Towards Verifiable FHE in Practice

Proving Correct Execution of TFHE's Bootstrapping using plonky2

Louis Tremblay Thibault
Zama
`louis.tremblay.thibault@zama.ai`

Michael Walter
Zama
`michael.walter@zama.ai`

Abstract

In this work we demonstrate for the first time that a full FHE bootstrapping operation can be proven using a SNARK in practice. We do so by designing an arithmetic circuit for the bootstrapping operation and prove it using plonky2. We are able to prove the circuit on an AWS `Hpc7a` instance in under 20 minutes. Proof size is about 200 kB and verification takes less than 10 ms. As the basis of our bootstrapping operation we use TFHE's programmable bootstrapping and modify it in a few places to more efficiently represent it as an arithmetic circuit (while maintaining full functionality and security). In order to achieve our results in a memory-efficient way, we take advantage of the structure of the computation and plonky2's ability to efficiently prove its own verification circuit to implement a recursion-based IVC scheme. Lastly, we present a security proof in the UC model that captures active attacks in real world applications of verifiable FHE and augment our prototype to fit such applications.

1 Introduction

There are two emerging cryptographic technologies with a host of applications in practice: Fully Homomorphic Encryption (FHE) and Succinct Non-interactive Arguments of Knowledge (SNARKs). FHE allows arbitrary computation on encrypted data, while SNARKs enable proving the correct execution of arbitrary computation with short proofs and verification time sublinear in the size of the computation. It is not hard to see the vast number of possible use cases for each of these technologies in practice, but in this work we are interested in the combination of the two. Specifically, we investigate to what extent it is possible to prove the correct execution of FHE operations using SNARKs in practice.

Combining FHE with SNARKs is enormously appealing as this has the potential to entirely replace solutions to secure outsourcing of computing based on hardware modules, which are riddled with practical attacks and ultimately only achieve a shift of trust to the hardware vendor. In contrast, a verifiable FHE scheme would allow outsourcing computation and reduce trust to cryptographic, i.e., mathematical assumptions using minimal interaction. Furthermore, such a scheme would thwart CCA-style attacks, to which FHE schemes are known to be inherently vulnerable [CGG16], which means that in a practical deployment, one needs to be very prudent in its use of FHE in order not to fall victim to attacks outside of the security model.

Unfortunately, despite a large amount of research and significant progress over the past one and a half decades, FHE operations still incur a significant overhead over their cleartext counterparts. Even worse, any truly *fully* homomorphic scheme we know of to date relies on a bootstrapping operation to reduce noise in ciphertexts, which accumulates during homomorphic operations and may lead to incorrect decryption if not handled correctly. In all current FHE schemes, this bootstrapping is the costliest operation.

On the other hand, SNARKs themselves incur a significant overhead over the computation to prove, with many practical SNARKs having proving complexity superlinear in the size of the computation.[1] Furthermore,

[1] With *size* we mean here the size of the circuit used to perform the computation in the arithmetic circuit model.

since the proof generation typically requires to keep the entire trace in memory, the memory requirement of SNARKs grows at least linearly in the computation length, which renders the memory complexity the bottleneck for long computations. There are techniques to mitigate this issue for *structured* computations. We will come back to this later.

So it is not surprising that the bootstrapping operation represents a formidable challenge for SNARKs. While some works have considered proving *levelled* homomorphic operations [VKH23, GNS23], as far as we are aware, there are no published attempts of applying a SNARK to an FHE bootstrapping operation, let alone successful attempts. In this work we seek to remedy this state of affairs and demonstrate the practicality of a fully verifiable bootstrapping.

Finally, we note that the security of FHE schemes is usually only considered under passive attacks (IND-CPA), while classic PKE schemes are typically expected to be secure also under active attacks (IND-CCA). Unfortunately, capturing active attacks using classic game-based indistinguishability notions is notoriously hard for FHE due to the inherent malleability of ciphertexts. This leads to weak or arguably over-complex definitions [AGHV22, VKH23, MN24, CFP$^+$24, Wal24]. We address this issue by modeling a typical use case of verifiable FHE as an MPC application and prove its security in the UC model.

1.1 Contribution

In this work we use an argument system for general purpose computation, plonky2 [Pol22], in order to allow the evaluator of a TFHE bootstrapping [CGGI20] to prove that it did so correctly. Note that the proof is publicly verifiable, not just by the party holding the decryption key (or some other kind of secret verification key). In order to use plonky2, we need to rewrite the bootstrapping algorithm in terms of an arithmetic circuit over a finite field. In this work we present a number of tweaks to TFHE to reduce its circuit size. Still, the main challenge is the sheer size of the bootstrapping circuit, which is too large to be handled in practice. To address this, we show how to exploit incrementally verified computation (IVC) [Val08] to take advantage of its inherent structure. We provide an implementation[2] and an experimental evaluation.

We compare our experimental results to using general-purpose zero-knowledge virtual machines (zkVMs) to prove correct execution of the PBS [BGZ23, Suc24]. Such zkVMs promise to be easy to use at the cost of introducing an overhead. To evaluate the extent of the overhead compared to our specially crafted circuit, we use a straight-forward implementation of the PBS and apply the zkVMs using a variety of test machines. The first observation is that neither of the two zkVMs we tested were actually able to prove an entire PBS due to performance limitations even on powerful machines (and in one case even a computer cluster), while this was no problem at all for our plonky2-based implementation even on moderate machines. It is plausible that tweaking the implementation on the zkVMs could solve the issue, but this was out of scope of this project. To still obtain a quantitative comparison, we performed micro-benchmarks. The results indicate that our implementation outperforms the zkVMs by at least two orders of magnitude.[3]

To the best of our knowledge, our results demonstrate for the first time that generating a proof for a bootstrapping operation is practically feasible: we are able to prove correctness of a TFHE-like bootstrapping with secure parameters in under 20 minutes on an AWS `Hpc7a` instance. While this is still likely to be too costly for many applications, others might already be able to take advantage of a fully verifiable FHE scheme. As an example, consider a blockchain protocol that allows smart contracts on encrypted data [DDD$^+$23, Tea23]. Here, verifiable FHE operations have the potential to replace certain consensus protocols and thus reduce the computational load of validators. The given proving time could be acceptable in this setting, if this is deployed akin to hybrid rollups, where a proof is only required in case of a dispute.

Furthermore, we argue for a security model that formalizes the settings in which our verifiable FHE construction can be used securely. In more detail, we note that adding a proof of correct evaluation to the ciphertext allows to strengthen the security model to some forms of active attacks, but it is not obvious what the precise security model is. A sound definition of a security notion achieved by this construction was

[2] https://github.com/zama-ai/verifiable-fhe-paper

[3] We remark that some zkVMs like [Suc24] have some advanced features, like precomiled circuits, that we did not explore in this project. We believe it is an interesting open question if more advanced usage of the zkVMs could yield results comparable to ours in a simpler way.

recently provided [Wal24], but it suffers from similar issues as previous work in being overly complex. We observe that the key issue in game-based security definitions for FHE meant to capture active security is that FHE ciphertexts are inherently malleable. The difficulty is to cleanly separate "benign" modifications of the ciphertext from malicious ones. To address this issue, we diverge from the game-based definitional framework and turn to simulation-based definitions. Specifically, inspired by [Sma23], we model typical secure outsourcing of computation as an MPC application and give a proof in the UC model. The advantage of the simulation-based approach is that we do not need to explicitely define malicious and benign modifications of the ciphertext, since the definition simply states that the adversary learns everything the ideal functionality leaks about the plaintexts, but nothing beyond that.

Lastly, by slightly augmenting our implementation, we show that we are able to prove the correct evaluation of simple functions over ciphertexts (e.g., a weighted sum followed by a unitary function) at a very low cost, potentially unlocking applications in the field of privacy-preserving machine learning (ML) that fit within our MPC model.

1.2 Choice of FHE scheme and SNARK

FHE Scheme Since our goal is fairly ambitious, we try to make our lives as easy as possible. In particular, we choose as our target the FHE scheme with the lightest known bootstrapping operation, namely TFHE [CGGI20, CLOT21, BBB$^+$22]. We take the liberty to modify TFHE at a few places to make it more amendable to our target SNARK. These modifications maintain the functionality of the bootstrap, but might make it slightly less efficient. If the modifications yield a faster proof generation, this is likely a worthwhile trade-off depending on the overall system. The most significant modification we apply is to use a SNARK-friendly prime modulus $q \approx 2^{64}$ instead of a power of 2, because most efficient SNARKs only natively support arithmetic circuits over finite fields. This way we avoid emulating the arithmetic in the ring $\mathbb{Z}_{2^{64}}$ within the SNARK field. While there are attempts to construct SNARKs for ring arithmetic [GNS23], this comes with its own caveats, like designated verifier and relatively poor performance.

SNARK There are a number of SNARK implementations for general purpose computation available and we selected the SNARK for our work based on the following criteria. In order to enable as many applications as possible, we target a transparent, publicly verifiable SNARK with sublinear verifier. For efficiency reasons, we require native support for arithmetic in fields of size $\approx 2^{64}$ and, ideally, support for efficiency improvements for structured computation like loops, since the core of TFHE's bootstrapping is essentially a large loop.

With these criteria in mind, plonky2 provides a suitable candidate. It relies on the PLONK arithmetization [GWC19] in combination with a polynomial commitment scheme based on hash functions, namely FRI [BBHR18]. It uses as a base field $\mathbb{F}_p$ with $p = 2^{64} - 2^{32} + 1$, which meets our requirement on the modulus, and, as an added bonus, is plausibly post-quantum secure, which is also true of TFHE. plonky2 is optimized for recursion, which allows us to construct *incrementally verifiable computation* (IVC) [Val08], a technique to prove the correct execution of loops more efficiently than simply rolling them out in a circuit, which will come in handy.

Looking ahead to the security proof in Section 6, we note that we do not actually need knowledge soundness, but require only regular soundness. In other words, a SNARG is sufficient for the applications we consider. However, plonky2 does provide knowledge soundness and we will continue using the term "SNARK", except in the security proof.

1.3 Related Work

There is a line of research considering verifiable computation on encrypted data. The first results in this area [GGP10, GKP$^+$13] are mainly of theoretical interest as they rely on heavy machinery like combining garbled circuits with FHE or functional encryption. The study of systems combining mechanisms for verifiable computation with FHE, as we do in this work, was initiated in [FGP14] and continued in [FNP20, GNS23, BCFK21, VKH23]. While these works promise better concrete efficiency than the aforementioned schemes, they still seem to be impractical and sidestep the complexity of bootstrapping by restricting to levelled HE schemes.

