Verifiable FHE via Lattice-based SNARKs

Shahla Atapoor[1], Karim Baghery[1], Hilder V. L. Pereira[2] and Jannik Spiessens[1]

[1] COSIC, KU Leuven, Leuven, Belgium
[2] Universidade de Campinas (UNICAMP), Instituto de Computação, Campinas, Brazil

Abstract. Fully Homomorphic Encryption (FHE) is a prevalent cryptographic primitive that allows for computation on encrypted data. In various cryptographic protocols, this enables outsourcing computation to a third party while retaining the privacy of the inputs to the computation. However, these schemes make an honest-but-curious assumption about the adversary. Previous work has tried to remove this assumption by combining FHE with Verifiable Computation (VC). Recent work has increased the flexibility of this approach by introducing integrity checks for homomorphic computations over rings. However, efficient FHE for circuits of large multiplicative depth also requires non-ring computations called maintenance operations, i.e. modswitching and keyswitching, which cannot be efficiently verified by existing constructions. We propose the first efficiently verifiable FHE scheme that allows for arbitrary depth homomorphic circuits by utilizing the double-CRT representation in which FHE schemes are typically computed, and using lattice-based SNARKs to prove components of this computation separately, including the maintenance operations. Therefore, our construction can theoretically handle bootstrapping operations. We also present the first implementation of a verifiable computation on encrypted data for a computation that contains multiple ciphertext-ciphertext multiplications. Concretely, we verify the homomorphic computation of an approximate neural network containing three layers and >100 ciphertexts in less than 1 second while maintaining reasonable prover costs.

Keywords: Fully-Homomorphic Encryption · Verifiable FHE · Lattice-based SNARKs · Computation on Encrypted Data

1 Introduction

Fully Homomorphic Encryption (FHE) schemes can be used to add privacy-preserving properties to cloud applications by encrypting the client's inputs such that the server can still (homomorphically) perform computations on them, resulting in encrypted outputs that are sent back to the client. Examples of such applications are oblivious RAM (ORAM) [PCDN23], privacy-preserving machine learning [BGGJ19] and more recently confidential smart contracts in general-purpose blockchains [ZAM23]. Normally, FHE schemes can only be used in settings where the server is assumed to be honest-but-curious, meaning the server is trusted to perform the homomorphic computations correctly but not trusted with access to the confidential plaintext values on which the computation is performed. This trust assumption can be removed by adding verifiablitity to existing FHE schemes and thereby constructing verifiable FHE (vFHE) [VKH23]. By adding integrity to the FHE primitive, vFHE could be used to maintain confidentiality of FHE against active adversaries performing e.g. key-recovery attacks [CT15, CGG16]. More generally, vFHE

E-mail: shahla.atapoor@esat.kuleuven.be (Shahla Atapoor), karim.baghery@esat.kuleuven.be (Karim Baghery), hilder@unicamp.br (Hilder V. L. Pereira), jannik.spiessens@esat.kuleuven.be (Jannik Spiessens)

enables verifiable computation on encrypted data, aka Private Verifiable Computation (PVC) [FGP14], in which a client can outsource computation to a server in a verifiable way while preserving the privacy of its inputs and outputs.

A common approach for constructing vFHE is to combine an FHE scheme with a Verifiable Computation (VC) scheme which is used to prove the correctness of the homomorphic computations. However, combining these two primitives in an efficient way turns out to be a highly non-trivial task. Namely, VC can usually prove arithmetic circuits whose gates are additions or multiplications over some field $\mathbb{F}_p$, while the homomorphic computation that the server wants to prove is performed over polynomial rings. Simply representing the computation as circuits over $\mathbb{F}_p$ introduces many significant overheads on the size of the proofs or on the size of CRS (common reference strings), and also on the running time of the prover and of the verifier.

To overcome this, recent works [GNS23, BCFK21] have studied how to modify the VC protocols to work over rings, in an attempt to have proofs that match the type of computation done by the server and do not require representing operations over rings with gates over $\mathbb{F}_p$. We propose a fundamentally different approach, namely, to exploit the well-known decomposition of the polynomial rings used in FHE as direct product of fields[1] which evokes the use of a lattice-based SNARK over fields to generate the proofs.

1.1 Our Contributions

We propose the first verifiable fully homomorphic encryption scheme combining FHE and SNARKs (succinct non-interactive argument of knowledge) in a non-trivial way. Our approach is modular, meaning that it is possible to construct blocks of verifiable homomorphic circuits that can be assembled together to build larger circuits. Our construction is the first one to handle real homomorphic computation, including the fundamental maintenance operations known as modulus switching and key-switching (aka relinearization). In addition, this implies that we can handle bootstrapping and achieve fully homomorphic encryption. Moreover, we provide a public C++ implementation of our construction and ran experiments that can serve as baselines for future works when it comes to practical results.

1.2 Exploiting double-CRT to make FHE more VC-friendly

Most FHE schemes work over cyclotomic rings $\mathcal{R} = \mathbb{Z}[X]/\langle X^N + 1\rangle$, where N is a power of two. In particular, ciphertexts are composed by elements of $\mathcal{R}_Q := \mathcal{R}/Q\mathcal{R}$, where Q is a large integer. Thus, when a server performs computation on encrypted data, it operates on elements from $\mathcal{R}_Q$, i.e., polynomials modulo $X^N + 1$ and Q. At first glance, this type of computation is not easily represented by circuits over $\mathbb{F}_p$, which is the set VC typically handles.

However, FHE schemes are commonly implemented using a double-CRT representation, which works by choosing Q as a product of a few small primes $q_0, ..., q_L$, then using the isomorphism

$$\mathcal{R}_Q = \frac{\mathbb{Z}_{q_0}[X]}{\langle X^N + 1\rangle} \times ... \times \frac{\mathbb{Z}_{q_L}[X]}{\langle X^N + 1\rangle}$$

to represent operations on $\mathcal{R}_Q$ as independent operations on each $\mathcal{R}_{q_i}$. Since for each prime q_i, it holds that $\mathbb{Z}_{q_i}$ is a field, this gives us a hint that it could be possible to instantiate different VC instances, defined over fields $\mathbb{F}_{q_0}, ..., \mathbb{F}_{q_L}$ and then have $L + 1$ proofs to prove the actual computation over $\mathcal{R}_Q$.

[1]This is often called residual number representation (RNS) or double-CRT representation.

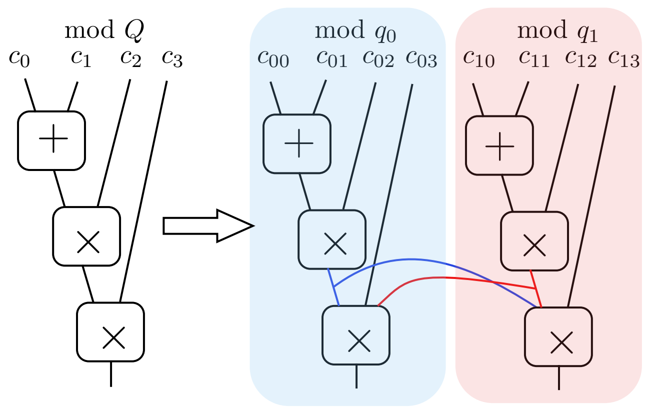
mod Q
c_0 c_1 c_2 c_3
mod q_0
c_{00} c_{01} c_{02} c_{03}
mod q_1
c_{10} c_{11} c_{12} c_{13}

Figure 1: Homomorphic computation of $(c_0 + c_1) \cdot c_2 \cdot c_3$ modulo $Q = q_0 \cdot q_1$. Every gate represents a homomorphic addition or multiplication, which are composed by many operations over $\mathcal{R}_Q$. When we represent the circuit as two circuits with low-level operations defined modulo q_0 and q_1, and inputs $c_{i,j} = c_j \bmod q_i$, the output of the first multiplication gate is used as input of the following gates in all the circuits.

However, between the homomorphic operations, one needs to execute two "maintenance operations" that are not defined in terms of additions and multiplications on $\mathcal{R}_Q$, and thus do not respect the above isomorphism. These operations are key switching, which is used to guarantee that ciphertexts have a valid format during the whole computation, and modulus switching, which controls the noise growth. Concretely, as discussed in more detail in Section 2.6, both operations require non-arithmetic modular reductions. Moreover, since the isomorphism does not hold, the computations modulo q_i become dependent of values modulo q_j for $j \neq i$. This means that instead of having $L + 1$ independent circuits defined modulo different primes q_i's, which could be proved independently, we actually have $L + 1$ circuits that are interconnected, with intermediary wires being shared among them. This is illustrated in Figure 1 and discussed in more detail in Sections 2.4 and 2.5.

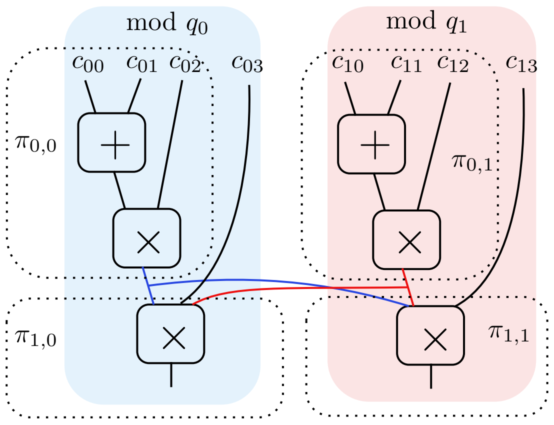
mod q_0
mod q_1
c_{00} c_{01} c_{02} c_{03}
c_{10} c_{11} c_{12} c_{13}
$\pi_{0,0}$
$\pi_{0,1}$
$\pi_{1,0}$
$\pi_{1,1}$

Figure 2: Two circuits representing the homomorphic computation of $(c_0 + c_1) \cdot c_2 \cdot c_3$ modulo $Q = q_0 \cdot q_1$. We divide the circuits in two layers, the first one is composed by the proofs $\pi_{0,0}$ and $\pi_{0,1}$, and the second layer corresponds to the proofs $\pi_{1,0}$ and $\pi_{1,1}$. Proof $\pi_{1,0}$ considers as input the value $c_{0,3}$ and the two outputs of the first layer. And similarly for $\pi_{1,1}$ and $c_{1,3}$.

Because of the non-arithmetic operations and the wires shared among the circuits, we cannot simply have one proof for each prime q_i. Thus, we subdivide each circuit into layers, such that the input wires can come from any circuit, and the output wires can be connected to any other circuit, but the internal wires are connected only to gates with respect to the same q_i. As such, we have "boxes" defined entirely modulo one single prime and we can finally have a proof for them, which gives us a proof for the original homomorphic computation modulo Q as a concatenation of the proofs of these small subcircuits. This is illustrated in Figure 2. Notice that simply breaking a proof with respect to $Q = \prod_{i=0}^{L} q_i$ into proofs with respect to q_i's does not increase the proof size, since each proof now is smaller (containing elements modulo q_i). However, adding k layers multiplies the proof size by $O(k)$. On the other hand, the prover's running time is basically the same and the CRS can even become smaller. Moreover, if a non-interactive VC protocol is used, then our solution remains non-interactive, since the prover can generate all $\pi_{i,j}$'s and only then send them to the client for verification.

1.3 Optimizations and efficiency

By looking at the homomorphic operations more closely, we see that there are different ways of grouping them or changing the order they are executed in, such that we add no or very little overhead to the prover and reduce the proof size and the verifier's running time. First of all, all the operations between plaintexts and ciphertexts, and also the homomorphic additions can be grouped in single proofs, since they do not require key switching or modulus switching, and these are the only two operations that mix the wires. Also, the homomorphic multiplication is usually composed by a tensor product, then a key switching, then a modulus switching. Thus, at first glance, a block of operations finishing with a ciphertext-ciphertext multiplication would require 3 layers of proofs, however, switching the modulus from Q to $Q' := Q/q_L$ means that the following computation is executed modulo Q', thus there is no subsequent computation modulo q_L. As a result, we can actually finish the proof with respect to q_L, then pass its output as inputs to the other proofs and save one layer. This is shown in Figure 3. Also, each layer has one less column than the previous layer, which almost halves the proof size.

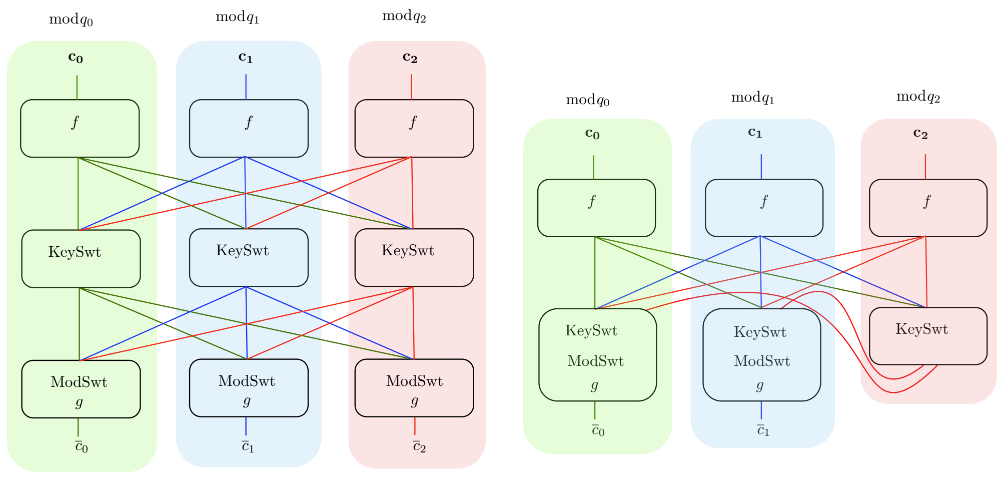

Figure 3: Proof for homomorphic computation of the composition $g \circ f$, where f ends with a multiplication. On the right, we show how we can merge two layers by ending the proof corresponding to the last prime.

