Optimized Homomorphic Evaluation of Boolean Functions

Nicolas Bon[1,2], David Pointcheval[2] and Matthieu Rivain[1]

[1] CryptoExperts, Paris, France
firstname.lastname@cryptoexperts.com
[2] Ecole Normale Supérieure - PSL, Paris, France
firstname.lastname@ens.fr

Abstract. We propose a new framework to homomorphically evaluate Boolean functions using the Torus Fully Homomorphic Encryption (TFHE) scheme. Compared to previous approaches focusing on Boolean gates, our technique can evaluate more complex Boolean functions with several inputs using a single bootstrapping. This allows us to greatly reduce the number of bootstrapping operations necessary to evaluate a Boolean circuit compared to previous works, thus achieving significant improvements in terms of performances. We define theoretically our approach which consists in adding an intermediate homomorphic layer between the plain Boolean space and the ciphertext space. This layer relies on so-called p-encodings embedding bits into $\mathbb{Z}_p$. We analyze the properties of these encodings to enable the evaluation of a given Boolean function and provide a deterministic algorithm (as well as an efficient heuristic) to find valid sets of encodings for a given function. We also propose a method to decompose any Boolean circuit into Boolean functions which are efficiently evaluable using our approach. We apply our framework to homomorphically evaluate various cryptographic primitives, and in particular the AES cipher. Our implementation results show significant improvements compared to the state of the art.

Keywords: FHE · TFHE · Boolean Functions · Implementation

1 Introduction

Homomorphic encryption (HE) is a cryptographic technique allowing the computation of operations on encrypted messages (which directly reflect on the original messages once decrypted), using only knowledge of public data. For example, an additive homomorphic encryption scheme is able to encrypt two messages m_1 and m_2 in ciphertexts c_1 and c_2 and to compute a third ciphertext c_3 from c_1 and c_2 that encrypts the sum $m_1 + m_2$, without knowledge of the secret key.

The security of these schemes typically relies on a small *noise* introduced in the data when encrypting. The problem arising is that this noise is growing while homomorphic computations are carried out, which bury the original data into the noise and makes it unrecoverable at decryption. In 2009, Gentry [Gen09] introduced the operation of *bootstrapping* to solve this problem. This operation resets the noise at a nominal level *without decryption* allowing a potentially infinite amount of operations, making the construction of a scheme achieving *Fully Homomorphic Encryption* (FHE) possible. This operation being extremely heavy and slow, it is considered as the main bottleneck for the development of schemes efficient enough to be used in practice.

Currently, the most popular schemes in the FHE ecosystem are lattice-based and rely on the hardness of the Learning With Errors assumption [Reg05] and/or its ring variant

RLWE [LPR10]. BFV [Bra12], BGV [BGV12] and CKKS [CKKS17] are *leveled* schemes, which means that they keep track of the "level" of noise in the data during the homomorphic evaluation. As soon as this level reaches a critical bound, no more computations can be performed. Some recent works (see e.g. [CHK+18], [CCS19], [LLL+20]) propose a bootstrapping operation for these schemes to overcome this limit in the future. On the other hand, TFHE [CGGI18] is built on top of a powerful bootstrapping technique known to currently be the most efficient but limiting the precision of encrypted data.

Each FHE scheme offers a set of basic homomorphic operations that can be used to build more complex algorithms. In general, these operations are homomorphic additions and multiplications, however some complex operations cannot be constructed only with these operations. TFHE offers homomorphic additions and multiplications by a plaintext as well, but its force lies in its operation of *programmable bootstrapping* allowing the evaluation of encrypted look-up tables (LUT) while resetting the noise level. However, for performance issues, these look-up tables can only handle a small amount of bits as input (around 8 bits maximum) so the scheme is best suited for applications requiring a small precision.

In particular, TFHE is the best option to evaluate Boolean circuits with encrypted inputs, but the performances of the existing frameworks are still limited. In [CGGI18], the authors propose a strategy to evaluate Boolean functions called the *gate bootstrapping*, in which they perform one bootstrapping for each bivariate Boolean gate of the underlying circuit. As a consequence, the conversion of the original Boolean circuit in a homomorphic circuit handling encrypted bits is straightforward, moreover the noise growth is contained thanks to the systematic use of bootstrapping. However, this approach is very expensive due to the high numbers of bootstrappings and makes it highly suboptimal for large circuits.

The authors of [CLOT21] propose a different approach: by leveraging a newer version of the TFHE scheme supporting a new operation named *TLWE ciphertexts multiplication*, Boolean circuits are evaluated with homomorphic sums for `XOR` gates and this new multiplication operation for `AND` gates. While this approach is clearly a progress from the vanilla framework, we note that a few bootstrappings are still required to control the noise growth and that this new operation of TLWE multiplications remains costly both in terms of performances and in terms of noise. Thus, we choose to stick to the first version of the TFHE scheme (while slightly modifying it) to keep the framework lighter and we tackle the performance issues of [CGGI18] with a different approach than the one of [CLOT21].

Our work introduces a new framework to homomorphically evaluate Boolean functions on encrypted data efficiently, i.e. by reducing the amount of necessary bootstrappings. Our approach introduces an *intermediate homomorphic layer* which encodes bits on a small ring $\mathbb{Z}_p$ before encrypting them. This allows us to evaluate Boolean functions with one cheap homomorphic sum followed by one bootstrapping. After formalizing the underlying concept of *p-encoding* and explaining our evaluation strategy, we investigate the issue of finding valid sets of encodings for a Boolean function. We characterize this problem and provide an exact constructive algorithm to solve it. We further provide a sieving heuristic finding solutions more efficiently but at the cost of loosing optimally. Since our method is only efficient for Boolean functions with limited number of inputs, we also propose a heuristic to decompose any Boolean circuit into Boolean functions which are efficiently evaluable using our approach. Finally, we apply our technique to various cryptographic primitives, namely the SIMON block cipher, the Trivium stream cipher, the Keccak permutation, the Ascon s-box and the AES s-box. Compared to previous works implementing the same primitives (for SIMON, Trivium and AES) our implementations achieve significant speedups.

After some technical preliminaries on TFHE (Section 2), we introduce a new concept of *intermediate homomorphic layer* and explain how bits are encoded in Section 3 and the algorithms to construct it in Sections 4, 5. Finally, we describe our modifications of the

TFHE scheme in Section 6 and our experimental results in Section 7.

2 Preliminaries on TFHE

2.1 Notations

Let $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ be the real torus, that is to say the additive group of real numbers modulo 1. In practice, torus elements are not represented with an infinite number of digits, but are discretized. Let us denote this precision in base 2 as Ω. We can define the discretized torus $\mathbb{T}_q = \{\frac{a}{q} \mid a \in \mathbb{Z}_q\}$ (the elements of the torus up to Ω bits of precision, q being 2^Ω) and identify it with the ring $\mathbb{Z}_q$. As a consequence, any element $\frac{a}{q}$ of $\mathbb{T}_q$ will be represented in machine by a without any loss of property of the group $\mathbb{T}_q$. The operations of sum $+$ and external product $\cdot$ have to be understood modulo q.

Moreover, for a natural integer N and a given q, we will denote by $\mathbb{T}_{N,q}[X]$ the ring of polynomial $\mathbb{T}_q[X]/(X^N + 1)$. The elements of this ring are polynomials of maximum degree $N - 1$ and with coefficients in $\mathbb{T}_q$. Like for the scalar version, this ring will be identified with the ring $\mathbb{Z}_q/(X^N + 1)$. N is usually taken as a power of two.

Finally, we will denote by $\mathbb{B}$ the set of binary digits $\{0, 1\}$. & and $\oplus$ denote the `AND` and `XOR` binary operations. For x and $q \in \mathbb{Z}$, $[x]_q$ denotes the reduction of x modulo q. For S a set, $x \xleftarrow{\$} S$ denotes a uniformly random sampling from the set. For χ a distribution, $x \xleftarrow{\$} \chi$ denotes a random sampling according to the distribution.