A few works have started investigating an approach based on performing the integrity check in the plaintext space [GGW23, ACGS23, CKPH22, CKP$^+$23], with [ACGS23] and [CKPH22, CKP$^+$23] claiming practical efficiency. However, this approach has the significant drawback that it requires decryption in order to verify the computation. This has two highly undesirable consequences: First, only the party with the secret key can verify the computation. While this might be acceptable in some applications, it does rule out many others. And second, it leaves the FHE scheme vulnerable to active attacks. As we will show in Section 6, performing integrity checks on the ciphertext allows to strengthen the security model significantly. Unfortunately, this is not the case if the client needs to decrypt the ciphertext before being able to verify. It can be verified that our proof of security does not hold for such constructions and indeed, it is not too hard to see that they are vulnerable to the attacks presented in [CGG16].

Finally, recent work [MN24, CFP$^+$24] has successfully managed to meaningfully define security of FHE under active attacks using game-based indistinguishability notions. However, these definitions do not cover the construction in this work and the corresponding constructions are inefficient in practice, since they require to include validity checks of ciphertexts into the proof of correct evaluation and make very strong assumptions about the employed SNARK. For example, these constructions cannot be instantiated using plonky2, since they require blackbox and straight-line extractability.

Future Work As hinted at above, our work makes use of recursion-based IVC. There is a recent line of work constructing more efficient IVC from folding schemes [BGH19, KST22, BC23]. We chose to focus on recursion, because folding schemes require the commitment scheme of the SNARK to be homomorphic. However, until very recently, there was no homomorphic scheme suitable for our application, due to large field size and/or trusted setup and/or inefficient verifier. This changed very recently with [BC24] and an exciting open question is if more efficient provers can be obtained using this new lattice-based folding.

Outline We begin with some background on LWE and TFHE in Section 2, which should be sufficient to follow the rest of the paper except for the security proof of Section 6. For the sake of readability, we introduce the required definitions for the security proof separately in Section 6. Section 3 describes our implementation of the core functionality of the bootstrapping (the blind rotation) and in Section 4 we show how to extend it to the full bootstrapping. In Section 5 we present experimental results and discuss applications in Section 7.

2 Preliminaries

Notation Throughout we will use the parameters $N, q \in \mathbb{Z}$, where N is a power of 2. If N is clear from context, we let $R = \mathbb{Z}[X]/(X^N + 1)$ and $R_q = R/qR$. Note that $R = \mathbb{Z}$ and $R_q = \mathbb{Z}_q$ when $N = 1$. Elements in R_q (for any N) are denoted by lower case letters, vectors over R_q by bold lower case letters and for a vector $\mathbf{a} \in R_q^k$ we denote by a_i its i-th component. Similarly, if $a \in R_q$ we refer to $a_i \in \mathbb{Z}_q$ as its i-th coefficient, i.e. we have $a = \sum_i a_i X^i$. For an element $a \in R$ we consider its norm $|a|$ to be the ∞-norm of its coefficient vector and we extend the norm to elements of R_q by lifting them to R, picking the representative with coefficients between $-q/2$ and $q/2$.

2.1 (G)LWE

Definition 1. *Let $N, q, k \in \mathbb{Z}$ with N a power of 2. Let $R = \mathbb{Z}[X]/(X^N + 1)$ and $R_q = R/qR$. Finally, let $\mathcal{X}$ be a "small" distribution over R_q. Then, for a fixed $\mathbf{s} \in R_q^k$ the* GLWE distribution *$GLWE^{\mathbf{s}}_{N,q,k,\mathcal{X}}$ is defined as $(\mathbf{a}, b = \langle \mathbf{a}, \mathbf{s} \rangle + e)$ where a is chosen uniformly at random from R_q^k and e is chosen from $\mathcal{X}$.*

Let $\mathcal{S}$ be some distribution over R_q^k. The GLWE problem *$GLWE^{\mathcal{S}}_{N,q,k,\mathcal{X}}$ is to distinguish the distribution $GLWE^{\mathbf{s}}_{N,q,k,\mathcal{X}}$ from the uniform distribution over R_q^{k+1}, where $\mathbf{s} \leftarrow \mathcal{S}$.*

The $\mathrm{LWE}^{\mathcal{S}}_{q,k,\mathcal{X}}$ problem is a special case of the $\mathrm{GLWE}^{\mathcal{S}}_{N,q,k,\mathcal{X}}$ where $N = 1$. TFHE assumes a secret distribution $\mathcal{S}$ that is uniform over elements in R_q^k with binary coefficients and thus we will assume this secret distribution throughout. With suitable choice for the error distribution $\mathcal{X}$ (e.g. discrete or rounded Gaussian with sufficiently large variance) and ring dimension k the corresponding $\mathrm{GLWE}^{\mathcal{S}}_{N,q,k,\mathcal{X}}$ problem is considered to be hard. It is standard practice to estimate the concrete security of a specific LWE instance using the lattice estimator [APS15].

2.2 TFHE

TFHE is a secret key FHE scheme based on (G)LWE. In the following we try to give a succinct intuitive description of TFHE that we hope is detailed enough to follow the rest of the work without cluttering it with too much formal notation. For a more rigorous description, we refer to [CGGI20] and follow up work, or the survey [Joy22].

The basic ciphertexts in TFHE are simple LWE ciphertexts, but internally it uses a range of other ciphertexts based on GLWE. Since we need to represent the entire bootstrapping as a circuit, we make use of all types of ciphertexts and thus we introduce them next.

2.2.1 Ciphertext Types

(G)LWE ciphertext Let $N, q, k \in \mathbb{Z}$, $\mathcal{X}$ be GLWE (or simply LWE in case $N = 1$) parameters. For a message $m \in R_p$, we define its (G)LWE encryption to be $(\mathbf{a}, b = \langle \mathbf{a}, \mathbf{s} \rangle + e + m)$, where $\mathbf{a} \in R_q^k$ is uniformly random, $\mathbf{s} \in R_q^k$ is chosen from the uniform binary distribution and e from $\mathcal{X}$. By the hardness of (G)LWE this is a semantically secure ciphertext. It can be decrypted using $\mathbf{s}$ if m represents a suitable encoding of a message that is robust w.r.t. the error distribution. For example, let $p < q \in \mathbb{Z}$ be a plaintext modulus and define $\Delta = \lfloor q/p \rceil$. For a message $m \in R_p$, we may define its encoding as $\Delta \cdot m$, which allows to recover $m \in \mathbb{Z}_p$ by rounding. In the context of LWE ciphertexts we typically denote the dimension by n instead of k. Note that (G)LWE ciphertexts are additively homomorphic and may be multiplied with “small” elements in R_q, where smallness is determined such that the resulting ciphertext can still be correctly decrypted given the error distribution and the encoding.

GLev Ciphertext GLev ciphertexts (where the “Lev” stands for *levelled*) are a way to extend (G)LWE ciphertexts in order to allow for multiplication with arbitrary constants. It is based on the standard approach of decomposition: for an element $a \in R_q$ and parameters B and ℓ, denote by $\mathrm{Dec}_{B,\ell}(a) \mapsto \mathbf{a}$ the transformation such that $\mathbf{a} \in R_q^\ell$, $|a_i| \leq B/2$ and $\sum_{i=1}^{\ell} \left\lfloor \frac{q}{B^i} \right\rceil a_i \approx a$. With this decomposition at hand, we define the GLev encryption of $m \in R_q$ with parameters B and ℓ to be the set of (G)LWE encryptions of $(\left\lfloor \frac{q}{B^i} \right\rceil) \cdot m$ for all $i \in \{1 \ldots \ell\}$. Note that such a ciphertext can be multiplied with an arbitrary element $a \in R_q$ by first decomposing a using parameters B and ℓ and taking the inner product with the GLev ciphertext. Since all components of $\mathrm{Dec}_{B,\ell}(a)$ are small the result is an (G)LWE encryption of $a \cdot m$ by the homomorphic properties of the (G)LWE ciphertexts (and assuming suitable parameters).

GGSW Ciphertext While GLev ciphertexts allow to multiply encrypted values with arbitrary constants, we would also like to be able to efficiently multiply encrypted values with each other. This can be achieved using GGSW ciphertexts (named after [GSW13]). The idea is to encrypt m as a GLev ciphertext and for each element s_i of the secret key $\mathbf{s} \in R_q^k$, additionally encrypt $m \cdot s_i$ as a GLev ciphertext. This set of $k+1$ GLev ciphertexts forms the GGSW ciphertext. By the properties of GLev ciphertexts, this allows to perform the multiplication while homomorphically decrypting a ciphertext $(\mathbf{a}, b = \langle \mathbf{a}, \mathbf{s} \rangle + m' + e)$ by homomorphically computing $b \cdot m$ and $a_i \cdot s_i \cdot m$ and using the additive homomorphism of GLWE ciphertexts. Note that m should not be too large as this would blow up the error. In TFHE, the message m is usually a key bit and thus binary, so clearly small. In summary, a GGSW ciphertext allows us to multiply a GLWE ciphertext with a GLev ciphertext and to obtain a GLWE ciphertext encrypting the product of the two plaintexts (as long as the plaintext in the GGSW ciphertext is sufficiently small). This operation is typically called the *external product*.

2.2.2 Programmable Bootstrapping

The PBS of TFHE receives as input

- the LWE ciphertext $(\mathbf{a}, b = \langle \mathbf{a}, \mathbf{s} \rangle + e + \Delta \cdot m) \in \mathbb{Z}_q^n$ to bootstrap, where the corresponding secret key is $\mathbf{s} \in \{0,1\}^n$,
- an element $t \in R_q$ that allows to encode a function[4] $f : \mathbb{Z}_p \mapsto \mathbb{Z}_p$ into the bootstrap,
- the bootstrapping key (bsk) as a collection of GGSW ciphertexts encrypting the individual bits $s_i \in \{0,1\}$ of the secret key under a bootstrapping secret key $\mathbf{s}' \in R_q^k$ with binary coefficients, and
- a key switching key (ksk) as a collection of GLev ciphertexts encrypting the coefficients of the bootstrapping key under the secret key $\mathbf{s}$.

It outputs a ciphertext $(\mathbf{a}', \mathbf{b}' = \langle \mathbf{a}', \mathbf{s} \rangle + e' + \Delta \cdot f(m))$, where e' only depends on the bsk and ksk, not on e. For suitable parameters, we have that $|e'| < |e|$. Combining this with the additive homomorphism of LWE ciphertexts we obtain a Fully Homomorphic Encryption scheme.