One problem that hinders the practical efficiency of our construction, is that to achieve soundness, state-of-the-art efficient VC schemes [Gro16, GWC19] need to prove computations over fields $\mathbb{F}_q$ where q is around 256 bits, while the q_i's typically used in FHE schemes are around 30 bits. One could emulate the smaller moduli in the bigger field $\mathbb{F}_q$, but this inevitably blows up the number of gates in the arithmetic circuit that the VC scheme verifies. We observe that recently proposed lattice-based approaches [GMNO18, ISW21] can achieve similar security while being flexible in terms of field choice. Note that the use of lattice-based constructions also comes with the added benefit of maintaining plausible post-quantum security.

In Section 3.4, we study the efficiency of our scheme when instantiated to verify building blocks such as ciphertext additions, plaintext-ciphertext multiplication, matrix-vector multiplication, and higher depth computations such as ciphertext-ciphertext multiplication, slot rotations and more general high-depth functions composed of these building blocks.

1.4 Implementation and practical results

We present the first implementation of a verifiable FHE construction that can be efficiently instantiated for fully homomorphic circuits i.e. with a possible multiplicative depth greater than one. We instantiate it for a homomorphic circuit representing a 3-layered neural network and implement it in C++ to show the practicality of our scheme. The only other vFHE implementation known to us [VKH23] proves the correct computation of a single ciphertext-ciphertext multiplication (without the required maintenance operations) in 443s while our implementation needs only 167s to prove a homomorphic computation on >100 ciphertexts that includes the maintenance operations required to compute higher depth computations. Verification times vary from 0.6s to 0.9s depending on the size of the input layer.

2 Preliminaries

2.1 Notations

We denote the security parameter as λ. The notation $y \leftarrow A(x)$ signifies the execution of a probabilistic polynomial-time (PPT) algorithm A which outputs y given the input x. The symbol $\mathbb{F}$ is used to denote a finite field, while $\mathcal{R}$ denotes a ring. We denote by $\mathsf{negl}(\lambda)$ an arbitrary negligible function in λ. Within the paper, square brackets are used to indicate a range $[n] = \{1, \ldots, n\}$, and also to represent the central remainder modulo q as $[n]_q$. Bold lowercase letters are used to denote vectors and bold uppercase for matrices.

2.2 Rank-1 Constraint System (R1CS)

An R1CS instance is a collection of constraints on a vector of values $\boldsymbol{c} \in \mathbb{F}^{N_w}$ called the wire values, where N_w is the number of wire values and $\mathbb{F}$ is a finite field. The first n values of this wire vector are called the *statement* $\boldsymbol{x} \in \mathbb{F}^n$. The last $N_w - n$ values are called the *witness* $\boldsymbol{w} \in \mathbb{F}^{N_w - n}$. There are N_g constraints which are also referred to as gates. An R1CS instance $\mathcal{CS}$ can be defined as a tuple $\mathcal{CS} = \left(N_g, n, N_w, \{\boldsymbol{a}_i, \boldsymbol{b}_i, \boldsymbol{c}_i\}_{i \in [N_g]}\right)$ for $N_g, n, N_w \in \mathbb{N}$ (with $n \leq N_w$) and $\boldsymbol{a}_i, \boldsymbol{b}_i, \boldsymbol{c}_i \in \mathbb{F}^{N_w+1}$ for all $i \in [N_g]$. Define a function $\mathcal{CS} : \mathbb{F}^n \times \mathbb{F}^{N_w - n} \to \{0, 1\}$ for some constraint system $\mathcal{CS}$. This function has the property that for some statement $\boldsymbol{x} \in \mathbb{F}^n$ and witness $\boldsymbol{w} \in \mathbb{F}^{N_w - n}$, $\mathcal{CS}(\boldsymbol{x}, \boldsymbol{w}) = 1$ if and only if $\boldsymbol{z}\boldsymbol{a}_i \cdot \boldsymbol{z}\boldsymbol{b}_i = \boldsymbol{z}\boldsymbol{c}_i$ for $\boldsymbol{z} = [1 \ \boldsymbol{x}^\top \boldsymbol{w}^\top]$ and $i \in [N_g]$. We call the constraint system *satisfiable* for some statement $\boldsymbol{x}$ iff there exists some $\boldsymbol{w}$ such that $\mathcal{CS}(\boldsymbol{x}, \boldsymbol{w}) = 1$. The set of all satisfiable wire values $(\boldsymbol{x}, \boldsymbol{w})$ is a relation called $\mathcal{R}_{\mathcal{CS}}$. We can define the corresponding language as $\mathcal{L}_{\mathcal{CS}} = \{\boldsymbol{x} \mid \exists \boldsymbol{w} \text{ s.t. } \mathcal{CS}(\boldsymbol{x}, \boldsymbol{w}) = 1\}$.

2.3 Succinct Non-interactive ARguments of Knowledge (SNARKs)

We define Succinct Non-interactive ARgument of Knowledge (SNARK) schemes in the preproccesing model with a designated verifier. A SNARK consists of the following three probabilistic polynomial time (PPT) algorithms:

- $\mathsf{Setup}(1^\lambda, \mathcal{CS}) \to (\mathsf{crs}, \mathsf{st})$: given the security parameter λ and the constraint system $\mathcal{CS}$, it generates a common reference string crs and a verification state st.
- $\mathsf{Prover}(\mathsf{crs}, \boldsymbol{x}, \boldsymbol{w}) \to \pi$: given a common reference string crs, a statement $\boldsymbol{x}$ and a witness $\boldsymbol{w}$, it generates a proof π.
- $\mathsf{Verifier}(\mathsf{st}, \boldsymbol{x}, \pi) \to b$: given a verification state st, a statement $\boldsymbol{x}$ and a proof π, it generates a verification bit $b \in \{0, 1\}$.

These algorithms must satisfy the completeness and knowledge soundness properties.

Completeness. A SNARK scheme is complete iff for any security parameter λ, statement $\boldsymbol{x}$ and witness $\boldsymbol{w}$, and R1CS instance $\mathcal{CS}$,

$$\Pr\left[\ \mathsf{Verifier}(\mathsf{st}, \boldsymbol{x}, \pi) = 1 \ \middle|\ \begin{array}{c} \mathcal{CS}(\boldsymbol{x}, \boldsymbol{w}) = 1 \\ (\mathsf{crs}, \mathsf{st}) \leftarrow \mathsf{Setup}(1^\lambda, \mathcal{CS}) \\ \pi \leftarrow \mathsf{Prover}(\mathsf{crs}, \boldsymbol{x}, \boldsymbol{w}) \end{array} \right] = 1.$$

Knowledge Soundness. A SNARK scheme satisfies knowledge soundness iff for any PPT algorithm Prover^*, there exists a PPT extractor Extr such that for any security parameter λ, R1CS instance $\mathcal{CS}$ and state z,

$$\Pr\left[\begin{array}{c} \mathsf{Verifier}(\mathsf{st}, \boldsymbol{x}, \pi^*) = 1 \\ \wedge \\ \mathcal{CS}(\boldsymbol{x}, \boldsymbol{w}^*) \neq 1 \end{array} \ \middle|\ \begin{array}{c} (\mathsf{crs}, \mathsf{st}) \leftarrow \mathsf{Setup}(1^\lambda, \mathcal{CS}) \\ (\pi^*, \boldsymbol{x}) \leftarrow \mathsf{Prover}^*(\mathsf{crs}; z) \\ \boldsymbol{w}^* \leftarrow \mathtt{Extr}(\mathsf{crs}; z) \end{array} \right] \leq \mathsf{negl}(\lambda).$$

SNARKs schemes are also required to be succinct. Concretely, this requires that the proof size can be expressed as $poly(\lambda + \log|\mathcal{CS}|)$ and the $\mathsf{Verifier}$ algorithm runs in time $poly(\lambda + |\boldsymbol{x}| + \log|\mathcal{CS}|)$.

2.3.1 Latices-based SNARKs.

Traditionally, SNARKs defined over $\mathbb{F}_q$ rely on the large size of this field to guarantee security and soundness, which means that q typically has around 256 bits. Lattice-based SNARKs, on the other hand, base their security on hardness assumptions such as the learning with errors problem (LWE), which allow them to be instantiated with smaller fields, i.e., $\mathbb{F}_q$ with small q. Thus, they are a perfect tool for our construction, because we can use small values of q to match the small primes used in FHE schemes. As an additional benefit of lattice-based SNARKs, they are post-quantum, just like the FHE schemes are. Thus, combining them with FHE gives us constructions that are still post-quantum secure.

In our analysis, we assume that the verification runs in time $O(\lambda + |x|)$ and each proof is composed by a constant number of ring elements, which together have size $O(\lambda)$. Those assumptions are true for current constructions of lattice-based SNARKs[GMNO18, ISW21]. We refer to the appendix B for more details on the construction of lattice-based SNARKs.

2.4 Fully Homomorphic Encryption (FHE)

FHE schemes are encryption schemes with the property that arbitrary circuits C can be evaluated homomorphically in the ciphertext space. To make the presentation more concrete, we consider in this section the BGV scheme [BGV12], but notice that other schemes, like FV [FV12] and CKKS [CKKS17], are very similar. We give a high-level description of the construction, that suffices for our purposes. Moreover, to simplify the presentation, we just present a symmetric-key version of BGV. Transforming it into a asymmetric scheme is done via simple standard techniques.

Let $\mathcal{R} = \mathbb{Z}[X]/\langle X^N + 1\rangle$, where $N = 2^k$ for some $k \in \mathbb{N}$. Define a modulus $Q = \prod_{i=0}^{L} q_i$ where each q_i is a different prime, and $\mathcal{R}_Q = \mathbb{Z}_Q[X]/\langle X^N + 1\rangle$. The ciphertexts are defined as vectors over $\mathcal{R}_Q$, but the homomorphic operations are easier to understand if the ciphertexts are considered polynomials in $\mathcal{R}_Q[Y]$, where Y is a new variable. That is, we can view a ciphertext $\mathbf{c} \in \mathcal{R}_Q^u$ as $c(Y) = \sum_{i=0}^{u-1} c_i \cdot Y^i$. Fix a plaintext modulus t and an error distribution χ_{err} over $\mathcal{R}$ that samples coefficients according to a discrete Gaussian distribution with standard deviation σ_{err}. Then, we say that $c(Y)$ decrypts to a message $m \in \mathcal{R}_t$ if when we evaluate $c(Y)$ on the secret key, we get the message plus some small noise term, i.e., $c(\mathsf{sk}) \bmod Q = te + m$ for $e \in \mathcal{R}$. Notice that in this case, $(c(\mathsf{sk}) \bmod Q) \bmod t = m$.

2.4.1 Generic construction.

A homomorphic encryption scheme HE requires the following functions for parameter generation, secret key generation, encryption and decryption. For a basic (symmetric-key) version of the BGV scheme, they can be constructed as follows.

- $\mathsf{HE.ParamGen}(1^\lambda, L)$: given the security parameter λ and a multiplicative depth L, choose N, $Q = \prod_{i=0}^{L} q_i$ and $\sigma \in \mathbb{R}$ such that the (N, Q, σ)-RLWE problem achieves λ bits of security and the FHE scheme based on it can accommodate homomorphic circuits of depth L. Let $\mathcal{R} := \mathbb{Z}[X]\langle X^N + 1\rangle$, $\mathcal{R}_Q := \mathcal{R}/Q\mathcal{R}$ and $\mathcal{R}_t := \mathcal{R}/t\mathcal{R}$ for some plaintext modulus t. The message and the ciphertext spaces are $\mathcal{R}_t$ and $\mathcal{R}_Q[Y]$, respectively. Set $\mathsf{params} := (N, Q, \sigma, t)$, which is a default input to the following algorithms.

- $\mathsf{HE.KeyGen}(1^\lambda)$: Given the security parameter λ, uses params to output some secret key s and $\texttt{rlk}$ which is the relinearization key with respect to s (see Section 2.6).

- $\mathsf{HE.Enc}_{\mathsf{sk}}(m)$: Consider $m \in \mathcal{R}_t$. Sample a uniformly at random from $\mathcal{R}_Q$, and $e \leftarrow \chi_{\mathsf{err}}$. Compute $b := -a \cdot \mathsf{sk} + t \cdot e + m \in \mathcal{R}_Q$. Output $c(Y) := b + a \cdot Y$.

- $\mathsf{HE.Dec}_{\mathsf{sk}}(c)$: Compute $b^\star := [c(\mathsf{sk})]_Q$ over $\mathcal{R}_Q$. Output $b^\star \bmod t$. Notice that $b^\star \bmod t = m$ iff all coefficients of e remain smaller than $\lfloor Q/2t \rfloor$, i.e. the noise term's l_∞-norm remains below a certain bound.