2.2 Complexity Assumptions

The TFHE scheme, as other schemes using lattices, relies on the hardness of the LWE assumption. More precisely, it relies on the torus-based version of the problem. In the following, we consider the classic definition but over a discretized torus and with a binary secret:

Definition 1. (LWE problem over the discretized torus). Let $q, n \in \mathbb{N}$ and let $\mathbf{s} = (s_1, \dots, s_n) \xleftarrow{\$} \mathbb{B}^n$. Let χ be an error distribution over $\mathbb{Z}_q$. The *decisional Learning With Errors over discretized torus problem* is to distinguish samples chosen with the following distributions:

$$\mathcal{D}_0 = \{(\mathbf{a}, r) \mid \mathbf{a} \xleftarrow{\$} \mathbb{T}_q^n, r \xleftarrow{\$} \mathbb{T}_q\}$$

and:

$$\mathcal{D}_1 = \{(\mathbf{a}, b) \mid \mathbf{a} = (a_1, \dots, a_n) \xleftarrow{\$} \mathbb{T}_q^n, e \xleftarrow{\$} \chi, b = \sum_{j=1}^{n} a_j \cdot s_j + e\}$$

The *search* version of the problem is to recover $\mathbf{s}$ from the samples of $\mathcal{D}_1$.

Both the search and decisional problems are reducible to each other [Reg05] and their average case is as hard as worst-case lattice problems.

[Joy22] argues that identifying the discretized torus $\mathbb{T}_q$ as $\mathbb{Z}_q$ makes the LWE assumption over the discretized torus as hard as the standard LWE assumption.

TFHE relies as well on the generalized version of LWE over rings introduced in [BGV12] named GLWE.

Definition 2. (GLWE problem over the discretized torus). Let $N, q, k \in \mathbb{N}$ with N a power of two and let $\mathbf{s} = (s_1, \dots, s_k) \xleftarrow{\$} \mathbb{B}_N[X]^k$. Let χ be an error distribution over

$\mathbb{Z}_{N,q}[X]$. The *General decisional Learning With Errors over discretized torus problem* is to distinguish samples chosen with the following distributions:

$$\mathcal{D}_0 = \{(\mathbf{a}, r) \mid \mathbf{a} \xleftarrow{\$} \mathbb{T}_{N,q}[X]^k, r \xleftarrow{\$} \mathbb{T}_{N,q}[X]\}$$

and:

$$\mathcal{D}_1 = \{(\mathbf{a}, b) \mid \mathbf{a} = (a_1, \dots, a_k) \xleftarrow{\$} \mathbb{T}_{N,q}[X]^k, e \xleftarrow{\$} \chi, b = \sum_{j=1}^{k} a_j \cdot s_j + e\}$$

The *search* version is analogous to the LWE one.

Note that RLWE is simply an instantiation of GLWE with $k = 1$.

The complexity analysis is analogous to the LWE version. In practice, the error distribution χ is a centered Gaussian distribution parametrized by its standard deviation σ.

2.3 Plaintext Space

Before expliciting more in depth the TFHE scheme, it is useful to define the *plaintext space* and how it is embedded in the discretized torus.

The plaintext space is the ring $\mathbb{Z}_p$, with $p \in \mathbb{N}$. For now, let us assume that $p \mid q$ and identify $\mathbb{Z}_p$ with $\mathbb{T}_p$. As $p \mid q$, all elements of $\mathbb{T}_p$ are elements of $\mathbb{T}_q$ as well. Thus, we can define a mapping $\rho : \mathbb{Z}_p \to \mathbb{Z}_q$ as $\rho : m \mapsto \frac{mq}{p}$.

Of course, only p elements of $\mathbb{Z}_q$ are reached by such a mapping and they have the form $\left\{\frac{kq}{p} \mid k \in \mathbb{Z}_p\right\}$. As they are evenly distributed across $\mathbb{Z}_q$, they define what we call *sectors of* $\mathbb{Z}_q$ of the form:

$$\left\{ \left(\frac{(2k-1)q}{2p}, \frac{(2k+1)q}{2p} \right) \mid k \in \mathbb{Z}_p \right\} .$$

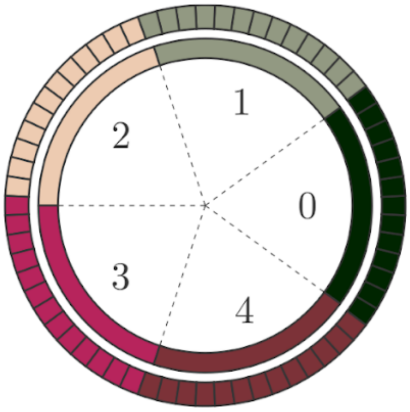


Figure 1: Embedding of $\mathbb{Z}_p$ in $\mathbb{Z}_q$

The embedding of $\mathbb{Z}_p$ in $\mathbb{Z}_q$ is illustrated in Figure 1.

During encryption of m, some small noise e is drawn from a Gaussian distribution over $\mathbb{Z}_q$ and is added to m. As e is small, the noisy message $m + e$ stays in the same sector as m but while homomorphic operations are carried out, the noise grows and may overflow out of the sector. When decrypting, one recovers the sum of the expected result and some noise $m' + e'$. As long as $e' < \frac{q}{2p}$, the message m' can be recovered by rounding to the closest center of sector.

In our work, we pick odd values for p. q being a power of 2 in practice, it implies that p does not divide q. This enables nice features explained in Section 6. Consequently, the centers and the bounds of sectors are computed by rounding the fractions to the closest integers. In practice, p is much smaller than q (p is restricted to a few bits, while q typically equals 2^{32} or 2^{64}), so this discrepancy makes this approximation sound. In the following, we will ignore this rounding.

2.4 Ciphertexts Types and Basic Operations

TFHE manipulates several different types of ciphertexts. In the following, we explain their structure:

- **TLWE ciphertexts**: The message m to be encrypted is encoded as an element of $\mathbb{T}_q$. A mask $\mathbf{a} = (a_1, \ldots, a_n)$ is drawn uniformely from $\mathbb{T}_q^n$ and a noise error e is sampled from χ. Using the secret key $\mathsf{sk} = (s_1, \ldots, s_n) \in \mathbb{B}^n$, the body of the ciphertext is defined by $b = \sum_{j=1}^{n} a_j \cdot s_j + m + e$. Finally, the TLWE ciphertext is $c = (\mathbf{a}, b)$. The decryption is performed by calculating the *phase*: $\phi(c) = b - \langle \mathbf{a}, \mathbf{s} \rangle = m + e$ and rounding to the closest center of sector.

- **TRLWE ciphertexts**: It has the same global structure as TLWE ones, except the mask $\mathbf{a}$ is sampled from $\mathbb{T}_{N,q}[X]^k$, the secret key from $\mathbb{B}[X]^k$ and the error from $\mathbb{T}_{N,q}[X]$. Some papers in the literature use the denomination TRLWE only if $k = 1$, and TGLWE otherwise. In this work, we do not make a difference between both cases.

During the bootstrapping phase presented in Section 2.5, another structure (the TRGSW ciphertext) is used but we do not mention it as we will not need it. More details about TRGSW can be found in [Joy22].

Two basics homomorphic operations are straightforward with these two structures: the component-wise sum of two TLWE (resp. TRLWE) ciphertexts c_1 and c_2 produces a ciphertext c_3 encrypting the sum modulo p of the two underlying messages m_1 and m_2. Moreover, the external product $\lambda \cdot c_1$ with $\lambda \in \mathbb{Z}$ also produces an encryption of the multiplication $[\lambda \cdot m_1]_p$.

In the framework introduced by this paper, the freshly encrypted ciphertexts are TLWE, as well as during homomorphic computations. We only manipulate TRLWE ciphertexts during the `BlindRotate` phase of the bootstrapping, presented in Section 2.5.

2.5 TFHE programmable bootstrapping (PBS)

As defined by Gentry in [Gen09], the procedure of bootstrapping can be defined as the homomorphic evaluation of the decryption circuit. In the context of TFHE, the hardest part to compute is the rounding of the value to an element of $\mathbb{T}_p$ by removing the noise. To achieve this homomorphically, it uses four procedures called `ModulusSwitch`, `BlindRotate`, `SampleExtract` and `KeySwitch`.

`ModulusSwitch`: The high level idea starts by homomorphically computing the phase $\mu \in \mathbb{Z}_q$ and reducing it to $\tilde{\mu} \in \mathbb{Z}_{2N}$ by computing $\tilde{\mu} = \left\lfloor \frac{\mu \cdot 2N}{q} \right\rceil$. In practice N takes values between 2^{10} and 2^{13} so the most significant bits carrying the true value modulo p are preserved.