The PBS consists of the following four steps. See Figure 1 for an illustration.

Mod Switch We embed the input ciphertext into the group $\langle X \rangle \subset R_q$, which is of size $2N$. So in order to match up the moduli, we first perform a modulus switch. In particular, this takes as input the ciphertext $(\mathbf{a}, b) \in \mathbb{Z}^{n+1}$ and outputs $(\mathbf{a}', b') \in \mathbb{Z}_{2N}^{n+1}$, where

$$a_i' = \left\lfloor \frac{a_i 2N}{q} \right\rceil$$

and similar for b'.

Blind Rotation The blind rotation is the core of the PBS. We begin its description by introducing a homomorphic ciphertext multiplexer (CMUX) operation: given two GLWE ciphertext $\mathbf{c}_0, \mathbf{c}_1 \in R_q^{k+1}$ and a GGSW encryption C_μ of a bit $\mu \in \{0,1\}$, all under the same key $\mathbf{s} \in R_q^k$, we can compute the GLWE ciphertext

$$\mathbf{c} = (\mathbf{c}_1 - \mathbf{c}_0) \odot C_\mu + \mathbf{c}_0$$

where $\odot$ corresponds to the external product described in Section 2.2.1. By the additive homomorphism and the properties of the external product, $\mathbf{c}$ will encrypt the same plaintext as $\mathbf{c}_\mu$.

We are now ready to describe the blind rotation. Let $(\mathbf{a}, b) \in \mathbb{Z}_{2N}^{n+1}$ be the ciphertext after the mod switch. The blind rotation begins by constructing a trivial GLWE ciphertext $(\mathbf{0}, X^{-b} \cdot t)$, where $\mathbf{0} \in R_q^k$ is the all zero vector of size k. Then, it iterates over the elements a_i of $\mathbf{a}$, where the output GLWE ciphertext $\mathbf{c}$ from the previous iteration is multiplied element-wise by X^{a_i}. The two ciphertexts $\mathbf{c}$ and $X^{a_i} \cdot \mathbf{c}$ are input to a homomorphic CMUX, with the control bit being the corresponding part of the bsk which is itself a GGSW ciphertext encrypting s_i. Accordingly, the result is a ciphertext encrypting the same plaintext as $X^{a_i s_i} \cdot \mathbf{c}$. After executing the full loop, the result is a GLWE ciphertext encrypting $X^{-b+\sum_i a_i s_i} \cdot t = X^{-b+\langle \mathbf{a}, \mathbf{s} \rangle} \cdot t = X^{-m-e} \cdot t$. Note that in R_q, this corresponds to a negacyclic rotation of t by $m+e$ positions. By redundantly embedding the function f into the test polynomial t, we can ensure that the error e is rounded away and the resulting ciphertext contains an encryption of $\Delta \cdot f(m)$ in its constant coefficient.

[4] There is a requirement for the function to be negacyclic, but we omit details since it is irrelevant for our work.

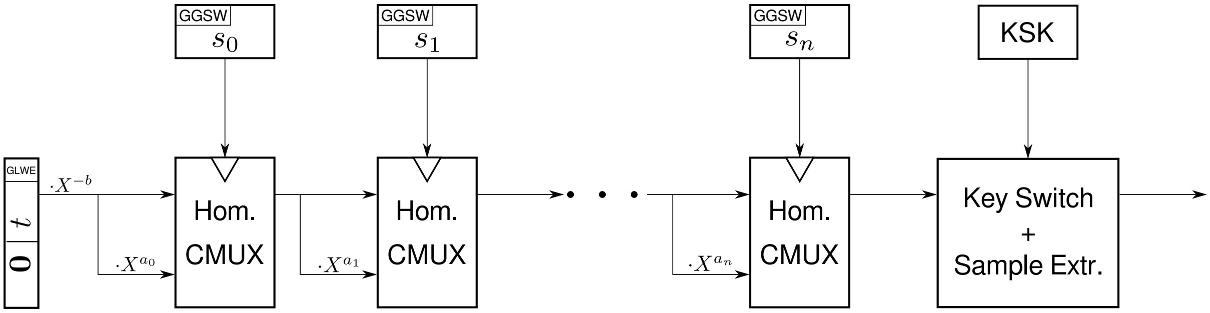

Figure 1: Illustration of TFHE's PBS (without mod switch)

Sample Extraction The goal of sample extraction is to convert a GLWE ciphertext into an LWE ciphertext encrypting the constant coefficient of the GLWE ciphertext, and where the key is a vector of bits corresponding to the concatenation of coefficient vectors in the GLWE secret key. We describe the special case of $k = 1$, since the generalization is straight-forward. So, given $(a, b) \in R_q^2$ we seek to construct $(\mathbf{a}', b') \in \mathbb{Z}_q^{N+1}$ such that $(b - a \cdot s)_0 = b' - \langle \mathbf{a}', \mathbf{s}' \rangle$, where $\mathbf{s}'$ is as described above. We note that

$$a \cdot s = \sum_i a \cdot s_i X^i = \sum_i \left(X^i a\right) s_i \quad .$$

Since addition in R_q is elementwise, we may set $a'_i = (X^i a)_0$ and $b' = b_0$ in order to achieve our goal.

Key Switch The key switch is a classic LWE type operation that follows from the observation that GLev ciphertexts can be used to homomorphically decrypt a GLWE ciphertext. Let $(\mathbf{a}, b) \in R_q^{k+1}$ be a GLWE ciphertext with corresponding secret key $\mathbf{s} \in R_q^k$. We would like to obtain a ciphertext $(\mathbf{a}', b') \in R_q^{k'+1}$ encrypting the same message as $(\mathbf{a}, b)$ but under the secret key $\mathbf{s}' \in R_q^{k'}$. We can do so by constructing a key switching key (ksk) that consists of GLev encryptions of s_i under $\mathbf{s}'$. Then, using the fact that we can multiply these ciphertexts with arbitrary constants using decomposition, we can homomorphically compute a ciphertext encrypting $b - \langle \mathbf{a}, \mathbf{s} \rangle$ under $\mathbf{s}'$, which yields the desired ciphertext. In the context of TFHE this operation has classically been applied to the LWE ciphertexts resulting from the sample extraction, but we remark that it may also be applied to GLWE ciphertexts.

Finally, we describe a slight modification of a GGSW ciphertext that also allows to perform a key switch, as already noticed in [BCL+23]. Recall that a GGSW ciphertext consists of a set of GLev ciphertexts of messages $m \cdot s_i$, where the s_i are the elements of the secret key $\mathbf{s}$. This allows to multiply a GLWE ciphertext and a GGSW ciphertext (both under secret key $\mathbf{s}$) to obtain a GLWE encryption of the product of the two messages under the same secret key $\mathbf{s}$. Now assume that we have a GLWE ciphertext encrypted under $\mathbf{s}$ and construct a GGSW ciphertext using GLev encryptions of the elements $m \cdot s_i$ but under a different key $\mathbf{s}'$. We can still apply the external product to obtain a GLWE ciphertext of the product, but the resulting ciphertext will be an encryption under $\mathbf{s}'$. In other words, by modifying the GGSW encryption and setting $m = 1$, we can also use the external product to perform a GLWE key switch. This will be useful in Section 4.2.

2.3 Parameters

As is plain from above description, there are a lot of parameters involved that impact the security, correctness and performance of TFHE. We do not go into details just yet but we remark that TFHE is typically instantiated with the ciphertext modulus $q = 2^{64}$ (see e.g. [CGGI16, Zam22b]). The other parameters are the result of a complex optimization procedure, but for concreteness the reader may consider Table 2.

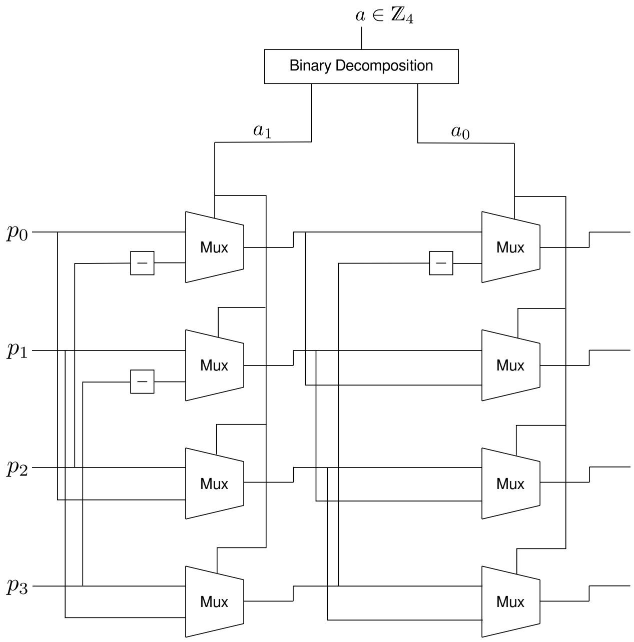

Figure 2: Circuit for multiplication of polynomial $p = \sum_i p_i X^i \in Z_q[X]/(X^4 + 1)$ by X^a

3 Blind Rotation

Since the blind rotation is the core of the PBS, we begin by describing our circuit for this part of the PBS. We start out with a circuit for one iteration and then explain how we scale to a full blind rotation.

3.1 One Step of the Blind Rotation

One of the bottleneck operations during a step of the blind rotation is polynomial multiplication in R_q. Implementations like [Zam22b] or the one accompanying [CGGI16] use an FFT on floating point numbers, which are very inefficient to realize in the arithmetic circuit model. Luckily, the choice of modulus $q = 2^{64} - 2^{32} + 1$ admits performing this multiplication using the NTT, so here we diverge from common implementations and use an NTT circuit instead.

The second main operation is multiplication by the monomial X^a, where $a \in \mathbb{Z}_{2N}$ is an input. This corresponds to a negacyclic rotation by a in the ring R_q, which is a rather trivial (and linear) operation on a CPU. However, in the circuit model it is not quite as easy, since a is not known during circuit construction and we cannot “rewire” a circuit during evaluation. Note that this operation would be trivial in the circuit model, if a was a fixed constant. So our solution to this problem is to implement subcircuits for negacyclic rotations by powers of two. Then we apply each of the subcircuits and each time select the rotated or not rotated polynomial using a MUX and the corresponding bit of the binary decomposition of a as control bit. See Figure 2 for an illustration. This is a circuit of size $O(N \log N)$ and thus significantly more expensive than on a CPU. All other operations (addition, decomposition) are readily available in plonky2 and are easily generalized to polynomials.