A homomorphic encryption scheme also requires a function $\mathsf{HE.Eval}$ that evaluates an arithmetic circuit C on some input ciphertexts $c_i(Y) = \mathsf{HE.Enc}(m_i)$ and outputs ciphertext $c'(Y)$ such that $\mathsf{HE.Dec}(c') = C(\{m_i\})$. We present the functions called by the $\mathsf{HE.Eval}$ function to homomorphically compute basic operations on ciphertexts.

- $\mathsf{HE.Add}(c_0, c_1)$: Output $c_{add}(Y) = c_0(Y) + c_1(Y) = (b_0 + b_1) + (a_0 + a_1) \cdot Y$. It is easy to see that

$$c_{add}(\mathsf{sk}) \bmod t = [c_0(\mathsf{sk}) + c_1(\mathsf{sk})]_Q \bmod t = m_0 + m_1.$$

  Thus, $c(Y)$ is an encryption of the sum of the messages, as desired. Notice that the noise terms are also added together, and therefore homomorphic addition increases the noise additively.

- $\mathsf{HE.Mult}(c_0, c_1)$: Output $c_{mult}(Y) = c_0(Y) \cdot c_1(Y) = b_0 \cdot b_1 + (a_0 \cdot b_1 + a_1 \cdot b_0) \cdot Y + a_0 \cdot a_1 \cdot Y^2$. We can see that

$$c_{mult}(\mathsf{sk}) \bmod t = c_0(\mathsf{sk}) \cdot c_1(\mathsf{sk}) \bmod t = m_0 \cdot m_1,$$

  which is an encryption of $m_0 \cdot m_1$, as desired. Notice that ciphertext multiplication leads to quadratic noise growth since the noise terms are multiplied.

  Furthermore, the degree of c_{mult} in Y is larger than the degree of both input ciphertexts, and thus requires more elements of $\mathcal{R}_Q$ to store it. In other words, this operation increases the size of the ciphertexts.

- $\mathsf{HE.MultPtxt}(c_0, m_1)$: Output $c_{multPtxt}(Y) = \sum_{i=0}^{k} m_1 \cdot c_{0,i} \cdot Y^i$ for input ciphertext $c_0(Y)$ of degree k. Notice that $c_{multPtxt}$ has the same degree as c_0 in Y, so the size of the output ciphertext remains constant. Also, the noise term of c_0 is only multiplied by m_0, and therefore the noise growth is small compared to ciphertext-ciphertext multiplication (at least when t is relatively small).

Notice that both the noise and ciphertext degree grow exponentially with the multiplicative depth of the homomorphic circuit being evaluated. In levelled FHE schemes, this is typically solved by performing maintenance operations after every ciphertext-ciphertext multiplication. More concretely, the ciphertext $c(Y) = c_0 + c_1 \cdot Y + c_2 \cdot Y^2 \in \mathcal{R}_Q$ is first relinearized using a relinearization key `rlk` resulting in $c'(Y) = c'_0 + c'_1 \cdot Y \in \mathcal{R}_Q$. This is followed by the modswitching operation which aims to remove the noise added by ciphertext-ciphertext multiplication (and relinearization). As the name implies, this is achieved by switching to a smaller modulus $Q^\star = Q/q_i$ such that $c''(Y) = c''_0 + c''_1 \cdot Y \in \mathcal{R}_{Q^\star}$, which essentially divides the noise by q_i. See Section 2.6 for a more detailed description.

2.5 Basics of RNS

All the homomorphic operations are composed of some operations over $\mathcal{R}_Q$, which boil down to adding and multiplying polynomials of degree less than N, then reducing them modulo $X^N + 1$, and reducing each coefficient modulo Q.

Because Q is typically large (say, with more than 1000 bits), to work directly with polynomials mod Q, we need to use libraries that implement arbitrary precision integers, which is inefficient. To overcome this, the residue number system (RNS), is typically used. It exploits the decomposition of $Q = \prod_{i=1}^{\ell} q_i$ to work with several polynomials modulo each q_i, which fit in the 32- or 64-bit native integer types of current processors.

In more detail, because $Q = \prod_{i=1}^{\ell} q_i$, by using the Chinese remainder theorem coefficient-wise, we have

$$\mathcal{R}_Q = \mathbb{Z}_Q[X]/\langle X^N + 1 \rangle = \prod_{i=1}^{\ell} \mathbb{Z}_{q_i}[X]/\langle X^N + 1 \rangle.$$

Thus, working with an element of $\mathcal{R}_Q$ is equivalent to working with a vector of size ℓ in $\prod_{i=1}^{\ell} \mathcal{R}_{q_i}$. Therefore, we could in principle verify the homomorphic computation over $\mathcal{R}_Q$ with ℓ independent proofs over each $\mathbb{Z}_{q_i}$. We will discuss the limitations of this method soon.

Since multiplying polynomials efficiently requires first performing a fast Fourier transform, or number-theoretic transform (NTT), it is common to go one step further and represent elements of $\mathcal{R}_Q$ in the “NTT form". Given $a \in \mathcal{R}_Q$, instead of simply storing its list of coefficients, we precompute the NTTs of a with respect to each q_i. For this, we choose each q_i as a prime congruent to 1 modulo $2N$. This guarantees that there is a primitive $2N$-th root of unity $\omega_i \in \mathbb{Z}_{q_i}$ and that the following is an isomorphism:

$$\mathsf{NTT}_i(a) = (a(\omega_i^0), a(\omega_i^1), a(\omega_i^3), ..., a(\omega_i^{2N-1})) \in \mathbb{Z}_{q_i}^{N+1}$$

Putting all together, we start with $a(X) \in \mathcal{R}_Q$, then we obtain a list of polynomials $(a_1(X), ..., a_\ell(X)) \in \mathcal{R}_{q_1} \times ... \times \mathcal{R}_{q_\ell}$, then we map each of them to a vector using the NTTs so, at the end, $a(X)$ is stored as a matrix

$$\mathtt{Mat}(a) := \begin{pmatrix} \mathsf{NTT}_1(a_1) \\ \vdots \\ \mathsf{NTT}_\ell(a_\ell) \end{pmatrix} = \begin{pmatrix} a_{1,0} & \dots & a_{1,N} \\ \vdots & \ddots & \vdots \\ a_{\ell,0} & \dots & a_{\ell,N} \end{pmatrix} \in \mathbb{Z}^{\ell \times (N+1)}.$$

By using a special type of NTT transform, called a negative-wrapped convolution (on which we will not elaborate here), we can avoid the polynomial reduction after multiplication [LMPR08, Zuc18]. Therefore, we can implement each addition and multiplication over $\mathcal{R}_Q$ with pointwise operations of the corresponding matrices. For example, $a \cdot b \in \mathcal{R}_Q$ is $\mathbf{C} := \mathtt{Mat}(a) \odot \mathtt{Mat}(b)$, that is, each entry (i, j) of $\mathbf{C}$ is $[a_{i,j} \cdot b_{i,j}]_{q_i}$

So computations over $\mathcal{R}_Q$ can instead be performed as vector computations over l different finite fields. We will refer to this as the double-CRT (dCRT) representation. In the following section, it will become clear that the FHE scheme presented in Section 2.4 is not practical when computed in dCRT representation entirely. Maintenance operations require inverting the NTT transform and then sharing elements between different rows in the matrix representation.

2.6 Maintenance operations

In Section 2.4, we explained that ciphertexts maintenance enables the scheme to manage arbitrary depth homomorphic circuits. Typically, one relinearizes after multiplication, followed by a modswitch to decrease the noise. We define the modulus at level i as $Q^{(i)} = \prod_{j=0}^{i} q_j$. Therefore, ciphertexts encrypted over $R_{Q^{(L)}}$ can manage homomorphic circuits of multiplicative depth L. It will become clear from a more detailed description below that the maintenance operations are not composed of additions and multiplications over $R_{Q^{(i)}}$. This implies that they can not be performed in dCRT representation. Therefore, one should first invert the NTT transform. However, one can avoid inverting the RNS decomposition by performing fast base extension `FastBaseExt`. To extend the decomposition of $c \in R_{\tilde{Q}}$ in base $Q^{(l)} = q_1 \cdot \ldots \cdot q_l$ to another base $\tilde{Q}^{(k)} = \tilde{q}_1 \cdot \ldots \cdot \tilde{q}_k$ that is coprime to $Q^{(l)}$, one computes

$$\mathtt{FastBaseExt}(c, Q^{(l)}, \tilde{Q}^{(k)}) := \left(\sum_{i=1}^{l} \left[c \cdot (Q^{(l)}/q_i)^{-1} \right]_{q_i} \cdot (Q^{(l)}/q_i) \mod \tilde{q}_j \right)_{j=1}^{k}$$

Notice that this does not exactly equal $[[c]_{Q^{(l)}}]_{\tilde{Q}^{(k)}}$, but it can be shown that using fast base extension in maintenance operations only adds negligible noise to the resulting ciphertexts.

2.6.1 Relinearization.

As discussed in Section 2.4, multiplying ciphertexts results in a degree 2 ciphertext $c(Y) = c_0 + c_1 \cdot Y + c_2 \cdot Y^2$ which should be relinearized to a ciphertext $c'(Y) = c'_0 + c'_1 \cdot Y$, that decrypts to the same plaintext. One approach would be to encrypt sk^2 as $\mathtt{rlk}(Y) = \mathtt{rlk}_0 + \mathtt{rlk}_1 \cdot Y$ and then compute $c'_j = c_j + c_2 \cdot \mathtt{rlk}_j$ for $j \in \{0, 1\}$. Notice however, that this would add a large noise term $c_2 \cdot e$, where e is the noise term of $\mathtt{rlk}(Y)$ and c_2 is an element modulo $Q^{(i)}$. Therefore, we instead use a decomposition of c_2 into its RNS base

$$\begin{aligned}\mathcal{D}_{Q^{(i)}}(c_2) = \Big(& \mathtt{FastBaseExt}([c_2]_{q_0}, q_0, Q^{(i)}), \\ & \ldots, \\ & \mathtt{FastBaseExt}([c_2]_{q_i}, q_i, Q^{(i)}) \Big) \in R_{Q^{(i)}}^{i+1},\end{aligned}$$

and define the relinearization key as a vector of ciphertexts $\texttt{rlk}_0, \texttt{rlk}_1 \in R_{Q^{(i)}}^{i+1}$, such that we can compute the relinearization as $c'_j = c_j + \langle \mathcal{D}_{Q^{(i)}}(c_2), \texttt{rlk}_j \rangle$ for $j \in \{0,1\}$. This ensures that noise terms of $\texttt{rlk}$ are only multiplied with smaller elements modulo q_j, since the elements of $\mathcal{D}_{Q^{(i)}}(c_2)$ were base extended from the base q_j to the base $Q^{(i)}$. Using a similar technique, one can construct from a ciphertext $c(Y)$ another ciphertext $c'(Y)$ such that $[c(\mathsf{sk})]_{Q^{(i)}} = [c'(\mathsf{sk}')]_{Q^{(i)}} \bmod t$, i.e. they decrypt to the same plaintext using a different secret key. This operation is referred to as key-switching.

2.6.2 Modulus Switching.

To decrease the noise in a ciphertext $c(Y) \in R_{Q^{(i)}}[Y]$ at level i, one switches the modulus of that ciphertext to $Q^{(i-1)} = Q^{(i)}/q_i$. This produces another ciphertext $c'(Y) \in R_{Q^{(i-1)}}[Y]$ such that $c(\mathsf{sk}) \mod Q^{(i)}$ and $c'(\mathsf{sk}) \mod Q^{(i-1)}$ are equivalent modulo t, i.e. they decrypt to the same plaintext. Given that the noise at $c(Y) = c_0 + c_1 Y$ satisfies a certain bound, the coefficients of c' can be calculated as

$$c'_l = \frac{1}{q_i}(c_l + \delta_l)$$

where $\delta_l = t(-c_l/t \bmod q_i)$ for $l \in \{0,1\}$. This operation can be performed in RNS decomposition by base extending $\delta \in R_{q_i}$ to the base $Q^{(i-1)}$.

For a more detailed description of these maintenance operations, as well as alternative methods, we refer to [Zuc18, KPZ21]. Importantly, notice that all computations required by the maintenance operations (and also the double-CRT vector operations) are easily representable by R1CS constraints.

2.7 Verifiable FHE (vFHE)

We present a definition for vFHE schemes adjusted from Viand et al. [VKH23]. This definition simply extends the definition of an FHE scheme by introducing a Verify algorithm that verifies the ciphertext c_y and proof π output by the Eval algorithm for a certain input ciphertext c_x and homomorphic circuit f. More concretely, a vFHE scheme consists of the following algorithms

- $\mathsf{params} \leftarrow \mathsf{ParamGen}(1^\lambda, f)$: given a security parameter λ and a homomorphic circuit f, it computes the parameters params which are a default input to all other algorithms.

- $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$: given a security parameter λ, it generates the public key pk and secret key sk.

- $c_x \leftarrow \mathsf{Enc}(x, \mathsf{pk})$: given plaintext input(s) x and a public key pk, it computes encryption(s) c_x.