`BlindRotate`: Then, for a polynomial $v(X) \in \mathbb{Z}_{N,q}[X]$, called the *accumulator*, one homomorphically multiplies $v(X)$ by $X^{-\tilde{\mu}}$ by *blind rotation* which yields an encryption of the polynomial $v_{\tilde{\mu}} + v_{\tilde{\mu}+1}X + \cdots \in \mathbb{Z}_{N,q}[X]$. By defining $v_j := \frac{1}{p} \left\lfloor \frac{jp}{2N} \right\rceil \, \forall j$, the blind rotation shall output an encrypted version of the message in the zero-degree coefficient. We do not explain here how this polynomial multiplication occurs, the reader is referred to [CGGI18] for a more elaborated explanation. The procedure outputs a TRLWE ciphertext of dimension k encrypting the polynomial $X^{-\tilde{\mu}} \cdot v(X)$. Note that the quotient polynomial of the ring has degree N but $\tilde{\mu}$ lives in $\mathbb{Z}_{2N}$ so each coefficient of v_i can be reached with a multiplication by $X^{-\tilde{\mu}}$ *and* by $X^{[N-\tilde{\mu}]_{2N}}$. In the latter case, the coefficient v_i gets negated because of the ring modulus $X^N + 1$: we will refer to this problem as the *negacyclicity problem*. One way to prevent this issue is to ensure that the most significant bit of μ is fixed at 0 [Joy22] but a recent work [CLOT21] proposes a more sophisticated way to solve this problem. In our case, we use a modified version of the accumulator detailed in Section 6.

`SampleExtract`: This step simply extracts the degree-zero coefficient of the previous polynomial. It takes as input the TRLWE ciphertext yielded by the `BlindRotate` step and outputs the TLWE ciphertext c' encrypting the original message m. However, this ciphertext is not immediately available for either further homomorphic computations or decryption, because it has a length $kN+1$ instead of $n+1$ (and as a consequence is encrypted under a different TLWE key).

`KeySwitch`: The previous step outputs the right value, but encrypted under a different set of parameters i.e. $c' \in \mathbb{Z}_q^{kN+1}$ while we are looking for $c \in \mathbb{Z}_q^n$. The only thing left is to convert c' to c, which requires *key switching keys* constructed from the secret key `sk` used at encryption. More details about this specific step can also be found in [CGGI18].

This "bland" procedure of bootstrapping simply refreshes the noise in the ciphertext to put it back at the "initial level", but can be very simply turned into a *Programmable* bootstrapping. Specifically it can simultaneously evaluate homomorphically any function f on the input. To achieve this, at the construction of the accumulator, the coefficient v_j is replaced by their evaluation by the function $f(v_j)$. This feature is extremely powerful and is the core of the huge potential of TFHE.

2.6 Basics on Boolean Functions and Boolean Circuits

In this paper, we focus on the evaluation of Boolean functions with TFHE. A Boolean function has the form $f : \mathbb{B}^\ell \longrightarrow \mathbb{B}$, with ℓ being called the *arity* of the function.

Definition 3. The Algebraic Normal Form (ANF) of a Boolean function $f : \{0,1\}^\ell \mapsto \{0,1\}$ is a polynomial expression in which each term corresponds to a specific input combination of n variables. The ANF is defined as follows:

$$f(x_1, x_2, \dots, x_l) = a_0 \oplus a_1 x_1 \oplus a_2 x_2 \oplus \dots \oplus a_{2^n-1} x_1 x_2 \dots x_l$$

$$\text{where: } a_0, a_1, a_2, \dots, a_{2^\ell - 1} \in \{0,1\} \quad \text{are the Boolean coefficients and}$$
$$x_1, x_2, \dots, x_\ell \quad \text{are called the Boolean variables}$$

This result means that any Boolean function can be evaluated by the means of `AND` and `XOR` operations. In the following, we will focus on the implementation of Boolean circuits composed of these operations exclusively.

A Boolean function can be represented by its *truth table*, which is simply a table gathering all the possible inputs and the corresponding result of the application by the function. It can also be represented with a Boolean formula. A third representation is the *Boolean circuit*:

Definition 4. A Boolean circuit associated to the Boolean function f is a finite Directed Acyclic Graph whose edges are *wires* and vertices are *Boolean gates* representing Boolean operations. We consider `AND` gates and `XOR` gates, of fan-in 2 and fan-out 1. We also consider copy gates, of fan-in 1 and fan-out > 1, that outputs several copies of its input. A circuit is further formally composed of input gates of fan-in 0 and fan-out 1, and output gates of fan-in 1 and fan-out 0.

Evaluating a ℓ-input m-output circuit consists in writing an input $\mathbf{x} \in \mathbb{B}^\ell$ in the input gates, processing the gates from input gates to output gates, then reading the outputs from the output gates.

This notion of Boolean circuit will be particularly useful in Section 5.

3 Boolean Encoding over $\mathbb{Z}_p$ and Homomorphic Evaluation Strategy Between $\mathbb{B}$ and $\mathbb{Z}_p$

To evaluate Boolean functions in TFHE, one could use the vanilla TFHE with $p = 2$. The problem is that the only evaluable function would be the `XOR` operation. To evaluate the other operators, the solution of [CGGI18] which is also implemented in the `tfhe-rs` library [Zam22b] is to take a larger p, specifically $p = 8$. This allows all the operations of the Boolean algebra to be carried out, however the negacyclicity problem introduced in Section 2.5 arises because 8 is even. Their solution to this issue is to keep a *bit of padding* fixed to zero, i.e., the values in $\mathbb{Z}_p$ have their most significant bit fixed to zero. This restriction has a heavy impact on performances, because it requires a bootstrapping after each Boolean gate to make sure no data ever overflows in the most significant bit.

Our solution makes use of *odd* values for p, which allows us to remove this constraint of padding and to perform more operations without bootstrapping. To do so, we had to slightly adapt the bootstrapping procedure of TFHE to support odd moduli. We explain this tweak in Section 6.

Moreover, the PBS described in Section 2.5 takes only one input and so can only evaluate univariate functions. The common solution to evaluate multivariate functions is to concatenate several input ciphertexts into one by shifting the MSB of each input and to sum them all. The problem is that the number of message bits cannot grow too much because the other parameters of the LWE problem must grow accordingly, degrading the performances. As a consequence, the performances quickly degrades as the arity of the function increases. Our approach consists in removing the padding bit and using a combination of homomorphic additions before a PBS to evaluate a function for *any* number of inputs with the cost of a single PBS.

To this purpose, we propose a construction in which we embed Boolean values in $\mathbb{Z}_p$ for well-chosen values of p, forming an "intermediate homomorphic layer" between $\mathbb{B}$ and $\mathbb{Z}_q$. In the following, we explain how we define such a layer, and we describe our new strategy to evaluate Boolean functions in a more efficient way without considering the circuit representation of the function.

3.1 Encoding of $\mathbb{B}$ over $\mathbb{Z}_p$

To represent Boolean values over $\mathbb{Z}_p$, we use a mapping function that we call a *p-encoding*:

Definition 5 (p-encoding)**.** A *p-encoding* is a function $\mathcal{E} : \mathbb{B} \mapsto 2^{\mathbb{Z}_p}$ that maps the Boolean space to a subset of the discretized torus. A p-encoding is *valid* if and only if:

$$\begin{cases} \mathcal{E}(0) \cap \mathcal{E}(1) = \emptyset \quad \text{and} \\ \text{if } p \text{ is even: } \forall\, x \in \mathbb{Z}_p, \forall\, b \in \mathbb{B} : x \in \mathcal{E}(b) \iff \left[x + \frac{p}{2}\right]_p \notin \mathcal{E}(b) \end{cases} \tag{1}$$

We call this last property *relaxed negacyclicity.*

In our approach when we need to encrypt a bit, we apply a p-encoding to embed it on $\mathbb{Z}_p$, then we encrypt the result using the classical setup of TFHE. When new values are freshly encrypted or produced by a PBS, only one element of $\mathbb{Z}_p$ is chosen for each bit. We call such an encoding a canonical p-encoding:

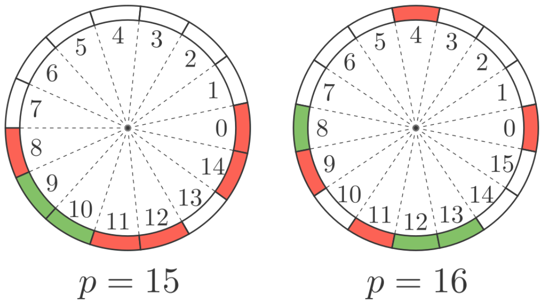


Figure 2: Representation of two valid p-encodings. The green part represents $\mathcal{E}(1)$, and the red part $\mathcal{E}(0)$. Note that the relaxed negacyclity is respected by the p-encoding on the right-hand figure as p is even.

