $\text{ct} \in \mathcal{E}$. For ease of notation we may sometimes remove mention of r where its exact value is not important.
- $\text{Dec}(\text{ct}, \text{sk})$: On input of an element $\text{ct} \in \mathcal{E}$ and the secret key sk this outputs the corresponding plaintext $m \in \mathcal{R}$.
- $\text{Eval}((\text{ct}_1, \dots, \text{ct}_m), F, \text{pk})$: On input of a sequence of m ciphertexts $(\text{ct}_1, \dots, \text{ct}_m) \in \mathcal{E}^m$, a function $F : \mathcal{M}^m \rightarrow \mathcal{M}^n$, and the public key pk , this operation produces a sequence of n ciphertexts $(\text{ct}'_1, \dots, \text{ct}'_n)$.

We require, for our main MPC protocol, that the Eval function is deterministic, i.e. execution on the same input ciphertexts, same function and same public key, will result in exactly the same output ciphertexts. See [SW26] on how this can be achieved even when the evaluation functions need to be “randomized” for correctness reasons.

2.3.1 Security of FHE

The standard security notion for SHE/FHE is that of IND-CPA. To make our advantage statements clearer we use a many-time variant of IND-CPA security, where we give the adversary access to a left-right oracle. Formally we define the game in Figure 1. We then have the definition

Definition 1 (FHE Security). An FHE scheme is said to be IND-CPA if for any PPT adversary $\mathcal{A}$, running in time polynomial in sec and making at most ℓ_{LR} queries to its left-right oracle, it holds that

$$\text{Adv}_{\text{IND-CPA}}^{\mathcal{A}}(\text{sec}, \ell_{\text{LR}}) = 2 \cdot \left| \Pr \left[\text{IND-CPA}^{\mathcal{A}}(\text{sec}) = 1 \right] - \frac{1}{2} \right|$$

is negligible in sec , where $\text{IND-CPA}^{\mathcal{A}}(\text{sec})$ is the game in Figure 1.

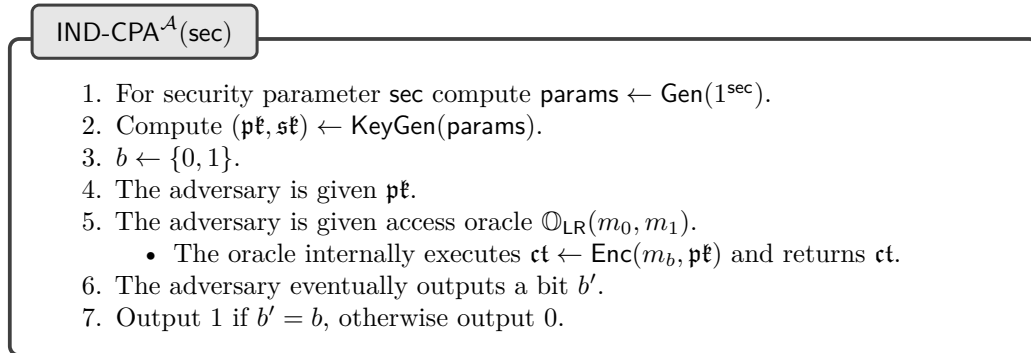


Figure 1: The security game for a FHE security

2.3.2 Correctness of FHE

The simplest notion of correctness is that of *perfect correctness* which requires

$$\Pr \left[\text{Dec}(\text{Eval}(\text{Enc}(m_1, \mathbf{pk}), \dots, \text{Enc}(m_k, \mathbf{pk}), F, \mathbf{pk}), \mathbf{sk}) = F(m_1, \dots, m_k) \right] = 1 \quad (1)$$

for all $(m_1, \dots, m_k)$ in the message space, all functions F which are desired to be homomorphically computable, and all $(\mathbf{pk}, \mathbf{sk}) \leftarrow \text{KeyGen}(\text{params})$, and all $\text{params} \leftarrow \text{Gen}(1^{\text{sec}})$. However, for many schemes, performance can be improved if one relaxes the definition such that Eq. (1) is only required to hold with probability $1 - \epsilon$ for small ϵ over the randomness of Gen , KeyGen , Enc and Eval . In the literature, this notion is typically referred to as *statistical correctness*.

This correctness notion is however not reactive, which is what we require in our protocol. Thus in [SW26], taking inspiration from [ABMP26], a reactive correctness notion is given called *computational correctness*, see Definition 2 and the related Figure 2.

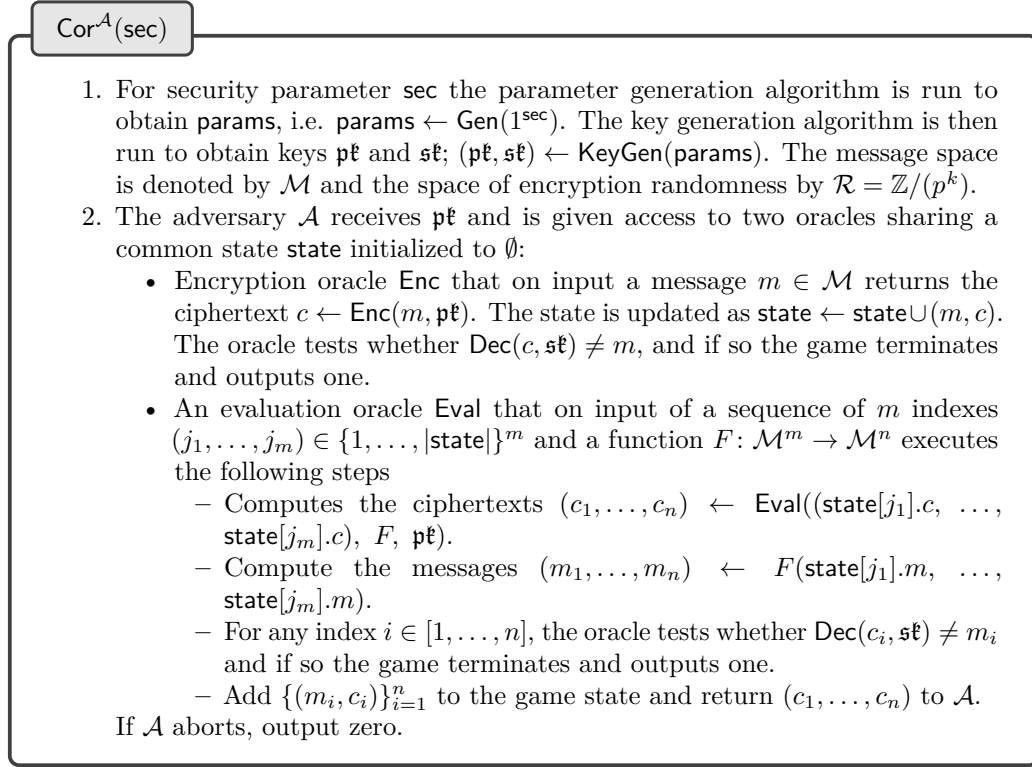


Figure 2: The security game for FHE correctness.

Definition 2 (Computational Correctness). An FHE scheme $(\text{Gen}, \text{KeyGen}, \text{Enc}, \text{Dec}, \text{Eval})$ is said to be *computationally correct* if for any PPT adversary $\mathcal{A}$, running in time polynomial in sec and making at most ℓ_{Enc} queries to the encryption oracle and at most ℓ_{Eval} queries to the evaluation oracle, it holds that

$$\text{Adv}_{\text{Cor}}^{\mathcal{A}}(\text{sec}, \ell_{\text{Enc}}, \ell_{\text{Eval}}) = \Pr \left[\text{Cor}^{\mathcal{A}}(\text{sec}) = 1 \right]$$

is negligible in sec , where $\text{Cor}^{\mathcal{A}}(\text{sec})$ is the game in Figure 2.

Note that, in the game $\text{Cor}^{\mathcal{A}}(\text{sec})$ the adversary gets to select the function F . Thus they may adaptively choose the function F depending on the ciphertexts, and also use

outputs of `Eval` as inputs to `Eval`. This definition of correctness closely matches how FHE schemes are used in practice, and in our protocol; namely data is encrypted, functions are evaluated, other data is encrypted, and then other functions are chosen to be evaluated.

The above correctness definition is closely related to the notion of IND-CPA-D security, see [LM21, CSBB24, CCP⁺24, BJSW25, SW26]. In particular an FHE scheme which is IND-CPA and computationally correct is IND-CPA-D. Note that to maintain deterministic evaluation of the `Eval` operation as well as computational correctness, we will need a de-randomized re-randomized version of the `Eval` algorithm as presented in [SW26].

2.4 Pseudo-Random Functions

Pseudo-Random Functions (PRFs) must always be keyed, and we will denote the key generation algorithm by $\text{PRF.KeyGen}(1^{\text{sec}})$, which will output a key $\mathbf{k} \in (\mathbb{Z}/(p))^{n_{\text{prf}}}$, with $n_{\text{prf}} = O(\text{sec})$. The PRF itself, on being called via $\text{PRF}_{\mathbf{k}}(x)$, for some input string x , will output a value in $\mathbb{Z}/(p^{k'})$, for some value of k' .

In terms of security the output of the PRF, on every call to $\text{PRF}_{\mathbf{k}}(x)$, should be indistinguishable from a uniformly random element of $\mathbb{Z}/(p^{k'})$. Formally we define the game in Figure 3. We then have the definition

Definition 3 (PRF Security). A PRF is said to be *indistinguishably secure* if for any PPT adversary $\mathcal{A}$, running in time polynomial in sec and making at most ℓ_{prf} queries to its PRF oracle, it holds that

$$\text{Adv}_{\text{PRF}}^{\mathcal{A}}(\text{sec}, \ell_{\text{prf}}) = 2 \cdot \left| \Pr \left[\text{PRF}^{\mathcal{A}}(\text{sec}) = 1 \right] - \frac{1}{2} \right|$$

is negligible in sec , where $\text{PRF}^{\mathcal{A}}(\text{sec})$ is the game in Figure 3.

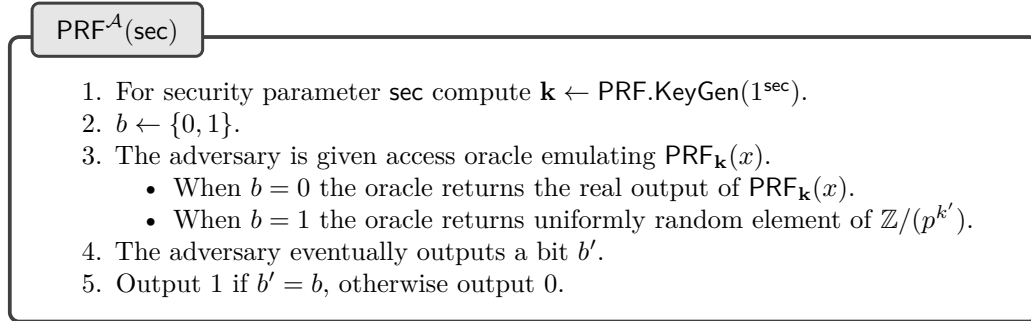


Figure 3: The security game for a PRF

2.5 FHE Friendly PRG

As well as PRFs, we also require an FHE friendly Pseudo-Random Generator (PRG). We assume a PRG given by two algorithms PRG.Init and PRG.Update .

- The algorithm $\text{PRG.Init}(\mathbf{k}_{\text{prg}})$ takes as input a key $\mathbf{k}_{\text{prg}} \in (\mathbb{Z}/(p))^{n_{\text{prg}}}$, with $n_{\text{prg}} = O(\text{sec})$, and outputs two state values $\text{state}_{\text{pub}} \subset (\mathbb{Z}/(p^{k_{\text{prg}}}))^r$ and $\text{state}_{\text{priv}} \subset (\mathbb{Z}/(p^{k_{\text{prg}}}))^s$, where $0 < k_{\text{prg}} \leq k$. The idea being that $\text{state}_{\text{pub}}$ is a public state, it does not depend on the secret key $\mathbf{k}_{\text{prg}}$, whilst $\text{state}_{\text{priv}}$ is a private state, i.e. it does depend on the secret key $\mathbf{k}_{\text{prg}}$.
- The algorithm $\text{PRG.Update}(k', \ell, \text{state}_{\text{pub}}, \text{state}_{\text{priv}})$ will produce three outputs, firstly a random output of ℓ elements in $\mathbb{Z}/(p^{k'})$, and then updated state values $\text{state}'_{\text{pub}} \subset (\mathbb{Z}/(p^{k_{\text{prg}}}))^r$ and $\text{state}'_{\text{priv}} \subset (\mathbb{Z}/(p^{k_{\text{prg}}}))^s$.

In terms of security the output of the PRG, on every call to PRG.Update should be indistinguishable from a uniformly random sequence of ℓ elements in $\mathbb{Z}/(p^{k'})$. Formally we define the associated security game in Figure 4. We then have the definition

Definition 4 (PRG Security). A PRG is said to be *indistinguishably secure* if for any PPT adversary $\mathcal{A}$, running in time polynomial in sec and making at most ℓ_{Update} calls to the PRG.Update oracle, each of which returns at most ℓ_{out} outputs, it holds that

$$\text{Adv}_{\text{PRG}}^{\mathcal{A}}(\text{sec}, \ell_{\text{Update}}, \ell_{\text{out}}) = 2 \cdot \left| \Pr \left[\text{PRG}^{\mathcal{A}}(\text{sec}) = 1 \right] - \frac{1}{2} \right|$$

is negligible in sec , where $\text{PRG}^{\mathcal{A}}(\text{sec})$ is the game in Figure 4.

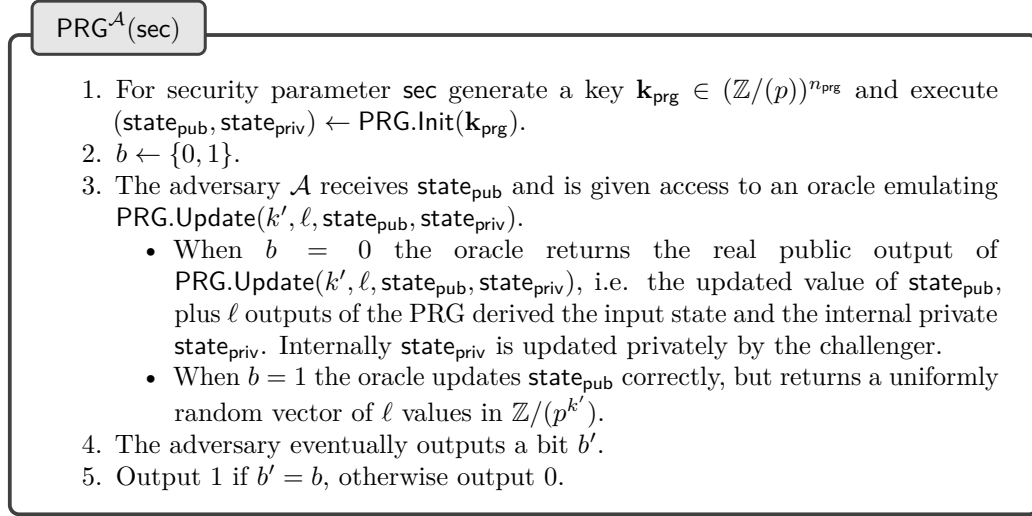


Figure 4: The security game for a PRG

When $\mathbb{Z}/(p^{k_{\text{prg}}}) = \mathbb{Z}/(p) = \mathbb{Z}/(2)$ there are a number of instantiations of such PRGs.

AES-CTR: A block cipher in CTR mode, such as AES, immediately fits into this framework. Here we set on initialization $\text{state}_{\text{priv}} = \mathbf{k}_{\text{prg}}$, where $\mathbf{k}_{\text{prg}}$ is an AES key of the appropriate length, and we set $\text{state}_{\text{pub}}$ to be the sequence of 128-bit zeros. The value of $\text{state}_{\text{priv}}$ is never updated, but the value of $\text{state}_{\text{pub}}$ will be. Then $\text{PRG.Update}(1, \ell, \text{state}_{\text{pub}}, \text{state}_{\text{priv}})$ will output the sequence of $\ell > 0$ bits obtained by evaluating AES in CTR mode, for the counters $\text{state}_{\text{pub}}, \text{state}_{\text{pub}} + 1, \text{state}_{\text{pub}} + 2, \dots, \text{state}_{\text{pub}} + \lceil \ell/128 \rceil - 1$, and then truncating the output to ℓ bits. The value of $\text{state}_{\text{pub}}'$ on return from PRG.Update is equal to $\text{state}_{\text{pub}} + \lceil \ell/128 \rceil$.