- $(c_y, \pi_y) \leftarrow \mathsf{Eval}(c_x, \mathsf{pk})$: given some input ciphertexts c_x and the public key pk, it computes the output ciphertexts c_y and a proof π_y.

- $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathsf{Verify}(c_x, c_y, \pi_y, \mathsf{sk})$: given some input ciphertexts c_x, some output ciphertexts c_y and a proof π_y, output either accept or reject.

- $y \leftarrow \mathsf{Dec}(c_y, \mathsf{sk})$: given some ciphertext(s) c_y and secret key sk, it computes the decryption(s) y.

Next, we define the properties that a vFHE scheme should satisfy.

Correctness. [VKH23] defines a correct vFHE scheme as a scheme that always decrypts to the correct plaintext, i.e., decryption works with probability one. However, most FHE schemes have a small failure probability, thus, we change the definition replacing "one" by overwhelming. Formally, for a certain security parameter λ, any function f and plaintext inputs x, using the parameters $\mathsf{params} \leftarrow \mathsf{ParamGen}(1^\lambda, f)$, it holds that

$$\Pr\left[\begin{array}{c|c} \mathsf{Dec}(c_y, \mathsf{sk}) = f(x) & \begin{array}{c} (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda) \\ c_x \leftarrow \mathsf{Enc}(x, \mathsf{pk}) \\ (c_y, \pi_y) \leftarrow \mathsf{Eval}(c_x, \mathsf{pk}) \end{array} \end{array}\right] = 1 - \mathsf{negl}(\lambda).$$

Completeness. A complete vFHE scheme always verifies for output ciphertexts and proof generated honestly for the corresponding input ciphertexts. More formally, for a certain security parameter λ and any function f and plaintext inputs x, using the parameters $\mathsf{params} \leftarrow \mathsf{ParamGen}(1^\lambda, f)$, it holds that

$$\Pr\left[\begin{array}{c|c} \begin{array}{c} \mathsf{Verify}(c_x, c_y, \pi_y, \mathsf{sk}) \\ = \mathsf{accept} \end{array} & \begin{array}{c} (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda) \\ c_x \leftarrow \mathsf{Enc}(x, \mathsf{pk}) \\ (c_y, \pi_y) \leftarrow \mathsf{Eval}(c_x, \mathsf{pk}) \end{array} \end{array}\right] = 1.$$

Soundness. A sound vFHE scheme only allows a negligible probability that some input and output ciphertexts verify if their corresponding plaintexts are not valid. More formally, for a certain security parameter λ and any function f, plaintext inputs x and adversary $\mathcal{A}$, using the parameters $\mathsf{params} \leftarrow \mathsf{ParamGen}(1^\lambda, f)$, it holds that

$$\Pr\left[\begin{array}{c|c} \begin{array}{c} \mathsf{Verify}(c_x, c_y, \pi_y, \mathsf{sk}) \\ = \mathsf{accept} \\ \wedge \\ \mathsf{Dec}(c_y, \mathsf{sk}) \neq f(x) \end{array} & \begin{array}{c} (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda) \\ x \leftarrow \mathcal{A}_{\mathsf{params}}(\mathsf{pk}) \\ c_x \leftarrow \mathsf{Enc}(x, \mathsf{pk}) \\ (c_y, \pi_y) \leftarrow \mathcal{A}_{\mathsf{params}}(c_x, \mathsf{pk}) \end{array} \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Security. The security of a vFHE scheme is defined basically in the same way as the security of a regular FHE scheme. Formally, for a certain security parameter λ, any function f, plaintext inputs x and adversary $\mathcal{A}$, using the parameters $\mathsf{params} \leftarrow \mathsf{ParamGen}(1^\lambda, f)$, we say that the vFHE scheme is CPA-secure if it holds that

$$\left|\Pr\left[\begin{array}{c|c} b' = b & \begin{array}{c} (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda) \\ (x_0, x_1, \mathsf{st}) \leftarrow \mathcal{A}_{\mathsf{params}}(\mathsf{pk}) \\ c_x \leftarrow \mathsf{Enc}(x_b, \mathsf{pk}) \text{ for } b \leftarrow \{0,1\} \\ b' \leftarrow \mathcal{A}_{\mathsf{params}}(c_x, \mathsf{st}) \end{array} \end{array}\right] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda).$$

Notice that [VKH23] defines CCA1 security and it is known that by combining FHE and SNARKs, we can achieve CCA1, but this introduces some technical and theoretical complications that would push us away from our goal of constructing a practical vFHE scheme. Thus, we prefer to stick to CPA security.

3 vFHE Schemes from Lattice-based SNARKs

In this section, we construct our new verifiable FHE (vFHE), as defined in Section 2.7, by combining a second-generation FHE scheme, such as BGV or CKKS, and a lattice-based SNARK. A vFHE scheme allows a client to outsource the computation of f on input x to a service provider, while keeping x private and also verifying the correctness of the final result $y = f(x)$. Without loss of generality, we assume that the homomorphic computation corresponding to the outsourced function is represented as a layered circuit, as explained in Section 3.1. The homomorphic computation is then performed as usual, and to allow the verification of the computation, the SNARK is used to generate proofs for each layer of the circuit. This is explained in detail in Section 3.2.

3.1 Layered circuits for homomorphic computation

In this section[2], we assume that the homomorphic computation can be represented as a circuit where each gate takes as input elements of the ring $\mathcal{R}_Q$ (polynomials modulo $X^N + 1$ and coefficients in $\mathbb{Z}_Q$) and performs a homomorphic addition, plaintext-ciphertext multiplication, or ciphertext-ciphertext multiplication, which is divided in a tensor product and the maintenance routines, the key-switching and the modulus switching. Moreover, it is always possible (and it is common in the FHE literature) to use layered circuits for the homomorphic computation, where each layer finishes with ciphertext-ciphertext multiplication gates, and the number of layers is then the multiplicative depth of the circuit. Thus, we consider the following structure for the circuits: First of all, define $Q^{(k)} = \prod_{i=0}^{k} q_i$, i.e., the product of $k+1$ small primes. For a circuit with multiplicative depth L, fresh ciphertexts are defined over $R_{Q^{(L)}} = R_Q$. Then we represent the circuit as the following composition of subcircuits

$$C_0(...(M_{L-1}(C_{L-1}(M_L(C_L(\cdot))))))$$

where each subcircuit C_k takes as input $\mathsf{input}(k)$ elements of $R_{Q^{(k)}}$, outputs $\mathsf{output}(k)$ elements of $R_{Q^{(k)}}$, and only the output gates can be tensor products. The subcircuit M_k then corresponds to the key- and modulus-switchings. Hence, it takes as input $\mathsf{input}(k-1) \leq \mathsf{output}(k)$ elements of $R_{Q^{(k)}}$, and outputs $\mathsf{input}(k-1)$ elements of $R_{Q^{(k-1)}}$.

3.2 Our vFHE scheme

Our scheme combines a second-generation FHE scheme $\mathcal{E} = (\mathsf{HE.ParamGen}, \mathsf{HE.KeyGen}, \mathsf{HE.Enc}, \mathsf{HE.Dec}, \mathsf{HE.Eval})$, such as BGV and CKKS, and a lattice-based SNARK $\Pi = (\mathsf{Setup}, \mathsf{Prover}, \mathsf{Verifier})$.

- $\mathsf{ParamGen}(1^\lambda, f)$: Given the function f, construct a layered circuit $C_0(...(M_{L-1}(C_{L-1}(M_L(C_L(\cdot))))))$ of multiplicative depth L that computes $f(\cdot)$. Run $\mathsf{HE.ParamGen}(1^\lambda, L)$ to generate the FHE parameters $\mathsf{HE.params}$. Return $\mathsf{params} = (\mathsf{HE.params}, L, \{C_i\}, \{M_i\})$ and consider them default inputs to the following algorithms.

- $\mathsf{KeyGen}(1^\lambda)$ Generate the secret, public, and relinearization key of the FHE scheme, i.e., run $(\mathsf{HE.sk}, \mathsf{HE.pk}, \mathsf{HE.}\mathtt{rlk}) \leftarrow \mathsf{HE.KeyGen}(1^\lambda)$. Then, run the setup algorithm of the SNARK scheme for each multiplicative layer i, over each field that is used in the dCRT representation of $\mathcal{R}_{Q^{(i)}}$. In more detail:

  1. For each prime q_j with $j = 0, \ldots, L$, use the SNARK setup algorithm to generate $(\mathsf{crs}_{L,j}, \mathsf{vrk}_{L,j}) \leftarrow \Pi.\mathsf{Setup}(1^\lambda, C_L)$ in $\mathbb{F}_{q_j}$.
  2. For each multiplicative layer $i = L-1$ to $i = 0$, use the SNARK setup algorithm to generate
     (a) $(\mathsf{crs}_{i,j}, \mathsf{vrk}_{i,j}) \leftarrow \Pi.\mathsf{Setup}(1^\lambda, C_i \circ M_{i+1})$ in $\mathbb{F}_{q_j}$, for $0 \leq j \leq i$.
     (b) $(\mathsf{crs}_{i,i+1}, \mathsf{vrk}_{i,i+1}) \leftarrow \Pi.\mathsf{Setup}(1^\lambda, M_{i+1})$ in $\mathbb{F}_{q_{i+1}}$.

  Output the secret/public keys: $\mathsf{sk} = (\mathsf{HE.sk}, \{\mathsf{vrk}_{i,j}\})$ and $\mathsf{pk} = (\mathsf{HE.pk}, \mathsf{HE.}\mathtt{rlk}, \{\mathsf{crs}_{i,j}\})$.

[2]Extending our construction to also implement slot rotations is straightforward, as discussed in Appendix A, where we also present an optimized way of adding all the slots minimizing the number of proof layers.

- $\mathsf{Enc}(x, \mathsf{pk})$: Return $c_x = \mathsf{HE.Enc}(x, \mathsf{HE.pk}) \in \mathcal{R}^2_{Q^{(L)}}$ for $x \in \mathcal{R}_t$.

- $\mathsf{Eval}(\bar{\mathbf{c}}, \mathsf{pk})$: For a layered circuit $C_0(...(M_{L-1}(C_{L-1}(M_L(C_L(\cdot))))))$ and vector of input ciphertexts $\bar{\mathbf{c}} \in (\mathcal{R}^2_{Q^{(L)}})^{\mathsf{input}(L)}$, run the evaluation algorithm of the FHE scheme as usual, but use the intermediate values as the wire values that are used to generate proofs in the SNARK scheme. In more detail:
  1. For each prime q_j with $j = 0, \ldots, L$:
     (a) Evaluate $\mathbf{c}_{L,j} \leftarrow \mathsf{HE.Eval}(\bar{\mathbf{c}}_j, C_L, \mathsf{HE.}\mathtt{rlk})$ where $\bar{\mathbf{c}}_j \in (\mathcal{R}^2_{q_j})^{\mathsf{input}(L)}$ and $\mathbf{c}_{L,j} \in (\mathcal{R}^3_{q_j})^{\mathsf{output}(L)}$ since C_L has $\mathsf{input}(L)$ inputs and $\mathsf{output}(L)$ outputs. Additionally store the results of all intermediate computations as $\mathbf{w}_{L,j} \in \mathbb{F}^*_{q_j}$.
     (b) Calculate proof $\pi_{L,j} \leftarrow \Pi.\mathsf{Prover}(\mathsf{crs}_{L,j}, \bar{\mathbf{c}}_j || \mathbf{c}_{L,j}, \mathbf{w}_{L,j})$.
  2. For each multiplicative layer $i = L - 1$ to $i = 0$:
     (a) Generate the intermediate outputs used for modulus switching $\mathbf{c}_{i,i+1} \leftarrow \mathsf{HE.Eval}(\mathbf{c}_{i+1,i+1}, M_{i+1}, \mathsf{HE.}\mathtt{rlk})$ where $\mathbf{c}_{i+1,i+1} \in (\mathcal{R}^3_{q_{i+1}})^{\mathsf{input}(i)}$ and $\mathbf{c}_{i,i+1} \in (\mathcal{R}^2_{q_{i+1}})^{\mathsf{input}(i)}$, since a maintenance circuit has as many ciphertext outputs as inputs. Also, for $0 \le j \le i$, generate intermediate outputs $\mathbf{c}_{i,j} \leftarrow \mathsf{HE.Eval}(\mathbf{c}_{i+1,j} || [\mathbf{c}_{i+1,i+1}]_{q_j}, C_i \circ M_{i+1}, \mathsf{HE.}\mathtt{rlk})$ where $\mathbf{c}_{i+1,j} \in (\mathcal{R}^3_{q_j})^{\mathsf{input}(i)}$, and $\mathbf{c}_{i,j} \in (\mathcal{R}^2_{q_j})^{\mathsf{output}(i)}$ since C_i has $\mathsf{input}(i)$ inputs and $\mathsf{output}(i)$ outputs. Again, store the results of all intermediate computations as $\mathbf{w}_{i,j} \in \mathbb{F}^*_{q_j}$ for $0 \le j \le i + 1$.
     (b) For $0 \le j \le i$, using additionally in the statement $\mathsf{HE.}\mathtt{rlk}$, modswitching outputs $\mathbf{c}_{i+1,i+1}$ and the decompositions $[\tilde{\mathbf{c}}_i]_{q_j} \in (\mathcal{R}^{i+1}_{q_j})^{\mathsf{input}(i)}$ used for relinearization, calculate the proofs

$$\pi_{i,j} \leftarrow \Pi.\mathsf{Prover}(\mathsf{crs}_{i,j}, \mathbf{c}_{i+1,j} || \mathbf{c}_{i,j} || [\mathbf{c}_{i+1,i+1}]_{q_j} || [\tilde{\mathbf{c}}_i]_{q_j} || \mathsf{HE.}\mathtt{rlk}, \mathbf{w}_{i,j}).$$