Definition 6 (Canonical Encoding)**.** A p-encoding $\mathcal{E}$ is said *canonical* if and only if it is valid and $|\mathcal{E}(0)| = |\mathcal{E}(1)| = 1$

Let c be a ciphertext encoding a bit b under a p-encoding $\mathcal{E}$, where $\mathcal{E}$ is an arbitrary valid encoding: its associated subsets can be any subset of $\mathbb{Z}_p$ as long as the validity requirements of (1) are fulfilled. One can transform the ciphertext c into another ciphertext c' encoded under any *canonical* p-encoding, possibly under a different p, by simply performing a PBS.

Property 1 (Reduction to a canonical encoding)**.** Let $\mathcal{E}$ be a valid p-encoding and $\mathcal{E}'$ a canonical p'-encoding. We denote $\alpha' = \mathcal{E}'(0)$ and $\beta' = \mathcal{E}'(1)$. Let c be a ciphertext encrypting a bit b under $\mathcal{E}$. Then, one can produce a ciphertext c' encrypting the same bit b under $\mathcal{E}'$ by applying a PBS on c. This PBS performs the function :

$$\begin{aligned}\texttt{Cast}_{\mathcal{E}\mapsto\mathcal{E}'} \ &: \ \mathbb{Z}_p \mapsto \mathbb{Z}_{p'} \\ x &\mapsto \begin{cases} \alpha' & \text{if } x \in \mathcal{E}(0) \\ \beta' & \text{if } x \in \mathcal{E}(1) \\ \perp & \text{otherwise.} \end{cases}\end{aligned}$$

Here, $\perp$ simply denotes a placeholder value for a state that cannot be reached.

Our goal is to represent the Boolean function we want to evaluate with a sum of p-encodings (we define what we mean by "sum of p-encoding" in Section 3.2). When sums are carried out on ciphertexts (and so homomorphically on the underlying p-encodings), the sets $\mathcal{E}(0)$ and $\mathcal{E}(1)$ of the p-encodings may move, grow, shrink, but they should never overlap as it would result in a loss of information. As we removed the need of a bit of padding, we do not need to track a potential overflow of data (informally we say that ciphertexts are free to "go around the torus"). After the sum, the encoding of the result can be reset to a canonical one with a PBS to allow further computation. Or, if the homomorphic computation is over, the result can be recovered by decrypting the ciphertext and checking in which partition the decrypted value lies.

The next subsection explains in further details the process of evaluating Boolean functions on with p-encodings.

3.2 A New Strategy for Homomorphic Boolean Evaluation

In the following, we consider two Boolean variables x and y and their two respective encodings over $\mathbb{Z}_p$:

$$\mathcal{E}_x = \begin{cases} 0 \mapsto \{\alpha_i\}_{0\le i\le l_0} \\ 1 \mapsto \{\beta_i\}_{0\le i\le l_1} \end{cases} \quad \text{and } \mathcal{E}_y = \begin{cases} 0 \mapsto \{\alpha'_i\}_{0\le i\le l'_0} \\ 1 \mapsto \{\beta'_i\}_{0\le i\le l'_1} \end{cases} \tag{2}$$

Let f be a bivariate Boolean function and let us construct two sets $\mathcal{P}_0$ and $\mathcal{P}_1$ such that:

$$\mathcal{P}_b = \{[\gamma+\delta]_p \mid (\gamma,\delta) \in \mathcal{E}_x(b_x) \times \mathcal{E}_y(b_y) \text{ and } f(b_x, b_y) = b \text{ with } (b_x, b_y) \in \mathbb{B}^2\} \ \forall\, b \in \mathbb{B}. \quad (3)$$

We say that the sum of p-encodings $\mathcal{E}_x + \mathcal{E}_y$ is *suitable for the evaluation of f* if and only if $\mathcal{P}_0 \cap \mathcal{P}_1 = \emptyset$. The definition can be generalized to any number of arguments ℓ for f. For a given f, finding such correct encodings is not trivial. We elaborate further on this point in Section 4.

If $\mathcal{E}_x$ and $\mathcal{E}_y$ are suitable for f, then one can use the computed sets $\mathcal{P}_b$ to construct a new p-encoding

$$\mathcal{E}_z = \begin{cases} 0 \mapsto \mathcal{P}_0 \\ 1 \mapsto \mathcal{P}_1 \end{cases}$$

that encodes the bit $f(x, y)$. As $\mathcal{E}_z$ is valid, then the clear value of the bit can be recovered by decryption, or further computations can be performed without the need of a bootstrapping.

Definition 7 (Application of a function to a vector of encodings)**.** Let $f : \mathbb{B}^\ell \mapsto \mathbb{B}$ be a Boolean function and let $\mathcal{E} = (\mathcal{E}_1, \ldots, \mathcal{E}_l)$ be a vector of p-encodings. We define $f(\mathcal{E})$ by:

$$f(\mathcal{E}) = \begin{cases} 0 \mapsto \mathcal{P}_0 \\ 1 \mapsto \mathcal{P}_1 \end{cases}$$

with:

$$\mathcal{P}_b = \left\{ \left[\sum_{i=1}^{l} \gamma_i\right]_p \mid (\gamma_1, \ldots, \gamma_l) \in \prod_{i=1}^{\ell} \mathcal{E}_i(b_i) \text{ and } f(b_1, \ldots, b_l) = b \right\} \forall b \in \mathbb{B}$$

We stress that $f(\mathcal{E})$ is a valid p-encoding if and only of $\mathcal{P}_0 \cap \mathcal{P}_1 = \emptyset$.

Let us illustrate the latter definition on two toy example. We consider the two Boolean operators & and $\oplus$. The p-encoding resulting of the function $f : (x, y) \mapsto x \ \& \ y$ is:

$$\mathcal{E}_{\&} = \begin{cases} 0 \mapsto \{\alpha_i + \alpha'_j\}_{\substack{0 \le i \le l_0 \\ 0 \le j \le l'_0}} \cup \{\alpha_i + \beta'_j\}_{\substack{0 \le i \le l_0 \\ 0 \le j \le l'_1}} \cup \{\alpha'_i + \beta_j\}_{\substack{0 \le i \le l'_0 \\ 0 \le j \le l_1}} \\ 1 \mapsto \{\beta_i + \beta'_j\}_{\substack{0 \le i \le l_1 \\ 0 \le j \le l'_1}} \end{cases} \quad (4)$$

and the p-encoding resulting of the operation $f : (x, y) \mapsto x \oplus y$ is:

$$\mathcal{E}_{\oplus} = \begin{cases} 0 \mapsto \{\alpha_i + \alpha'_j\}_{\substack{0 \le i \le l_0 \\ 0 \le j \le l'_0}} \cup \{\beta_i + \beta'_j\}_{\substack{0 \le i \le l'_0 \\ 0 \le j \le l'_1}} \\ 1 \mapsto \{\alpha_i + \beta'_j\}_{\substack{0 \le i \le l_0 \\ 0 \le j \le l'_1}} \cup \{\alpha'_i + \beta_j\}_{\substack{0 \le i \le l'_0 \\ 0 \le j \le l_1}} \end{cases} \quad (5)$$

Figure 3 further illustrates this construction for these two operations.

To wrap up, here is our proposed framework to evaluate a Boolean function $f : \mathbb{B}^\ell \mapsto \mathbb{B}$ given a vector of suitable p-encodings $\mathcal{E} = (\mathcal{E}_1, \ldots, \mathcal{E}_l)$:

1. Encrypt each input b_i with some canonical p-encoding $\mathcal{E}_i$ into a ciphertext c_i such that $\mathcal{E}_{sum} = f(\mathcal{E}_1, \ldots, \mathcal{E}_\ell)$ is a valid encoding.