Kreyvium: A stream cipher, such as Kreyvium [CCF⁺16, CCF⁺18], works in exactly the opposite way. On initialization the value of $\text{state}_{\text{priv}}$ is created by going through the warm up of the stream cipher on input of $\mathbf{k}_{\text{prg}}$, whilst the value of $\text{state}_{\text{pub}}$ is set to the empty set. Then $\text{PRG.Update}(1, \ell, \text{state}_{\text{pub}}, \text{state}_{\text{priv}})$ will output the sequence of $\ell > 0$ bits obtained by clocking the stream cipher for ℓ iterations. The value of $\text{state}_{\text{pub}}'$ returned will still be empty, whilst the value of $\text{state}_{\text{priv}}'$ will be the updated internal stream cipher state.

What we require, whether a block cipher or stream cipher variant, is that the evaluation of the PRG should be FHE friendly. In other words one should be able to efficiently evaluate the PRG.Init and PRG.Update functions where the state $\text{state}_{\text{priv}}$ is FHE encrypted, but the $\text{state}_{\text{pub}}$ is kept in the clear.

Whilst much work has gone into trying to find efficient ways to evaluate AES in the homomorphic domain [GHS12, WWL⁺23, TCBS23a], the stream cipher Kreyvium is specifically designed to be easy to evaluate with various FHE schemes. In particular Kreyvium is easy to evaluate using TFHE [TCBS23b].

3 FHE Encrypted PRF

In this section let $G(x, y, z)$ denote the function

$$G(x, y, z) = \left\lceil \frac{x}{y} \cdot (z \pmod{y}) \right\rceil.$$

In [CPS18, DJL⁺25] the authors enable an efficient method for an FHE scheme to homomorphically evaluate a Pseudo-Random Function (PRF) using BGV and TFHE. In particular the constructions are based on the following PRF, where $P < Q$,

$$\text{PRF} : \begin{cases} \mathbb{Z}^{n_{\text{prf}}} \times (\mathbb{Z}/Q)^{n_{\text{prf}}} & \longrightarrow (\mathbb{Z}/(P))^{n_{\text{prf}}} \\ (\mathbf{k}_{\text{prf}}, \mathbf{a}) & \longmapsto G(P, Q, \mathbf{a} \cdot \mathbf{k}_{\text{prf}}). \end{cases}$$

The value $\mathbf{k}_{\text{prf}} \in \{0, 1\}^{n_{\text{prf}}}$ is considered as the secret key, and the value $\mathbf{a} \in (\mathbb{Z}/Q)^{n_{\text{prf}}}$ is the PRF's public input. This function is known to imply a secure weak-PRF, based on the hardness of LWR [BPR12]. In the Random Oracle Model, by allowing the vector $\mathbf{a}$ to be the output of a random oracle $\mathbf{a}(x) = H(x)$ it yields a direct (fully secure) PRF. This Random Oracle Model construction was first explicitly given in [BLMR13], but it is implicitly also in [BPR12].

In the paper [CPS18] the construction is applied, for BGV encryption, for the ring $\mathcal{R}$ of dimension $n_{\text{prf}} = m_{\text{prf}}$ with Q being the plaintext modulus p^k , $P = 2$. Thus the basic PRF is of the form

$$\text{PRF}^{\text{BGV}} : \begin{cases} \{0, 1\}^{n_{\text{prf}}} \times \{0, 1\}^* & \longrightarrow \mathcal{R} \\ (\mathbf{k}_{\text{prf}}, x) & \longmapsto G(2, p^k, H(x) \cdot \mathbf{k}_{\text{prf}}). \end{cases}$$

for some hash function, modelled as a random oracle,

$$H : \{0, 1\}^* \longrightarrow \mathcal{R}^{n_{\text{prf}}}.$$

That this construction, in the Random Oracle Model, forms a secure PRF follows from the Ring-LWR assumption. Note, the output of this basic PRF is simply elements with binary coefficients in the ring $\mathcal{R}$. This basic PRF, can be extended to a PRF which produces elements uniformly at random from the entire ring, by summing a number of such evaluations multiplied by appropriate powers of two. Thus we can produce a PRF whose codomain is the whole of $\mathcal{R}$.

In the paper [DJL⁺25] the authors adopt a slight variation on the basic PRF from [CPS18] above. In particular they focus on a PRF particularly suited for evaluation via the TFHE Fully Homomorphic Encryption scheme. In order to reduce the evaluation complexity under homomorphic operations, and also to ensure the plaintext space is compatible with future homomorphic operations, they change the basic PRF to one of form

$$\text{PRF}^{\text{TFHE}} : \begin{cases} \{0, 1\}^{n_{\text{prf}}} \times \{0, 1\}^* & \longrightarrow \mathbb{Z}/(p^k) \\ (\mathbf{k}_{\text{prf}}, x) & \longmapsto (-1)^{\text{msb}(H(x) \cdot \mathbf{k}_{\text{prf}} \pmod{2N})} \cdot G(p^k, N, H(x) \cdot \mathbf{k}_{\text{prf}}). \end{cases}$$

for some hash function, modelled as a random oracle,

$$H : \{0, 1\}^* \longrightarrow (\mathbb{Z}/(2N))^{n_{\text{prf}}}.$$

This modification, producing a PRF negacyclic in $H(x) \cdot \mathbf{k}_{\text{prf}}$, is applied with $2 \cdot N$ being the modulus after the TFHE modulus-switch, i.e. N is the RLWE ring dimension for TFHE, and n is the LWE secret key dimension, with $\mathbf{k}_{\text{prf}} \in \{0, 1\}^{n_{\text{prf}}}$ being the LWE secret key. Again, under the LWR assumption, this modification can be proved to be a secure PRF in the random oracle model. Note, the PRF can be used to produce output modulo $p^{k'}$ for any k' less than k if desired via an obvious modification.

Note, one can produce t values from either PRF by utilizing the PRF with a matrix $A \in \mathcal{R}^{t \times n_{\text{prf}}}$ instead of a single vector $\mathbf{a} \in \mathcal{R}^{n_{\text{prf}}}$. Thus we have, for example in the case of the PRF PRF^{TFHE} ,

$$\text{PRF}^{\text{TFHE}} : \begin{cases} \{0, 1\}^{n_{\text{prf}}} \times \{0, 1\}^* & \longrightarrow (\mathbb{Z}/(p^k))^t \\ (\mathbf{k}_{\text{prf}}, x) & \longmapsto (-1)^{\text{msb}(H(x) \cdot \mathbf{k}_{\text{prf}} \pmod{2N})} \cdot G(p^k, N, H(x) \cdot \mathbf{k}_{\text{prf}}). \end{cases}$$

for some hash function, modelled as a random oracle,

$$H : \{0, 1\}^* \longrightarrow (\mathbb{Z}/(2N))^{t \times n_{\text{prf}}}.$$

In this work we will abstract away the details of the above PRFs using the following formalism. We assume an underlying (secure) PRF, with domain $\{0, 1\}^{n_{\text{prf}}}$ for the secret key, which produces a sequence of t uniformly random outputs in the subset of the FHE schemes plaintext space $\mathbb{Z}/(p^{k'}) \subseteq \mathbb{Z}/(p^k) = \mathcal{R}$, i.e. a PRF of the

$$\text{LWR-PRF}_{k', t} : \{0, 1\}^{n_{\text{prf}}} \times \{0, 1\}^* \longrightarrow (\mathbb{Z}/(p^{k'}))^t,$$

for $0 \leq k' \leq k$. This PRF can be instantiated, in the case of TFHE, via the definition

$$\text{LWR-PRF}_{k', t}(\mathbf{k}_{\text{prf}}, x) := (-1)^{\text{msb}(H(x \| k' \| t) \cdot \mathbf{k}_{\text{prf}} \pmod{2N})} \cdot G(p^{k'}, N, H(x \| k' \| t) \cdot \mathbf{k}_{\text{prf}})$$

Using an FHE encryption $\mathbf{pk}_{\text{prf}}$ of the secret key $\mathbf{k}_{\text{prf}}$, and the FHE evaluation key $\mathbf{pk}$, one can easily homomorphically evaluate the PRF via a function call of the form $\text{FHE-PRF}_{k', t}(x, \mathbf{pk}, \mathbf{pk}_{\text{prf}})$, which produces a vector of t encryptions of the form $\text{Enc}(\text{LWR-PRF}_{k', t}(\mathbf{k}_{\text{prf}}, x), \mathbf{pk})$. With this encrypted PRF we think of the FHE KeyGen operation as outputting the pair of pairs of keys $((\mathbf{pk}, \mathbf{pk}_{\text{prf}}), (\mathbf{sk}, \mathbf{k}_{\text{prf}}))$, and the parameter generation algorithm Gen as including the value n_{prf} and an upper bound T on the number of outputs t which may be requested in a single evaluation within the output set params .

3.1 Security Model for Encrypted PRFs

Despite being introduced in [CPS18, DJL⁺25], no formal security model for what an encrypted PRF should achieve has been given. We hence formalize this via the game in Figure 5 and Definition 5.

Definition 5. An FHE scheme with encrypted PRF, $(\text{Gen}, \text{KeyGen}, \text{Enc}, \text{Dec}, \text{Eval}, \text{FHE-PRF})$ is said to be *EncPRF* secure if for any PPT adversary $\mathcal{A}$, running in time polynomial in sec and making at most $\ell_{\mathbb{O}}$ queries to its $\mathbb{O}$ oracle, it holds that

$$\text{Adv}_{\text{EncPRF}}^{\mathcal{A}}(\text{sec}, \ell_{\mathbb{O}}) = 2 \cdot \left| \Pr \left[\text{EncPRF}^{\mathcal{A}}(\text{sec}) = 1 \right] - \frac{1}{2} \right|$$

is negligible in sec , where $\text{EncPRF}^{\mathcal{A}}(\text{sec})$ is the game in Figure 5.

We can then show that above encrypted PRF based on TFHE is secure, assuming that the TFHE scheme is IND-CPA secure and the LWR-PRF is a secure PRF.

Theorem 1. *The encrypted PRF FHE-PRF instantiated using TFHE and LWE-PRF is EncPRF-secure assuming that TFHE is IND-CPA secure and LWR-PRF is a secure PRF. In particular if $\mathcal{A}$ is an adversary against EncPRF security, making at most $\ell_{\mathbb{O}}$ queries to its oracle $\mathbb{O}(x, k', t)$, then there are adversaries $\mathcal{A}_1$ and $\mathcal{A}_2$ such that*

$$\text{Adv}_{\text{EncPRF}}^{\mathcal{A}}(\text{sec}, \ell_{\mathbb{O}}) \leq \text{Adv}_{\text{IND-CPA}}^{\mathcal{A}_1}(\text{sec}, \ell_{\mathbb{O}}) + \text{Adv}_{\text{PRF}}^{\mathcal{A}_2}(\text{sec}, \ell_{\mathbb{O}}).$$

EncPRF^A(sec)

1. For security parameter **sec** the parameter generation algorithm is run to obtain **params**, i.e. $\text{params} \leftarrow \text{Gen}(1^{\text{sec}})$. The key generation algorithm is then run to obtain keys $\mathbf{pk}$ and $\mathbf{sk}$; $((\mathbf{pk}, \mathbf{pk}_{\text{prf}}), (\mathbf{sk}, \mathbf{k}_{\text{prf}})) \leftarrow \text{KeyGen}(\text{params})$. The message space is denoted by $\mathcal{M}$ and the space of encryption randomness by $\mathcal{R} = \mathbb{Z}/(p^k)$.
2. The challenger picks a bit $b \in \{0, 1\}$.
3. The adversary $\mathcal{A}$ receives $(\mathbf{pk}, \mathbf{pk}_{\text{prf}})$, the adversary can encrypt any message of their choice (it is a public key scheme after all), and can evaluate the encrypted PRF on inputs x of their choice (again as this is an operation made available via access to the public key).
4. The adversary also has access to an oracle $\mathbb{O}(x, k', t)$, where $0 < t \leq T$, which maintains an internal state **state**, and operates as follows:
 - If $b = 0$ then $\mathbb{O}(x, k', t)$ will return the value $\text{LWR-PRF}_{k', t}(\mathbf{k}_{\text{prf}}, x)$.
 - If $b = 1$ and $\exists(x, k', t, r) \in \text{state}$ then return r .
 - If $b = 1$ and $\nexists(x, k', t, r) \in \text{state}$ then pick r from $(\mathbb{Z}/(p^k))^t$, add (x, k', t, r) to **state**, and return r .
5. Eventually $\mathcal{A}$ returns its guess b' as to the hidden bit b .
6. Return one if $b' = b$, and zero otherwise.

Figure 5: The security game for an FHE scheme with encrypted PRF.

Proof. We let $\mathbb{G}_0$ denote the game in Figure 5 with $b = 0$. In game $\mathbb{G}_1$ we switch to a modified game in which the key $\mathbf{pk}_{\text{prf}}$ now “encrypts” not $\mathbf{k}_{\text{prf}}$, but a different key $\mathbf{k}'_{\text{prf}}$. So the oracle $\mathbb{O}(x, k', t)$ still returns LWR-PRF values for the key $\mathbf{k}_{\text{prf}}$, but the evaluation of the encrypted PRF produces encrypted LWR-PRF values with respect to the key $\mathbf{k}'_{\text{prf}}$. Clearly, any adversary that can distinguish between $\mathbb{G}_0$ and $\mathbb{G}_1$ can be turned into an adversary $\mathcal{A}_1$ against the IND-CPA security of the FHE scheme, i.e.

$$\left| \Pr[1 \leftarrow \mathcal{A}(\mathbb{G}_0)] - \Pr[1 \leftarrow \mathcal{A}(\mathbb{G}_1)] \right| \leq \text{Adv}_{\text{IND-CPA}}^{\mathcal{A}_1}(\text{sec}, \ell_{\mathbb{O}}),$$

where $\mathcal{A}(\mathbb{G}_i)$ denotes the output of the adversary $\mathcal{A}$ in game $\mathbb{G}_i$.

In game $\mathbb{G}_2$ we switch the output of the $\mathbb{O}(x, k', t)$ oracle to be a random value, i.e. for $b = 1$ in the game. Thus we obviously have, that there is an adversary $\mathcal{A}_2$ such that

$$\left| \Pr[1 \leftarrow \mathcal{A}(\mathbb{G}_1)] - \Pr[1 \leftarrow \mathcal{A}(\mathbb{G}_2)] \right| \leq \text{Adv}_{\text{PRF}}^{\mathcal{A}_2}(\text{sec}, \ell_{\mathbb{O}}).$$

Finally, in game $\mathbb{G}_3$ we switch back from $\mathbf{pk}_{\text{prf}}$ encrypting $\mathbf{k}'_{\text{prf}}$, back to the original key $\mathbf{k}_{\text{prf}}$. Clearly, as the adversary has no access to the messages encrypted via the encrypted PRF operation, and the two sets of encrypted messages in game $\mathbb{G}_2$ and $\mathbb{G}_3$ are valid LWR-PRF outputs, we have

$$\Pr[1 \leftarrow \mathcal{A}(\mathbb{G}_2)] = \Pr[1 \leftarrow \mathcal{A}(\mathbb{G}_3)].$$

From this the result follows. $\square$

Note, the above theorems advantage statement does not need to account for the “ciphertexts” encrypting the PRF key, since these are more akin to encryption of the secret keys in the TFHE “bootstrapping” keys; thus they form part of the public key of the underlying TFHE scheme.

3.2 Modification to Correctness Definition of FHE

The introduction of this encrypted PRF creates an additional aspect which needs to be taken into account in our correctness game. In our main protocol below we will not really use the correctness game from Figure 2, but the correctness game from Figure 6. In particular there will be no use of the public key encryption mechanism. The only way new FHE encryptions are brought into existence will either be via the FHE-PRF primitive, or via executions of the Eval operation. In our MPC protocol the adversary can evaluate arbitrary ciphertexts offline, but only ciphertexts evaluated by a majority of the computing parties can ever be used within the protocol. In the MPC protocol below the only fresh ciphertexts entered into protocol come from the FHE-PRF, thus the adversary does not have the ability to introduce arbitrary ciphertexts. Indeed removing the ability for the adversary (or anyone) to introduce fresh ciphertexts of their choosing is exactly what enables us to remove the ZKPoKs from the previous protocol.