     Also calculate the proofs for the maintenance circuit M_{i+1}

$$\pi_{i,i+1} \leftarrow \Pi.\mathsf{Prover}(\mathsf{crs}_{i,i+1}, \mathbf{c}_{i+1,i+1} || \mathbf{c}_{i,i+1} || [\tilde{\mathbf{c}}_i]_{q_j} || \mathsf{HE.}\mathtt{rlk}, \mathbf{w}_{i,i+1}).$$

  Return all ciphertexts $\mathbf{c}_{i,j}$ output by $\mathsf{HE.Eval}$ and all proofs $\pi_{i,j}$ output by $\Pi.\mathsf{Prover}$

- $\mathsf{Verify}(\bar{\mathbf{c}}, \{\mathbf{c}_{i,j}\}, \{\pi_{i,j}\}, \mathsf{sk})$: Given the input ciphertexts $\bar{\mathbf{c}} \in (\mathcal{R}^2_{Q^{(L)}})^{\mathsf{input}(L)}$, all intermediate outputs $\mathbf{c}_{i,j} \in (R^3_{q_j})^{\mathsf{output}(i)}$, the proofs $\pi_{i,j}$ and verification keys $\mathsf{vrk}_{i,j}$, run $\Pi.\mathsf{Verifier}$ for every partial circuit in every field $\mathbb{F}_{q_j}$ and output reject if any SNARK verifier rejects. Otherwise output accept. In more detail:
  1. For each prime q_j with $j = 0, \ldots, L$: verify the circuit C_L by running $b_{L,j} \leftarrow \Pi.\mathsf{Verifier}(\mathsf{vrk}_{L,j}, \bar{\mathbf{c}}_j || \mathbf{c}_{L,j}, \pi_{L,j})$ and output reject if $b_{L,j} = 0$.
  2. For each multiplicative layer $i = L - 1$ to $i = 0$:
     (a) For $0 \le j \le i$, verify the circuit $C_i \circ M_{i+1}$ in $\mathbb{F}_{q_j}$ by using the relevant input and output ciphertexts, modswitching outputs and inputs for relinearization to run

$$b_{i,j} \leftarrow \Pi.\mathsf{Verifier}(\mathsf{vrk}_{i,j}, \mathbf{c}_{i+1,j} || \mathbf{c}_{i,j} || [\mathbf{c}_{i+1,i+1}]_{q_j} || [\tilde{\mathbf{c}}_i]_{q_j} || \mathsf{HE.}\mathtt{rlk}, \pi_{i,j})$$

     and output reject if $b_{i,j} = 0$.
     (b) Verify the circuit M_{i+1} by using the relevant input and output ciphertexts and inputs for relinearization to run

$$b_{i,i+1} \leftarrow \Pi.\mathsf{Verifier}(\mathsf{vrk}_{i,i+1}, \mathbf{c}_{i+1,i+1} || \mathbf{c}_{i,i+1} || [\tilde{\mathbf{c}}_i]_{q_j} || \mathsf{HE.}\mathtt{rlk}, \pi_{i,i+1})$$

     and output reject if $b_{i,i+1} = 0$.

(c) Output accept, since all subcircuits verified correctly.

- $\mathsf{Dec}(c_y, \mathsf{sk})$: Output $y = \mathsf{HE.Dec}(c, \mathsf{HE.sk}) \in \mathcal{R}_t$ and $c_y \in (\mathcal{R}^3_{Q(0)})^{\mathsf{output}(0)}$.

Subcircuit blueprinting. In our basic construction, the vFHE public key contains $O(L^2)$ different SNARK crs instances which are used in the Eval algorithm to generate the vFHE proof. This could easily be decreased to only $O(L)$ instances by generating a "blueprint" crs for each prime q_j. These crs's encode a blueprint circuit C_B which is able to compute all other subcircuits $C_0(M_1(\cdot)), \ldots, C_{L-1}(M_L(\cdot)), C_L$ by setting certain input wires to zero. Since these subcircuits would be very similar, mostly differing in the number of inputs and outputs, the added number of gates would be minimal, which means that the added cost of proof generation would also be minimal. Note that one can also choose to make this tradoff for certain similar layers but not for others. This blueprinting technique also effects the number of SNARK vrk instances in the secret key but their size is negligible compared to the crs's.

3.3 Security analysis

The FHE and the SNARK schemes are used independently in our construction, hence, the security of our scheme is trivially inherited from them. In this section, we briefly discuss the security requirements that were defined in Section 2.7.

3.3.1 Correctness & Completeness.

These properties follow from the correctness and completeness of the FHE and SNARK scheme respectively. Our construction simply divides the FHE computation in an exhaustive set of subcircuits. The Eval algorithm evaluates every subcircuit, propagating outputs as intended by the FHE scheme. The Verify algorithm will accept when all the SNARKs that prove these subcircuits verify.

3.3.2 IND-CPA Security.

Suppose there is an algorithm $\mathcal{A}$ that breaks the CPA-security of our scheme. Then we can construct an algorithm $\mathcal{B}$ that breaks the CPA-security of the underlying FHE scheme by simply letting $\mathcal{B}$ generate the public parameters of the SNARK scheme (as they are all independent of the secret values of the FHE scheme, thus $\mathcal{B}$ is able to do so), and providing them to the $\mathcal{A}$ algorithm. The remainder of the IND-CPA security game is the same for the vFHE adversary, so $\mathcal{B}$ can forward the messages between $\mathcal{A}$ and its challenger. Therefore, our construction remains CPA-secure if the base FHE scheme is already so.

3.3.3 Soundness.

The knowledge-soundness of the SNARK scheme implies that the probability ε that a PPT algorithm can produce a verifying but non-valid assignment is negligible in the security parameter λ. If we denote this probability for the subcircuit in layer i with the modulus q_j as $\varepsilon_{i,j}$ and call this event $V_{i,j}$ (such that $\Pr[V_{i,j}] = \varepsilon_{i,j}$), and denote the soundness probability from Section 2.7 as $\varepsilon_{\text{vFHE}}$, we can use the union bound to state that

$$\varepsilon_{\text{vFHE}} \leq \Pr[\bigcup_{\substack{i=0,\ldots,d \\ j=0,\ldots,i+1}} V_{i,j}] \leq \sum_{\substack{i=0,\ldots,d \\ j=0,\ldots,i+1}} \varepsilon_{i,j} \leq d^2\varepsilon = \mathsf{negl}(\lambda)$$

where d is the multiplicative depth of the homomorphic circuit.

3.4 About the compactness and client's efficiency

To avoid trivial constructions of FHE where the server does not actually compute anything, but just attaches the circuit to the ciphertext and the client decrypts and evaluates the circuit on their own, one usually requires that FHE schemes be *compact* [DGHV10], i.e., the running time of decrypting a ciphertext is independent of the circuit that was evaluated. We notice that as all existing leveled FHE schemes, our construction does not satisfy this compactness property, but it is not equivalent to the trivial construction either. Indeed, there are functions for which it is more expensive for the client to compute them locally than to outsource their computation, run the decryption and the verification.[3] Moreover, we notice that there are other reasons for a client to outsource the computation, for example, if the client has already outsourced the storage or if the computation to be performed depends on inputs from the server, which are unknown to the client (as is common when the server offers machine learning as a service, since in this case the server trains a large model and the client cannot download the model to run it locally). Finally, we would like to stress that in practice, our construction is still much more efficient than existing constructions and allows the computation of complicated and deep circuits that could not be handled previously.

All that said, we now present an analysis of the verification cost, then we provide examples of some functions families that are "outsourceable".

3.4.1 Verification cost

As stated in Section 2.3.1, the time complexity of verifying a proof π that a circuit with input size ℓ_{in} and output size ℓ_{out} was correctly computed is $O(\ell_{in} + \ell_{out} + \lambda)$. Notice that we instantiate our SNARKs over $\mathbb{F}_{q_j}$ for $0 \leq j \leq L$, thus, each gate of the circuit that operates on polynomials is actually a point-wise addition or multiplication of N-dimensional vectors corresponding to the NTTs of the polynomials (see Section 2.5). Thus, because the circuit is defined in terms of polynomials, its verification cost is $O(N \cdot (\ell_{in} + \ell_{out}) + \lambda)$.

Lemma 1. *Verifying $C_L(\cdot)$ can be done in $O(N \cdot L \cdot (\mathsf{input}(L) + \mathsf{output}(L)) + L \cdot \lambda)$ basic operations, i.e., operations modulo small primes q_j's.*

Proof. One just has to note that C_L is a circuit from $R_{Q^{(L)}}^{\mathsf{input}(L)}$ to $R_{Q^{(L)}}^{\mathsf{output}(L)}$, thus, the verification for each prime q_j can be done in time $O(N \cdot (\mathsf{input}(L) + \mathsf{output}(L)) + \lambda)$. Since we have $L+1$ primes, verifying $C_L(\cdot)$ costs, in total, $O(N \cdot L \cdot (\mathsf{input}(L) + \mathsf{output}(L)) + L \cdot \lambda)$ basic operations. $\square$

Lemma 2. *Verifying $C_k(M_{k+1}(\cdot))$ can be done in $O(N \cdot \mathsf{output}(k+1) \cdot (k+1)^2 + N \cdot \mathsf{output}(k) \cdot (k+1) + \lambda \cdot (k+1))$ basic operations, i.e., operations modulo small primes q_j's.*

Proof. Since M_{k+1} runs the relinearization and the modulus-switching, it uses the algorithm `FastBaseExt`, which takes as input the outputs of C_{k+1} used as inputs for C_k, for each of the $k+1$ primes used to compute C_{k+1}. Thus, M_{k+1} is a circuit from $R_{Q^{(k+1)}}^{\mathsf{input}(k)\cdot(k+1)}$ to $R_{Q^{(k)}}^{\mathsf{input}(k)}$.

Then, we have $C_k : R_{Q^{(k)}}^{\mathsf{input}(k)} \to R_{Q^{(k)}}^{\mathsf{output}(k)}$. Thus, $C_k \circ M_{k+1} : R_{Q^{(k+1)}}^{\mathsf{input}(k)\cdot(k+1)} \to R_{Q^{(k)}}^{\mathsf{output}(k)}$, which means that for each prime $q_0, ..., q_k$, we can verify $C_k(M_{k+1}(\cdot))$ in time $O(N \cdot \mathsf{input}(k) \cdot (k+1) + N \cdot \mathsf{output}(k) + \lambda)$. Since $\mathsf{input}(k) \leq \mathsf{output}(k+1)$, verifying it for all $k+1$ primes has time complexity $O(N \cdot \mathsf{output}(k+1) \cdot (k+1)^2 + N \cdot \mathsf{output}(k) \cdot (k+1) + \lambda \cdot (k+1))$. $\square$

[3]In works that construct private verifiable computation (PVC), this is related to the outsourceability of the functions.

Lemma 3 (Client's verification cost)**.** *Verifying a layered circuit* $C(\cdot) = C_0(...(M_{L-1}(C_{L-1}(M_L(C_L(\cdot))))))$ *can be done in*

$$O\left(\lambda \cdot L^2 + N \cdot L \cdot \mathsf{input}(L) + \sum_{k=0}^{L} N \cdot \mathsf{output}(k) \cdot (k+1)^2\right) \tag{1}$$

operations modulo small primes q_j*'s.*

Proof. Since the verification is done by verifying C_L, then verifying the composition $C_k \circ M_{k+1}$ for $0 \le k \le L-1$, we just have to compute $t_0 + t_1$, where t_0 is cost from Lemma 1 and t_1 is the sum of costs from Lemma 2 for each k. It holds that

$$\begin{aligned} t_1 &= \sum_{k=0}^{L-1} O(N \cdot \mathsf{output}(k+1) \cdot (k+1)^2 + N \cdot \mathsf{output}(k) \cdot (k+1) + \lambda \cdot (k+1)) \\ &= O\left(\lambda \cdot L^2 + \sum_{k=0}^{L} N \cdot \mathsf{output}(k) \cdot (k+1)^2\right) \end{aligned}$$

Since $t_0 = O(N \cdot L \cdot (\mathsf{input}(L) + \mathsf{output}(L)) + L \cdot \lambda)$, the result follows. $\square$

We stress that thanks to the slot structure of the plaintext space of the FHE schemes we are considering, the client can encrypt $s := \Theta(N)$ messages per ciphertext and the homomorphic computation actually evaluates the circuit on N different inputs in parallel. Thus, the expression in Lemma 3 can be divided by N when comparing to the cost of evaluating the function locally. Generally speaking, a circuit is outsourceable if it is wide and the number of "inner gates" is much larger than the number of inputs and outputs. For example, supposing that $\mathsf{input}(k)$ and $\mathsf{output}(k)$ are constants for all k, then Lemma 3 simplifies to $O(N \cdot L^3 + \lambda \cdot L^2) = O(N \cdot L^3)$, because $N = \Theta(\lambda)$ for security reasons. Hence, since the cost of encryption/decryption is negligible wrt the cost of verification, a circuit with S gates and multiplicative depth L is outsourceable if $S = \Omega(L^3)$.