3.2 Scaling to Full Blind Rotation

The obvious way to scaling the blind rotation step to n steps is to build a large circuit with n subcircuits performing one step each. While this works in theory, the circuit size blows up, since n is very large. In fact, in our experiments we were only able to do this for small n, cf. Section 5.2. For larger n, our test machines

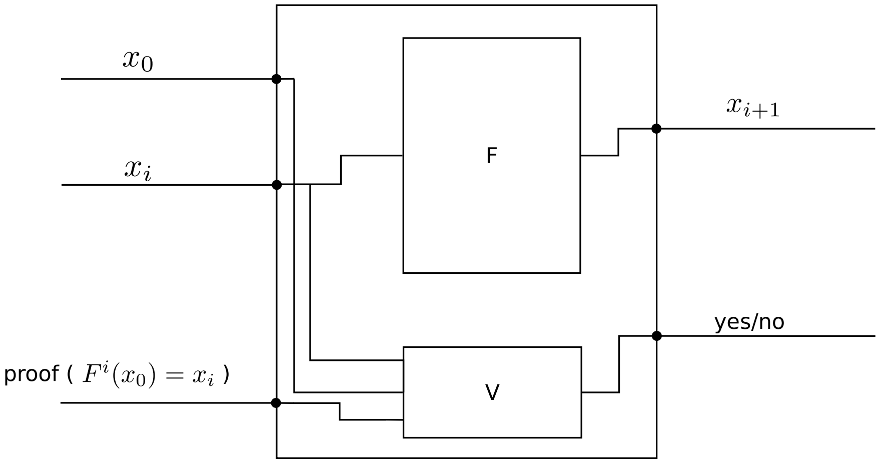

Figure 3: Illustration of recursion-based IVC. F is the circuit computing the loop iteration, V is the verifier circuit verifying a proof for F′, the illustrated circuit itself. The public input is x_0 along with the public outputs x_{i+1} and the verifier output, the other two inputs are prover inputs.

ran out of memory, which quickly becomes the bottleneck.

Another easy approach to scaling the blind rotation is to simply prove each step individually and send the proofs and intermediate results to the verifier. The verifier can check each of the proofs. This achieves a proving time that is linear in the number of steps and can be performed with memory equivalent to one step. The issue with this approach is the proof size and verifier complexity: the proof now consists of n smaller proofs and n GLWE ciphertexts (the intermediate results) and the verifier needs to check all individual proofs. In other words, the proof and verifier are not succinct as they are linear in the circuit size. With our example parameters (cf. Table 2), each ciphertext has size about 4 kB, so $n > 2^9$ ciphertexts alone amount to about 2 MB, without even considering the substantially larger proofs.[5] In some applications this might be acceptable, but typically this is considered too large and the burden on the verifier too costly.

Clearly, we can use a hybrid strategy to reduce the proof size and verifier complexity. If we are working on a machine that is able to prove t steps of the blind rotation at a time, we may take advantage of this and cut the number of intermediate results and inner proofs down by a factor t.

3.2.1 Blind Rotation Based on IVC

As noted in Section 1, plonky2 supports recursion and thus allows constructing IVC. The general idea of IVC to prove the correct execution of a loop is the following. Let F be the function describing the step function of the loop, i.e., we want to prove $y = F^n(x)$, where $F^n(x)$ corresponds to applying F successively n times to x. We may augment F to obtain a function F' that takes as input the (public) initial value x, a private prover input y_i, and, also as a private witness, a proof π_i. F' outputs y_{i+1} while also verifying that the proof π_i is valid with respect to F' itself (for input x and output y_i). For an illustration of a circuit F′ computing F' see Figure 3. A proof for this circuit attests to the correctness of the combined statement: 1) $y_{i+1} = F(y_i)$ and 2) the verifier accepts π_i as a proof of $y_i = F(x)$. The prover may now successively prove $y_i = F'(y_{i-1}, \pi_{i-1})$ to obtain π_i and obtain the output y_n along with a succinct proof π_n. See e.g. [Tha22] for more details and references.

This approach seems like the ideal tool to prove a blind rotation. The overhead for the prover of proving the verifier circuit for each iteration is relatively small compared to our function F implementing a step of the blind rotation, due to plonky2's focus on optimization of recursion. There is one caveat, in that in our description above there is no public input beyond the initial x. In particular, the individual loop iterations

[5] We remark that it might be possible to compress the set of proofs into a single smaller proof using recursion, but this will certainly not work for the intermediate results, which need to be sent and checked in any case.

do not receive any public input specific to the iteration. In contrast, in our application of blind rotation, every loop iteration receives a different part of the bootstrapping key and ciphertext element. One way to solve this is by passing the entire bootstrapping key as input to the step function and use a counter that keeps track of the loop iteration. Then we could use a selector subcircuit that picks out the correct part of the key and the LWE mask for the current iteration. Note that this subcircuit grows linearly with the size of the bootstrapping key, which consists of $n(k+1)^2\ell N$ elements in $\mathbb{Z}_q$. For small n this circuit is smaller than the circuit for our step function and thus does not incur too large of an overhead, but as n grows it quickly becomes the bottelneck.

So we opt for another solution based on hashing: since plonky2 is optimized for recursion and its verifier needs to perform hashing operations, it necessarily supports efficient proofs for evaluating hash functions. Accordingly, we let the elements of the bootstrapping key and LWE ciphertext be private prover inputs, which may differ across iterations, and extend the circuit computing the loop to compute a running hash chain over them. The final hash is part of the output and the verifier may recompute the hash in order to verify that the prover used the correct bootstrapping key and LWE ciphertext in the correct order. In fact, we split the hash over the bootstrapping key and the ciphertext into two seperate hash chains. This has the advantage that for a fixed bootstrapping key the verifier needs to compute the corresponding hash chain only once, e.g. during key generation. Since the bootstrapping key is orders of magnitude larger than the ciphertext, this significantly speeds up the verifier in case multiple PBS operations per bootstrapping key are to be evaluated. Note that the verifier does not even need to store the bootstrapping key after computing this hash and may perform verification with the hash only. We remark that we do not claim novelty for the idea of replacing a large public input with a large private input and a small public hash value. This seems to be folklore in the zkVM literature and even plonky2 already employs this technique itself. The novelty here is in the observation that it provides an elegant solution to our problem of different, and potentially very large, inputs to each loop iteration.

4 Extension to Full PBS

We now outline how we extend the IVC-based prover to a full PBS. In contrast to a regular, non-recursive prover, this is not trivial and we cannot simply plug together the step circuits and obtain a prover for the complete functionality. However, we will see that we can still extend the prover efficiently to the full PBS. The resulting IVC circuit is illustrated in Figure 4.

4.1 Mod Switch

Recall that we need to switch the modulus of the input ciphertext $c = (\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$ to turn it into a ciphertext $c' = (\mathbf{a}', b') \in \mathbb{Z}_{2N}^{n+1}$. The resulting elements of c' are used as input to the negacyclic rotation operation (cf. Section 3.1), where they are binary decomposed and the individual bits are used as control bits of MUX operations that pick the shifted or unshifted polynomial, where the shift is fixed. It follows that an easy way to perform the mod switch is to consider the element a_i in each iteration (or $-b$ in the first iteration), perform a bit decomposition and use the $\log N + 1$ most significant bits as input to the polynomial rotation. In fact, in order to round to the closest integer, we use the $\log N + 2$ most significant bits and the final shift by one position is performed twice, once with the $(\log N + 1)$st most significant bit and again with the $(\log N + 2)$nd bit (the latter leading to the correct rounding). While this approach does not perform the mod switch exactly as described in Section 2.2, it is a close enough approximation as we quantify next.

For an element $a \in \mathbb{Z}_q$, the mod switch operation would require to perform the operation $a \mapsto \lfloor a \cdot 2N/q \rceil$. The circuit we describe above instead performs the operation $a \mapsto \lfloor a \cdot 2N/2^{64} \rceil$.

Lemma 1. *Let $c = (\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$ be an LWE ciphertext with binary key $\mathbf{s} \in \{0,1\}^n$ and let $p \in \mathbb{Z}$ such that $q/p = (1 - \epsilon)$. Then performing the mod switch to $2N$ using p instead of q increases the error by at most $\epsilon \cdot 2N$.*

Proof. Let $\epsilon_b = \frac{b \cdot 2N}{p} - \left\lfloor \frac{b \cdot 2N}{p} \right\rceil$ and $\epsilon_i = \frac{a_i \cdot 2N}{p} - \left\lfloor \frac{a_i \cdot 2N}{p} \right\rceil$. Then we have

$$\begin{aligned}
\left\lfloor \frac{b \cdot 2N}{p} \right\rceil - \sum_i \left\lfloor \frac{a_i \cdot 2N}{p} \right\rceil s_i &= \frac{2N}{p} \left(b - \langle \mathbf{a}, \mathbf{s} \rangle \right) + \epsilon_b - \sum_i \epsilon_i s_i \\
&= \frac{q}{p} \cdot \frac{2N}{q} \left(b - \langle \mathbf{a}, \mathbf{s} \rangle \right) + \epsilon_b - \sum_i \epsilon_i s_i \\
&= \frac{2N}{q} \left(b - \langle \mathbf{a}, \mathbf{s} \rangle \right) - \epsilon \frac{2N}{q} \left(b - \langle \mathbf{a}, \mathbf{s} \rangle \right) + \epsilon_b - \sum_i \epsilon_i s_i \ .
\end{aligned}$$

The lemma follows, since $\left(b - \langle \mathbf{a}, \mathbf{s} \rangle \right) / q < 1$ and the rounding errors ϵ_b and ϵ_i have a similar distribution as they would when mod switching using q. □ □

Since the mod switch operation itself incurs an error of $O(\sqrt{n})$ (cf. the $\epsilon_b - \sum_i \epsilon_i s_i$ term), it is clear that for typical parameters the additional error of $\epsilon \cdot 2N$ is negligible if $\epsilon \ll 1/N$.

4.2 Key Switch

The biggest challenge in extending the blind rotation to a full PBS is the key switch as it is structurally quite different from the blind rotation. There are essentially two options to add the key switch in a straightforward manner. First, one could extend the circuit to perform the full key switch in each round on the accumulator value in parallel to the blind rotation step and select the output value depending on the loop counter using a MUX. The drawback of this solution is that the circuit for a full key switch is quite large compared to a step of the blind rotation and thus would slow down each step significantly.

The second approach would be to perform just one of the $k \cdot N + 1$ steps of the key switch in every iteration and again select the output depending on the loop counter. The overhead in each iteration would be very small and thus each iteration would be just as fast to prove as without the key switch. However, we now need to perform $n + k \cdot N$ steps of the loop iteration instead of just n. Since $k \cdot N$ is typically larger than n, this incurs a slowdown of at least a factor 2.