We then have the definition

$\text{Cor}_{\text{prf}}^{\mathcal{A}}(\text{sec})$

1. For security parameter sec the parameter generation algorithm is run to obtain params , i.e. $\text{params} \leftarrow \text{Gen}(1^{\text{sec}})$. The key generation algorithm is then run to obtain keys pk and sk ; $((\text{pk}, \text{pk}_{\text{prf}}), (\text{sk}, \text{k}_{\text{prf}})) \leftarrow \text{KeyGen}(\text{params})$. The message space is denoted by $\mathcal{M}$ and the space of encryption randomness by $\mathcal{R} = \mathbb{Z}/(p^k)$.
 2. The adversary $\mathcal{A}$ receives $(\text{pk}, \text{pk}_{\text{prf}})$ and is given access to two oracles sharing a common state state initialized to $\emptyset$:
 - An encrypted PRF FHE-PRF that on input of a string $x \in \{0,1\}^*$, an integer value $0 < k' \leq k$ and an integer value $0 < t \leq T$, returns the ciphertexts $(c_1, \dots, c_t) \leftarrow \text{FHE-PRF}_{k',t}(x, \text{pk}, \text{pk}_{\text{prf}})$. The state is updated as $\text{state} \leftarrow \text{state} \cup \{(m_i, c_i)\}_{i=1}^t$, where $(m_1, \dots, m_t) = \text{LWR-PRF}_{k',t}(\text{k}_{\text{prf}}, x)$. If for any index $i \in [1, \dots, t]$ we have $\text{Dec}(c_i, \text{sk}) \neq m_i$ then the game terminates and outputs one.
 - An evaluation oracle Eval that on input of a sequence of m indexes $(j_1, \dots, j_m) \in \{1, \dots, |\text{state}|\}^m$ and a function $F: \mathcal{M}^m \rightarrow \mathcal{M}^n$ executes the following steps
 - Computes the ciphertexts $(c_1, \dots, c_n) \leftarrow \text{Eval}((\text{state}[j_1].c, \dots, \text{state}[j_m].c), F, \text{pk})$.
 - Compute the messages $(m_1, \dots, m_n) \leftarrow F(\text{state}[j_1].m, \dots, \text{state}[j_m].m)$.
 - If for any index $i \in [1, \dots, n]$ we have $\text{Dec}(c_i, \text{sk}) \neq m_i$ then the game terminates and outputs one.
 - Add $\{(m_i, c_i)\}_{i=1}^n$ to the game state and return $(c_1, \dots, c_n)$ to $\mathcal{A}$.
- If $\mathcal{A}$ aborts, output zero.

Figure 6: The security game for FHE correctness, using FHE-PRF. The highlighted item reflects the difference with Figure 2.

Definition 6 (Computational Correctness). An FHE scheme with encrypted PRF, (Gen , KeyGen , Enc , Dec , Eval , FHE-PRF) is said to be *computationally correct* if for any PPT adversary $\mathcal{A}$, running in time polynomial in sec and making ℓ_{prf} queries to the encrypted PRF oracle and ℓ_{Eval} queries to the evaluation oracle it holds that

$$\text{Adv}_{\text{Cor}_{\text{prf}}}^{\mathcal{A}}(\text{sec}, \ell_{\text{prf}}, \ell_{\text{Eval}}) = \Pr \left[\text{Cor}_{\text{prf}}^{\mathcal{A}}(\text{sec}) = 1 \right]$$

is negligible in sec , where $\text{Cor}_{\text{prf}}^A(\text{sec})$ is the game in Figure 6.

3.3 Computational Correctness via Pseudo-Randomization

The computational correctness in the sense of Definition 6 can be achieved using (de-randomized) randomization techniques from [SW26]. There are two main steps in the proof of [SW26]: first, it is shown that a randomized version of **Eval** achieves ACER correctness⁴. Second, it is shown how to de-randomize **Eval** using a random oracle and ZKPoKs.

Adapting the first step to Definition 6 is relatively straight-forward and we only sketch the adaptation in Section 3.3.1 below. To cleanly capture this minor modification without going into the proof details, we simply state the correctness of the randomized scheme as Assumption 1.

The second step involves removing ZKPoKs, which is more delicate and thus we give full proof details in Section 3.3.4, where we reduce the correctness of the de-randomized scheme to the correctness of the randomized scheme, i.e. to Assumption 1.

We also need another technical assumption about the unpredictability of the randomized scheme, see Assumption 2. This assumption already appeared in [SW26] and is required to ensure that the adversary is not able to reference a ciphertext in a random oracle query that is only later added to the game state. If the adversary was able to do so, the reduction would get stuck. See Theorem 2 for details.

We remark that the proof techniques are similar to the ones in [SW26] and a reader familiar with the previous work may safely take Theorem 2 for granted and skip ahead to Section 4.

3.3.1 Randomization to obtain Computational Correctness

We first define a re-randomization procedure as in Figure 7. This algorithm simply executes the basic encryption procedure **Enc** to obtain encryptions of zero, and then it adds the result to the ciphertexts needing re-randomization. Each re-randomization of ciphertext ct_i is controlled by a seed value seed_i , which determines the underlying randomness used within the encryption algorithm.

FHE Re-Randomization

$\text{ReRand}((\text{ct}_0, \dots, \text{ct}_{m-1}), (\text{seed}_0, \dots, \text{seed}_{m-1}), \text{prf})$:

1. For $i \in [0, \dots, m-1]$ do
 - (a) $\text{zt}_i \leftarrow \text{Enc}(0, \text{prf}; \text{seed}_i)$.
 - (b) $\text{ct}'_i \leftarrow \text{FHE.Add}(\text{ct}_i, \text{zt}_i)$.
2. Return $(\text{ct}'_0, \dots, \text{ct}'_{m-1})$.

Figure 7: FHE Re-Randomization

The specific randomized variant of FHE we will use is the following one: In the evaluation procedure the evaluator first generates uniformly random seeds for each ciphertext, and then they apply the algorithm **ReRand** to the *input* ciphertexts to **Eval**. This provides the specific randomized evaluation procedure, Eval^R , given in Figure 8. Furthermore, we randomize the FHE-PRF algorithm by applying **ReRand** to its *output* ciphertext, to obtain the algorithm FHE-PRF^R .

⁴This is a stronger form of correctness than Definition 2, see Theorem 4 of [SW26], in which the adversaries correctness game is augmented by an encryption oracle to which it can also supply the encryption randomness. We do not need to use ACER correctness in our results.

FHE Fully Randomized FHE with Encrypted PRF

$\text{Eval}^R((\text{ct}_1, \text{aux}_1^{\text{seed}}), \dots, (\text{ct}_m, \text{aux}_m^{\text{seed}}), F, \text{pk})$:
 1. For $i \in [1, \dots, m]$ do $\text{seed}_i \leftarrow \{0, 1\}^{2 \cdot \text{sec}}$.
 2. $(\text{rt}_1, \dots, \text{rt}_m) \leftarrow \text{ReRand}((\text{ct}_1, \dots, \text{ct}_m), (\text{seed}_1, \dots, \text{seed}_m), \text{pk})$.
 3. $(\text{ct}'_1, \dots, \text{ct}'_n) \leftarrow \text{Eval}((\text{rt}_1, \dots, \text{rt}_m), F, \text{pk})$.
 4. $\text{aux}^{\text{seed}} \leftarrow (\text{seed}_1, \dots, \text{seed}_m)$.
 5. Return $((\text{ct}'_1, \text{aux}^{\text{seed}}), \dots, (\text{ct}'_n, \text{aux}^{\text{seed}}))$.

$\text{FHE-PRF}_{k',t}^R(x, \text{pk}, \text{pk}_{\text{prf}})$:
 1. $(\text{rt}_1, \dots, \text{rt}_t) \leftarrow \text{FHE-PRF}_{k',t}(x, \text{pk}, \text{pk}_{\text{prf}})$
 2. For $i \in [1, \dots, t]$ do $\text{seed}_i \leftarrow \{0, 1\}^{2 \cdot \text{sec}}$
 3. $(\text{ct}_1, \dots, \text{ct}_t) \leftarrow \text{ReRand}((\text{rt}_1, \dots, \text{rt}_t), (\text{seed}_1, \dots, \text{seed}_t))$
 4. $\text{aux}^{\text{seed}} \leftarrow (\text{seed}_1, \dots, \text{seed}_t)$
 5. Return $((\text{ct}_1, \text{aux}^{\text{seed}}), \dots, (\text{ct}_t, \text{aux}^{\text{seed}}))$

Figure 8: FHE Fully Randomized FHE with Encrypted PRF

In Figure 8 we have an extra technical condition in that the evaluation algorithm not only returns the output evaluation, but it also returns the seeds which have been used to produce the re-randomization as auxiliary information aux^{seed} attached to each ciphertext. Since the auxiliary information is attached to ciphertexts, we also pass this into the Eval^R function (but then ignore it). Outputting such information obviously can only make security harder to obtain. Thus if we prove security with this information leakage then we will have security of an implemented scheme where no such information is leaked. This leakage is technically needed later in our final proof technique for the DeRandEval evaluation method in the random oracle model.

For the rest of this section we will make the following assumption

Assumption 1. *A Fully Randomized FHE algorithm as in Figure 8 is computationally correct in the sense of Definition 6.*

It can be shown that Assumption 1 holds for a scheme like TFHE with typical parameter selection. The proof of this is similar to the one in [SW26], where it was shown that a similar assumption holds, but with respect to ACER correctness. The only change in the proof is that the underlying assumption on the average-case hardness now needs to take into account worst-case (over the input (x, k', t)) noise of ciphertexts output by FHE-PRF instead of encryption with adversarially chosen encryption randomness. Otherwise, the proof remains the same.

3.3.2 Unpredictability of FHE Ciphertexts

Before proceeding we need to define a notion of unpredictability of ciphertexts. This notion is needed to ensure that our reduction in Theorem 2 below can program the random oracle and simulate the Cor_{prf} game correctly. This assumption is related to the unpredictability of the output ciphertexts from an evaluation operation related to a *specific* re-randomization procedure for general FHE, and was first introduced in [SW26].

We first formally define unpredictability as follows:

Definition 7. For any randomized function $F : \mathcal{X} \mapsto \mathcal{Y}$ we say that it is *unpredictable* if

for any 2-stage PPT adversary $(\mathcal{A}_1, \mathcal{A}_2)$ and for any $(y, \text{state}) \leftarrow \mathcal{A}_1()$ we have that

$$\Pr \left[y = y' \mid x \leftarrow \mathcal{A}_2(\text{state}), y' \leftarrow F(x) \right]$$

is negligible in sec .

Definition 8. Let $(\text{FHE.KeyGen}, \text{FHE.Enc}, \text{FHE.Dec}, \text{FHE.Eval}, \text{FHE-PRF})$ be an FHE scheme with encrypted PRF. We say that the scheme is *unpredictable* if $\text{FHE.Eval}(\cdots, \cdot, \text{pk})$ and $\text{FHE-PRF}(\cdot, \text{pk}, \text{pk}_{\text{prf}})$ are unpredictable.⁵

Intuitively, Definition 8 states that for any ciphertext that an adversary produces, the evaluation and encrypted PRF are randomized enough that it is unlikely that the adversary can produce a set of inputs that yields this specific ciphertext.

Combining our unpredictability definition and the above randomized evaluation algorithm leads to another assumption.

Assumption 2. Any FHE algorithm from Figure 8 with appropriately chosen ReRand is unpredictable.

Specifically in the context of TFHE, we believe this is a reasonable assumption. For a detailed discussion of this definition and assumption we refer to [SW26].

3.3.3 De-Randomizing Eval^R

In our main MPC protocol below we will require that the computing parties perform a majority vote on the output of FHE evaluation calls, in order to ensure that the evaluation has been performed correctly (assuming a majority of the computing parties are honest). This provides a simple mechanism for Verifiable Computation of the evaluation. However, to do this we require that the evaluation procedure is deterministic. Thus, as in [SW26], we need to de-randomize the above randomization procedure. This is done essentially the same as methodology as in [SW26], see Figure 9. We assume in this section, as before, that F is a function mapping m ciphertext inputs to n ciphertext outputs. Note, the input of $\text{aux}_i^{\text{seed}}$ in DeRandEval in Figure 9 is a proof artifact; it enables bookkeeping in the following simulations.

3.3.4 DeRandEval Provides Computational Correctness in the Random Oracle Model

We shall now show that in Eval^R from Figure 8 we can replace the generation of the random seeds with the evaluation of a hash function. Thus we obtain exactly the evaluation operation of DeRandEval . The proof of the following theorem, follows roughly the template of the equivalent theorem from [SW26]. However, the introduction of the encrypted PRF leads to some differences in details, hence, we provide a full proof below.

Theorem 2. The derandomized FHE scheme from Figure 9 is computationally correct, in the sense of Definition 6, assuming

- Assumption 1 holds.
- Assumption 2 holds.
- The hash function $\text{Hash}^{2\text{-sec}}$ is modeled as a random oracle.

Proof. Let $\mathcal{A}$ be an adversary that is able to break the computational correctness of the FHE scheme from Figure 9 in the random oracle model. We will show that $\mathcal{A}$ can be

⁵In [SW26] unpredictability was defined with a second case that allowed for invertibility of FHE.Eval as long as it is generally and publicly easy to efficiently generate this ciphertext through encryption and evaluation. For simplicity, we do not consider this case here.

FHE Fully De-randomized FHE with Encrypted PRF

```

DeRandEval(((ct1, aux1seed), ..., (ctm, auxmseed)), F, pℓ):
  1. For i ∈ [1, ..., m] do
    (a) seedi ← Hash2-sec(ct0, ..., ctm-1, F, i).
  2. (rt1, ..., rtm) ← ReRand((ct1, ..., ctm), (seed1, ..., seedm), pℓ).
  3. (ct'1, ..., ct'n) ← Eval((rt1, ..., rtm), F, pℓ).
  4. auxseed ← (seed1, ..., seedm).
  5. Return ((ct'1, auxseed), ..., (ct'n, auxseed)).

DeRandFHE-PRFk',t(x, pℓ, pℓprf):
  1. (rt1, ..., rtt) ← FHE-PRFk',t(x, pℓ, pℓprf)
  2. For i ∈ [1, ..., t] do seedi ← Hash2-sec(x, k', t, i)
  3. (ct1, ..., ctt) ← ReRand((rt1, ..., rtt), (seed1, ..., seedt))
  4. auxseed ← (seed1, ..., seedt)
  5. Return ((ct1, auxseed), ..., (ctt, auxseed))

```

Figure 9: FHE Fully De-randomized FHE with Encrypted PRF

used to create an adversary $\mathcal{B}$ which breaks the computational correctness of the fully randomized FHE algorithm from Figure 8.

We first provide a sketch of the proof. The basic idea is that $\mathcal{B}$ will simulate the Cor_{prf} game to $\mathcal{A}$ as if using DeRandEval and DeRandFHE-PRF . It does so by submitting queries to its own Eval and FHE-PRF oracle, which are truly randomized but returns the seeds used for re-randomization. $\mathcal{B}$ can use these seeds to program the random oracle, such that $\mathcal{A}$'s view is consistent with DeRandEval and DeRandFHE-PRF using the random oracle. A little care must be taken, because $\mathcal{A}$ may query the random oracle itself. But the way that DeRandEval is defined, such queries are only useful when they include all inputs to DeRandEval . So, $\mathcal{B}$ can submit such queries to its evaluation oracle, obtain the seeds and program the random oracle consistently. If $\mathcal{A}$ later submits such a query to its own evaluation oracle, $\mathcal{B}$ simply looks up the result. So $\mathcal{B}$ may make the evaluation queries in a different order than $\mathcal{A}$, but since evaluation is stateless, this is not a problem.

The main difference to the proof of Theorem 2 in [SW26] is the lack of ZKPoKs. In [SW26] they were used in cases, where $\mathcal{A}$ queries the random oracle on a ciphertext that $\mathcal{B}$ has not seen so far (and is thus not in the game state). Since the calls to the random oracle must include the ZKPoK of each fresh input ciphertext (for the call to be useful), $\mathcal{B}$ can use the ZKPoKs to extract the message and the randomness it needs to add the ciphertext to the game state using the encryption oracle. Here, we handle this case differently: since the randomization at the end of FHE-PRF^R renders the function unpredictable (in a similar sense as Definition 8), we must have that for any ciphertext ct occurring in a random oracle call of the form $(\text{ct}_0, \dots, \text{ct}_1, F, i)$, there must have been another call to the random oracle of the form (x, k', t, i) such that ct is one of the t ciphertexts output by $\text{DeRandFHE-PRF}_{k',t}(x, p_{\ell}, p_{\ell}^{\text{prf}})$, except with negligible probability. The adversary $\mathcal{B}$ may use this call to add ct to the game state and can thus translate the random oracle query into an evaluation query and program the random oracle accordingly.