3.4.2 Outsourceability of matrix-vector multiplication.

Let $f(\mathbf{v}) = \mathbf{M} \cdot \mathbf{v}$ for some matrix $\mathbf{M} \in \mathbb{Z}^{m \times n}$ and some vector $\mathbf{v} \in \mathbb{Z}^n$. Also, let $s \in \Theta(N)$ be the number of plaintext slots. Then, computing $f(\mathbf{v}_1), f(\mathbf{v}_2), ..., f(\mathbf{v}_s)$ locally costs at least $s \cdot m \cdot n \in \Omega(N \cdot m \cdot n)$ basic operations. However, since we can compute f with a circuit of depth one, we can set $L = 1$ and the verification cost in Lemma 3 becomes $O(N(n+m) + \lambda)$. Because $N = \Theta(\lambda)$, this is actually $O(N(n+m))$, which is cheaper than $\Omega(N \cdot m \cdot n)$. This comparison could be extended to affine maps $f(\mathbf{v}) = \mathbf{M} \cdot \mathbf{v} + \mathbf{b}$ for a vector $\mathbf{b} \in \mathbb{Z}^m$. Notice that homomorphically calculating these linear functions does not require ciphertext-ciphertext multiplications.

3.4.3 Outsourceability of depth-one circuits.

Now we extend the comparison to non-linear functions. Let $f(\mathbf{v}_1, \mathbf{v}_2) = (\mathbf{M}_1 \cdot \mathbf{v}_1) \odot (\mathbf{M}_2 \cdot \mathbf{v}_2)$ for $\mathbf{v}_1, \mathbf{v}_2 \in \mathbb{Z}^n$ and $\mathbf{M}_1, \mathbf{M}_2 \in \mathbb{Z}^{m \times n}$ where $\odot$ represents the Hadamard product. Here, homomorphically calculating f would require one layer of ciphertext-ciphertext multiplications, so again we set $L = 1$. Similar to the previous comparison we can show that the verification cost is $O(N(n+m))$. Also, for $s \in \Theta(N)$ plaintext slots, the cost of local computation is at least $\Omega(N \cdot m \cdot n)$, so the function remains outsourceable. Notice again that this comparison can be extended to functions of the form $f(\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3) = (\mathbf{M}_1 \cdot \mathbf{v}_1) \odot (\mathbf{M}_2 \cdot \mathbf{v}_2) + \mathbf{M}_3 \cdot \mathbf{v}_3 + \mathbf{b}$.

3.4.4 Outsourceability of higher depth circuits.

To homomorphically calculate circuits of higher depth we divide them into consecutive depth-one circuits as described in Section 3.1. Consider for example homomorphically calculating a function f that approximately represents the feedforward computation of a neural network. For a d-layered network, f can be defined as $f(\mathbf{v}) = f_d(\dots f_2(f_1(\mathbf{v}))\dots)$ where $f_i(\mathbf{v}) = \sigma(\mathbf{M}_i \cdot \mathbf{v} + \mathbf{b}_i)$ for a sequence of compatible weight matrices $\mathbf{M}_i$, bias vectors $\mathbf{b}_i$ and an activation function σ. In FHE, the function σ is typically approximated by a low-degree polynomial, e.g. $\sigma(\mathbf{v}) = \mathbf{v} \odot \mathbf{v}$, thus each neural network layer is a depth-one circuit and we can set $L = O(d)$. To ease the comparison, let's say the number of inputs or outputs in each layer (i.e. neurons) is upper bounded by w. Then, the cost of locally evaluating f on $s \in \Theta(N)$ different inputs becomes $\Omega(N \cdot d \cdot w^2)$. From Lemma 3, we can conclude that the verification cost becomes $O(N \cdot w \cdot d^3)$. Therefore, f would be (asymptotically) outsourceable in terms of basic operations when $w \in \Omega(d^2)$, meaning the neural network is sufficiently wide. In Section 5, we instantiate our construction for this specific example.

4 Related work and comparisons

4.0.1 Comparison with [BCFK21]

In [BCFK21], two homomorphic hash functions over Galois rings are proposed and they are used together with a variant of the GKR protocol [GKR08] to obtain verifiable computation over encrypted data. On the positive side, their solution is publicly verifiable. However, the types computation they can verify is rather limited.

In more detail, instead of verifying computation over $\mathcal{R}_Q$, they actually verify circuits over $\mathbb{Z}_Q[X]$, meaning that no reduction modulo $X^N + 1$ is performed, which means that the degrees of the polynomials involved in the homomorphic computation are no longer bounded by N, but on depth d, they have degree $O(2^d \cdot N)$. Then, they use the homomorphic hash functions to compress these polynomials and reduce the proof size. But because the maintenance operations do not respect the homomorphic properties of the hashes, i.e., they are not composed by additions and multiplications on $\mathbb{Z}_Q[X]$, they cannot prove the relinearization and the key-switching. Without relinearization, the number of polynomials in each ciphertexts is no longer constant, but at depth d, we have $\Theta(2^d)$ of them. Thus, by exponentially increasing the number of polynomials and the degree of each polynomial, we end up with ciphertexts of size $\Theta(2^{2d} \cdot N \log Q)$ instead of $\Theta(N \log Q)$. Of course, operating with larger ciphertexts is also more costly timewise. In other words, there is a huge time and memory overhead for the server depending on the depth.

In our case, the server has essentially no overhead, as the homomorphic computation is basically unchanged.

Furthermore, no modulus switching means that the noise in the ciphertexts grows exponentially fast. Essentially, at depth d, the noise is $\Theta(\sigma^{2^d})$ where σ is a constant bounding the initial noise of fresh ciphertexts. If the noise is too big, then the correctness of decryption stops holding. In more detail, we need the final noise to be bounded by Q, so,

$$2^d \cdot \log \sigma < \log Q \;\Rightarrow\; d \in O\left(\log(\log Q / \log \sigma)\right) = O\left(\log \log Q\right)$$

Therefore, on top of the limitation related to the size of the ciphertexts (efficiency), there is another limitation related to the correctness, which implies that, in the best possible scenario, the depth of the circuits that [BCFK21] can verify is only $O\left(\log \log Q\right)$. In our case, because we support modulus-switching, the FHE scheme itself supports much deeper computation.

From Theorem 7 of [BCFK21], the time complexity of their verification of a circuit of depth L, having S gates over $\mathcal{R}_Q$, and input size $\mathsf{input}(L)$ is $\tilde{O}((\mathsf{input}(L) + L^2)N + \lambda L \log S)$

operations over $\mathbb{Z}_Q$. Since additions and multiplications modulo Q cost at least $\log Q$ basic operations, supposing Q polynomial in N, as it is usual in FHE, their verification cost becomes $\tilde{O}(((\mathsf{input}(L) + L^2)N + \lambda L \log S) \log N)$ in terms of basic integer operations. By setting L as a constant, since basically that is the multiplicative depth that their construction can handle, their verification cost becomes $\tilde{O}(\mathsf{input}(L)N \log N + \lambda \log S \log N)$, while in our case, from Equation 1, we obtain verification cost $O(\lambda + N \cdot \mathsf{input}(L))$.

4.0.2 Comparison with Rinocchio [GNS23]

In [GNS23], the authors propose another approach for verifying computations over encrypted data. Their work defines a zkSNARK for computation over rings by extending the classical Quadratic Arithmetic Programs (QAPs) to Quadratic Ring Programs (QRPs), and defining compatible ring-based encoding schemes. They claim that this proof system enables privacy-preserving verifiable computation by instantiating it over the ring R_Q and combining it with RLWE-based FHE schemes. However, similar to [BCFK21], their approach does not natively support the maintenance operations that are crucial for the efficiency of modern FHE schemes such as [BGV12] and [FV12]. As discussed above, not performing the maintenance operations means that both the modulus Q and the ciphertext degree depend exponentially on the multiplicative depth d. This puts significant overhead on the prover for any practical applications.

As the authors briefly remark, it is possible to simulate the non-arithmetic operations in the QRP and in that way still incorporate maintenance operations in the proof. Again this causes additional overhead in contrast to our construction, which we will analyze here. Relinearization, for example, requires modular reduction $[c]_{q_j}$ of an element $c \in R_{Q_i}$ for $j = 0, \ldots, i$. To prove this modular reduction, one needs to prove the modular reduction of each of its N coefficients. Proving the reduction $[a]_{q_j}$ for $a \in \mathbb{Z}_Q$ requires $\mathcal{O}(\log Q)$ constraints, since it is only possible by first bitwise decomposing the coefficient a. Therefore, proving relinearizations alone would add $\mathcal{O}(dLN \log Q)$ constraints to the QRP instance for each ciphertext. In our construction, we avoid having to express these reductions using R1CS constraints and therefore avoid the increased CRS size and the increased cost of proof generation.

Modulus-switching also significantly increases the amount of constraints in the QRP since it requires non-arithmetic modular reduction. Moreover, ciphertexts coefficients are defined modulo Q_{i-1} after this operation. The authors of Rinocchio suggest to emulate this modulus switch by multiplying by the constant $(1, 1, \ldots, 1, 0, \ldots, 0)$ in RNS decomposition, for $i-1$ non-zero elements. Since this only emulates the reduction in the RNS decomposition of the ring, one eventually has to properly reduce (requiring bitwise decomposition) before the next round of maintenance operations. The authors do remark that modulus-switching can be avoided by using a scale-invariant FHE scheme [FV12]. However, in that case the homomorphic multiplication of ciphertexts would require non-arithmetic computations. Both approaches, in contrast to our construction, also imply that the size of ciphertexts and therefore the amount of constraints, does not decrease linearly w.r.t. the current multiplicative depth.

One advantage of Rinocchio is that while our proof size depends on the multiplicative depth of the circuit, their proofs consist of a constant number of encoding elements. However, [GNS23] describe this encoding as a Regev-style encoding for plaintext space $\mathcal{R}_Q$, which is impractical since Q typically has hundreds of bits. To be fair, we mention an improvement of the encoding in Rinocchio introduced by Viand et al. [VKH23] where each of the RNS digits of a ring element is encoded separately such that the plaintext modulus of the encoding scheme corresponds to a modulus q_i instead of Q. In this case the encoding size *per ring element* would be similar to our construction, but we would have to encode less elements because of the reasons mentioned above, as well as the blueprinting technique mentioned in Section 3.2.

To summarize: our construction is generally more efficient than Rinocchio except in proof size when multiplicative depth is large, but we have shown that in this case their CRS size and proof generation are impractically costly.

5 Implementation and Performance

In this section, we discuss an instantiation of our scheme that uses the construction by Ishai et al. [ISW21] as lattice-based SNARK to verify the following computation of a BGV homomorphic circuit

$$[\mathbf{c}_{l+1}]_{j=1}^{k_{l+1}} = \left[\left(\sum_{i=1}^{k_l} a_{l,ij}\mathbf{c}_{l,i} + b_{l,j} \right)^2 \right]_{j=1}^{k_{l+1}} \tag{2}$$

for two layers $l = 0$ and $l = 1$. Here, a and b are elements of the plaintext space $\mathcal{R}_t$, and ciphertexts $\mathbf{c} \in \mathcal{R}_Q[Y]$. Notice that this computation approximates the feedforward evaluation of a basic neural network layer. Since each layer has a multiplicative depth of one, we can evaluate one neural network layer before performing relinearization and modswitch on the ciphertexts $\mathbf{c}_{1,i}$. We refer to Section 3.4 for a more detailed description of this instantiation.

It is trivial to show that for standard BGV parameters that provide 128-bit security, it suffices to select a BGV modulus, $Q = Q^{(1)} = q_0q_1q_2$, the product of three 30-bit primes, and a post-modswitch modulus $Q^{(0)} = q_0q_1$. (Concretely, the output ciphertexts $\mathbf{c}_{2,i}$ satisfy the noise bound for decryptability and the relinearized ciphertexts $\mathbf{c}'_{1,i}$ satisfy the noise bound required before modswitching.) Therefore, we claim that this computation is verifiable using 6 proofs in 2 layers and 3 finite fields, as in Figure 3. To simplify parameter selection for this specific instantiation, the moduli $Q^{(0)}, Q^{(1)}$ contain an extra prime. Remark that our construction also allows for other similar adjustments, e.g. removing multiple primes per modswitch for increased noise reduction.

As shown in Section 3.3, we can select the security parameters for the FHE and SNARK scheme independently. We now select the parameters of the lattice-based SNARK scheme such that 128-bit security is achieved (for instantiations over all prime fields $\mathbb{F}_{q_i}$). This scheme consists of a linear PCP and a linear-only vector encryption. To ensure efficiency of the former, we select the primes q_i such that the prime fields $\mathbb{F}_{q_i}$ contain N_g-th roots of unity for $N_g = 2^{20}$ which determines the maximum number of gates in one R1CS instance. This allows for $O(n \log n)$ construction of the QAP that the linear PCP is based on. Notice that the existence of the N_g-th roots of unity ensure the existence of the N-th roots of unity required for the NTT transformation in BGV (since $N = 2^{12}$). Also, a soundness amplification parameter ρ is determined in order to achieve sufficient knowledge soundness for the PCP.