2. For a Boolean function f to be evaluated on $b_1, \ldots, b_l$, compute homomorphically the sum of the ciphertexts $c = c_1 + \cdots + c_l$. This yields an encryption of $b = f(b_1, \ldots, b_l)$, encoded with a valid p-encoding $\mathcal{E}_{sum} = f(\mathcal{E}_1, \ldots, \mathcal{E}_l)$.

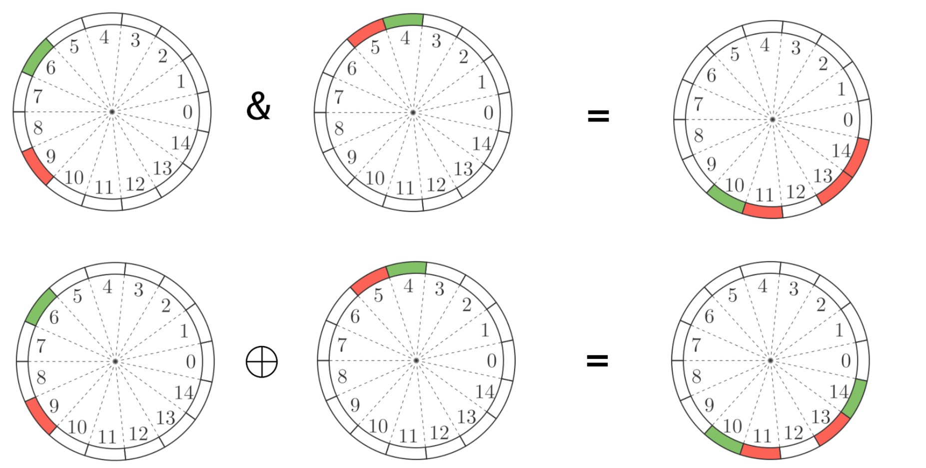


Figure 3: Starting from two canonical encodings, we produce two new p-encodings corresponding to the results of the & and the $\oplus$ operations.

3. (a) If the result is directly required by the client, c is returned as ciphertext which can be decrypted to get the result in $\mathbb{Z}_p$ and associated to the right Boolean value.

   (b) If the result is reused in further homomorphic computations, a PBS calculating $\texttt{Cast}_{\mathcal{E}_{sum} \mapsto \mathcal{E}_{new}}$ on the result is computed (like introduced in Property 1), with $\mathcal{E}_{new}$ a new canonical p-encoding. The resulting value can then be used as an input for a next Boolean function.

Let us formalize this process by defining the notion of *gadget associated to a Boolean function f* :

Definition 8 (Gadget)**.** Let f be a Boolean function of arity ℓ. A gadget associated to f is an homomorphic operator defined by a tuple $\Gamma = \left(\mathcal{E}_{\mathbf{in}} = (\mathcal{E}_{in}^{(1)}, \dots, \mathcal{E}_{in}^{(\ell)}), \mathcal{E}_{out}, p_{in}, p_{out}\right)$ such that:

- All the elements of $\mathcal{E}_{\mathbf{in}}$ are p_{in}-encodings, and $\mathcal{E}_{out}$ is a canonical p_{out}-encoding.
- The encoding $\mathcal{E}_{sum} = f(\mathcal{E}_{in}^{(1)}, \dots, \mathcal{E}_{in}^{(\ell)})$ is a valid encoding.

Applying a gadget to ciphertexts $c1, \dots, c_\ell$, that encrypt the bits $b_1, \dots, b_\ell$, produces a new ciphertext c' encrypting the bit $f(b_1, \dots, b_\ell)$ under the p_{out}-encoding $\mathcal{E}_{out}$. To do so, we perform the following algorithm:

- Constructing an intermediate ciphertext $c_{inter} = \sum_{i=1}^{\ell} c_i$ using the homomorphic sum of TFHE. This ciphertext encrypts $f(b_1, \dots, b_\ell)$ under the p_{in}-encoding $f(\mathcal{E}_1, \dots, \mathcal{E}_\ell)$.
- Reducing the encoding of c_{inter} from $\mathcal{E}_{inter}$ to $\mathcal{E}_{out}$ by applying a PBS on c_{inter} performing the function $\texttt{Cast}_{\mathcal{E}_{inter} \mapsto \mathcal{E}_{out}}$. This produces the expected result c'.

The advantage of this construction is that only one PBS is performed to apply the function. Moreover, depending on the function, the input size of the PBS lookup table might be much smaller than the arity of the function. Gadgets can be seen as a way to compress several Boolean operators into a single evaluation of univariate look-up table. Of course, for a given p_{in} and a given f, such a gadget may not exist. In such a case, two solutions can be considered:

- Increasing the value of p_{in} (e.g. taking $p_{in} \geq 2^\ell$ always works, but is very inefficient).
- Splitting the function into a graph of subfunctions, and evaluating each one with a gadget.

The question of constructing valid gadgets for a given f is treated in Section 4. The question of efficiently splitting a function is treated in Section 5.

Example: We illustrate our approach with a simple working example: let f be a basic multiplexing function, such that

$$f(a,b,c) = \begin{cases} a \text{ if } c = 1 \\ b \text{ if } c = 0 \end{cases}$$

Instead of leveraging its Boolean representation $f(a,b,c) = a\&c \oplus b\&\bar{c}$, which would cost 3 PBS with the approach of [CGGI18], our strategy consists in constructing a gadget and apply it to the inputs a, b and c, which takes only one PBS. Here is the step-by-step procedure:

1. Encrypting the bits with the 7-encodings:

$$\mathcal{E}_a = \mathcal{E}_b = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{1\} \end{cases} \quad \text{and } \mathcal{E}_c = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{2\} \end{cases}$$

.

2. Applying the function f on the 7-encodings by summing the ciphertexts, producing a valid 7-encoding:

$$\mathcal{E}_{sum} = \begin{cases} 0 \mapsto \{0,1,2,5\} \\ 1 \mapsto \{3,4,6\} \end{cases}$$

At this point, only sums have been performed on the ciphertexts.

3. With one PBS, resetting the result to a target canonical p-encoding (with any p), for example

$$\mathcal{E}_{new} = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{1\} \end{cases} \quad \text{with } p = 7$$

A visualization of this procedure can be found in Figure 4. We just defined the gadget $\Gamma = ((\mathcal{E}_a, \mathcal{E}_b, \mathcal{E}_c), \mathcal{E}_{new}, 7, 7)$.

3.3 Encoding Switching

To apply a gadget to a given ciphertext, it has to be encrypted under the right encoding. Thus, we need a method to homomorphically switch the encoding of a ciphertext. This allows as well to plug the output of any gadget on the input of any other one, and so to evaluate a chain of gadgets as long as we want. In the following, we explore different possibilities of encoding switching. Let us begin with some trivial cases:

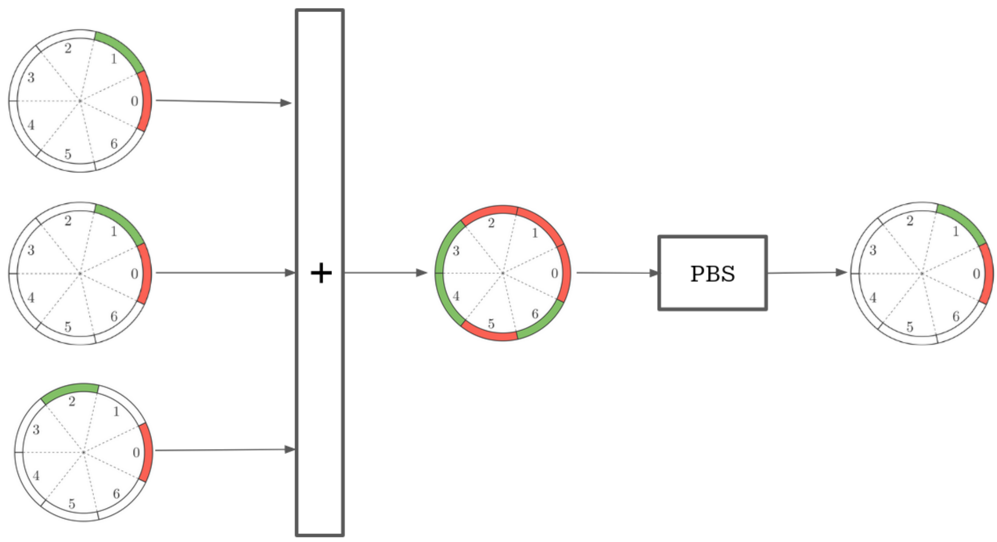


Figure 4: Illustration of an execution of the framework for the multiplexing function.