In more detail, the algorithm $\mathcal{B}$ runs $\mathcal{A}$ and responds to its queries in the following way:

For each evaluation and FHE-PRF query, we first check if this query has been made before. This includes queries as part of random oracle queries (see below). If so, we simply map the indices of the output ciphertexts in the simulated state to the ciphertexts in the

state that corresponds to the previous invocation. Otherwise, for new queries

- FHE-PRF queries (x, k', t) are forwarded keeping track of the correct indices⁶. When the Cor_{prf} challenger returns the ciphertexts $(\text{ct}_1, \dots, \text{ct}_t)$ and seeds $(\text{seed}_1, \dots, \text{seed}_t)$, $\mathcal{B}$ programs the random oracle to return seed_i on input (x, k', t, i) for all i , records $\text{ct}_1, \dots, \text{ct}_t$ in its internal game state state' and returns them to $\mathcal{A}$.
- Evaluation queries $(j_1, \dots, j_m, g)$ are forwarded, potentially re-mapping the indices. Upon receiving the resulting ciphertexts $(\text{ct}_1, \dots, \text{ct}_n)$ and seed_i such that

$$(\text{ct}_1, \dots, \text{ct}_n) \leftarrow \text{Eval}((\text{rt}_1, \dots, \text{rt}_m), g, \text{pk}),$$

where

$$(\text{rt}_1, \dots, \text{rt}_m) \leftarrow \text{ReRand}((\text{ct}_1, \dots, \text{ct}_m), (\text{seed}_1, \dots, \text{seed}_m), \text{pk}),$$

we set the random oracle to return seed_i on input of $(\text{state}[j_1], \dots, \text{state}[j_k], g, i)$ for all i . We then record ct_i , for $i = 1, \dots, n$, in the internal game state state' and return it to $\mathcal{A}$.

- There are two types of (useful) random oracle queries:
 1. Queries of the form (x, k', t, i) are handled identically to FHE-PRF queries, except that seed_i is returned to $\mathcal{A}$ instead of the ciphertexts
 2. For a query of the form $(\text{ct}_1, \dots, \text{ct}_m, g, k)$, we first check if all ct_i are in state . If not, choose $\text{seed} \leftarrow \mathcal{R}$, set the output to the query to seed and return it to $\mathcal{A}$. Otherwise, let $j_1, \dots, j_m$ correspond to the indices such that $\text{ct}_i = \text{state}[j_i].c$ for all i . Send evaluation query $(j_1, \dots, j_m, g)$ and receive $(c'_1, \dots, c'_n, \text{seed}_1, \dots, \text{seed}_m)$. Update the internal game state state' and program the random oracle such that on input $(c_1, \dots, c_m, g, i)$ it outputs seed_i for all $i \in [1, \dots, m]$. Return seed_k to $\mathcal{A}$.

Let E be the event that $\mathcal{A}$ submits a query $q = (\text{ct}_1, \dots, \text{ct}_m, g, k)$ to the random oracle that contains a ciphertext ct_α for some $\alpha \in (1, \dots, m)$ that is not in the internal game state state' (and thus also not in the game state state). Conditioned on the event $\neg E$, $\mathcal{B}$ responds to every one of $\mathcal{A}$'s queries with a ciphertext that is consistent with DeRandFHE-PRF and DeRandEval using the random oracle as the hash function.

The probability of E is not necessarily small, since $\mathcal{A}$ may compute an arbitrary ciphertext ct_α and query the random oracle on it. In this case, $\mathcal{B}$ is unable to add ct_α to state and thus cannot make an evaluation query on $q = (\text{ct}_1, \dots, \text{ct}_m, g)$ to obtain re-randomization randomness. However, in this case, picking a uniformly random seed is still consistent, unless $\mathcal{B}$ needs to add ct_α to the game state state at some point and then $\mathcal{A}$ submits $(j_1, \dots, j_m, g)$ to its evaluation oracle and $\text{state}[j_i].c = \text{ct}_i$ for all i . Let E' be this bad event. When $\mathcal{B}$ attempts to send this query to its own evaluation query, the resulting seed_k will be different to what $\mathcal{B}$ set the random oracle to return on $(\text{ct}_1, \dots, \text{ct}_m, g, k)$. We now analyze the probability of the bad event E' .

The only way that ciphertexts are added to the game state is through queries to the FHE-PRF and Eval oracles, so clearly for ct_α is the result of one of those queries. First, assume, that it is a query to the Eval query and let $q' = (j'_1, \dots, j'_{m'}, g')$ be the query that results in ct_α being added to state . Note that $\mathcal{A}$ must have made the query q' after query q , since otherwise ct_α would be in state when query q is submitted. Furthermore, the way that $\mathcal{B}$ handles random oracle queries, no query of the form $(\text{ct}'_1, \dots, \text{ct}'_{m'}, g', k)$ (where $\text{state}[j'_i].c = \text{ct}'_i$) could have been made to the random oracle before query q (since $\mathcal{B}$ would have sent this query to its own evaluation oracle and thus ct_α would be in state). Now consider $\mathcal{A}_1$ as the part of $\mathcal{A}$ up to the query q , where it sends ct_α . Define $\mathcal{A}_2$ to be $\mathcal{A}$ after sending query q up query q' , which is the query that results in the ciphertext ct_α being added to the game state. Clearly, $(\mathcal{A}_1, \mathcal{A}_2)$ violates Definition 8 and thus Assumption 2.

The case where ct_α is the result of a FHE-PRF query is identical. $\square$

⁶The adversary may make hash queries $H(x, k', t, i)$ before querying FHE-PRF on (x, k', t) . This case is handled with the random oracle queries below.

4 The Programming Model

We aim to enable the parties to carry out a reactive computation, meaning that the function being evaluated may depend on values that are publicly revealed during the computation. The computation itself is defined over the ring $\mathcal{R} = \mathbb{Z}/(p^k)$ supported by the FHE scheme.

Following [Sma23a], we describe the target functions not as arithmetic circuits over $\mathbb{Z}/(p^k)$, but rather as “extended” arithmetic circuits. That is, we consider circuits augmented with additional gadgets or instructions. Such circuits are referred to as F-Circuits, a notion originally introduced (as M-Circuits in the MPC setting) in [BDO⁺21]. We note, however, that we introduce some modifications to the F-Circuit framework as presented in [Sma23a].

4.1 Registers

Our machine state **state** consists of a set of registers $\mathbb{V} = \{\text{varid}_i\}$. Each register has a *fixed* type. The type consists of two values, the domain of definition, and whether it is *sensitive* (i.e. the value is encrypted under FHE).

In essence we have three main variants of the types.

- $(\mathbb{Z}/(p^{k'}), -)$: This refers to a variable which holds a non-sensitive value in the ring $\mathbb{Z}/(p^{k'})$, with $0 < k' \leq k$.
- $(\mathbb{Z}/(p^{k'}), s)$: This refers to a variable which holds a *sensitive* value in the ring $\mathbb{Z}/(p^{k'})$, with $0 < k' \leq k$.
- $(-, -)$: This refers to a variable which holds a non-sensitive integer value (in some bounded range, say $[0, \dots, 2^{64}]$).

We refer to the fixed aspects of the type of a register **varid** by the notation $\text{type}(\text{varid}) = (\text{type}^1(\text{varid}), \text{type}^2(\text{varid}))$. The integer types allow for the simple description of loop counters etc within functions; and other operations which enable a richer form of programmatic expression. In particular, there is a special integer register **pc** of type $(-, -)$. The register **pc** can be altered by a program, and will be used as the program counter.

4.2 Instructions

An instruction **instr**, in this paper, will be restricted to come from the following list

$$\text{instr} \in \{\text{Input}, \text{Output}, \text{Evaluate}, \text{Random}, \text{Declassify}, \text{Terminate}\}.$$

An instruction operates on the machine state as follows:

- **Input**(**varid**, i): This accepts an input (of the type associated with **varid**) from player $\mathcal{I}_i$ for $i \in [1, \dots, I]$. If $\text{type}^2(\text{varid}) = s$ then it is assumed that this is private input from player $\mathcal{I}_i$, otherwise it is considered public input.
- **Output**(**varid**, i): This outputs the value of **varid** to player $\mathcal{O}_i$ for $i \in [1, \dots, O]$. If $\text{type}^2(\text{varid}) = s$ then it is assumed that player $\mathcal{O}_i$ (and only this player) learns the clear value of **varid**, even though it is not stored in clear form.
- **Evaluate**($\{\text{varid}_{y_j}\}_{j=1}^n, \{\text{varid}_{x_i}\}_{i=1}^m, F$): Given a deterministic function F on the m -input variables $\{\text{varid}_{x_i}\}_{i=1}^m$ this computes the output variables $\{\text{varid}_{y_j}\}_{j=1}^n$. In terms of the type consistency we require:
 - The values of $\text{type}^1(\text{varid}_{y_j})$ are implicitly inferred from the syntax and semantics of the description of F .
 - The values of $\text{type}^2(\text{varid}_{y_j})$ are derived from standard information flow analysis; in particular a variable varid_{y_j} is marked sensitive if its computation involves reading a sensitive variable.

- **Random**($\{\text{varid}_{y_j}\}_{j=1}^n, k'$): This computes $\{\text{varid}_{y_j}\}_{j=1}^n = \text{LWR-PRF}_{k',n}(\mathbf{k}_{\text{prf}}, \text{nonce})$ for some global, but hidden, secret value $\mathbf{k}_{\text{prf}}$ and some unique-per-call (non-repeating) value nonce . It assumes that $\text{type}(\text{varid}_{y_j}) = (\mathbb{Z}/(p^{k'}), s)$.
- **Declassify**($\text{varid}_y, \text{varid}_x$): This takes an input register of type $\text{type}(\text{varid}_x) = (\mathbb{Z}/(p^{k'}), s)$ and produces an output register of type $\text{type}(\text{varid}_y) = (\mathbb{Z}/(p^{k'}), -)$, which holds the same value.
- **Terminate**: This simply stops the computation and deletes the internal state, i.e. the contents of all registers.

A program $\mathcal{P}$ is an ordered sequence of instructions, $\mathcal{P} = [\text{instr}_1, \dots, \text{instr}_l]$. We execute the program by setting $\text{pc} \leftarrow 1$ to start with and then executing the following steps in turn.

1. Fetch the instruction instr indexed by pc .
2. Increment pc by one.
3. Execute the operation defined by instr .

Note, conditional branching on non-sensitive values is allowed via assigning to the pc value via evaluating, via the **Evaluate** command, the function

$$\text{pc} \leftarrow \text{varid}_b \cdot 100 + (1 - \text{varid}_b) \cdot 200$$

which will jump to instruction 100 if $\text{varid}_b = 1$ and jump to instruction 200 if $\text{varid}_b = 0$.

Also note that, the command **Random** simply provides access to an encrypted random number generator for the program. It does not enable the programmer to repeatedly sample the same encrypted value.

4.3 Extensions

As noted in [Sma23a], the basic state machine described above can be naturally extended to support stacks of the various register types (rather than only flat collections of registers), as well as memory banks indexed by values of type $(-, -)$. It is also useful to allow stack elements to be accessed via relative indices of type $(-, -)$. These extensions are straightforward and we omit further details.

Support for subroutine execution can be incorporated by using clear integer variables to maintain a call-return stack for the program counter pc . A subroutine call corresponds to pushing the current pc onto this stack, followed by updating pc , while a return corresponds to popping from the stack and assigning the resulting value back to pc . Parameter passing can be implemented in the usual way using stacks for the relevant data types, and local variables within subroutines can likewise be handled via standard stack-based mechanisms.

A crucial aspect is that each register, stack, or memory bank has a fixed type that remains unchanged throughout the computation. Hence, any “casting” must be carried out during assignment. For convenience, we allow implicit type casting from types of the form $(\mathbb{Z}/(p^{k''}), -)$ (resp. $(\mathbb{Z}/(p^{k''}), s)$) to $(\mathbb{Z}/(p^{k'}), -)$ (resp. $(\mathbb{Z}/(p^{k'}), s)$) whenever $k'' < k$, but not in the reverse direction. Additionally, we permit implicit casting between types of the form $(\mathbb{Z}/(p^{k'}), -)$ and $(-, -)$.

4.4 Compilation

In practice, programs are written in a high-level language, and must be compiled into the representation described above. This compilation step itself introduces potential security concerns. As highlighted in [BDO⁺21] for M-Circuits, two compilations of the same high-level program, while functionally equivalent, may differ in that one preserves privacy whereas the other may leak sensitive information.

In this work, we assume that the compilation process has undergone an appropriate security analysis, ensuring that any values revealed (for instance through **Declassify** operations) do not compromise sensitive data. Consequently, our focus is on the security

of executing the resulting F-Circuit, under the assumption that it faithfully captures the intended functionality of the original program.

However, we do assume that the precise functions evaluated can be chosen by the adversary, in terms of their representations. i.e. the parties will check that obvious security problems do not arise from the semantics of the submitted programs, but not their syntax. For example, two functions could compute the same allowed function, but one could result (by carefully crafting of the valid operations) in a failure of the correctness of the FHE operations. The adversary could select the function which had a greater probability of failure of correctness, despite the two functions being semantically equivalent. In practice this problem will be eliminated due to our FHE schemes correctness notions.

Our security requirement is the standard robust MPC requirement that the adversary only learns data which is revealed to it via the program (i.e. any declassified values, and values which it enters as inputs, or receives as output, and any values which can be deduced from that), in addition if the adversary deviates from the program then the honest parties will still compute the values as required. This is captured in an ideal functionality given in Section 5.2.

5 Ideal Functionalities

In this section we present three ideal functionalities, the first two are the standard ideal functionalities for (robust) threshold FHE, whilst the third is the ideal functionality which captures the execution of an F-Circuit “program”.

5.1 Threshold-FHE Functionality

Formally, we model threshold decryption using two ideal functionalities. The first, $\mathcal{F}_{\text{KeyGen}}$ (see Figure 10), serves as a setup assumption for our protocol and is required for the UC proof. It produces a key pair $((\mathbf{pk}, \mathbf{pk}_{\text{prf}}), (\mathbf{sk}, \mathbf{k}_{\text{prf}}))$, and distributes shares of the FHE secret key $\mathbf{sk}$ among the computing parties $\mathbb{T}$ via a linear secret sharing scheme $\langle \cdot \rangle$ that tolerates up to $t_{\mathbb{T}}$ corruptions. The share of a value x held by party $\mathcal{T}_i$ is denoted $\langle x \rangle_i$.

This functionality can be instantiated using a general MPC protocol; see, for example, [BCD⁺25]. Note that, although we aim for active security, we do not “complete” adversarial input shares into a full sharing (as is commonly done), since we may assume that the MPC protocol implementing $\mathcal{F}_{\text{KeyGen}}$ does not require this step. Also note that we add the generation of the keys for the encrypted PRF $(\mathbf{pk}_{\text{prf}}, \mathbf{k}_{\text{prf}})$ into the functionality.

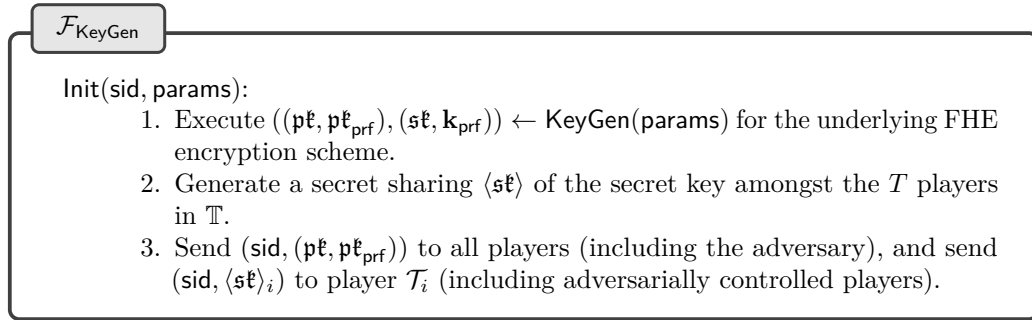


Figure 10: The ideal functionality for distributed key generation

The key functionality required in the following is $\mathcal{F}_{\text{KeyGenDec}}$ given in Figure 11. Note, that this functionality always returns the correct result, irrespective of what the adversary does. The functionality also requires that a majority of a set of privileged parties query

DistDecrypt before it executes the command, this is to ensure that the functionality is not triggered by a single adversarial party. A protocol which realises $\mathcal{F}_{\text{KeyGenDec}}$ in the $\mathcal{F}_{\text{KeyGen}}$ -hybrid model is given in [DDE⁺23] when $t_{\mathbb{T}} < T/3$, for all the FHE schemes above.