Clearly, one could attempt to mitigate above issues by implementing a hybrid, but we chose a different path. Inspired by [BCL+23] we do not perform sample extraction and then an LWE key switch, but rather first perform a GLWE key switch to a partial key of size n and then perform a trivial sample extraction on the verifier side (cf. Section 4.3). The advantage is that the GLWE key switch has the same structure as the external product but with a key switching key instead of a GGSW encryption as input. This means, we can re-use the largest part of the blind rotation circuit, the external product, for the key switch. The additional logic of selecting the input and output to the external product circuit is small in comparison and does not affect prover time, and this increases the overall number of loop iterations only by one. The drawback is that the key switch needs to use the same parameters (decomposition base and level, ring and GLWE dimension) as the blind rotation, but the key switching key needs to carry larger noise for security due to the key being partial. So this requires tweaking the parameters. Looking ahead, we note that we use the parameter optimization approach from [BCL+23] but restricting the search space such that the bootstrapping and the key switch use the same parameters.

4.3 Sample Extraction

Sample extraction takes as input a GLWE ciphertext and outputs an LWE ciphertext of dimension $n = kN$ where the key of the resulting ciphertext is the (concatenation of the) coefficient vector(s) of the GLWE secret key. This conversion consists of a simple, fixed re-ordering and negation of a few elements and is thus very cheap and easy to perform. Hence, we may assume that the verifier performs it itself, i.e. we may assume the prover sends the GLWE ciphertext resulting from the PBS and GLWE key switch to the verifier and the verifier will perform the sample extraction itself. In the following we note that we can trivialize

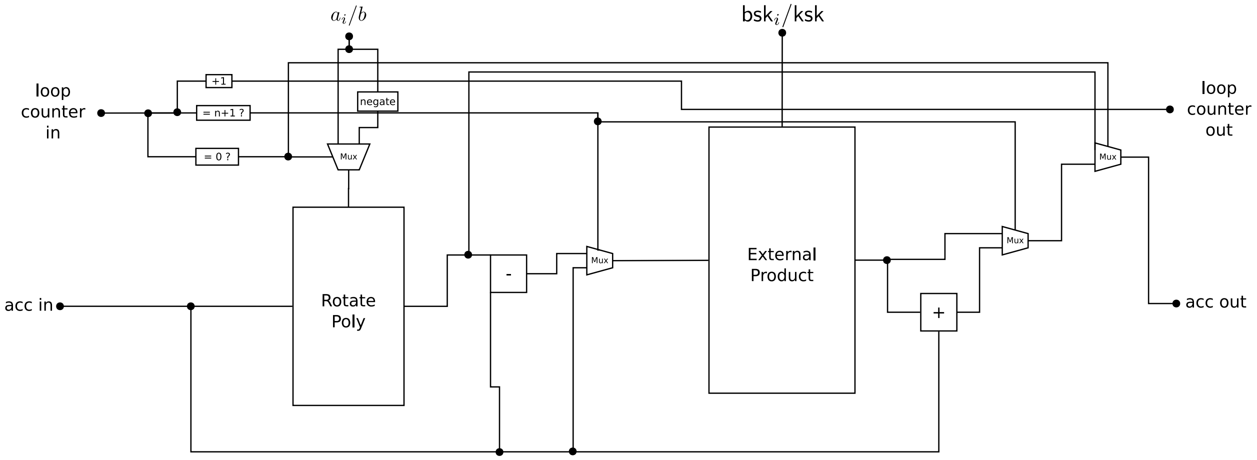

Figure 4: IVC Circuit for TFHE's PBS. This corresponds to the subcircuit F from Figure 3. We omit the hash chains over the bootstrapping key and ciphertext $(\mathbf{a}, b)$. The mod switch is not depicted since we consider it integrated into the polynomial rotation as described in Section 4.1.

the sample extraction even further by modifying the key switching key, ensuring that the LWE sample is obtained by literally copying a subset of the coefficients from the GLWE sample.

In the following we assume $k = 1$ for simplicity, but the generalization to $k > 1$ is straight-forward. Let $\mathbf{s}' \in \mathbb{Z}^n$ be the target LWE key, i.e. the key under which the output ciphertext of the PBS should be encrypted. (Typically, this is the same key under which the input ciphertext to the PBS is encrypted.) Let $s \in R_q$ be the key of the GLWE ciphertext that is the result of the key switch. For a GLWE ciphertext (a, b), write $\tilde{m} = b - a \cdot s$. Then we have

$$\tilde{m}_0 = b_0 - (a \cdot s)_0 = b_0 - \left(\left(\sum_i a_i \cdot X^i \right) \cdot s \right)_0 = b_0 - \sum_i a_i \cdot (X^i \cdot s)_0 \ .$$

So if we set s such that $(X^i \cdot s)_0 = s'_i$, we see that the coefficient vector of a together with b_0 forms a valid LWE ciphertext encrypting $\tilde{m}_0$ under $\mathbf{s}'$. So by modifying the key switching key to switch to s as defined above, we may think of this modification as integrating the usual sample extraction into the key switch. This also works for $n < N$, since we can view $\mathbf{s} \in \{0, 1\}^n$ as an N-dimensional vector, where the last $N - n$ elements are 0. This also means we can drop the corresponding elements of the extracted mask $\mathbf{a}$.

5 Experimental Results

In this section, we describe our experimental results. As we have observed, building a circuit for the PBS is a tedious task. As such, an easier way to approach proving the correct execution of a PBS would be to implement it in a zkVM. While easy to use, zkVMs introduce a significant overhead and designing a circuit for a SNARK is typically much more efficient, especially in terms of proving time. To quantify how much exactly we are gaining from the latter approach, we first give some results on our implementation in existing zkVMs and then proceed to experimental results of our design in plonky2.

5.1 Zero Knowledge Virtual Machines

Modern general purpose zkVMs are computing platforms based on STARKs [BSBHR18] and the RISC-V instruction set architecture. To use such zkVMs, one must write the program whose execution they would like to prove in a general purpose programming language such as Rust, which supports RISC-V as a compilation

	Prover time (min)	Verifier time (s)
RISC Zero	6.4	0.5
SP1	2.1	1.1

Table 1: Performance of proving and verifying a single step of the blind rotation using zkVMs on an AWS `Hpc7a.96xlarge` machine.

target. The compiled program is then given as input to the zkVM, which executes it and produces a proof of correct execution. Notice the subtle difference between this approach and more common verifiable computation techniques: the circuit for which the proof is generated is not that of the compiled program, but that of the virtual machine which receives the compiled program as *input*. The versatility of zkVMs makes them a powerful tool in that they allow users unfamiliar with arithmetic circuit generation or domain specific languages [BIM+23, Sta20, Azt23] to easily generate proofs for any program they have already built. However, the overhead caused by the VM logic is significant and as such, there is a trade-off between ease of use and performance.

To set a performance baseline, we use general purpose zkVMs RISC Zero [BGZ23] and SP1 [Suc24] to generate proofs of correct execution for a step of TFHE's bootstrapping. Since we are not able to prove a complete PBS in one go due to performance limitations, we extrapolate from micro-benchmarks to estimate real performance.

Using RISC Zero, we are able to measure that one step of the blind rotation takes about 26 million RISC-V instructions to complete. The time taken to generate a proof of this computation is about 6 minutes on average on an AWS `Hpc7a.96xlarge` machine. In contrast, using the SP1 zkVM, we measure that one step of the blind rotation takes about 27 million instructions. We measure a timing of approximately 2 minutes per step of the blind rotation on average with the same type of machine.

A performance comparison between the two zkVMs is presented in Table 1. Note that the timings included in this table represent a single step of the blind rotation (cf. Section 2.2.2). The time taken to prove a full PBS using a zkVM increases by a factor approximately equal to the parameter n (cf. Table 2). This is explained by proof composition and recursion techniques allowing the proving time to grow only linearly in the size of the circuit even though the underlying SNARK may not offer a linear time prover.

5.2 Our plonky2 based Implementation

Parameters There are a multitude of parameters of TFHE's PBS that may be tweaked and optimized, all of which impact the correctness, security and performance of the PBS. This optimization is very complex. Indeed, it is a subject of scientific research in its own right [BBB+23]. As pointed out in Section 4.2, we need to tweak parameters to ensure correctness and security. For this, we follow the approach of [BCL+23] (which in turn uses an adaptation of [BBB+23]), since some of the proposed algorithms are similar to our circuit design. In order to obtain a usable and performant set of paramters, we tweaked the optimization code from [BCL+23] by restricting the search space to fit our needs (cf. Section 4.2). We show the corresponding parameters in Table 2. We remark though that the optimization targets a computational model that is different from the arithmetic circuit model, and, as we saw in Section 3.1, some of the operations have significantly different cost in different computational models. It follows that the parameters we obtained might not minimize the circuit for the PBS and might not be optimal. However, fully optimizing parameters for our setting is out of scope of this work and we believe our results already demonstrate the progress of our approach towards practical verifiable FHE.

Results We experimented with our plonky2-based implementation on a few different machines: a modern consumer laptop (M3 MacBook Pro) and a two AWS EC2 compute-optimized instances. Proof size is obviously independent of the machine and was a little less than 200 kB. In our tests with a non-recursive approach (cf. Section 3.2) even AWS instances with large amounts of memory struggled to prove even a

q	n	N	k	B	ℓ
$2^{64} - 2^{32} + 1$	728	2^{10}	1	2^5	4

Table 2: TFHE parameters suitable for our circuit. The noise parameters are set such that we may claim 128 bits of security relying on the lattice estimator. This parameterization allows for a plaintext space of size 4.

	CPU Cores	Memory (GB)	Prover time (min)	Verifier time (ms)
M3 MacBook Pro	8	16	40	4
`C6i.metal`	128	256	21	8
`Hpc7a.96xlarge`	192	768	18	8

Table 3: Performance of proving and verifying a PBS operation using our plonky2 based implementation. Proving benefits more from parallelization than verifying, which explains why a consumer laptop may outperform a multi-CPU machine for verification only.

small-ish number of steps ($n \approx 50$) due to the memory requirement. As expected, this was not the case in our experiments with the recursive IVC approach, where memory consumption is independent of the number of loop iterations n. In fact, even an older laptop with just 8 GB of memory was able to run the prover, albeit taking significantly longer than the better performing machines we report timings for in Table 3.