As for the linear-only vector encryption, the parameters were selected similarly to the method in Section 4.2 of [ISW21]. Recall that the plaintext space $\mathbb{F}_{p_i}$ of this scheme needs to match the finite field of the PCP. In this context, q_i are the ciphertext space moduli. In Table 1, we summarize our selection of the necessary parameters. Lastly, we slightly adjusted both the linear PCP and the vector encryption to remove components that form the zero-knowledge property of the resulting SNARK.

We have implemented[4] this instantiation of our vFHE scheme for the homomorphic circuit described by Equation (2). Using the `libsnark` library, we implemented the R1CS constraint systems. This includes the elementary BGV computations in double-CRT representation, as well as the circuits required for the relinearization and the modswitch operations. The latter include sub-circuits for the NTT transformations which are most

[4] https://github.com/jannikspiessens/vFHE

Table 1: Parameter selection for the lattice-based SNARK. p is the plaintext modulus, while q is the ciphertext modulus. n and s are the lattice dimension and the Gaussian width of the lattice-based vector encryption. ρ is the soundness amplification parameter. τ is the ciphertext sparsification parameter which ensures the linear-only property and q' is the post-modswitch modulus which decreases proof size. For a more thorough explanation, we refer to Ishai et al. [ISW21].

p	ρ	τ	$\log_2 q$	q'	n	s
1085276161	15	5	87	389942329959458	3500	4
1092616193	15	5	87	410854793832210	3500	4
1095761921	15	5	87	420467605951707	3500	4

expensive in terms of the number of R1CS gates required. We used the `lattice-zksnark` library by Ishai et al. to implement the `Setup`, `Prover` and `Verifier` SNARK methods that are used in our scheme. The timings of these algorithms, aggregated over all 6 R1CS instances, are most relevant for the performance of our scheme and are shown in Table 2. The total `crs` size over all 6 proofs is 11.6GB while the proof size is about 53.2kB. There are 3 133 440 gates in all R1CS instances combined. Note that `crs` size and setup time could be greatly reduced (possibly 6x smaller) when using the blueprinting technique discussed in Section 3.2.

Table 2: Performance results for SNARK methods of our construction on the computation of Equation (2) for different k_0 and $k_1 = 3, k_2 = 1$.

k_0	`Setup` time	`Prover` time	`Verifier` time
5	1 821 s	116 s	597 ms
15	1 922 s	131 s	618 ms
25	2 023 s	138 s	664 ms
35	2 025 s	138 s	692 ms
50	2 129 s	146 s	731 ms
75	2 448 s	167 s	824 ms
100	2 431 s	167 s	925 ms

We compare our results with [VKH23], who implemented the Rinocchio scheme, and also the naive approach of using field-incompatible proof schemes to verify the dCRT computations. Both approaches are unable to efficiently verify homomorphic circuits with depth greater than one, therefore they only verify a circuit that performs one ciphertext-ciphertext multiplication (followed by a modswitch). Comparison is unfair since it is unclear how their constructions would scale for higher depth circuits. However, we note that our implementation still achieves a 4-6x improvement in prover time while verification times remains practical, even for circuits with a high number of public inputs.

Acknowledgments

We would like to thank Frederik Vercauteren for his valuable comments. This work was supported by CyberSecurity Research Flanders with reference number VR20192203. In addition, this work was partially supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-21-C-0034.

References

[BCFK21] Alexandre Bois, Ignacio Cascudo, Dario Fiore, and Dongwoo Kim. Flexible and efficient verifiable computation on encrypted data. In Juan A. Garay, editor, *Public-Key Cryptography - PKC 2021 - 24th IACR International Conference on Practice and Theory of Public Key Cryptography, Virtual Event, May 10-13, 2021, Proceedings, Part II*, volume 12711 of *Lecture Notes in Computer Science*, pages 528–558. Springer, 2021. doi:10.1007/978-3-030-75248-4_19.

[BCI$^+$13] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, and Omer Paneth. Succinct non-interactive arguments via linear interactive proofs. In Amit Sahai, editor, *TCC 2013: 10th Theory of Cryptography Conference*, volume 7785 of *Lecture Notes in Computer Science*, pages 315–333. Springer, Heidelberg, March 2013. doi:10.1007/978-3-642-36594-2_18.

[BGGJ19] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. Simulating homomorphic evaluation of deep learning predictions. In Shlomi Dolev, Danny Hendler, Sachin Lodha, and Moti Yung, editors, *Cyber Security Cryptography and Machine Learning*, pages 212–230, Cham, 2019. Springer International Publishing.

[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. Association for Computing Machinery, January 2012. doi:10.1145/2090236.2090262.

[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, *Advances in Cryptology – EUROCRYPT 2017, Part III*, volume 10212 of *Lecture Notes in Computer Science*, pages 247–277. Springer, Heidelberg, April / May 2017. doi:10.1007/978-3-319-56617-7_9.

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

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer, Heidelberg, December 2017. doi:10.1007/978-3-319-70694-8_15.

[CT15] Massimo Chenal and Qiang Tang. On key recovery attacks against existing somewhat homomorphic encryption schemes. In Diego F. Aranha and Alfred Menezes, editors, *Progress in Cryptology - LATINCRYPT 2014: 3rd International Conference on Cryptology and Information Security in Latin America*, volume 8895 of *Lecture Notes in Computer Science*, pages 239–258. Springer, Heidelberg, September 2015. doi:10.1007/978-3-319-16295-9_13.

[DGHV10] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully Homomorphic Encryption over the Integers. In *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*. Springer, 2010.

[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: 21st Conference on Computer and Communications Security*, pages 844–855. ACM Press, November 2014. doi:10.1145/2660267.2660366.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. https://eprint.iacr.org/2012/144.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 626–645. Springer, Heidelberg, May 2013. doi:10.1007/978-3-642-38348-9_37.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In Richard E. Ladner and Cynthia Dwork, editors, *40th Annual ACM Symposium on Theory of Computing*, pages 113–122. ACM Press, May 2008. doi:10.1145/1374376.1374396.

[GMNO18] Rosario Gennaro, Michele Minelli, Anca Nitulescu, and Michele Orrù. Lattice-based zk-SNARKs from square span programs. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 556–573. ACM Press, October 2018. doi:10.1145/3243734.3243845.

[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.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 305–326. Springer, Heidelberg, May 2016. doi:10.1007/978-3-662-49896-5_11.

[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. https://eprint.iacr.org/2019/953.

[ISW21] Yuval Ishai, Hang Su, and David J. Wu. Shorter and faster post-quantum designated-verifier zksnarks from lattices. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 212–234. ACM, 2021. doi:10.1145/3460120.3484572.

[KPZ21] Andrey Kim, Yuriy Polyakov, and Vincent Zucca. Revisiting homomorphic encryption schemes for finite fields. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021, Part III*, volume 13092 of *Lecture Notes in Computer Science*, pages 608–639. Springer, Heidelberg, December 2021. doi:10.1007/978-3-030-92078-4_21.

[LMPR08] Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. SWIFFT: A modest proposal for FFT hashing. In Kaisa Nyberg, editor, *Fast Software Encryption – FSE 2008*, volume 5086 of *Lecture Notes in Computer Science*, pages 54–72. Springer, Heidelberg, February 2008. doi:10.1007/978-3-540-71039-4_4.

[PCDN23] Jeongeun Park, Kelong Cong, Debajyoti Das, and Georgio Nicolas. Poster: Panacea—stateless and non-interactive oblivious ram. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, pages 3585–3587, 2023.

[VKH23] Alexander Viand, Christian Knabenhans, and Anwar Hithnawi. Verifiable fully homomorphic encryption. *CoRR*, abs/2301.07041, 2023. arXiv:2301.07041, doi:10.48550/arXiv.2301.07041.

[ZAM23] ZAMA. fhEVM, 2023. https://github.com/zama-ai/fhevm.

[Zuc18] Vincent Zucca. *Towards Efficient Arithmetic for Ring-LWE based Homomorphic Encryption. (Vers une arithmétique efficace pour le chiffrement homomorphe basé sur le Ring-LWE)*. PhD thesis, Pierre and Marie Curie University, Paris, France, 2018. URL: https://tel.archives-ouvertes.fr/tel-02492236.

A More building blocks for vFHE

A.1 Rotations

For any odd integer $k \in [N]$, homomorphically rotating the slots by k positions is done in two steps: First, we apply an automorphism $\tau_k : X \mapsto X^k$ to both components $c_i \in \mathcal{R}_Q$ of a ciphertext $c(Y)$, then we apply a key switching from $s(X^k)$ to $s(X)$.

Since the automorphism is implemented by simply permuting the entries of the matrices corresponding to the double-CRT representation of $c(Y)$, it does not add any cost to the proof and just implies that the wires of the proof of key switching have to be renamed to match the permutation. Then, the key-switching procedure is basically the same as the relinearization, which was already implemented for the homomorphic multiplication.

A.2 Adding all the slots via rotations

Let $\mathsf{AddSlots}_k : \mathcal{R}_Q^2 \to \mathcal{R}_Q^2$ be a procedure that takes one ciphertext encrypting $(\mu_0, ..., \mu_{S-1})$ and outputs an encryption of a vector $\mathbf{u}$ such that $\mathbf{u}[k] = \sum_{i=0}^{S-1} \mu_i$ and $\mathbf{u}[i] = 0$ for $i \neq k$, i.e., the sum is located in the k-th slot and all the other slots are zero.

Algorithm 1: Standard way of adding all the slots homomorphically

```
   Input: Ciphertext $c(Y) \in \mathcal{R}_Q[Y]$, key-switching keys $\mathbf{K}_i$ from $\psi_{2^i}(s)$ to $s$, for
          $0 \leq i < \log S$, where $s$ is the secret key. An integer $k \in [0, S-1]$.
   Output: Ciphertext $c'(Y) \in \mathcal{R}_Q[Y]$,
 1 $c'(Y) = c(Y)$
 2 for $0 \leq i < \log S$ do
 3     $r(Y) = \psi_{2^i}(c(Y))$                ▷ Automorphism that rotates by $2^i$
 4     $r(Y) = KeySwt(c(Y), \mathbf{K}_i)$
 5     $c'(Y) = c'(Y) + r(Y)$
   ▷ Apply a mask to zero other slots
 6 $\mathbf{u} = (0, ..., 0) \in \mathbb{Z}^S$
 7 $\mathbf{u}[k] = 1$
 8 $u(X) = Pack(\mathbf{u})$
 9 $c'(Y) = c'(Y) \cdot u$
10 return $c'(Y)$
```

This procedure is usually implemented as shown in Algorithm 1, since it requires only $\Theta(\log S)$ rotations. However, since each rotation requires a key switching, this algorithm would require $\Theta(\log S)$ layers of proofs in our construction. Instead, we propose Algorithm 2, which computes the same, but only applies the key switchings at the end and in parallel, requiring thus, just 2 layers of proofs, one for the main block, and another one for the key switchings. The main idea is that we can loop rotating the slots and adding, as in the original algorithm, but then we are producing a ciphertext that depends not only on the original secret key, but also on "rotated keys" obtained after the automorphisms. Namely, we are adding terms like $r_0^{(i)} \psi_{2^i}(\mathsf{sk})$ to the ciphertext. Thus, if we store the values $r_0^{(i)}$, we can use them to key switch at the end, producing encryptions of $-r_0^{(i)} \psi_{2^i}(\mathsf{sk})$, which can then be added to the final ciphertext so that we only keep the term that depends on sk itself.

The number of additions on $\mathcal{R}_Q$ is basically the same in both algorithms. The number of key switchings also doesn't change. The only overhead is that now the prover has to store all the $O(\log S)$ ring elements $r_0^{(i)} \in \mathcal{R}_Q$ produced in the first loop. That is, we reduce the number of proof layers from $\Theta(\log S)$ to 2 basically for free.

Algorithm 2: Verification-friendly slot addition

```
Input: Ciphertext $\mathbf{c} \in \mathcal{R}_Q^2$, Key-switching keys $\mathbf{K}_i$ from $s(X^{2^i})$ to $s(X)$, for
       $0 \le i < \log S$. An integer $k \in [0, S-1]$.
Output: Ciphertext $\mathbf{c} \in \mathcal{R}_Q^2$,
1  $(a', b') = (\mathbf{c}[0], \mathbf{c}[1])$                      ▷ Encryption of $m$ under $s$
2  for $0 \le i < \log S$ do
3      $(r_0^{(i)}, r_1^{(i)}) = \psi_{2^i}(\mathbf{c})$          ▷ $r_1^{(i)} = r_0^{(i)}\psi_{2^i}(s) + e^{(i)} + \psi_{2^i}(m)$
4      $b' = b' + r_1^{(i)}$      ▷ Now $b'$ is of the form $a's + \sum_{j=0}^{i} r_0^{(i)}\psi_{2^i}(s) + e' + m'$
   ▷ Apply the key switchings
5  for $0 \le i < \log S$ do
6      $\mathbf{v}_i = Decompose(r_0^{(i)}) \cdot \mathbf{K}_i \in \mathcal{R}_Q^2$     ▷ $\mathbf{v}_i[1] = \mathbf{v}_i[0]s + e_i - r_0^{(i)}\psi_{2^i}(s)$
7  $c'(Y) = (a', b') + \sum_{i=0}^{\log S - 1} \mathbf{v}_i$
   ▷ Apply a mask to zero other slots
8  $\mathbf{u} = (0, ..., 0) \in \mathbb{Z}^S$
9  $\mathbf{u}[k] = 1$
10 $u(X) = Pack(\mathbf{u})$
11 $c'(Y) = c'(Y) \cdot u$
12 return $c'(Y)$
```

B Lattice-based SNARKs

We recall a lattice-based designated-verifier SNARK, proposed by Ishai et al.[ISW21] which combines linear PCPs and linear-only vector encryption following the Bitansky et al. [BCI+13, BISW17] compiler. Note that these components can be easily adjusted such that they construct a designated-verifier zkSNARK.