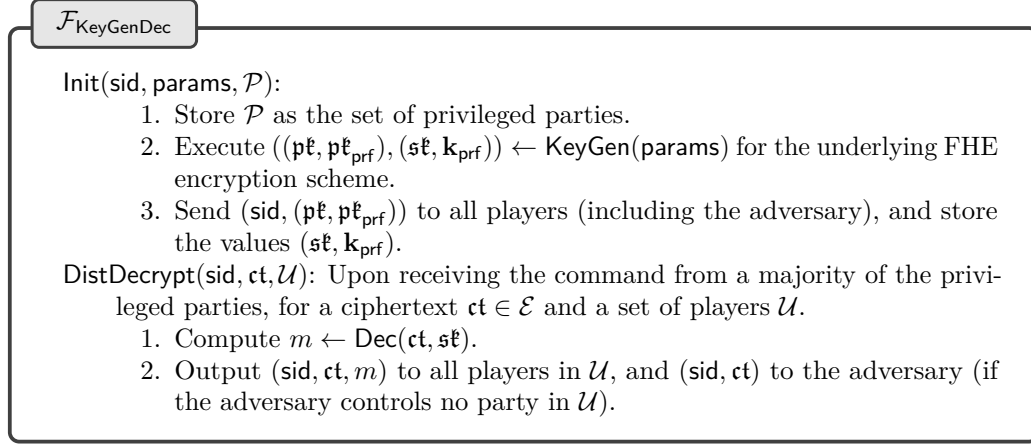


Figure 11: The ideal functionality for distributed key generation and decryption

The functionality $\mathcal{F}_{\text{KeyGenDec}}$ implies that if the target set of players $\mathcal{U}$ in the DistDecrypt cannot be adversarially corrupted, then, by definition, the functionality is connected to the players in $\mathcal{U}$ via private channels. When the parties in $\mathbb{T}$ realize this functionality, it follows that (for such sets $\mathcal{U}$) communication between $\mathbb{T}$ and $\mathcal{U}$ must occur over private channels.

In our protocol the functionality is used in two situations:

- When $\mathcal{U}$ consists of a single output party. This corresponds to using DistDecrypt to deliver a private output to that party. As not all output parties may be corrupted, private channels between the functionality and the party in $\mathcal{U}$ are required.
- When $\mathcal{U}$ is the set of computing parties $\mathbb{C}$. As it is not security critical if the returned value is private, we only need authentic channels between the functionality and the parties in $\mathbb{C}$.

5.2 Ideal F-Circuit Execution Functionality

The ideal functionality, shown in Figure 12 below, precisely captures the behavior we expect from an ideal execution of the F-Circuit described in Section 4. We emphasize that we do not claim the F-Circuit itself guarantees application-level security, this responsibility lies with the compiler. Our goal here is solely to define, at an abstract level, what capabilities an execution of the F-Circuit should provide. To cope with the reactive nature of our protocol, whenever the functionality receives a command from a party, it sends a, possibly modified version, of the command to all other parties, including the adversary. This makes it possible for the simulator to properly react to honest parties' behavior.

As before, we assume a register bank $\mathbb{V} = \{\text{varid}_i\}$, modeled for simplicity as a single flat collection of registers, without stacks or indexable memory. The ideal functionality maintains a distinction between registers that are in the clear and those that are intended to remain securely encapsulated within the functionality.

For ease of presentation, we allow the ideal functionality to output an **abort**. However, such a situation would only arise from an ill-typed program, which should already be detected by the compiler. Therefore, these **abort** outputs should be viewed as a notational convenience for the reader, rather than as genuine behaviors of the functionality itself.

Also note that, the ideal functionality ‘implements’ an ideal PRF under the Random command, i.e. not the actual PRF LWR-PRF, but an ideal PRF.

$\mathcal{F}_{F\text{-Circuit}}$

Init: On input $(\text{sid}, \text{Init}, \mathcal{R} = \mathbb{Z}/(p^k), \text{policy})$, store $(\mathcal{R}, \text{policy})$.

Input: On input of $(\text{sid}, \text{Input}, \mathcal{I}_i, \text{varid}, x)$ from $\mathcal{I}_i \in \mathbb{I}$ and $(\text{sid}, \text{Input}, \mathcal{I}_i, \text{varid}, -)$ from more than one half of the parties in $\mathbb{C}$:

1. If $\text{type}^1(\text{varid}) = \mathbb{Z}/(p^{k'})$ and $x \in \mathbb{Z}/(p^{k'})$ then store x in **varid**.
With a caveat here in relation to the implicit coercions mentioned earlier.
2. If $\text{type}^1(\text{varid}) = -$ and $x \in \mathbb{Z}$ then store x in **varid**.
3. If $\text{type}^2(\text{varid}) = -$ then send $(\text{sid}, \text{Input}, \mathcal{I}_i, \text{varid}, x)$ to all parties, otherwise send $(\text{sid}, \text{Input}, \mathcal{I}_i, \text{varid})$ to all parties.
4. If $x = \perp$ then store the value zero in **varid**.

This last case corresponds to an adversarial input x which is mistyped.

Output: On input of $(\text{sid}, \text{Output}, \mathcal{O}_i, \text{varid})$ for $\mathcal{O}_i \in \mathbb{O}$ and from more than one half of the parties in $\mathbb{C}$:

1. If $(\text{sid}, \text{Output}, \mathcal{O}_i, \text{varid}) \notin \text{policy}$ then do nothing.
2. Send the value x stored in **varid** to $\mathcal{O}_i$ and $(\text{sid}, \text{Output}, \mathcal{O}_i, \text{varid})$ to all the parties.
3. If $\text{type}^2(\text{varid}) = -$ then also send x to the adversary.

Evaluate: On input of $(\text{sid}, \text{Evaluate}, \{\text{varid}_{y_j}\}_{j=1}^n, \{\text{varid}_{x_i}\}_{i=1}^m, F)$ from a party and from more than one half of the parties in $\mathbb{C}$:

1. If $(\text{sid}, \text{Evaluate}, \{\text{varid}_{y_j}\}_{j=1}^n, \{\text{varid}_{x_i}\}_{i=1}^m, F) \notin \text{policy}$ then do nothing.
2. Send $(\text{sid}, \text{Evaluate}, \{\text{varid}_{y_j}\}_{j=1}^n, \{\text{varid}_{x_i}\}_{i=1}^m, F)$ to all parties.
3. If F is not a deterministic function F on the m input variables, producing n output variables, for which the types of the inputs and outputs match the values of $\text{type}(\cdot)$ of the associated **varid**’s, then ignore.
4. For $i = 1, \dots, m$, retrieve x_i from **varid** $_{x_i}$.
5. Compute $(y_1, \dots, y_n) \leftarrow F(x_1, \dots, x_m)$.
6. For $j = 1, \dots, n$, store y_j in **varid** $_{y_j}$.

Random: On input of $(\text{sid}, \text{Random}, \{\text{varid}_{y_j}\}_{j=1}^n, k')$ from a party and from more than one half of the parties in $\mathbb{C}$:

1. If $(\text{sid}, \text{Random}, \{\text{varid}_{y_j}\}_{j=1}^n, k') \notin \text{policy}$ then do nothing.
2. Send $(\text{sid}, \text{Random}, \{\text{varid}_{y_j}\}_{j=1}^n, k')$ to all parties.
3. If for some $j \in [1, \dots, n]$ we have $\text{type}(\text{varid}_{y_j}) \neq (\mathbb{Z}/(p^{k'}), s)$ then abort.
Again, with a caveat here in relation to the implicit coercions mentioned earlier.
4. For $j = 1, \dots, n$ generate $y_j \leftarrow \mathbb{Z}/(p^{k'})$ and store y_j in **varid** $_{y_j}$.

Declassify: On input of $(\text{sid}, \text{Declassify}, \text{varid}_y, \text{varid}_x)$ from a party and from more than one half of the parties in $\mathbb{C}$:

1. If $(\text{sid}, \text{Declassify}, \text{varid}_y, \text{varid}_x) \notin \text{policy}$ then do nothing.
2. Send $(\text{sid}, \text{Declassify}, \text{varid}_y, \text{varid}_x)$ to all parties.
3. If $\text{type}(\text{varid}_x) \neq (\mathbb{Z}/(p^{k'}), s)$ or $\text{type}(\text{varid}_y) \neq (\mathbb{Z}/(p^{k'}), -)$ then abort.
4. Assign the contents x of **varid** $_x$ to **varid** $_y$ and send x to the adversary if $\mathbb{C}$ contains a corrupt party.

Terminate: On input of $(\text{sid}, \text{Terminate})$ from all parties:

1. This simply stops the computation and deletes the internal state, i.e. the contents of all registers.

Figure 12: The ideal functionality for executing a program represented by a F-Circuit

The reader will notice that the ideal functionality refers to a policy policy . This is defined at the start of the protocol and it defines what types of functions and outputs/declassifications are allowed by the functionality. This is needed to avoid trivial attacks due to the ability for parties to request the declassification of data, or evaluation of arbitrary functions. For example without such a policy an adversary could request the declassification of an honest players input.

The reader should think of the policy as defining what possible programs can be executed, which do not result in trivial breaks, i.e. any data leakage which is consistent with the given policy is allowed.

Since at least half of the parties in $\mathbb{C}$ are considered to be honest and have to “input” the commands on the ideal functionality, this checking of policy could be removed from the ideal functionality and be asked to be executed by the parties in $\mathbb{C}$ beforehand. However, to make it clear that there is some implicit policy as to what commands are allowed, we include statements to check the policy within the ideal functionality.

6 The MPC-FHE Protocol

Here, we combine all the components together into one protocol which executes the F-Circuit functionality. The protocol is given in Figure 13–Figure 15, where we assume the input F-Circuit is validly typed. The protocol is given in the $\{\mathcal{F}_{\text{KeyGenDec}}\}$ -hybrid model, and works in the robust setting, i.e. the protocol should not abort, under malicious adversaries. When we specify **abort** below, this is due to bad typing of the input operations.

All players are assumed to be connected by authentic channels, except in the following cases:

- Messages from $\mathcal{F}_{\text{KeyGenDec}}$ to those in $\mathbb{I}$ and $\mathbb{O}$ are assumed sent by private channels.
- Messages from $\mathcal{F}_{\text{KeyGenDec}}$ to those in $\mathbb{C}$ are assumed sent by authentic channels, as the values sent are public.
- Messages from the players in $\mathbb{I}$ to the players in $\mathbb{C}$ are assumed to be by *reliable broadcast channels*, which are reliable in the presence of Byzantine faults, including by the sender. This can be ensured by the players in $\mathbb{C}$ executing an echo-broadcast upon receiving data from a player in $\mathbb{I}$, to ensure the value has indeed been broadcast correctly. As $\mathbb{C}$ contains an honest majority, the players in $\mathbb{C}$ can take a majority vote on what the precise input is, or declare the party’s input to be invalid.

Input of data is divided into three routines, the first one InitInput_1 initializes for each user $\mathcal{I}_i$ in $\mathbb{I}$ an input context j for $j \in \mathbb{N}$. The underlying key for the PRG is obtained by evaluating the encrypted PRF. This sets up a homomorphically encrypted PRG key held by the entities in $\mathbb{C}$. Note that all n_{prg} key elements are obtained from a single invocation of the encrypted PRF, as are the n outputs of a call to **Random**; the parameters must hence be chosen so that the bound T of **params** satisfies $T \geq \max(n_{\text{prg}}, n)$ for every n occurring in a call to **Random**. In line 3 of InitInput_1 , the homomorphic evaluation in the case of AES-CTR could be homomorphically evaluating the AES key schedule, whilst in the case of Kreyvium it is homomorphically evaluating the warm-up phase.

The routine InitInput_2 allows the user $\mathcal{I}_i$ to obtain the same PRG state, but in the clear. Then in the **Input** routine the user actually enters data, using the clear PRG-state on their end, and the encrypted PRG-state by the entities in $\mathbb{C}$. The use of multiple contexts j per player, i.e. with distinct channels indexed by j , allows the user to maintain different local states perhaps depending on which device they are using to communicate with the system. If the reader wishes they can think of setting $j = 0$ always.

Note, clearly for correct operation the order of execution must be InitInput_1 , followed by InitInput_2 , and then many executions of **Input**. However, an adversarial user could invoke the operations in the order InitInput_1 , then some invocations of **Input**, then InitInput_2 and then some more invocations of **Input**. In this setting the adversarial user will essentially

$\Pi_{F\text{-Circuit}}$ (Part I)

Init(sid, params):

1. The players call Init(sid, params, $\mathbb{C}$) on $\mathcal{F}_{\text{KeyGenDec}}$ so that all players obtain $(\mathbf{pk}, \mathbf{pk}_{\text{prf}})$ and the functionality uses $\mathbb{C}$ as the privileged set.
2. Sample $\text{cnt} \leftarrow [0, \dots, 2^{\text{sec}})$. This can either be done via a coin-tossing protocol, or alternatively cnt could be set to a fixed initial value of, say, zero.

InitInput₁(sid, i, j):

1. The parties in $\mathbb{C}$ invoke $\text{DeRandFHE-PRF}_{1, n_{\text{prg}}}(\text{cnt}, \mathbf{pk}, \mathbf{pk}_{\text{prf}})$, with the resulting n_{prg} encrypted variables being assigned to $\{\text{varid}_{k_{\text{prg}}}^{i,j,\ell}\}_{\ell=1}^{n_{\text{prg}}}$, each of type $(\mathbb{Z}/(p), s)$.
2. $\text{cnt} \leftarrow \text{cnt} + 1$.
3. The parties in $\mathbb{C}$ execute $\text{DeRandEval}(\{\text{varid}_{k_{\text{prg}}}^{i,j,\ell}\}_{\ell=1}^{n_{\text{prg}}}, \text{PRG.Init}, \mathbf{pk})$. The resulting public $\text{state}_{\text{pub}}$ variables are assigned to varid 's of type $(\mathbb{Z}/(p^{k_{\text{prg}}}), -)$, whilst the resulting encrypted state variables in $\text{state}_{\text{priv}}$ are assigned to varid 's of type $(\mathbb{Z}/(p^{k_{\text{prg}}}), s)$. To ease notation these sets of varid 's will be denoted by $\text{state}_{\text{pub}}^{i,j}$ and $\text{state}_{\text{priv}}^{i,j}$.

InitInput₂(sid, i, j):

1. On receiving the instruction from $\mathcal{I}_i$, parties in $\mathbb{C}$ call $\mathcal{F}_{\text{KeyGenDec}}.\text{DistDecrypt}(\text{sid}, \text{ct}_{k_{\text{prg}}}^{i,j,\ell}, \{\mathcal{I}_i\})$ for $\ell = 1, \dots, n_{\text{prg}}$ where $\text{ct}_{k_{\text{prg}}}^{i,j,\ell}$ is the ciphertext contained in the registers $\text{varid}_{k_{\text{prg}}}^{i,j,\ell}$.
2. Party $\mathcal{I}_i$ denotes the received values as a PRG key $\mathbf{k}_{i,j} \in (\mathbb{Z}/(p))^{n_{\text{prg}}}$. *Note, that $\mathcal{I}_i$ only receiving the value requires private channels from the entities in $\mathbb{T}$ to the entities in $\mathbb{I}$.*
3. The party $\mathcal{I}_i$ executes $\text{PRG.Init}(\mathbf{k}_{i,j})$ in-the-clear, in order to obtain a local in-the-clear value of $\text{state}_{\text{priv}}^{i,j}$, as well as the in-the-clear value $\text{state}_{\text{pub}}^{i,j}$.

Figure 13: The protocol for executing a program represented by a F-Circuit – Part I

be inputting, in the first invocations of **Input**, unknown random values. However, in our simulation we will ensure that the input random values are consistent with subsequent values decrypted.

Note: The ciphertexts contained in $\{\text{varid}_{k_{\text{prg}}}^{i,j,\ell}\}_{\ell=1}^{n_{\text{prg}}}$ and in the encrypted state $\text{state}_{\text{priv}}^{i,j}$ held by the entities in $\mathbb{C}$ cannot be accessed, or written to, by independent calls to the procedures **Evaluate**, **Random** or **Declassify**, as they are “local” variables purely for use only by the input routines. In the same way the public state values $\text{state}_{\text{pub}}^{i,j}$ may not be written to as the output of **Evaluate** or **Declassify**.

6.1 Removing the PRG

For case where using a PRG is not desirable, either due to complex state synchronization between the input party and the computing parties, or the heavy cost of warming up a PRG such as Kreyvium, a stateless protocol may be preferred. We outline the four-part modification needed for Figure 13–Figure 15 to support a version of the protocol that does not require a PRG. Instead, we will use the PRF output directly to encrypt the plaintext. This version of the protocol already appeared in [Sma23b].

1. Skip every step that uses **PRG.Init** and **PRG.Update**: the mask y used in **Input** is now the PRF output itself, so no PRG state is maintained and $\text{state}_{\text{pub}}$, $\text{state}_{\text{priv}}$ remain

$\Pi_{F\text{-Circuit}}$ (Part II)

Input(sid, $\mathcal{I}_i$, varid, x, j): When player $\mathcal{I}_i$ wishes to input a value x of valid type $\text{type}(\text{varid})$, with respect to its j -th input channel, we execute:

1. If $\text{type}^2(\text{varid}) = -$

- (a) Player $\mathcal{I}_i$ *broadcasts* x , in the clear, to all players in $\mathbb{C}$.
- (b) The parties in $\mathbb{C}$ store the received x in the register **varid**.