6 Security in the UC Model

The goal of this section is to rigorously prove a strong security notion achieved by combining a passively secure FHE scheme and a SNARG. We do so by capturing a classic use case as an MPC application and prove its security in the UC model [Can01]. For this, we need to first give formal definitions of the primitives involved.

6.1 Preliminaries

Definition 2. *Let $\mathcal{P}$, $\mathcal{C}$ and $\mathcal{F}$ be the plaintext space, ciphertext space and a circuit family, respectively. A symmetric FHE scheme $\mathcal{E}$ for $\mathcal{F}$ is a tuple of algorithms:*

- $\mathcal{E}.\mathrm{Gen}(1^\lambda)$*: generates a secret key s and an evaluation key p*
- $\mathcal{E}.\mathrm{Enc}(s, m)$*: takes a key s and message $m \in \mathcal{P}$ and outputs a ciphertext $c \in \mathcal{C}$*
- $\mathcal{E}.\mathrm{Dec}(s, c)$*: takes a secret key and a ciphertext $c \in \mathcal{C}$ and returns a message in $\mathcal{P}$.*
- $\mathcal{E}.\mathrm{Eval}(p, f, (c_1, \dots, c_\ell))$*: takes an evaluation key p, a circuit f in $\mathcal{F}$ and a tuple of input ciphertexts $(c_1, \dots, c_\ell) \in \mathcal{C}^\ell$ and returns a ciphertext $\mathcal{C}$.*

In the following we will assume that $\mathcal{E}.\mathrm{Dec}$ and $\mathcal{E}.\mathrm{Eval}$ are deterministic. We extend encryption and decryption to vectors of messages and ciphertexts, resp., in the natural way, i.e. element-wise.

An FHE scheme $\mathcal{E}$ is *correct* if for all $(m_1, \dots, m_\ell) \in \mathcal{P}^\ell$ and all $f \in \mathcal{F}$ we have

$$\Pr\left[m \neq f(m_1, \dots, m_\ell) \;\middle|\; \begin{array}{c} (s,p) \leftarrow \mathcal{E}.\mathrm{Gen}(1^\lambda) \\ (c_1, \dots, c_\ell) \leftarrow \mathcal{E}.\mathrm{Enc}(s, m_1, \dots, m_\ell) \\ c \leftarrow \mathcal{E}.\mathrm{Eval}(p, f, c_1, \dots, c_\ell) \\ m \leftarrow \mathcal{E}.\mathrm{Dec}(s, c, f, c_1, \dots, c_\ell) \end{array} \right] = \mathrm{negl}(\lambda) \ .$$

Definition 3. *A symmetric FHE scheme $\mathcal{E}$ is IND-CPA secure if for all PPT adversaries $\mathcal{A}$ it holds that*

$$\left| 2 \cdot \Pr\left[b = b' \,\middle|\, \begin{array}{c} (s,p) \leftarrow \mathcal{E}.\,\mathrm{Gen}(1^\lambda) \\ b \leftarrow U(\{0,1\}) \\ b' \leftarrow \mathcal{A}^{O^b_{Enc_s}}(p) \end{array} \right] - 1 \right| = \mathrm{negl}(\lambda)$$

where $O^b_{Enc_s}(m_0, m_1)$ is the oracle that returns $\mathcal{E}.\,\mathrm{Enc}(s, m_b)$.

In this work we are interested in SNARGs with pre-processing. The following definition is adapted from [Chi14, BISW17].

Definition 4. *A preprocessing succinct non-interactive argument (ppSNARG) $\Pi = (\mathrm{Gen}, \mathrm{Prove}, \mathrm{Verify})$ for a circuit family $\mathcal{F}$ over some domain $\mathcal{D}$ is a triple of algorithms such that:*

1. Gen *takes as input a security parameter $\lambda \in \mathbb{N}$ and circuit $f \in \mathcal{F} : \mathcal{D}^\ell \mapsto \mathcal{D}$ and outputs a reference string σ and verification state τ.*
2. Prove *takes as input σ, a statement $(x, y) \in \mathcal{D}^{\ell+1}$ outputs a proof π when $f(x) = y$.*
3. Verify *takes as input τ, a statement (x, y) and a proof π and outputs* Acc *or* Rej.

For succinctness, we require the size of π and the running time of Verify *to be sublinear in the size of f.*

A ppSNARG is *complete* if for all $f \in \mathcal{F}$, $(x, y) \in \mathcal{D}^{\ell+1}$ such that $f(x) = y$,

$$\Pr\left[\mathrm{Verify}(\tau, (x, y), \pi) = \mathrm{Acc} \,\middle|\, \begin{array}{c} (\sigma, \tau) \leftarrow \mathrm{Gen}(\lambda, f) \\ \pi \leftarrow \mathrm{Prove}(\sigma, (x, y)) \end{array} \right] = 1 \ .$$

A ppSNARG is *sound* if for all $f \in \mathcal{F}$ and all PPT adversaries $\mathcal{A}$

$$\Pr\left[\begin{array}{c} \mathrm{Verify}(\tau, (x, y), \pi) = Acc \\ \wedge f(x) \neq y \end{array} \,\middle|\, \begin{array}{c} (\sigma, \tau) \leftarrow \mathrm{Gen}(\lambda, f) \\ ((x, y), \pi) \leftarrow \mathcal{A}(\sigma) \end{array} \right] = \mathrm{negl}(\lambda) \ .$$

6.2 UC Security for FHE plus SNARG

Setting We will model the basic outsourcing application as a secure 2-party computation. So let $\mathcal{I}$ be the party holding $f, m_1, \ldots, m_\ell$ that wants to obtain $f(m_1, \ldots, m_\ell)$, where the $m_1, \ldots, m_\ell$ are private and f is public. Let $\mathcal{A}$ be the computing party. We define the ideal functionality we are targeting below (see Algorithm 1). With every query we let $\mathcal{I}$ specify whether or not $m = f(m_1, \ldots, m_\ell)$ is secret or public by sending a public bit b. If $b = 0$, then only $\mathcal{I}$ should receive the output m, otherwise m is also sent to $\mathcal{A}$. In the language of encryption schemes, this models decryption oracles and thus provides security against active attacks.

Algorithm 1: Ideal Functionality

```
receive $(f, m_1, \ldots, m_\ell, b) \in \mathcal{F} \times \mathcal{P}^\ell \times \{0,1\}$ from $\mathcal{I}$
receive $a \in \{0,1\}$ from $\mathsf{S}$
if $a = 1$ then
    $m \leftarrow \perp$
else
    $m \leftarrow f(m_1, \ldots, m_\ell)$
if $b = 0$ then
    return $m$ to $\mathcal{I}$
else
    return $m$ to $\mathcal{I}$ and $\mathsf{S}$
```

We will realize the ideal functionality using symmetric key FHE (with plaintext $\mathcal{P}$, ciphertext $\mathcal{C}$ and supported circuit family $\mathcal{F}$) and a ppSNARG for the circuit family $\{\mathcal{E}.\operatorname{Eval}(\cdot, f, \cdots\} \mid f \in \mathcal{F}\}$. We assume that $\mathcal{I}$ is honest, but $\mathcal{A}$ may be fully corrupted. The protocol is the simplest one one can think of, see Algorithm 2 and Algorithm 3.

Algorithm 2: Setup

```
$\mathcal{I}$ computes $(s, p) \leftarrow \mathcal{E}.\operatorname{Gen}(1^\lambda)$
$\mathcal{I}$ sends the evaluation key $p$ to $\mathcal{A}$
```

Algorithm 3: Main Protocol

```
$\mathcal{I}$ receives inputs $(f, m_1, \ldots, m_\ell, b)$
$\mathcal{I}$ computes $(c_1, \ldots, c_\ell) \leftarrow \mathcal{E}.\operatorname{Enc}(s, m_1, \ldots, m_\ell)$
$\mathcal{I}$ sends $(f, c_1, \ldots, c_\ell)$ to $\mathcal{A}$
$\mathcal{A}$ sets $f' = \mathcal{E}.\operatorname{Eval}(p, f, \cdot, \ldots, \cdot)$
$\mathcal{A}$ computes $(\sigma, \tau) \leftarrow \Pi.\operatorname{Gen}(1^\lambda, f')$, $c \leftarrow f'(c_1, \ldots, c_\ell)$ and $\pi \leftarrow \Pi.\operatorname{Prove}(\sigma, c_1, \ldots, c_\ell, c)$
$\mathcal{A}$ sends $(c, \pi)$ to $\mathcal{I}$
$\mathcal{I}$ sets $f' = \mathcal{E}.\operatorname{Eval}(p, f, \cdot, \ldots, \cdot)$
$\mathcal{I}$ computes $(\sigma, \tau) \leftarrow \Pi.\operatorname{Gen}(1^\lambda, f')$ and $d \leftarrow \Pi.\operatorname{Verify}(\tau, (c_1, \ldots, c_\ell, c), \pi)$
if $d = \mathrm{Rej}$ then
    $\mathcal{I}$ sets $m \leftarrow \perp$
else
    $\mathcal{I}$ computes $m \leftarrow \mathcal{E}.\operatorname{Dec}(s, c)$
if $b = 1$ then
    $\mathcal{I}$ sends $m$ to $\mathcal{A}$
    $\mathcal{A}$ outputs $m$
$\mathcal{I}$ outputs $m$
```

Theorem 1. *The protocol given in Algorithm 2 and Algorithm 3 realizes the functionality from Algorithm 1 in the UC model.*

Proof. We first give the simulator S that has access to the ideal functionality (but not the secret input of $\mathcal{I}$) in Algorithm 4. The setup in Algorithm 2 is simulated by S simply by running it itself.

It remains to show that the simulation is indeed indistinguishable to the environment $\mathcal{Z}$ from an execution of the real protocol. Recall that $\mathcal{Z}$ controls the adversary $\mathcal{A}$ and the inputs of $\mathcal{I}$ and gets to see the output of $\mathcal{I}$.

The simulation differs from the execution of the real protocol in two places: 1) it uses dummy messages $m_1, \ldots, m_\ell = 0^\ell$, and 2) it ignores the ciphertext sent by $\mathcal{A}$ and uses the output of the ideal functionality instead. We define a modified protocol that proceeds as in the real protocol, but mirrors the ideal functionality instead of decrypting the ciphertext from $\mathcal{A}$ (but uses the real input messages), see Algorithm 5.