B.0.1 Linear Probabilistically Checkable Proofs (LPCPs).

A linear PCP is a PCP where the oracle is restricted to respond with linear functions $\boldsymbol{a} = \boldsymbol{Q}^\top \boldsymbol{\pi}$ of the queries $\boldsymbol{Q}$. Following Ishai et al., we will define them using three PPT algorithms. A LPCP with k queries, query length l and knowledge error ε consists of a tuple of algorithms $\Pi_{LPCP} = (\mathsf{Q}, \mathsf{P}, \mathsf{V})$ with the following properties

- $\mathsf{Q}(\mathcal{CS}) \to (\mathsf{st}, \boldsymbol{Q})$: given the the constraint system $\mathcal{CS}$, it generates some query matrix $\boldsymbol{Q} \in \mathbb{F}^{l \times k}$ and a verification state st.
- $\mathsf{P}(\mathcal{CS}, \boldsymbol{x}, \boldsymbol{w}) \to \boldsymbol{\pi}$: given the statement $\boldsymbol{x} \in \mathbb{F}^n$, the witness $\boldsymbol{w} \in \mathbb{F}^{N_w - n}$ and the constraint system $\mathcal{CS}$, it generates a proof $\boldsymbol{\pi} \in \mathbb{F}^l$.
- $\mathsf{V}(\mathsf{st}, \boldsymbol{x}, \boldsymbol{a}) \to b$: given the verification state st, the statement $\boldsymbol{x} \in \mathbb{F}^n$ and some responses $\boldsymbol{a} \in \mathbb{F}^k$, it generates a verification bit $b \in \{0, 1\}$

For our purposes, these LPCPs need to satisfy completeness and knowledge soundness properties described below.

Completeness. A LPCP scheme $\Pi_{LPCP} = (\mathsf{Q}, \mathsf{P}, \mathsf{V})$ is complete iff for every statement $\boldsymbol{x}$ and witness $\boldsymbol{w}$, and R1CS instance $\mathcal{CS}$,

$$\Pr\left[\ \mathsf{V}(\mathsf{st}, \boldsymbol{x}, \boldsymbol{Q}^\top \boldsymbol{\pi}) = 1 \ \middle|\ \begin{array}{c} \mathcal{CS}(\boldsymbol{x}, \boldsymbol{w}) = 1 \\ (\mathsf{st}, \boldsymbol{Q}) \leftarrow \mathsf{Q}(\mathcal{CS}) \\ \boldsymbol{\pi} \leftarrow \mathsf{P}(\mathcal{CS}, \boldsymbol{x}, \boldsymbol{w}) \end{array} \right] = 1.$$

Knowledge Soundness. A LPCP scheme $\Pi_{LPCP} = (\mathsf{Q}, \mathsf{P}, \mathsf{V})$ has knowledge soundness iff for every statement $\boldsymbol{x}$ and proof $\boldsymbol{\pi}^*$ where

$$\Pr[\mathsf{V}(\mathsf{st}, \boldsymbol{x}, \boldsymbol{Q}^\top \boldsymbol{\pi}^*) = 1 \mid (\mathsf{st}, \boldsymbol{Q}) \leftarrow \mathsf{Q}(\mathcal{CS})] > \varepsilon$$

there exists an efficient extractor $\texttt{Extr}$ such that

$$\Pr[\mathcal{CS}(\boldsymbol{x}, \boldsymbol{w}^*) = 1 \mid \boldsymbol{w}^* \leftarrow \texttt{Extr}(\boldsymbol{x}, \boldsymbol{\pi}^*)] = 1.$$

This implies that for every $\boldsymbol{x} \notin \mathcal{L}_{\mathcal{CS}}$ and proof vector $\boldsymbol{\pi}^*$

$$\Pr[\mathsf{V}(\mathtt{st}, \boldsymbol{x}, \boldsymbol{Q}^\top \boldsymbol{\pi}^*) = 1] \leq \varepsilon.$$

Ishai et al. use the claim written below, to construct a linear PCP for R1CS over any field $\mathbb{F}$ by utilizing Quadratic Arithmetic Programs (QAPs) [GGPR13].

Linear PCPs for R1CS (adapted from [ISW21]). Let $\mathcal{CS}$ be an R1CS instance over a finite field $\mathbb{F}$, where $\mathcal{CS} = (N_g, n, N_w, \{\boldsymbol{a}_i, \boldsymbol{b}_i, \boldsymbol{c}_i\}_{i \in [N_g]})$. Then, there exists a 4-query linear PCP for CS with knowledge error $2N_g/(|\mathbb{F}| - N_g)$ and query length $4 + N_w + N_g - n$.

They consider linear PCPs over quadratic extensions $\mathbb{F}_{p^2}$ as well as over the base field $\mathbb{F}_p$. As will be obvious from the construction later, this field needs to match the plaintext space of the vector encryption scheme. Ishai et al. achieve this by either instantiating the vector encryption over the same field or compiling the $\mathbb{F}_{p^2}$ PCP to a $\mathbb{F}_p$ PCP. We will only consider PCPs over $\mathbb{F}_p$ since only they are compatible with our construction.

B.0.2 Linear-Only Vector Encryption.

We recall the definition of a vector encryption scheme by Ishai et al. and then define the linear-only property required by the [BCI+13, BISW17] compiler. Let $\mathbb{F}$ be a finite field. A secret-key additively-homomorphic vector encryption scheme over a vector space $\mathbb{F}^l$ consists of a tuple of algorithms $\Pi_{Enc} = (\mathsf{Setup}, \mathsf{Encrypt}, \mathsf{Decrypt}, \mathsf{Add})$ with the following properties:

- $\mathsf{Setup}(1^\lambda, 1^l) \to (\mathsf{p}, \mathsf{sk})$: On input the security parameter λ and the plaintext dimension l, the setup algorithm outputs public parameters p and a secret key sk.
- $\mathsf{Encrypt}(\mathsf{sk}, v) \to \mathtt{ct}$: On input the secret key sk and a vector $v \in \mathbb{F}^l$, the encryption algorithm outputs a ciphertext $\mathtt{ct}$.
- $\mathsf{Decrypt}(\mathsf{sk}, \mathtt{ct}) \to v/\perp$: On input the secret key sk and a ciphertext $\mathtt{ct}$, the decryption algorithm either outputs a vector $v \in \mathbb{F}^l$ or a special symbol $\perp$.
- $\mathsf{Add}(\mathsf{p}, \{\mathtt{ct}_i\}_{i \in [n]}, \{c_i\}_{i \in [n]}) \to \mathtt{ct}^\star$: On input the public parameters, a collection of ciphertexts $\mathtt{ct}_1, \cdots, \mathtt{ct}_n$ and scalars $c_1, \cdots, c_n \in \mathbb{F}$, the addition algorithm outputs a new ciphertext $\mathtt{ct}^\star$.

Moreover, Π_{Enc} should be additively homomorphic and satisfy CPA security. Additionally, Π_{Enc} should satisfy the following property.

Linear-only homomorphic encryption (adapted from [ISW21]). An additively-homomorphic vector encryption scheme $\Pi_{Enc} = (\mathsf{Setup}, \mathsf{Encrypt}, \mathsf{Decrypt}, \mathsf{Add})$ is linear-only iff for any PPT adversary $\mathcal{A}$, there exists an PPT extractor $\texttt{Extr}$ that outputs $\Pi \in \mathbb{F}^l$ such that for any security parameter λ, auxiliary state $z \in \{0,1\}^{\mathsf{poly}(\lambda)}$, plaintext dimension l and plaintext generator $\mathcal{M}$

$$\Pr\left[\begin{array}{c|c} \mathsf{Decrypt}(\mathtt{ct}') \neq \perp & (\mathsf{p}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda, 1^l) \\ \wedge & (v_1, \ldots, v_k) \leftarrow \mathcal{M}(\mathsf{p}) \\ \mathsf{Decrypt}(\mathtt{ct}') \neq v_i \text{ for } i \in [k] & \mathtt{ct}_i \leftarrow \mathsf{Encrypt}(\mathsf{sk}, v_i) \text{ for } i \in [k] \\ & \mathtt{ct}' \leftarrow \mathcal{A}(\mathsf{p}, \mathtt{ct}_1, \ldots, \mathtt{ct}_k; z) \\ & \Pi \leftarrow \mathtt{Extr}(\mathsf{p}, \mathtt{ct}_1, \ldots, \mathtt{ct}_l; z) \\ & \mathtt{ct}' = [\mathtt{ct}_1, \ldots, \mathtt{ct}_l]^\top \Pi \end{array}\right] \leq \mathsf{negl}(\lambda).$$

B.0.3 SNARKs from Linear-Only Encryption.

We recall the Bitansky et al. [BCI+13] compiler for constructing SNARKs from linear PCPs and linear-only vector encryption (specifically the variant by Boneh et al. [BISW17] based on linear-only vector encryption) following Ishai et al. [ISW21].

Let $\mathcal{CS}$ be an R1CS instance over a finite field $\mathbb{F}$. The construction relies on the following building blocks:

- Let $\Pi_{LPCP} = (\mathsf{Q}_{LPCP}, \mathsf{P}_{LPCP}, \mathsf{V}_{LPCP})$ be a k-query LPCP for $\mathcal{CS}$. Let m be the query length of Π_{LPCP}.
- Let $\Pi_{Enc} = (\mathsf{Setup}_{Enc}, \mathsf{Encrypt}_{Enc}, \mathsf{Decrypt}_{Enc}, \mathsf{Add}_{Enc})$ be a linear-only additively-homomorphic vector encryption scheme for $\mathbb{F}^k$.

The single-theorem, designated-verifier SNARK $\Pi_{SNARK} = (\mathsf{Setup}, \mathsf{Prover}, \mathsf{Verifier})$ for RCS is defined as follows:

- $\mathsf{Setup}(1^\lambda, \mathcal{CS}) \rightarrow (\mathsf{crs}, \mathsf{st})$: On input the security parameter λ, run $(\mathsf{st}_{LPCP}, \boldsymbol{Q}) \leftarrow \mathsf{Q}_{LPCP}(\mathcal{CS})$ where $\boldsymbol{Q} \in \mathbb{F}^{m \times k}$. For each $i \in [m]$, let $q_i^\top \in \mathbb{F}^k$ denote the i th row of $\boldsymbol{Q}$. Then sample $(\mathsf{p}, \mathsf{sk}) \leftarrow \mathsf{Setup}_{Enc}(1^\lambda, 1^k)$ and compute $\mathtt{ct}_i \leftarrow \mathsf{Encrypt}_{Enc}(\mathsf{sk}, q_i^\top)$ for each $i \in [m]$. Output the common reference string $\mathsf{crs} = (\mathcal{CS}, \mathsf{p}, \mathtt{ct}_1, \cdots, \mathtt{ct}_m)$ and the verification state $\mathsf{st} = (st_{LPCP}, \mathsf{sk})$.
- $\mathsf{Prover}(\mathsf{crs}, \boldsymbol{x}, \boldsymbol{w}) \rightarrow \pi$: On input the common reference string $\mathsf{crs} = (\mathcal{CS}, \mathsf{p}, \mathtt{ct}_1, \cdots, \mathtt{ct}_m)$, a statement $\boldsymbol{x}$, and a witness $\boldsymbol{w}$, the prover constructs an LPCP proof $\boldsymbol{\pi} \leftarrow \mathsf{P}_{LPCP}(\mathcal{CS}, \boldsymbol{x}, \boldsymbol{w})$. The prover then homomorphically computes the linear PCP response $\mathtt{ct}^\star \leftarrow \mathsf{Add}_{Enc}(\mathsf{p}, \{\mathtt{ct}_1, \cdots, \mathtt{ct}_m\}, \{\pi_1, \cdots, \pi_m\})$. It outputs the proof $\pi = \mathtt{ct}^\star$.
- $\mathsf{Verifier}(\mathsf{st}, \boldsymbol{x}, \pi)$: On input the verification state $\mathsf{st} = (\mathsf{st}_{LPCP}, \mathsf{sk})$, the statement $\boldsymbol{x}$, and the proof $\pi = \mathtt{ct}^\star$, the verifier first decrypts $\boldsymbol{a} \leftarrow \mathsf{Decrypt}_{Enc}(\mathsf{sk}, \mathtt{ct}^\star)$. If $\boldsymbol{a} = \perp$, the verifier outputs 0. Otherwise, it outputs $\mathsf{V}_{LPCP}(\mathsf{st}_{LPCP}, \boldsymbol{x}, \boldsymbol{a})$.