2. Else

- (a) Party $\mathcal{I}_i$ executes

$$(y, \text{state}_{\text{pub}}^{i,j}, \text{state}_{\text{priv}}^{i,j}) \leftarrow \text{PRG.Update}(k', 1, \text{state}_{\text{pub}}^{i,j}, \text{state}_{\text{priv}}^{i,j})$$

to obtain the output $y \in \mathbb{Z}/(p^{k'})$, where $\text{type}^1(\text{varid}) = \mathbb{Z}/(p^{k'})$.

- (b) Party $\mathcal{I}_i$ computes $z \leftarrow y + x$.
- (c) Party $\mathcal{I}_i$ *broadcasts* $z \in \mathbb{Z}/(p^{k'})$, in the clear, to all players in $\mathbb{C}$.
- (d) If **InitInput**₁(sid, i, j) has not yet been executed, i.e. the parties in $\mathbb{C}$ do not hold any encrypted value of $\text{state}_{\text{priv}}^{i,j}$,
 - i. The parties in $\mathbb{C}$ let ct_x denote a default encryption of zero.
- (e) Else
 - i. The parties in $\mathbb{C}$ homomorphically execute

$$(\text{ct}_y, \text{state}_{\text{pub}}^{i,j}, \text{state}_{\text{priv}}^{i,j}) \leftarrow \text{PRG.Update}(k', 1, \text{state}_{\text{pub}}^{i,j}, \text{state}_{\text{priv}}^{i,j}),$$

using their public value of $\text{state}_{\text{pub}}^{i,j}$ and its homomorphically encrypted value of $\text{state}_{\text{priv}}^{i,j}$.

- ii. The parties in $\mathbb{C}$ homomorphically evaluate $\text{ct}_x \leftarrow z - \text{ct}_y$.
- (f) The parties in $\mathbb{C}$ store ct_x in the register **varid**.

Figure 14: The protocol for executing a program represented by a F-Circuit – Part II

$\perp$ throughout.

2. Combine **InitInput**₁, **InitInput**₂ and **Input** from Figure 13–Figure 15 into a single routine.
3. Add domain separation for calls to DeRandFHE-PRF via **Random** and via **Input**, e.g., 1 is used in DeRandFHE-PRF($1 \parallel \text{cnt}, \text{pk}, \text{pk}_{\text{prf}}$) when called by **Random** and 0 is used in DeRandFHE-PRF when called by **Input**.
4. When calling DeRandFHE-PRF from **Input**, use

$$\text{DeRandFHE-PRF}_{k',1}(0 \parallel \text{sid} \parallel i \parallel j \parallel \text{nonce}, \text{pk}, \text{pk}_{\text{prf}}),$$

where every element of the string has a fixed length, $\text{nonce} \in \{0, 1\}^{2 \cdot \text{sec}}$ is sampled uniformly at random by the input party. The single resulting encrypted variable is of type $(\mathbb{Z}/(p^{k'}), s)$ and plays the role of ct_y in **Input**; in particular there is no longer a PRG key $\{\text{varid}_{k_{\text{prg}}^{i,j,\ell}}\}_{\ell=1}^{n_{\text{prg}}}$ of type $(\mathbb{Z}/(p), s)$. If $\mathcal{I}_i$ inputs m values of the same type within one routine, use $\text{DeRandFHE-PRF}_{k',m}$ and take one output per value.

After the above modification, the new protocol can be summarized below. For every input, the input party $\mathcal{I}_i$ sends a fresh **nonce** to the parties in $\mathbb{C}$ using a reliable broadcast channel from $\mathbb{I}$ to $\mathbb{C}$; the parties in $\mathbb{C}$ evaluate the encrypted PRF to obtain ct_y , and a majority of them submit the same ct_y to $\mathcal{F}_{\text{KeyGenDec}}$; the parties in $\mathbb{T}$ realizing this functionality threshold-decrypt ct_y and privately return y to $\mathcal{I}_i$; finally, $\mathcal{I}_i$ broadcasts $z \leftarrow y + x$ to the parties in $\mathbb{C}$, who homomorphically remove the mask by computing

$\Pi_{F\text{-Circuit}}$ (Part III)

Output(sid, $\mathcal{O}_i$, varid): When player $\mathcal{O}_i$ is expecting an output value x of valid type $\text{type}(\text{varid})$ we execute:

1. If $\text{type}^2(\text{varid}) = -$ then
 - (a) The contents of register **varid** is sent to player $\mathcal{O}_i$ by all players in $\mathbb{C}$.
 - (b) Player $\mathcal{O}_i$ takes the majority verdict as the value of the register sent.
 - (c) The value sent is the output value.
2. If $\text{type}^2(\text{varid}) = s$ then
 - (a) Parties in $\mathbb{C}$ call $\mathcal{F}_{\text{KeyGenDec}}.\text{DistDecrypt}(\text{sid}, \text{ct}_x, \{\mathcal{O}_i\})$, where ct_x is the content of register **varid**.
 - (b) The party $\mathcal{O}_i$ receives the output plaintext value, and takes this as the output value.

*Note, that $\mathcal{O}_i$ only receiving the value implies **private channels** between $\mathcal{F}_{\text{KeyGenDec}}$ and the entities in $\mathbb{O}$.*

Evaluate(sid, $(\{\text{varid}_{y_j}\}_{j=1}^n, \{\text{varid}_{x_i}\}_{i=1}^m, F)$):

1. The players in $\mathbb{C}$, for $i \in [1, \dots, m]$, retrieve the ciphertexts containing x_i from **varid** $_{x_i}$.
2. The players in $\mathbb{C}$ compute $(y_1, \dots, y_n) \leftarrow \text{DeRandEval}((x_1, \dots, x_m), F, \text{pt})$.
3. The players in $\mathbb{C}$, for $j \in [1, \dots, n]$, store y_j in **varid** $_{y_j}$.

Random(sid, $\{\text{varid}_{y_j}\}_{j=1}^n, k')$:

1. The players in $\mathbb{C}$ invoke $\text{DeRandFHE-PRF}_{k',n}(\text{cnt}, \text{pt}, \text{pt}_{\text{prf}})$, with the resulting n encrypted variables being assigned to $\{\text{varid}_{y_j}\}_{j=1}^n$.
2. $\text{cnt} \leftarrow \text{cnt} + 1$.

Declassify(sid, **varid** $_y$, **varid** $_x$):

1. The players in $\mathbb{C}$ retrieve ct_x from **varid** $_x$ and call $\mathcal{F}_{\text{KeyGenDec}}.\text{DistDecrypt}(\text{sid}, \text{ct}_x, \mathbb{C})$, where ct_x is the agreed content of register **varid** $_x$ held by a majority of the parties in $\mathbb{C}$.
2. The public result y is sent back to all parties in $\mathbb{C}$, and assigned to the register **varid** $_y$.

Terminate:

1. This simply stops all parties and deletes the internal state of all parties.

Figure 15: The protocol for executing a program represented by a F-Circuit – Part III

$\text{ct}_x \leftarrow z - \text{ct}_y$. Note that just like the old InitInput_2 from Figure 13, the majority of $\mathbb{C}$ must have computed the same ct_y to call DistDecrypt .

The additive pad used in the secret-input routine can also be replaced by an one-time-secure symmetric encryption scheme **Sym**. In this variant, y is used as the encryption key: the input party reliably broadcasts $z \leftarrow \text{Sym.Enc}_y(x)$, and the parties in $\mathbb{C}$ homomorphically evaluate $\text{Sym.Dec}_y(z)$ using ct_y to obtain ct_x . This generalization is useful when the decryption algorithm of **Sym** is efficient under homomorphic evaluation and a long input needs to be submitted. The current operations $z = y + x$ and $x = z - y$ are the special case in which **Sym** is the additive one-time pad.

7 Security Proof

This proof is in the UC model, so we first outline the basic overall structure. As well as the honest parties executing a protocol we have an adversarial party $\mathcal{A}$ which performs

the actions for the parties which are adversarially controlled. The adversary $\mathcal{A}$ mounts an attack on the protocol, by statically corrupting⁷ as many input and output players as it wants, as well as up to t_C of the computing parties $\mathbb{C}$.

In addition there is an additional entity $\mathcal{Z}$ called the environment. The environment sits in one of two worlds; a real world or an ideal world. The goal of the environment is to output a single bit. If $\mathcal{Z}$ outputs one, then it thinks it is sitting in a real world. On the other hand if $\mathcal{Z}$ outputs zero, then it thinks it is sitting in a simulated world.

When $\mathcal{Z}$ is sitting in the *real world* it selects the honest parties inputs, which it gives to the honest parties. The environment also receives the outputs of the honest parties at the end of the protocol execution. However, the environment does not get to see the random tapes/internal variables (e.g. the keys, key shares etc) of the honest parties. The environment executes the adversarial entity and during the attack, $\mathcal{Z}$ takes complete control over the actions of the corrupted parties.

When $\mathcal{Z}$ is sitting in the *ideal world*, there is now another entity called a simulator $\mathcal{S}$. The simulator must simulate the environment's view of the protocol so that it looks like what $\mathcal{Z}$ would see in a real attack. The simulator has access to the ideal functionality $\mathcal{F}_{F-Circuit}$ from Figure 12 on behalf of the corrupted parties. The environment in the ideal world still selects the honest parties inputs, which it gives to the (dummy) honest parties. If the inputs are private, these inputs are passed directly to the ideal functionality, and are not available to the simulator. Again during the attack, the environment takes complete control over the actions of the corrupted parties. Once again, at the end of the protocol, the environment $\mathcal{Z}$ gets access to the outputs of the honest parties, by having them directly passed to $\mathcal{Z}$ from the ideal functionality. The simulator $\mathcal{S}$ does not get access to the private outputs of the honest parties, but does get access to the public outputs of all parties.

Our protocol $\Pi_{F-Circuit}$, from Figure 13–Figure 15, is said to (computationally) securely implement $\mathcal{F}_{F-Circuit}$, if a poly-time bounded environment $\mathcal{Z}$ outputs one with essentially the same probability in the real and the ideal process.

In a $\mathcal{G}$ -hybrid model, the protocol executes just as in the real world, however the parties have access to an ideal functionality $\mathcal{G}$ for some sub-task. During the protocol, the parties communicate with $\mathcal{G}$ as in the ideal world. The main UC-composition theorem [Can00, Can01] says that an ideal functionality proved secure in the $\mathcal{G}$ -hybrid model, will be secure when the sub-task represented by $\mathcal{G}$ is realized by a UC-secure protocol that realizes $\mathcal{G}$.

The formal statement we aim to prove is

Theorem 3. *The protocol $\Pi_{F-Circuit}$ (in Figure 13–Figure 15) UC realizes the functionality $\mathcal{F}_{F-Circuit}$ from Figure 12 in the $\{\mathcal{F}_{KeyGenDec}\}$ -hybrid model, assuming the underlying FHE encryption scheme with encrypted PRF is both IND-CPA and computationally correct (in the sense of Definition 6), the function PRG is a secure PRG, LWR-PRF is a secure PRF, $t_C < C/2$, and DeRandEval is deterministic.*

More formally, for any environment $\mathcal{Z}$ there are algorithms $\mathcal{A}_1, \dots, \mathcal{A}_5$ such that,

$$\begin{aligned} \text{Adv}^{\mathcal{Z}} &:= \Pr[1 \leftarrow \mathcal{Z}(\text{Real})] - \Pr[1 \leftarrow \mathcal{Z}(\text{Ideal})] \\ &\leq \text{Adv}_{\text{Cor}_{\text{prf}}}^{\mathcal{A}_1}(\text{sec}, \ell_{\text{Random}} + I \cdot \ell_J, \ell_{\text{Evaluate}} + I \cdot \ell_J) \\ &\quad + \text{Adv}_{\text{IND-CPA}}^{\mathcal{A}_2}(\text{sec}, \ell_{\text{Random}} + I_A \cdot \ell_J) \\ &\quad + \text{Adv}_{\text{PRF}}^{\mathcal{A}_3}(\text{sec}, \ell_{\text{Random}} + I_A \cdot \ell_J) \\ &\quad + (I - I_A) \cdot \ell_J \cdot \text{Adv}_{\text{PRG}}^{\mathcal{A}_4}(\text{sec}, \ell_{\text{Input}}, 1) \\ &\quad + \text{Adv}_{\text{IND-CPA}}^{\mathcal{A}_5}(\text{sec}, (I - I_A) \cdot \ell_J \cdot \ell_{\text{Input}}). \end{aligned}$$

where

⁷Static corruption implies that *all* the corrupted parties are actually defined at the start of the protocol.

- ℓ_{Random} is the number of calls to the operation **Random** made by the program,
- ℓ_{Evaluate} is the number of calls to the operation **Evaluate** made by the program,
- ℓ_{Input} is the maximum number of calls to **Input** for a fixed (i, j) value.
- ℓ_J is the maximum number of j values used in InitInput_1 etc per party,
- I_A is the number of dishonest input parties.

Before proceeding with the proof we note that if $t_{\mathbb{T}} < T/3$ then there is a UC-secure protocol which implements the functionality $\mathcal{F}_{\text{KeyGenDec}}$, i.e. the one from [DDE⁺23]. However, in our hybrid model, $\mathbb{T}$ is abstracted away by the ideal functionality $\mathcal{F}_{\text{KeyGenDec}}$, and thus we do not need to worry about the threshold $t_{\mathbb{T}}$ in our proof.

Proof. We let $\mathbb{C}_A, \mathbb{I}_A$ and $\mathbb{O}_A$ denote the adversarially controlled parties in $\mathbb{C}, \mathbb{I}$ and $\mathbb{O}$ respectively. Since $t_{\mathbb{C}} < C/2$, the homomorphic **DeRandEval** algorithm is deterministic, and the protocol uses an honest majority mechanism to detect the correct outputs of the parties in $\mathbb{C}$ we need not worry about adversarial behaviour of the parties in $\mathbb{C}$. This fact, combined with the fact that we are working in the $\{\mathcal{F}_{\text{KeyGenDec}}\}$ -hybrid model, means that the only dishonest parties we need to consider are those in the sets $\mathbb{I}$ and $\mathbb{O}$. However, the parties in $\mathbb{O}$ only act passively in our protocol.

The simulator is given in Figure 16–Figure 17. It will play the honest player’s parts, whilst the environment $\mathcal{Z}$ will play the parts of the corrupt players⁸. The simulator runs an internal copy of $\mathcal{F}_{\text{KeyGen}}$ in order to simulate calls to these functionalities.

The key problem in creating the simulator, is that to call the ideal functionality, the simulator needs to somehow extract the dishonest parties’ private inputs from the output of the adversary, in order to pass the adversarial inputs to the ideal functionality. However, since the simulator knows the underlying secret key of the PRG it can decrypt every z value sent on behalf of the corrupt players. This can then be used to obtain the required input, for the corrupt parties, to the functionality $\mathcal{F}_{F\text{-Circuit}}$.

The proof will proceed via a sequence of games.

Game $\mathbb{G}_0$: In the first game $\mathbb{G}_0$ we model a situation that is perfectly indistinguishable from the real world for the environment $\mathcal{Z}$. The environment $\mathcal{Z}$ is interacting with a “challenger” $\mathcal{X}_0$ which (for reasons which will become apparent later) we will divide into two entities $\mathcal{X}_0 = (\mathcal{S}_0, \mathcal{B}_0)$. The entity $\mathcal{S}_0$ controls the hybrid functionality $\mathcal{F}_{\text{KeyGenDec}}$ and interacts with the parties controlled by the environment $\mathcal{Z}$, whilst $\mathcal{B}_0$ controls the honest parties and will be performing “book-keeping”.

The entity $\mathcal{S}_0$ executes the key generation algorithm of the functionality $\mathcal{F}_{\text{KeyGenDec}}$, to obtain $\text{pk}, \text{pk}_{\text{prf}}, \text{sk}$ and k_{prf} . It passes $(\text{pk}, \text{pk}_{\text{prf}})$ to the adversary.

Whenever an honest party executes a command, the entity $\mathcal{B}_0$ notifies $\mathcal{S}_0$ of this, so that $\mathcal{S}_0$ knows it needs to simulate the honest parties behavior to the adversary. The challenger answers the environments queries and inputs as follows:

- When the environment passes to the challenger the honest parties inputs these are assumed to go directly to $\mathcal{B}_0$. Then $\mathcal{B}_0$ passes them to $\mathcal{S}_0$. The entity $\mathcal{S}_0$ runs the real protocol, with these inputs, in order to simulate the input of the said inputs to the environment.
- When the environment executes the input routines for parties in $\mathbb{I}_A$, the entity $\mathcal{S}_0$ can decrypt them, since it knows all the associated the secret keys sk and k_{prf} . This is true, even if the adversary executes an **Input** command, before the appropriate InitInput_2 command has been executed. The entity $\mathcal{S}_0$ then passes the dishonest players inputs to $\mathcal{B}_0$.
- When the **Random** command is executed, the entity $\mathcal{S}_0$ can obtain the underlying LWR-PRF value, as it knows k_{prf} . This underlying PRF value is passed to entity $\mathcal{B}_0$.