Property 2 (Encoding switching with a sum by a constant)**.** Let x be a ciphertext encoded under $\mathcal{E}_x = \begin{cases} 0 \mapsto \{\alpha_i\}_{0\leq i\leq l_0} \\ 1 \mapsto \{\beta_i\}_{0\leq i\leq l_1} \end{cases}$ and $a \in \mathbb{Z}_p$ a constant. The encoding of x can be switched to:

$$\mathcal{E}'_x = \begin{cases} 0 \mapsto \{[\alpha_i + a]_p\}_{0\leq i\leq l_0} \\ 1 \mapsto \{[\beta_i + a]_p\}_{0\leq i\leq l_1} \end{cases}$$

by an homomorphic addition of the ciphertext x and the clear value a.

Proof. All the elements of $\mathcal{E}'_x(0)$ (resp. $\mathcal{E}'_x(1)$) are offset by exactly a from their counterparts in $\mathcal{E}_x(0)$ (resp. $\mathcal{E}(1)$). Thus, if the original encoding $\mathcal{E}_x$ was valid, then $\mathcal{E}_x(0) \cap \mathcal{E}_x(1) = \emptyset$. So we trivially get $\mathcal{E}'_x(0) \cap \mathcal{E}'_x(1) = \emptyset$ and thus the validity of $\mathcal{E}'_x$. $\square$

Property 3 (Encoding switching with multiplication by a constant)**.** Let x be a ciphertext encoded under $\mathcal{E}_x = \begin{cases} 0 \mapsto \{\alpha_i\}_{0\leq i\leq l_0} \\ 1 \mapsto \{\beta_i\}_{0\leq i\leq l_1} \end{cases}$ and $a \in \mathbb{Z}_p$ a constant value prime with p. The encoding of x can be switched to:

$$\mathcal{E}'_x = \begin{cases} 0 \mapsto \{[a \cdot \alpha_i]_p\}_{0\leq i\leq l_0} \\ 1 \mapsto \{[a \cdot \beta_i]_p\}_{0\leq i\leq l_1} \end{cases}$$

by an homomorphic multiplication of the ciphertext x by the clear value a.

Proof. As a is prime with p, the multiplication by a is a bijection from $\mathbb{Z}_p$ to $\mathbb{Z}_p$. By definition, all the α_i's are different of the β_i's. If we apply a bijection on them, the inequalities are conserved. $\square$

Note that the condition of primality between a and p is a sufficient condition for the multiplication to be a valid encoding switching, but is not necessary. In particular, one other case is particularly useful in practice:

Property 4 (Encoding switching for a canonical encoding containing a zero)**.** Let x be a ciphertext encoded under the p-encoding: $\mathcal{E}_x = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{1\} \end{cases}$ and let $a \in \mathbb{Z}_p \setminus \{0\}$. Then, it can be switched to: $\mathcal{E}'_x = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{a\} \end{cases}$ by a simple homomorphic multiplication between the ciphertext x and the clear value a. This holds as well if $\mathcal{E}(0)$ and $\mathcal{E}(1)$ swapped.

Proof. The property is trivial by the linear homomorphism of the TFHE scheme. □

These techniques are powerful because they do not require any bootstrapping, so they can be considered as free in terms of performances. However, any valid p-encoding can be turned into any other one with a programmable bootstrapping, even with a different modulus p. A reduced version of this is given by Property 1, but it can be extended to any valid output p-encoding.

Property 5 (Arbitrary encoding switching with a PBS)**.** Let c be a ciphertext encoded under $\mathcal{E}$. Its encoding can be switched to $\mathcal{E}'$ (even with a different modulus p') by applying a PBS on c evaluating the function

$$\texttt{Cast}_{\mathcal{E}\mapsto\mathcal{E}'} \ : \ \mathbb{Z}_p \mapsto \mathbb{Z}_{p'} \tag{6}$$

$$x \mapsto \begin{cases} \alpha' \in \mathcal{E}'(0) & \text{if } x \in \mathcal{E}(0) \\ \beta' \in \mathcal{E}'(1) & \text{if } x \in \mathcal{E}(1) \\ \perp & \text{otherwise.} \end{cases} \tag{7}$$

Here, $\perp$ simply denotes an arbitrary placeholder value, as it will never be reached.

See Sections 2.5 and 6.2 for a more in-depth insight on the actual procedure of programmable bootstrapping.

4 Algorithms of construction of gadgets

Let $f : \mathbb{B}^\ell \mapsto \mathbb{B}$ a Boolean function with ℓ entries. This section addresses the problem of constructing a gadget for f. To do so, we pick a value for p and we search a vector of ℓ p-encodings $\mathcal{E}_{\mathbf{in}}$ suitable for f.

4.1 Reduction of the Search Space

While exhaustive search is a first option, it quickly becomes impractical due to the explosion of the number of possibilities as p grows. As a consequence, a reduction of the search space is needed without leaving out a potential solution.

We introduce two lemmas that will be used to reduce the search space:

Lemma 1 (Reducibility to singletons)**.** *Let $f : \mathbb{B}^\ell \longrightarrow \mathbb{B}$ and let $(\mathcal{E}_1, \ldots, \mathcal{E}_l)$ be a vector of p-encodings suitable for f and having the form: $\forall\ i \in \{1, \ldots, \ell\}, \mathcal{E}_i = \begin{cases} 0 \mapsto \{x_j^{(i)}\}_{1\leq j\leq l_0^{(i)}} \\ 1 \mapsto \{y_j^{(i)}\}_{1\leq j\leq l_1^{(i)}} \end{cases}$. Then any vector of canonical p-encodings $(\mathcal{E}'_1, \ldots, \mathcal{E}'_l)$ of the form: $\forall\ i \in \{1, \ldots, \ell\}, \mathcal{E}'_i = \begin{cases} 0 \mapsto \{x^{(i)}\} \\ 1 \mapsto \{y^{(i)}\} \end{cases}$ with $x^{(i)} \in \mathcal{E}_i(0)$ and $y^{(i)} \in \mathcal{E}_i(1)$ is suitable for the function f as well.*

Proof. Let us assume that the vector $\mathcal{E} = (\mathcal{E}_1, \ldots, \mathcal{E}_l)$ of Lemma 1 is suitable for the function f. Then the sets $\mathcal{P}_0$ and $\mathcal{P}_1$ constructed like in Equation 3 are disjoint. Now, let us consider the vector of canonical p-encodings $\mathcal{E}' = (\mathcal{E}'_1, \ldots, \mathcal{E}'_l)$ respecting the property:

$$\forall\ b\ \in \mathbb{B}, \forall\ i \in \{0, \ldots, \ell\}, \mathcal{E}'_i(b) \subset \mathcal{E}_i(b).$$

As a consequence, if we build the sets $\mathcal{P}'_0$ and $\mathcal{P}'_1$ relative to the encodings $\mathcal{E}'$, then we naturally get $\mathcal{P}'_0 \subset \mathcal{P}_0$ and $\mathcal{P}'_1 \subset \mathcal{P}_1$. So we get $\mathcal{P}'_0 \cap \mathcal{P}'_1 = \emptyset$, proving Lemma 1. □

Lemma 2 (Reducibility to the singleton zero)**.** *Let $f : \mathbb{B}^\ell \longrightarrow \mathbb{B}$ and let $(\mathcal{E}_1, \ldots, \mathcal{E}_l)$ be a vector of p-encodings suitable for f and of the form: $\forall i \in \{1, \ldots, \ell\}, \mathcal{E}_i = \begin{cases} 0 \mapsto \{x^{(i)}\} \\ 1 \mapsto \{y^{(i)}\} \end{cases}$ Then any vector of canonical p-encodings $(\mathcal{E}'_1, \ldots, \mathcal{E}'_l)$ of the form: $\forall\ i \in \{1, \ldots, \ell\}, \mathcal{E}'_i = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{y^{(i)} - x^{(i)}\} \end{cases}$ is suitable for the function f as well.*

Proof. Let $f : \mathbb{B}^\ell \longrightarrow \mathbb{B}$ be a function and $\mathcal{E}$ be a vector of canonical p-encodings $(\mathcal{E}_1, \ldots, \mathcal{E}_l)$ suitable for f with:

$$\forall\ i \in \{1, \ldots, \ell\}, \mathcal{E}_i = \begin{cases} 0 \mapsto \{x^{(i)}\} \\ 1 \mapsto \{y^{(i)}\} \end{cases} .$$

Let us build the sets $\mathcal{P}_0$ and $\mathcal{P}_1$ according to Equation 3. Each element of these sets is the sum of exactly one element of each p-encoding, that is to say an element $\mathcal{E}_i(0) \cup \mathcal{E}_i(1)$.