From the view of $\mathcal{Z}$, the real and the simulation of the modified protocol (Algorithm 5) are identical up to the point where the output m is computed in case $d \neq \mathrm{Rej}$. In order to reach that part, we must have $d = \mathrm{Acc}$, so the proof sent by $\mathcal{A}$ must verify. In order to distinguish the two worlds, we must now have $\mathcal{E}.\operatorname{Dec}(s, c) \neq f(m_1, \ldots, m_\ell)$. There are two cases: either, we have $\mathcal{E}.\operatorname{Eval}(p, f, c_1, \ldots, c_\ell) \neq c$, which means that $\mathcal{A}$ can be used to break the soundness of Π; or we have $\mathcal{E}.\operatorname{Eval}(p, f, c_1, \ldots, c_\ell) = c$, meaning that $\mathcal{E}.\operatorname{Dec}(s, c) \neq f(m_1, \ldots, m_\ell)$. This violates the correctness of $\mathcal{E}$ and happens only with negligible probability. Note that the correctness conditions requires this to be the case for any input messages $(m_1, \ldots, m_\ell) \in \mathcal{P}^\ell$ and any $f \in \mathcal{F}$, so this is independent of $\mathcal{Z}$'s choice of $\mathcal{I}$'s input.

The only difference between a simulation of the real protocol (i.e. the ideal world) and of the modified protocol in $\mathcal{Z}$'s view is that the ciphertexts $c_1, \ldots, c_\ell$ encrypt the actual input messages in the modified

Algorithm 4: Simulator S (Main Protocol)

```
receive $\mathcal{I}$'s public inputs $f$ and $b$
$(m_1, \ldots, m_\ell) \leftarrow 0^\ell$
$(c_1, \ldots, c_\ell) \leftarrow \mathcal{E}.\,\mathrm{Enc}(s, m_1, \ldots, m_\ell)$
send $(f, c_1, \ldots, c_\ell)$ to $\mathcal{A}$
receive $(c, \pi)$ from $\mathcal{A}$
set $f' = \mathcal{E}.\,\mathrm{Eval}(p, f, \cdot, \ldots, \cdot)$
compute $(\sigma, \tau) \leftarrow \Pi.\,\mathrm{Gen}(1^\lambda, f')$ and $d \leftarrow \Pi.\,\mathrm{Verify}(\tau, c_1, \ldots, c_\ell, c), \pi)$
if $d = \mathrm{Rej}$ then
    $a \leftarrow 1$
else
    $a \leftarrow 0$
send $a$ as $\mathcal{A}$'s input to the functionality
if $b = 1$ then
    receive the output $m$ from the functionality and forward it to $\mathcal{A}$
```

protocol, while in the ideal world they encrypt 0^ℓ. Note that in both cases the output message is computed from the actual inputs. Clearly, the indistinguishability of the modified protocol and the ideal world follows from the IND-CPA security of $\mathcal{E}$. □

Our proof shows that the combination of FHE and a suitable SNARG provides security even against active adversaries as long as the verifier knows the function and input ciphertexts. Accordingly, the proof does not need to rely on knowledge extraction to obtain any of these inputs. A natural setting is secure outsourcing of computation, where a single party (holding a secret key) asks a server to securely evaluate a function on ciphertexts it encrypted itself. In other applications, where multiple parties use a public key scheme to provide different inputs and/or the function, these may not be known to the verifying party and serve as witnesses instead. In that case, the situation becomes significantly more complex, as the simulator will need to extract the information it does not know. Unfortunately, most efficient SNARGs rely on rewinding extractors for knowledge soundness, which poses troubles in simulation-based proofs, especially in the UC setting.

One may wonder, if our proposed protocol is useful from an efficiency standpoint. This is reasonable, since, after all, the user $\mathcal{I}$ needs to run $\Pi.\,\mathrm{Gen}(\cdot)$ for each query in order to obtain τ and ppSNARGs give no guarantee that this is more efficient then simply performing the computation itself. However, from the protocol it is clear that $\Pi.\,\mathrm{Gen}(\cdot)$ only needs to be run once per distinct $f \in \mathcal{F}$, so there is enormous potential for amortization over many queries to the same function. Furthermore, note that the preprocessing as we defined it is public and transparent. This means that the computing party $\mathcal{A}$ could perform the preprocessing, send τ to the user $\mathcal{I}$ and commit to it by, e.g. uploading a hash of (f, τ) to a blockchain. Now any party, including competitors in a market-based application, may check the validity of the setup. If the user $\mathcal{I}$ is willing to rely on such non-cryptographic incentives to ensure integrity of the verification state τ, it never needs to perform $\Pi.\,\mathrm{Gen}(\cdot)$. In fact, $\mathcal{I}$ never even has to read the entire circuit f in this setting.

7 Applications

Equipped with a verifiable bootstrapping prototype and with a good understanding of its security, we slightly augment it to demonstrate how it can enable novel applications.

Privacy-preserving machine learning applications such as [Zam22a, LKL+22] rely on FHE to compute common ML operations (e.g., weighted sums and activation functions) over encrypted user data. Such frameworks allow for a typical two-party computation protocol where a client sends a private input (e.g., an encrypted prompt) to a server, which evaluates a machine learning model on this prompt and returns the

Algorithm 5: Modified Protocol

```
$\mathcal{I}$ receives inputs $(f, m_1, \ldots, m_\ell, b)$
$\mathcal{I}$ computes $(c_1, \ldots, c_\ell) \leftarrow \mathcal{E}.\operatorname{Enc}(s, m_1, \ldots, m_\ell)$
$\mathcal{I}$ sends $(f, c_1, \ldots, c_\ell)$ to $\mathcal{A}$
$\mathcal{A}$ sets $f' = \mathcal{E}.\operatorname{Eval}(p, f, \cdot, \ldots, \cdot)$
$\mathcal{A}$ computes $(\sigma, \tau) \leftarrow \Pi.\operatorname{Gen}(1^\lambda, f')$, $c \leftarrow \mathcal{E}.\operatorname{Eval}(p, f, c_1, \ldots, c_\ell)$ and $\pi \leftarrow \Pi.\operatorname{Prove}(\sigma, c_1, \ldots, c_\ell, c))$
$\mathcal{A}$ sends $(c, \pi)$ to $\mathcal{I}$
$\mathcal{I}$ sets $f' = \mathcal{E}.\operatorname{Eval}(p, f, \cdot, \ldots, \cdot)$
$\mathcal{I}$ computes $(\sigma, \tau) \leftarrow \Pi.\operatorname{Gen}(1^\lambda, f')$ and $d \leftarrow \Pi.\operatorname{Verify}(\tau, (p, c_1, \ldots, c_\ell, c), \pi)$
if $d = \mathrm{Rej}$ then
    $\mathcal{I}$ sets $m \leftarrow \perp$
else
    $\mathcal{I}$ computes $m \leftarrow f(m_1, \ldots, m_\ell)$
if $b = 1$ then
    $\mathcal{I}$ sends $m$ to $\mathcal{A}$
    $\mathcal{A}$ outputs $m$
$\mathcal{I}$ outputs $m$
```

encrypted result to the client who can then decrypt and use the result. However, clients in this scenario can suffer from attacks if the server deviates from the prescribed protocol. Hence we show how our prototype can be used to strengthen security in this use case by generating a proof of correct inference for a rudimentary feedforward neural network.

Weighted sum The weighted sum operation is at the center of contemporary machine learning models. Therefore, the prototype is modified to compute a weighted sum of the form $\sum_{i=1}^{n} w_i \cdot x_i$ where the weights w_i are in cleartext and the user inputs x_i are ciphertexts. The prover and verifier algorithms must now take in a vector of ciphertexts $\mathbf{c} = (c_1, \ldots, c_n)$ as input. The weights are assumed to be constants known in advance[6] and built in the arithmetic circuit. This assumption is realistic because model weights are fixed after training and do not change from one inference to another. Now remember from Section 2.2.2 that the blind rotation step circuit takes in as input a single element a of the ciphertext being bootstrapped and uses this element to compute two outputs: 1) the output of the MUX operation, which involves switching the modulus of a and performing a negacyclic rotation, and 2) a hash chain over these ciphertext elements. These two functionalities must be amended as follows for the prover to be able to generate proofs of correct weighted sum computation.

For the first part, the step circuit is modified to receive as input a vector of ciphertext elements $\mathbf{a} = (a_{c_1}, \ldots, a_{c_n})$ instead of a single ciphertext element. At blind rotation step j, this vector is comprised of all j-th elements of the ciphertexts in $\mathbf{c}$. Before the modulus switch step, the circuit now multiplies each element a_{c_i} with its respective weight w_i before summing them together, resulting in a combined element $\hat{a} = \sum_{i=1}^{n} w_i \cdot a_{c_i}$. Note that given two LWE ciphertexts $c = (a_1, \ldots, a_n, b)$, $c' = (a'_1, \ldots, a'_n, b')$ encrypting m and m' respectively and a scalar $w \in \mathbb{Z}_p$, we have that $w \cdot c = (w \cdot a_1, \ldots, w \cdot a_n, w \cdot b)$ and $c + c' = (a_1 + a'_1, \ldots, a_n + a'_n, b + b')$ which decrypt to $w \cdot m$ and $m + m'$ respectively provided ciphertext noise keeps small. As such, the element $\hat{a}$ computed by the circuit is equivalent to the j-th element of the ciphertext $\hat{c} = \sum_{i=1}^{n} w_i \cdot c_i$. This ciphertext element $\hat{a}$ is then used as is for the modulus switch procedure and the negacyclic rotation, meaning that the step circuit is now effectively performing a step of the blind rotation for the ciphertext $\hat{c}$. Therefore, the ciphertext output by the circuit is the result of a bootstrapped weighted sum of ciphertexts.

Second, the circuit now computes one hash chain per input ciphertext. This implies that the input and

[6] Specifically, the weights are agreed upon by the prover and verifier during the transparent setup phase of the SNARK protocol.

output of the step circuit increase in size linearly with the number of input ciphertexts n, but has little impact on the prover and verifier complexity and does not affect the security guarantees discussed in Section 3.2.1.

Activation function The original prototype takes in as input an LWE ciphertext c encrypting m and encodes the identity function $\mathsf{id}(m) = m$ in the test polynomial used in the bootstrapping. As such, both the input and output ciphertexts of the bootstrapping encrypt the same message and the operation only serves to reduce the noise. However, as mentionned in Section 2.2.2, it is possible to exploit the structure of the bootstrapping to evaluate a univariate function f on m by encoding f into the test polynomial. For example, the test polynomial can be set to encode the commonly used rectified linear unit (ReLU) activation function in order to obtain an encryption of $\mathsf{ReLU}(m)$ "for free" as a result of bootstrapping c. By combining this with the weighted sum modification, one can now generate a proof of correct execution of a rudimentary feedforward neural network of the form $\mathsf{ReLU}(\sum_{i=1}^{n} w_i \cdot x_i)$ where the x_i are encrypted and potentially sensitive user inputs.