⁸Thus, for simplicity of exposition, we assume only one adversarial entity, and do not split into an “environment” and an “adversary”.

$\mathcal{S}_{F-Circuit}$ (Part I)

Init(sid):

1. The simulator simulates the calls to $\mathcal{F}_{KeyGenDec}$ by internally running the algorithm **KeyGen**, in order to obtain the pair of key pairs $((\mathbf{pk}, \mathbf{pk}_{prf}), (\mathbf{sk}, \mathbf{k}_{prf}))$, as well as shares $\langle \mathbf{sk} \rangle_i$. The pair $(\mathbf{pk}, \mathbf{pk}_{prf})$ is sent to all players.
2. $\text{cnt} \leftarrow [0, \dots, 2^{\text{sec}})$.

InitInput₁(sid, i, j):

1. There is nothing to simulate here. The honest parties in $\mathbb{C}$ simply follow the protocol, by executing the call to **DeRandFHE-PRF** and performing the homomorphic evaluation of **PRG.Init**.
2. The simulator generates $\{k_{prg}^{i,j,\ell}\}_{\ell=1}^{n_{prg}}$ by selecting uniformly random elements from $(\mathbb{Z}/(p))^{n_{prg}}$.
3. The simulator also computes the values of $\text{state}_{priv}^{i,j}$ and $\text{state}_{pub}^{i,j}$, using the just generated values of $\{k_{prg}^{i,j,\ell}\}_{\ell=1}^{n_{prg}}$, and stores the states for later use.
4. $\text{cnt} \leftarrow \text{cnt} + 1$.

InitInput₂(sid, i, j):

1. Let the ciphertexts held in $\text{varid}_{k_{prg}^{i,j,\ell}}$ be denoted by $\text{ct}_{k_{prg}^{i,j,\ell}}$.
2. The simulator knows the values $\{k_{prg}^{i,j,\ell}\}_{\ell=1}^{n_{prg}}$ from above. When $\text{ct}_{k_{prg}^{i,j,\ell}}$ is passed to $\mathcal{F}_{KeyGenDec}$ the simulator forwards the query and simulates the output using the stored value $k_{prg}^{i,j,\ell}$.
3. Note, if $\mathcal{I}_i \in \mathbb{I}_A$ then the simulator will send $\mathcal{I}_i$ the values $\{k_{prg}^{i,j,\ell}\}_{\ell=1}^{n_{prg}}$.

Input(sid, $\mathcal{I}_i$, varid , x, j):

1. If $\mathcal{I}_i \in \mathbb{I} \setminus \mathbb{I}_A$, i.e. $\mathcal{I}_i$ is honest:
 - (a) If $\text{type}^2(\text{varid}) = -$ then the functionality will output x to the adversary, thus the simulator will also learn x . The simulator essentially does nothing for such operations.
 - (b) If $\text{type}^2(\text{varid}) = s$ then the simulator generates z uniformly at random and passes it to the parties in $\mathbb{C}$.
 - (c) The honest parties in $\mathbb{C}$ then process the incoming value as specified in the protocol.
2. If $\mathcal{I}_i \in \mathbb{I}_A$, i.e. $\mathcal{I}_i$ is dishonest:
 - (a) If $\text{type}^2(\text{varid}) = -$ then the adversary will send x to the simulator in the clear.
 - (b) If $\text{type}^2(\text{varid}) = s$ then the adversary will send $z = x + y$ to the simulator in the clear. The simulator can then recover y , as it knows $\text{state}_{priv}^{i,j}$ and $\text{state}_{pub}^{i,j}$. From y and z it can recover the adversary's input x .
 - (c) The value x is then passed to $\mathcal{F}_{F-Circuit}$ by calling $(\text{sid}, \text{Input}, \mathcal{I}_i, \text{varid}, x)$ on $\mathcal{F}_{F-Circuit}$.

Figure 16: The simulator for $\mathcal{F}_{F-Circuit}$ – Part I

- When a **Declassify** operation or an **Output** operation is performed, the entity $\mathcal{S}_0$ can execute the code of $\mathcal{F}_{KeyGenDec}$, as it knows the underlying secret key $\mathbf{sk}$.

Clearly $\mathcal{X}_0$ is acting as the challenger in the real world, and so we have

$$\Pr[1 \leftarrow \mathcal{Z}(\text{Real})] = \Pr[1 \leftarrow \mathcal{Z}(\mathbb{G}_0)].$$

$\mathcal{S}_{F-Circuit}$ (Part II)

Output(sid, $\mathcal{O}_i$, varid):

1. If $\mathcal{O}_i \in \mathbb{O} \setminus \mathbb{O}_A$, i.e. $\mathcal{O}_i$ is honest:
 - (a) If $\text{type}^2(\text{varid}) = -$ then the simulator knows the value x , and can thus transmit the value to $\mathcal{O}_i$ for honest parties in $\mathbb{C}$.
 - (b) If $\text{type}^2(\text{varid}) = s$ then the simulator simulates the operation by
 - i. The simulator obtains the ciphertext ct_x stored in varid, and generates a random message x ,
 - ii. The simulator queries $\mathcal{F}_{\text{KeyGenDec}}$ with ct_x , and simulates the resulting interaction to the adversary using the value x .
2. If $\mathcal{O}_i \in \mathbb{O}_A$, i.e. $\mathcal{O}_i$ is dishonest.
 - (a) If $\text{type}^2(\text{varid}) = -$ then the simulator knows the value x , and can thus simulate the transmission of x to $\mathcal{O}_i$ from the honest parties in $\mathbb{C}$.
 - (b) If $\text{type}^2(\text{varid}) = s$ then
 - i. The simulator obtains the underlying message x encrypted by the ciphertext ct_x stored in varid, by calling (sid, Output, $\mathcal{O}_i$, varid) on $\mathcal{F}_{F-Circuit}$.
 - ii. The simulator queries $\mathcal{F}_{\text{KeyGenDec}}$ with ct_x , and simulates the resulting interaction to the adversary using the value x .

Evaluate(sid, ($\{\text{varid}_{y_j}\}_{j=1}^n, \{\text{varid}_{x_i}\}_{i=1}^m, F$):

1. There is nothing to simulate here, as the procedure is purely a deterministic computation by the parties in $\mathbb{C}$ on their known inputs.

Random(sid, $\{\text{varid}_{y_j}\}_{j=1}^n, k'$):

1. Again there is nothing to simulate here, as the procedure is purely a deterministic computation by the parties in $\mathbb{C}$ on their known inputs.

Declassify(sid, varid_y, varid_x):

1. The simulator obtains the underlying message y encrypted by the ciphertext ct_x stored in varid_x, by calling (sid, Declassify, varid_y, varid_x) on $\mathcal{F}_{F-Circuit}$.
2. The simulator queries $\mathcal{F}_{\text{KeyGenDec}}$ with ct_x , and simulates the resulting interaction to the adversary using the value y .
3. The simulator stores y in varid_y.

Terminate:

1. The simulator stops, and outputs the values obtained from the functionality $\mathcal{F}_{F-Circuit}$ as to the functionalities inputs and outputs of the honest parties.

Figure 17: The simulator for $\mathcal{F}_{F-Circuit}$ – Part II

Notice, the book-keeper $\mathcal{B}_0$ is given all of the input values (either directly from $\mathcal{Z}$, or from $\mathcal{S}_0$). It is also given all of the underlying random values obtained from executing the Random command. This means that $\mathcal{B}_0$ can compute all of the values of the varid values used in the execution, since it can recover the others by evaluating the functions F sent to Evaluate in-the-clear. It is for this reason we call $\mathcal{B}_0$ a book-keeper. At all times, $\mathcal{B}_0$ (and subsequent $\mathcal{B}_i$'s) will maintain a consistent plaintext view of all registers. Notice, that at this stage the only communication from $\mathcal{B}_0$ to $\mathcal{S}_0$ is the honest parties inputs.

At the end of the game the book-keeper $\mathcal{B}_0$ outputs the honest parties outputs to the environment. It can do this, since it has a consistent view of all the values held in the registers.

Game $\mathbb{G}_1$: In game $\mathbb{G}_1$ the entity $\mathcal{S}_0$ is modified (to obtain $\mathcal{S}_1$) so that it decrypts, using the knowledge of the secret key $\mathbf{sk}$, every encrypted `varid` value. This value is sent to a modified version of the book-keeper $\mathcal{B}_1$, which aborts if the value received is not equal to the values it maintains for these `varid` values.

It is clear that if $\mathcal{Z}$ can distinguish between running in $\mathbb{G}_0$ and $\mathbb{G}_1$ then we can construct an adversary $\mathcal{A}_1$ against the Cor_{prf} security from Definition 6. Thus we have

$$\begin{aligned} & \left| \Pr[1 \leftarrow \mathcal{Z}(\mathbb{G}_0)] - \Pr[1 \leftarrow \mathcal{Z}(\mathbb{G}_1)] \right| \\ & \leq \text{Adv}_{\text{Cor}_{\text{prf}}}^{\mathcal{A}_1}(\text{sec}, \ell_{\text{Random}} + I \cdot \ell_J, \ell_{\text{Evaluate}} + I \cdot \ell_J). \end{aligned}$$

This is because, firstly, the number of calls to `DeRandFHE-PRF` is equal to the number of calls to `Random` plus the maximum number of calls to `InitInput1`, each such call being a single query to the encrypted PRF oracle which requests $t = n$ resp. $t = n_{\text{prg}}$ outputs, and secondly, the number of calls to `DeRandEval` made is equal to the number of calls to `Evaluate` plus the maximum number of calls to `InitInput1`.

Game $\mathbb{G}_2$: Assuming game $\mathbb{G}_1$ does not abort, we can here return $(\mathcal{S}_2, \mathcal{B}_2)$ to behave just as $(\mathcal{S}_1, \mathcal{B}_1)$, except $\mathcal{S}_2$ will now not, when answering the `Declassify` queries decrypt the ciphertexts itself, but will instead be given the values which the book-keeper $\mathcal{B}_2$ is storing for these values.

Thus in game $\mathbb{G}_2$ we can simulate the queries to $\mathcal{F}_{\text{KeyGenDec}}$ used in `Declassify` using the stored values from $\mathcal{B}_2$. Since these values are correct, as $\mathbb{G}_1$ did not abort we have

$$\Pr[1 \leftarrow \mathcal{Z}(\mathbb{G}_1)] = \Pr[1 \leftarrow \mathcal{Z}(\mathbb{G}_2)]$$

Game $\mathbb{G}_3$: We make a similar change in $\mathbb{G}_3$ for the $\mathcal{F}_{\text{KeyGenDec}}$ used in `InitInput2`. Now, instead of using the real $\mathcal{F}_{\text{KeyGenDec}}$ operation, which requires the secret key $\mathbf{sk}$, the entity $\mathcal{S}_3$ now simulates the queries to $\mathcal{F}_{\text{KeyGenDec}}$ using its known values $\mathbf{k}_{\text{prg}}^{i,j,\ell}$. Again as $\mathbb{G}_1$ did not abort, these values are correct, and we have

$$\Pr[1 \leftarrow \mathcal{Z}(\mathbb{G}_2)] = \Pr[1 \leftarrow \mathcal{Z}(\mathbb{G}_3)]$$

Game $\mathbb{G}_4$: At this point the only usage of $\mathbf{sk}$ by $\mathcal{S}_3$ is in executing $\mathcal{F}_{\text{KeyGenDec}}$ in the `Output` to honest and dishonest parties.

In the case of dishonest parties we proceed as above, i.e. $\mathcal{S}_4$ will obtain the value to output by requesting this from $\mathcal{B}_4$, and then simulating the query to $\mathcal{F}_{\text{KeyGenDec}}$.

In the case of output to honest parties the entity $\mathcal{S}_4$ will generate a random value x , and then uses this to simulate the query to $\mathcal{F}_{\text{KeyGenDec}}$. Note, at the end of game the environment will still be passed the correct value, by $\mathcal{B}_4$, which is output to the honest party, and not the fake value x .

$$\Pr[1 \leftarrow \mathcal{Z}(\mathbb{G}_3)] = \Pr[1 \leftarrow \mathcal{Z}(\mathbb{G}_4)]$$

Note, at this point $\mathcal{S}_4$ no longer uses the FHE secret key $\mathbf{sk}$.

Game $\mathbb{G}_5$: Now instead of $\mathcal{S}_5$ using the key $\mathbf{k}_{\text{prf}}$ to obtain the underlying PRF values and then forwarding them to $\mathcal{B}_5$. Now we work in reverse, and let $\mathcal{B}_5$ generate the internally random values as if they were from an ideal PRF. Note, at this point $\mathcal{S}_5$ does not require knowledge of the LWR-PRF secret key $\mathbf{k}_{\text{prf}}$ in order to operate.

The entity $\mathcal{S}_5$ will also be modified in order to generate the outputs of the PRF needed for `InitInput1` for all parties at random too. Thus it can recover the inputs from the dishonest parties in $\mathbb{I}_A$ as before, and pass them onto $\mathcal{B}_5$.

Again it is clear that if $\mathcal{Z}$ can distinguish between $\mathbb{G}_4$ and $\mathbb{G}_5$ then we can construct an adversary $\mathcal{D}$ against the encrypted PRF. Thus we have there is an adversary $\mathcal{D}$ such that

$$\left| \Pr[1 \leftarrow \mathcal{Z}(\mathbb{G}_4)] - \Pr[1 \leftarrow \mathcal{Z}(\mathbb{G}_5)] \right| \leq \text{Adv}_{\text{EncPRF}}^{\mathcal{D}}(\text{sec}, \ell_{\text{Random}} + I_A \cdot \ell_J),$$

since, as above, the number of calls to `DeRandFHE-PRF` which are opened is bounded by the number of calls to `Random` plus the maximum number of calls to `InitInput1` made by dishonest input parties, each such call being a single query to the $\mathbb{O}$ oracle of Figure 5. By Theorem 1 we can translate this into the existence of adversaries $\mathcal{A}_2$ and $\mathcal{A}_3$ such that

$$\begin{aligned} \left| \Pr[1 \leftarrow \mathcal{Z}(\mathbb{G}_4)] - \Pr[1 \leftarrow \mathcal{Z}(\mathbb{G}_5)] \right| & \\ & \leq \text{Adv}_{\text{IND-CPA}}^{\mathcal{A}_2}(\text{sec}, \ell_{\text{Random}} + I_A \cdot \ell_J) \\ & \quad + \text{Adv}_{\text{PRF}}^{\mathcal{A}_3}(\text{sec}, \ell_{\text{Random}} + I_A \cdot \ell_J). \end{aligned}$$

Game $\mathbb{G}_6$: Our final change is in the input protocol for honest parties $\mathcal{I} \in \mathbb{I} \setminus \mathbb{I}_A$. Now instead of $\mathcal{B}_6$ passing the honest players input to $\mathcal{S}_6$, the entity $\mathcal{B}_6$ does nothing, and $\mathcal{S}_6$ now simulates the input routine by generating $z = y + x$ at random during the execution of `Input` for the honest parties. Since the adversary learns nothing about $\text{state}_{\text{priv}}^{i,j}$ for such calls generating z , it also learns nothing about the underlying PRG outputs y . Since y is (by PRG security) indistinguishable from random, this means z is indistinguishable from random. Moreover, knowing z and ct_y (through $\mathcal{C}_A$) the adversary can compute ct_x for the random x used in the simulation. However, by the IND-CPA security of the FHE scheme, ct_x is indistinguishable from an encryption of the *real* x , and thus the adversary learns nothing about x . Put another way, if the adversary can distinguish $\mathbb{G}_6$ from $\mathbb{G}_5$ then we can create an adversary $\mathcal{A}_4$ against the PRG security or an adversary $\mathcal{A}_5$ against the IND-CPA security of the FHE scheme.

$$\begin{aligned} \left| \Pr[1 \leftarrow \mathcal{Z}(\mathbb{G}_5)] - \Pr[1 \leftarrow \mathcal{Z}(\mathbb{G}_6)] \right| & \\ & \leq (I - I_A) \cdot \ell_J \cdot \text{Adv}_{\text{PRG}}^{\mathcal{A}_4}(\text{sec}, \ell_{\text{Input}}, 1) \\ & \quad + \text{Adv}_{\text{IND-CPA}}^{\mathcal{A}_5}(\text{sec}, (I - I_A) \cdot \ell_J \cdot \ell_{\text{Input}}). \end{aligned}$$

This is because we have at most $(I - I_A) \cdot \ell_J$ distinct PRG keys held by honest parties. For each PRG key we make at most ℓ_{Input} calls to the `Input` operation, each of which produces a PRG output of length one.