Let us pick an indice $k \in \{1, \ldots, \ell\}$, a value $a \in \mathbb{Z}_p$ and replace $\mathcal{E}_k$ in the vector $\mathcal{E}$ by:

$$\mathcal{E}'_k = \begin{cases} 0 \mapsto \{x^{(i)} - a\} \\ 1 \mapsto \{y^{(i)} - a\} \end{cases}$$

By using the Property 2, we directly have $\mathcal{P}'_0 \cap \mathcal{P}'_1 = \emptyset$ from $\mathcal{P}_0 \cap \mathcal{P}_1 = \emptyset$ (by suitability of the encodings for f).

By iterating this procedure on each of the ℓ elements of $\mathcal{E}$, and by picking each time $a = -x^{(i)}$, we prove Lemma 2. $\square$

Using both Lemmas 1 and 2, we can restrict the search to the encodings of the form

$$\mathcal{E}_i = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{d_i\} \end{cases}$$

with $d_i \neq 0$ without any loss of generality.

Moreover, we restrict the solution further: we only consider p-encodings with p *odd* and *prime*. The choice of an odd p allows to free ourselves from the negacyclicity constraint (more about that in Section 6.1). To explain the constraint of primality, we introduce the following lemma, that allows to drastically improve the performances of the search:

Lemma 3. *Let p be a prime and $f : \mathbb{B} \longrightarrow \mathbb{B}$ be a Boolean function and let $\mathcal{E} = (\mathcal{E}_1, \ldots, \mathcal{E}_l)$ be p-encodings suitable for f with: $\forall\ i \in \{1, \ldots, \ell\}, \mathcal{E}_i = \begin{cases} 0 \mapsto \{x^{(i)}\} \\ 1 \mapsto \{y^{(i)}\} \end{cases}$. For every $a \in \mathbb{Z}_p \setminus \{0\}$, the vector of p-encodings $\mathcal{E}' = (\mathcal{E}'_1, \ldots, \mathcal{E}'_l)$ with: $\mathcal{E}'_i = \begin{cases} 0 \mapsto \{[a \cdot x^{(i)}]_p\} \\ 1 \mapsto \{[a \cdot y^{(i)}]_p\} \end{cases}$ is suitable for f as well.*

Proof. This is an immediate consequence of Property 3. $\square$

As a consequence, if p is prime (which we shall always choose in practice), any solution can be turned into a solution with $d_1 = 1$ by simply multiplying all the p-encodings of the solution by $[d_1^{-1}]_p$. So we can fix $d_1 = 1$ without any loss of generality, reducing drastically the size of the search space.

4.2 Formalization of the Search Problem

According to the lemmas from Section 4.1, we can reduce the problem of finding a vector of p-encodings $(\mathcal{E}_1, \ldots, \mathcal{E}_l)$ such that $f(\mathcal{E}_1, \ldots, \mathcal{E}_l)$ is valid to the problem of finding a vector $\mathbf{d} = (d_1, \ldots, d_l)$ such that $\mathcal{E}_i = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{d_i\} \end{cases}$ and $f(\mathcal{E}_1, \ldots, \mathcal{E}_l)$ is valid. In the following, we describe an algorithm to find such a vector $\mathbf{d}$.

We denote V the matrix of elements of $\mathbb{B}$ of shape $2^\ell \times \ell$ gathering all the possible sequences of entries for the function f:

$$V = \begin{pmatrix} 0 & \ldots & 0 & 0 \\ 0 & \ldots & 0 & 1 \\ \vdots & \ddots & \vdots & \vdots \\ 1 & \ldots & 1 & 1 \end{pmatrix}$$

Also, we denote by $\mathbf{b}$ the vector of all the outputs of the function f, sorted in same order as the rows of V. Thus, we have: $\forall\ i \in \{1, \ldots, 2^\ell\}, b_i = f(V_i)$ for V_i the ith row of V. Let us define the vector $\mathbf{r}$ as: $\mathbf{r} = V\mathbf{d}$. To make $\mathbf{d}$ a solution of the problem, $\mathbf{r}$ has to verify the following property:

$$\forall\ i, j \in \{1, \ldots, 2^\ell\}^2, f(V_i) \neq f(V_j) \implies r_i \neq r_j$$

An alternative formulation is: we look for two disjoint subsets $\mathcal{P}_0$ and $\mathcal{P}_1$ of $\mathbb{Z}_p$, such that: $f(V_i) = b \iff r_i \in \mathcal{P}_b$.

The following section describes an algorithm finding a solution to this problem.

4.3 Algorithm

We start by constructing two sets $\mathcal{F}$ and $\mathcal{T}$ such that:

$$\mathcal{F} = \{V_i | b_i = 0\} \text{ and } \mathcal{T} = \{V_i | b_i = 1\}.$$

Each line V_i represents a linear combination of the d_j's, that verifies:

$$r_i = \sum_{j=0}^{\ell} V_{ij} \cdot d_j \mod p.$$

The values r_i produced by the elements of $\mathcal{F}$ must be different from the ones produced by $\mathcal{T}$. As a consequence, we can write:

$$\forall\ (V_i, V_j) \in \mathcal{F} \times \mathcal{T}, \sum_{k=0}^{\ell} V_{ik} \cdot d_k \neq \sum_{k=0}^{\ell} V_{jk} \cdot d_k,$$

which is equivalent to writing:

$$\forall\ (V_i, V_j) \in \mathcal{F} \times \mathcal{T}, \sum_{k=0}^{\ell} (V_{ik} - V_{jk}) \cdot d_k \neq 0.$$

So we can rewrite our constraints in the set $\mathcal{C} = \{V_i - V_j | (V_i, V_j) \in \mathcal{F} \times \mathcal{T}\}$. $\mathcal{C}$ contains vectors with coordinates in $\{0, 1, -1\}$ representing linear combinations that have to be non-zero. Note that if an element of the set $\mathcal{C}$ is the opposite of an other, it does not bring further constraint and can thus be safely discarded from the set.

The use of a set in the implementation at this point of the algorithm allows to remove a lot of duplicate constraints and to simplify the next step. Then, the problem reduces to solving a "linear system of inequalities" in the ring $\mathbb{Z}_p$:

$$\begin{cases} c_1^{(1)} \cdot d_1 + \cdots + c_l^{(1)} \cdot d_l \neq 0 \mod p \\ c_1^{(2)} \cdot d_1 + \cdots + c_l^{(2)} \cdot d_l \neq 0 \mod p \\ \vdots \end{cases} \quad \text{with } c_i^{(j)} \in \{0, \pm 1\}$$

After filtering, we pack all the elements of $\mathcal{C}$ in ℓ matrices $\{C_i\}_{1 \leq i \leq \ell}$ (each row being a linear combination), where the matrix C_i packs all the constraints involving only the i first inputs (i.e. all the coefficients of column index greater than i are zeros).

We then perform a recursive search (Algorithm 1), affecting at each step of depth i a possible value d_i for the i-th input. To do so, we call Algorithm 2 to construct the set of all possible values complying with the constraints of the matrix C_i and the previously set values for the preceding inputs. If we reach a dead-end, we backtrack by deleting the preceding input and assigning it the next possible value. Algorithms 1 and 2 formalize this idea: Algorithm 1 is a basic recursive backtracking algorithm using calls to the set construction function (Algorithm 2) to get the possibilities for the next value of $\mathbf{d}$. The latter, when called at depth $j+1$, takes as input the j values already computed at higher depth for $\mathbf{d}$ and the matrix of constraints C_{j+1}. Each line of C_{j+1} creates a (potentially duplicate) forbidden value for d_{j+1}, these values are all computed and the complement of this set in $\mathbb{Z}_p$ is returned by the algorithm (i.e. the set for possible values for d_{j+1} at this point of the search).