With 4 inputs, this combination only marginally increases the circuit size as well as prover and verifier complexity. We leave as future work finding better optimized TFHE parameters that would allow for more extensive testing of this application. This amendment to our prototype demonstrates how it can be used to construct verifiable private machine learning inference and thus strengthen security guarantees in the typical two-party computation protocol proven secure above.

Ackowledgements

We would like to thank Samuel Tap for help with the parameter optimization and the RISC Zero team for assistance with their zkVM.

References

[ACGS23] Diego F. Aranha, Anamaria Costache, Antonio Guimarães, and Eduardo Soria-Vazquez. HELIOPOLIS: verifiable computation over homomorphically encrypted data from interactive oracle proofs is practical. *IACR Cryptol. ePrint Arch.*, page 1949, 2023.

[AGHV22] Adi Akavia, Craig Gentry, Shai Halevi, and Margarita Vald. Achievable CCA2 relaxation for homomorphic encryption. In Eike Kiltz and Vinod Vaikuntanathan, editors, *TCC 2022, Part II*, volume 13748 of *LNCS*, pages 70–99. Springer, Cham, November 2022. doi:10.1007/978-3-031-22365-5_3.

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Math. Cryptol.*, 9(3):169–203, 2015.

[Azt23] Aztec. The noir programming language. https://noir-lang.org/, 2023. Accessed: 2024-03-01.

[BBB$^+$22] Loris Bergerat, Anas Boudi, Quentin Bourgerie, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Parameter optimization & larger precision for (T)FHE. Cryptology ePrint Archive, Report 2022/704, 2022. URL: https://eprint.iacr.org/2022/704.

[BBB$^+$23] Loris Bergerat, Anas Boudi, Quentin Bourgerie, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Parameter optimization and larger precision for (T)FHE. *Journal of Cryptology*, 36(3):28, July 2023. doi:10.1007/s00145-023-09463-5.

[BBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *ICALP 2018*, volume 107 of *LIPIcs*, pages 14:1–14:17. Schloss Dagstuhl, July 2018. doi:10.4230/LIPIcs.ICALP.2018.14.

[BC23] Benedikt Bünz and Binyi Chen. Protostar: Generic efficient accumulation/folding for special sound protocols. Cryptology ePrint Archive, Paper 2023/620, 2023. https://eprint.iacr.org/2023/620. URL: https://eprint.iacr.org/2023/620.

[BC24] Dan Boneh and Binyi Chen. Latticefold: A lattice-based folding scheme and its applications to succinct proof systems. Cryptology ePrint Archive, Paper 2024/257, 2024. https://eprint.iacr.org/2024/257. URL: https://eprint.iacr.org/2024/257.

[BCFK21] Alexandre Bois, Ignacio Cascudo, Dario Fiore, and Dongwoo Kim. Flexible and efficient verifiable computation on encrypted data. In Juan Garay, editor, *PKC 2021, Part II*, volume 12711 of *LNCS*, pages 528–558. Springer, Cham, May 2021. doi:10.1007/978-3-030-75248-4_19.

[BCL$^+$23] Loris Bergerat, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, Adeline Roux-Langlois, and Samuel Tap. Faster secret keys for (t)fhe. Cryptology ePrint Archive, Paper 2023/979, 2023. https://eprint.iacr.org/2023/979. URL: https://eprint.iacr.org/2023/979.

[BGH19] Sean Bowe, Jack Grigg, and Daira Hopwood. Halo: Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Report 2019/1021, 2019. URL: https://eprint.iacr.org/2019/1021.

[BGZ23] Jeremy Bruestle, Paul Gafni, and RISC Zero. Risc zero zkvm: Scalable, transparent arguments of risc-v integrity. https://dev.risczero.com/proof-system-in-detail.pdf, 2023. Accessed: 2024-02-29.

[BIM$^+$23] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina Melé. Circom: A circuit description language for building zero-knowledge applications. *IEEE Trans. Dependable Secur. Comput.*, 20(6):4733–4751, 2023. doi:10.1109/TDSC.2022.3232813.

[BISW17] Dan Boneh, Yuval Ishai, Amit Sahai, and David J. Wu. Lattice-based SNARGs and their application to more efficient obfuscation. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part III*, volume 10212 of *LNCS*, pages 247–277. Springer, Cham, April / May 2017. doi:10.1007/978-3-319-56617-7_9.

[BSBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Paper 2018/046, 2018. https://eprint.iacr.org/2018/046. URL: https://eprint.iacr.org/2018/046.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001. doi:10.1109/SFCS.2001.959888.

[CFP$^+$24] Sébastien Canard, Caroline Fontaine, Duong Hieu Phan, David Pointcheval, Marc Renard, and Renaud Sirdey. Relations among new CCA security notions for approximate FHE. Cryptology ePrint Archive, Report 2024/812, 2024. URL: https://eprint.iacr.org/2024/812.

[CGG16] Ilaria Chillotti, Nicolas Gama, and Louis Goubin. Attacking FHE-based applications by software fault injections. Cryptology ePrint Archive, Report 2016/1164, 2016. URL: https://eprint.iacr.org/2016/1164.

[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, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 3–33. Springer, Berlin, Heidelberg, December 2016. 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.

[Chi14] Alessandro Chiesa. *Succinct non-Interactive arguments.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2014.

[CKP+23] Sylvain Chatel, Christian Knabenhans, Apostolos Pyrgelis, Carmela Troncoso, and Jean-Pierre Hubaux. Poster: Verifiable encodings for maliciously-secure homomorphic encryption evaluation. In *CCS*, pages 3525–3527. ACM, 2023.

[CKPH22] Sylvain Chatel, Christian Knabenhans, Apostolos Pyrgelis, and Jean-Pierre Hubaux. Verifiable encodings for secure homomorphic analytics. *CoRR*, abs/2207.14071, 2022.

[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, *ASIACRYPT 2021, Part III*, volume 13092 of *LNCS*, pages 670–699. Springer, Cham, December 2021. doi:10.1007/978-3-030-92078-4_23.

[DDD+23] Morten Dahl, Clément Danjou, Daniel Demmler, Tore Frederiksen, Petar Ivanov, Marc Joye, Dragos Rotaru, Nigel Smart, and Louis Tremblay Thibault. fhEVM: Confidential EVM Smart Contracts using Fully Homomorphic Encryption. https://github.com/zama-ai/fhevm/blob/main/fhevm-whitepaper.pdf, 2023. Accessed: 2023-11-22.

[FGP14] Dario Fiore, Rosario Gennaro, and Valerio Pastro. Efficiently verifiable computation on encrypted data. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 844–855. ACM Press, November 2014. doi:10.1145/2660267.2660366.

[FNP20] Dario Fiore, Anca Nitulescu, and David Pointcheval. Boosting verifiable computation on encrypted data. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020, Part II*, volume 12111 of *LNCS*, pages 124–154. Springer, Cham, May 2020. doi:10.1007/978-3-030-45388-6_5.

[GGP10] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 465–482. Springer, Berlin, Heidelberg, August 2010. doi:10.1007/978-3-642-14623-7_25.

[GGW23] Sanjam Garg, Aarushi Goel, and Mingyuan Wang. How to prove statements obliviously? *IACR Cryptol. ePrint Arch.*, page 1609, 2023.

[GKP+13] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. How to run Turing machines on encrypted data. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 536–553. Springer, Berlin, Heidelberg, August 2013. doi:10.1007/978-3-642-40084-1_30.

[GNS23] Chaya Ganesh, Anca Nitulescu, and Eduardo Soria-Vazquez. Rinocchio: SNARKs for ring arithmetic. *Journal of Cryptology*, 36(4):41, October 2023. doi:10.1007/s00145-023-09481-3.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 75–92. Springer, Berlin, Heidelberg, August 2013. doi:10.1007/978-3-642-40041-4_5.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. URL: https://eprint.iacr.org/2019/953.

[Joy22] Marc Joye. SoK: Fully homomorphic encryption over the [discretized] torus. *IACR TCHES*, 2022(4):661–692, 2022. doi:10.46586/tches.v2022.i4.661-692.

[KST22] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 359–388. Springer, Cham, August 2022. doi:10.1007/978-3-031-15985-5_13.

[LKL$^+$22] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, and Jong-Seon No. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *IEEE Access*, 10:30039–30054, 2022. doi:10.1109/ACCESS.2022.3159694.

[MN24] Mark Manulis and Jérôme Nguyen. Fully homomorphic encryption beyond IND-CCA1 security: Integrity through verifiability. In Marc Joye and Gregor Leander, editors, *EUROCRYPT 2024, Part II*, volume 14652 of *LNCS*, pages 63–93. Springer, Cham, May 2024. doi:10.1007/978-3-031-58723-8_3.

[Pol22] Polygon. Plonky2. https://github.com/mir-protocol/plonky2, 2022. Accessed: 2023-11-22.

[Sma23] 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 *LNCS*, pages 263–283. Springer, Cham, December 2023. doi:10.1007/978-3-031-47818-5_14.

[Sta20] Starknet. The cairo programming language. https://www.cairo-lang.org/, 2020. Accessed: 2024-03-01.

[Suc24] Succinct. Sp1. https://github.com/succinctlabs/sp1/, 2024. Accessed: 2024-02-29.

[Tea23] The Fhenix Team. Fhe-rollups: Scaling confidential smart contracts on ethereum and beyond. https://www.fhenix.io/wp-content/uploads/2023/11/FHE_Rollups_Whitepaper-v0.1-1.pdf, 2023. Accessed: 2023-11-22.

[Tha22] Justin Thaler. Proofs, arguments, and zero-knowledge. *Found. Trends Priv. Secur.*, 4(2-4):117–660, 2022.

[Val08] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In Ran Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 1–18. Springer, Berlin, Heidelberg, March 2008. doi:10.1007/978-3-540-78524-8_1.

[VKH23] Alexander Viand, Christian Knabenhans, and Anwar Hithnawi. Verifiable fully homomorphic encryption, 2023. arXiv:2301.07041.

[Wal24] Michael Walter. What have SNARGs ever done for FHE? Cryptology ePrint Archive, Report 2024/1207, 2024. URL: https://eprint.iacr.org/2024/1207.

[Zam22a] Zama. Concrete ML: a privacy-preserving machine learning library using fully homomorphic encryption for data scientists, 2022. https://github.com/zama-ai/concrete-ml.

[Zam22b] Zama. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data, 2022. https://github.com/zama-ai/tfhe-rs.