Finally, we see that $\mathcal{S}_6$ is exactly the simulator described in Figure 16–Figure 17, with the book-keeper $\mathcal{B}_6$ being exactly the ideal functionality. Thus we have

$$\Pr[1 \leftarrow \mathcal{Z}(\text{Ideal})] = \Pr[1 \leftarrow \mathcal{Z}(\mathbb{G}_6)].$$

From this the result follows. $\square$

7.1 Proof Modification After Removing the PRG

Below we outline the changes to the proof and Theorem 3 when the PRG is removed. Let Q_H be the number of `Input` operations on secret data performed by honest parties, Q_A for the number of mask requests made by corrupt input parties, $Q = Q_H + Q_A$, and taking $\text{nonce} \leftarrow \{0, 1\}^{2 \cdot \text{sec}}$, the bound becomes

$$\begin{aligned} \text{Adv}^{\mathcal{Z}} & \leq \text{Adv}_{\text{Cor}_{\text{prf}}}^{\mathcal{A}_1}(\text{sec}, \ell_{\text{Random}} + Q, \ell_{\text{Evaluate}} + Q) \\ & \quad + \text{Adv}_{\text{IND-CPA}}^{\mathcal{A}_2}(\text{sec}, \ell_{\text{Random}} + Q) \\ & \quad + \text{Adv}_{\text{PRF}}^{\mathcal{A}_3}(\text{sec}, \ell_{\text{Random}} + Q) \end{aligned}$$

$$+ \text{Adv}_{\text{IND-CPA}}^{A_4}(\text{sec}, Q_H) + \binom{Q_H}{2} \cdot 2^{-2 \cdot \text{sec}}.$$

Comparing to Theorem 3, the parameters n_{prg} , ℓ_J and ℓ_{Input} no longer appear, the PRG term disappears, and a statistical term appears which bounds the probability that two **Input** operations of an honest party draw the same **nonce**.

The simulator of Figure 16 changes only in its input routines, which are merged into a single routine following the protocol. Instead of generating a PRG key $\{k_{\text{prg}}^{i,j,\ell}\}_{\ell=1}^{n_{\text{prg}}}$ and computing the associated states $\text{state}_{\text{pub}}^{i,j}$ and $\text{state}_{\text{priv}}^{i,j}$, the simulator samples a single mask $y \leftarrow \mathbb{Z}/(p^{k'})$ per request and stores it against (i, j, nonce) . When $\mathcal{I}_i \in \mathbb{I}_A$ the query to $\mathcal{F}_{\text{KeyGenDec.DistDecrypt}}$ is simulated using this y , and on receiving the broadcast value z the simulator recovers the input as $x \leftarrow z - y$ and passes it to $\mathcal{F}_{F\text{-Circuit}}$. For an honest $\mathcal{I}_i$ the simulator broadcasts a uniformly random z , as before.

Games $\mathbb{G}_0$ to $\mathbb{G}_5$ are essentially unchanged except for the parameters. Between $\mathbb{G}_5$ and $\mathbb{G}_6$ we insert a game which halts if two **Input** operations of an honest party use the same (i, j, nonce) , this accounts for the possibility of collision. Finally the PRG game $\mathbb{G}_6$ is simplified and only the IND-CPA step for ct_x remains.

Modification to the proof is trivial if the additive pad is replaced by an one-time-secure symmetric encryption scheme **Sym** mentioned in Section 6.1. The proof therefore gains a term for **Sym** and an additional game hop.

Acknowledgements

The author would like to thank Amit Deo and Benoit Libert for discussions whilst this work was carried out.

References

- [ABMP26] Andreea Alexandru, Ahmad Al Badawi, Daniele Micciancio, and Yuriy Polyakov. Application-aware approximate homomorphic encryption: Configuring FHE for practical use. *IACR Communications in Cryptology (CiC)*, 2(4):3, 2026. doi:10.62056/ayl83z10k.
- [BCD⁺25] Carl Bootland, Kelong Cong, Daniel Demmler, Tore Kasper Frederiksen, Benoit Libert, Jean-Baptiste Orfila, Dragos Rotaru, Nigel P. Smart, Titouan Tanguy, Samuel Tap, and Michael Walter. Threshold (fully) homomorphic encryption. Cryptology ePrint Archive, Report 2025/699, 2025. URL: <https://eprint.iacr.org/2025/699>.
- [BdBB⁺25] Mathieu Ballandras, Mayeul de Bellabre, Loris Bergerat, Charlotte Bonte, Carl Bootland, Benjamin R. Curtis, Jad Khatib, Jakub Klemsa, Arthur Meyre, Thomas Montaigu, Jean-Baptiste Orfila, Nicolas Sarlin, Samuel Tap, and David Testé. TFHE-rs: A (Practical) Handbook, 2025. URL: <https://github.com/zama-ai/tfhe-rs-handbook/blob/main/tfhe-rs-handbook.pdf>.
- [BDO⁺21] Karim Baghery, Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, Nigel P. Smart, and Titouan Tanguy. Compilation of function representations for secure computing paradigms. In Kenneth G. Paterson, editor, *Topics in Cryptology – CT-RSA 2021*, volume 12704 of *Lecture Notes in Computer Science*, pages 26–50, Virtual Event, May 17–20, 2021. Springer, Cham, Switzerland. doi:10.1007/978-3-030-75539-3_2.

- [BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012: 3rd Innovations in Theoretical Computer Science*, pages 309–325, Cambridge, MA, USA, January 8–10, 2012. Association for Computing Machinery. doi:[10.1145/2090236.2090262](https://doi.org/10.1145/2090236.2090262).
- [BHY09] Mihir Bellare, Dennis Hofheinz, and Scott Yilek. Possibility and impossibility results for encryption and commitment secure under selective opening. In Antoine Joux, editor, *Advances in Cryptology – EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 1–35, Cologne, Germany, April 26–30, 2009. Springer Berlin Heidelberg, Germany. doi:[10.1007/978-3-642-01001-9_1](https://doi.org/10.1007/978-3-642-01001-9_1).
- [BJSW25] Olivier Bernard, Marc Joye, Nigel P. Smart, and Michael Walter. Drifting towards better error probabilities in fully homomorphic encryption schemes. In Serge Fehr and Pierre-Alain Fouque, editors, *Advances in Cryptology – EUROCRYPT 2025, Part VIII*, volume 15608 of *Lecture Notes in Computer Science*, pages 181–211, Madrid, Spain, May 4–8, 2025. Springer, Cham, Switzerland. doi:[10.1007/978-3-031-91101-9_7](https://doi.org/10.1007/978-3-031-91101-9_7).
- [BK11] Dan Bogdanov and Liina Kamm. Constructing privacy-preserving information systems using secure multiparty computation. Cybernetica Research Report T-4-13, 2011. Accessed: 2026-08-25.
- [BKL⁺14] Dan Bogdanov, Liina Kamm, Sven Laur, Pille Pruulmann-Vengerfeldt, Riivo Talviste, and Jan Willemson. Privacy-preserving statistical data analysis on federated databases. In Bart Preneel and Demosthenes Ikononou, editors, *Proceedings of the Annual Privacy Forum. APF’14*, volume 8450 of *LNCS*, pages 30–55. Springer, 2014. doi:[10.1007/978-3-319-06749-0_3](https://doi.org/10.1007/978-3-319-06749-0_3).
- [BLMR13] Dan Boneh, Kevin Lewi, Hart William Montgomery, and Ananth Raghunathan. Key homomorphic PRFs and their applications. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 410–428, Santa Barbara, CA, USA, August 18–22, 2013. Springer Berlin Heidelberg, Germany. doi:[10.1007/978-3-642-40041-4_23](https://doi.org/10.1007/978-3-642-40041-4_23).
- [BNP08] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM CCS 2008: 15th Conference on Computer and Communications Security*, pages 257–266, Alexandria, Virginia, USA, October 27–31, 2008. ACM Press. doi:[10.1145/1455770.1455804](https://doi.org/10.1145/1455770.1455804).
- [BPR12] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 719–737, Cambridge, UK, April 15–19, 2012. Springer Berlin Heidelberg, Germany. doi:[10.1007/978-3-642-29011-4_42](https://doi.org/10.1007/978-3-642-29011-4_42).
- [Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886, Santa Barbara, CA, USA, August 19–23, 2012. Springer Berlin Heidelberg, Germany. doi:[10.1007/978-3-642-32009-5_50](https://doi.org/10.1007/978-3-642-32009-5_50).

- [Can00] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. URL: <https://eprint.iacr.org/2000/067>.
- [Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press. doi:10.1109/SFCS.2001.959888.
- [CCF⁺16] Anne Canteaut, Sergiu Carpov, Caroline Fontaine, Tancrede Lepoint, María Naya-Plasencia, Pascal Paillier, and Renaud Sirdey. Stream ciphers: A practical solution for efficient homomorphic-ciphertext compression. In Thomas Peyrin, editor, *Fast Software Encryption – FSE 2016*, volume 9783 of *Lecture Notes in Computer Science*, pages 313–333, Bochum, Germany, March 20–23, 2016. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-662-52993-5_16.
- [CCF⁺18] Anne Canteaut, Sergiu Carpov, Caroline Fontaine, Tancrede Lepoint, María Naya-Plasencia, Pascal Paillier, and Renaud Sirdey. Stream ciphers: A practical solution for efficient homomorphic-ciphertext compression. *Journal of Cryptology*, 31(3):885–916, July 2018. doi:10.1007/s00145-017-9273-9.
- [CCP⁺24] Jung Hee Cheon, Hyeongmin Choe, Alain Passelègue, Damien Stehlé, and Elias Suvanto. Attacks against the IND-CPA^D security of exact FHE schemes. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *ACM CCS 2024: 31st Conference on Computer and Communications Security*, pages 2505–2519, Salt Lake City, UT, USA, October 14–18, 2024. ACM Press. doi:10.1145/3658644.3690341.
- [CGGI16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 3–33, Hanoi, Vietnam, December 4–8, 2016. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-662-53887-6_1.
- [CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, January 2020. doi:10.1007/s00145-019-09319-x.
- [CLOT21] Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021, Part III*, volume 13092 of *Lecture Notes in Computer Science*, pages 670–699, Singapore, December 6–10, 2021. Springer, Cham, Switzerland. doi:10.1007/978-3-030-92078-4_23.
- [CPS18] Eric Crockett, Chris Peikert, and Chad Sharp. ALCHEMY: A language and compiler for homomorphic encryption made easY. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 1020–1037, Toronto, ON, Canada, October 15–19, 2018. ACM Press. doi:10.1145/3243734.3243828.
- [CSBB24] Marina Checri, Renaud Sirdey, Aymen Boudguiga, and Jean-Paul Bultel. On the practical CPA^D security of “exact” and threshold FHE schemes

- and libraries. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology – CRYPTO 2024, Part III*, volume 14922 of *Lecture Notes in Computer Science*, pages 3–33, Santa Barbara, CA, USA, August 18–22, 2024. Springer, Cham, Switzerland. doi:[10.1007/978-3-031-68382-4_1](https://doi.org/10.1007/978-3-031-68382-4_1).
- [DDE⁺23] Morten Dahl, Daniel Demmler, Sarah El Kazdadi, Arthur Meyre, Jean-Baptiste Orfila, Dragos Rotaru, Nigel P. Smart, Samuel Tap, and Michael Walter. Noah’s ark: Efficient threshold-FHE using noise flooding. In Michael Brenner, Anamaria Costache, and Kurt Rohloff, editors, *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Copenhagen, Denmark, 26 November 2023*, pages 35–46. ACM, 2023. doi:[10.1145/3605759.3625259](https://doi.org/10.1145/3605759.3625259).
- [DI05] Ivan Damgård and Yuval Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 378–394, Santa Barbara, CA, USA, August 14–18, 2005. Springer Berlin Heidelberg, Germany. doi:[10.1007/11535218_23](https://doi.org/10.1007/11535218_23).
- [DJL⁺25] Amit Deo, Marc Joye, Benoît Libert, Benjamin R. Curtis, and Mayeul de Bellabre. Fast homomorphic evaluation of LWR-based PRFs. In Chun-Ying Huang, Jyh-Cheng Chen, Shiuh-Pyng Shieh, David Lie, and Véronique Cortier, editors, *ACM CCS 2025: 32nd Conference on Computer and Communications Security*, pages 2951–2965, Taipei, Taiwan, October 13–17, 2025. ACM Press. doi:[10.1145/3719027.3765177](https://doi.org/10.1145/3719027.3765177).
- [DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662, Santa Barbara, CA, USA, August 19–23, 2012. Springer Berlin Heidelberg, Germany. doi:[10.1007/978-3-642-32009-5_38](https://doi.org/10.1007/978-3-642-32009-5_38).
- [FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. URL: <https://eprint.iacr.org/2012/144>.
- [Gen09] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.
- [GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 850–867, Santa Barbara, CA, USA, August 19–23, 2012. Springer Berlin Heidelberg, Germany. doi:[10.1007/978-3-642-32009-5_49](https://doi.org/10.1007/978-3-642-32009-5_49).
- [KN08] Gillat Kol and Moni Naor. Cryptography and game theory: Designing protocols for exchanging information. In Ran Canetti, editor, *TCC 2008: 5th Theory of Cryptography Conference*, volume 4948 of *Lecture Notes in Computer Science*, pages 320–339, San Francisco, CA, USA, March 19–21, 2008. Springer Berlin Heidelberg, Germany. doi:[10.1007/978-3-540-78524-8_18](https://doi.org/10.1007/978-3-540-78524-8_18).
- [Lib24] Benoît Libert. Vector commitments with proofs of smallness: Short range proofs and more. In Qiang Tang and Vanessa Teague, editors, *PKC 2024: 27th International Conference on Theory and Practice of Public Key Cryptography*,

- Part II*, volume 14602 of *Lecture Notes in Computer Science*, pages 36–67, Sydney, NSW, Australia, April 15–17, 2024. Springer, Cham, Switzerland. doi:10.1007/978-3-031-57722-2_2.
- [LM21] Baiyu Li and Daniele Micciancio. On the security of homomorphic encryption on approximate numbers. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 648–677, Zagreb, Croatia, October 17–21, 2021. Springer, Cham, Switzerland. doi:10.1007/978-3-030-77870-5_23.
- [PVW08] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In David Wagner, editor, *Advances in Cryptology – CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 554–571, Santa Barbara, CA, USA, August 17–21, 2008. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-540-85174-5_31.
- [Sma23a] Nigel P. Smart. Practical and efficient FHE-based MPC. In Elizabeth A. Quaglia, editor, *19th IMA International Conference on Cryptography and Coding*, volume 14421 of *Lecture Notes in Computer Science*, pages 263–283, London, UK, December 12–14, 2023. Springer, Cham, Switzerland. doi:10.1007/978-3-031-47818-5_14.
- [Sma23b] Nigel P. Smart. Practical and efficient FHE-based MPC. Cryptology ePrint Archive, Report 2023/981, 2023. URL: <https://eprint.iacr.org/2023/981>.
- [SW26] Nigel P. Smart and Michael Walter. Reactive correctness, sIND-CPAD-security and deterministic evaluation for TFHE. *IACR Communications in Cryptology (CiC)*, 3(1), 2026. doi:10.62056/aksdkmol.
- [TCBS23a] Daphné Trama, Pierre-Emmanuel Clet, Aymen Boudguiga, and Renaud Sirdey. A homomorphic aes evaluation in less than 30 seconds by means of tfhe. In Michael Brenner, Anamaria Costache, and Kurt Rohloff, editors, *11th Workshop on Encrypted Computing and Applied Homomorphic Cryptography (WAHC 2023)*, pages 79–90. ACM, 2023. doi:10.1145/3605759.3625260.
- [TCBS23b] Daphné Trama, Pierre-Emmanuel Clet, Aymen Boudguiga, and Renaud Sirdey. Trivial transciphering with trivium and TFHE. In Michael Brenner, Anamaria Costache, and Kurt Rohloff, editors, *11th Workshop on Encrypted Computing and Applied Homomorphic Cryptography (WAHC 2023)*, pages 69–78. ACM, 2023. doi:10.1145/3605759.3625255.
- [WWL⁺23] Benqiang Wei, Ruida Wang, Zhihao Li, Qinju Liu, and Xianhui Lu. Fregata: Faster homomorphic evaluation of AES via TFHE. In Elias Athanasopoulos and Bart Mennink, editors, *ISC 2023: 26th International Conference on Information Security*, volume 14411 of *Lecture Notes in Computer Science*, pages 392–412, Groningen, The Netherlands, November 15–17, 2023. Springer, Cham, Switzerland. doi:10.1007/978-3-031-49187-0_20.