Theorem 1. *Running Algorithm 1 with increasing values of p ensures that the first solution $\mathbf{d}$ found is optimal for the function f, i.e. the solution works and its associated p is the smallest as possible.*

Optimizations: Several optimizations are possible to improve the performances of the search. First, in Algorithm 2, one can check the size of the set $\bar{S}$ at each iteration and stop as soon as the size of the set is p. Such a set means that a dead-end has been reached and that no value will be returned by the function. Then, one can leverage symmetries existing in the table but also in the function. For example, if we consider the function $f : (x, y) \longrightarrow x \oplus y$, the two variables x and y have symmetric roles. Thus, if the pair of encodings $(\mathcal{E}_x, \mathcal{E}_y)$ is valid, then the pair $(\mathcal{E}_y, \mathcal{E}_x)$ is valid as well. As a consequence, one can arbitrarily set $d_x \leq d_y$ and removing half the possibilities for (x, y).

Development of an heuristic: This algorithm of the previous section is deterministic and finds any existing set of encodings compliant with the function f *for a given value of p*. However, the right value for p is not known *a priori*, so we have to run the full algorithm for each possible value of p until we find one that works. For these reasons, we might prefer an efficient heuristic over the previous algorithm in some contexts. In Section 4.5, we define such a heuristic which allows to drastically improve the performance by executing directly the algorithm with realistic values for p.

4.4 Performances measurements

In this section, we present some experimental results to demonstrate the performances of the algorithm. We ran Algorithm 1 for a lot of random Boolean functions of arity ℓ. Two metrics are particularly interesting for us:

- The running time of the algorithm, especially in the cases where there is no solution.

Algorithm 1 Recursive function `search` that adds an element to the vector $\mathbf{d}$

Require:
$\mathbf{d} := (d_i)_{1 \leq i \leq j}$ ▷ The vector of values for the inputs already computed
$\{C_i | i \in \{0, \ldots, \ell - 1\}\}$ ▷ The matrices of constraints, pre-computed
$p \in \mathbb{N}^*$ ▷ the modulus of the input encodings
$\ell \in \mathbb{N}^*$ ▷ The target number of encodings required
Ensure: f is evaluable using the encodings $\mathbf{d}$.
if $j = \ell$ **then**
  return $\mathbf{d}$ ▷ Base case of recursion when a complete solution has been found
else
  $\mathcal{P} \leftarrow$ `get_possible_values`$(\mathbf{d}, C_{j+1}, p)$ ▷ Retrieving the set of possible values for d_k.
  for $x \in \mathcal{P}$ **do**
    $\mathbf{d} \leftarrow (\mathbf{d} || x)$ ▷ Affecting one of the possible value to d_k
    $\mathbf{d}_{sol} \leftarrow$ `search`$(\mathbf{d}, C, p, \ell)$ ▷ Recursively calling the algorithm
    if $\mathbf{d}_{sol} \neq \perp$ **then**
      return $\mathbf{d}_{sol}$ ▷ If a final solution has been found, propagating it to the higher level
    else
      $\mathbf{d} \leftarrow \mathbf{d}[: j + 1]$ ▷ If the previous call failed, we remove the last value and try an other one.
    end if
  end for
  return $\perp$ ▷ If all of the possibilities have been tested and none of them work, we need to backtrack
end if

Algorithm 2 Function `get_possible_values` that builds the set of possible values for the next slot of $\mathbf{d}$ given the slots already filled in.

Require:
$\mathbf{d} := (d_i)_{\{1 \leq i \leq j\}}$ ▷ The set of values for the inputs already computed. Note that d_1 is fixed to 1
C_{j+1} ▷ The matrix of constraints of this step, pre-computed
$p \in \mathbb{N}^*$ ▷ the modulus of input encoding

Ensure: The set S contains only values suitable for the $j + 1$-th slot of $\mathbf{d}$.
$\bar{S} \leftarrow \{\}$ ▷ $\bar{S}$ is the set of forbidden values for d_{j+1}
for $c \in C_{j+1}$ **do**
  $\bar{c} \leftarrow c[j + 1]$ ▷ We retrieve the $(j + 1)$th coefficient of the inequation c
  $\bar{S} \leftarrow \bar{S} \cup \{ \left[-\bar{c} \cdot \sum_{k=1}^{j} c_k \cdot d_k \right]_p \}$ ▷ We compute the value forbidden by c
end for
$S \leftarrow \mathbb{Z}_p \setminus \bar{S}$
return S

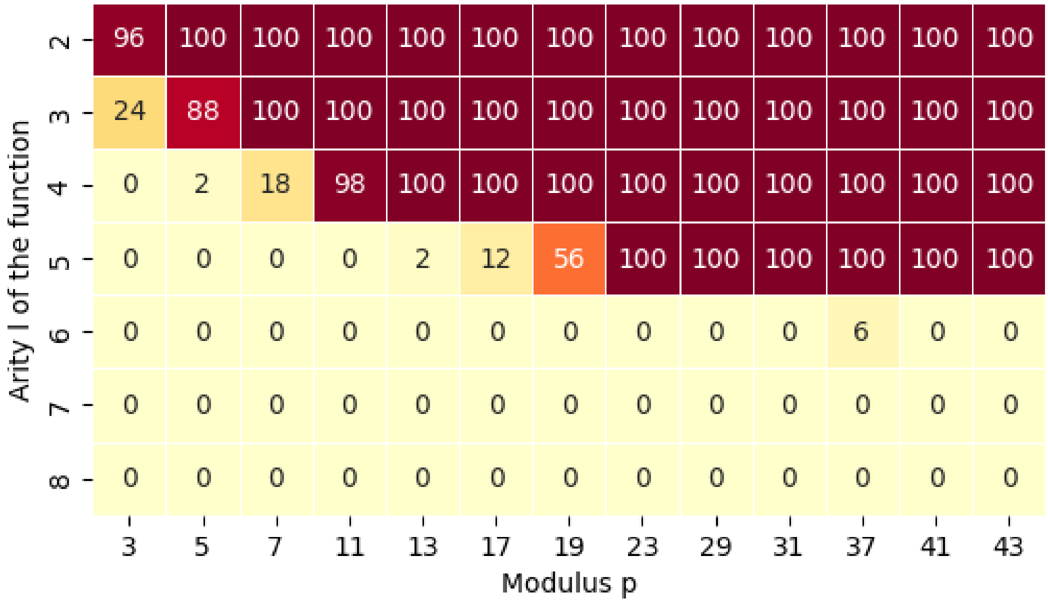


Figure 5: Rate of success of the algorithm for 100 random Boolean functions for different values of ℓ and p.

- The probability of success, for a random function

Figure 5 shows the rate of success for random Boolean functions of arity $\ell \in \{2, 9\}$ and for prime values of $p \in [3, 39]$. It illustrates the intuitive idea that one has to increase p to evaluate functions of bigger arity ℓ. It also give a rough idea of the value of p required for a given function of arity ℓ.

Figure 6a shows the evolution of the time of execution of the algorithm for random Boolean functions *for which no solution exists*. It shows the explosion of the complexity for high values of p, and justifies the need of a more efficient algorithm for those function (we introduce one in Section 5).

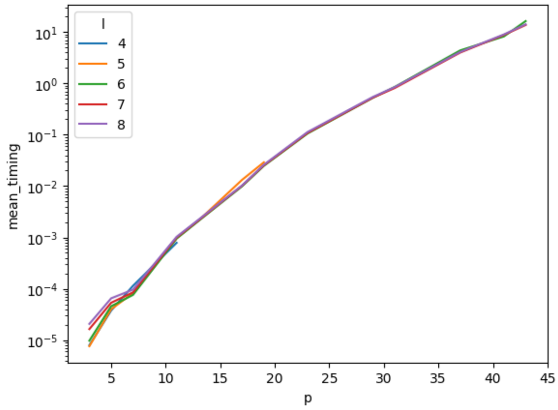
Lastly, Figure 6b shows how long it takes to find a solution when one exists, relatively to the running time when no solution exist at all. It illustrates a form of "speed of convergence" and shows that it is located around $\frac{1}{3}$.

4.5 An Efficient Sieving Heuristic to Find Suitable Encodings

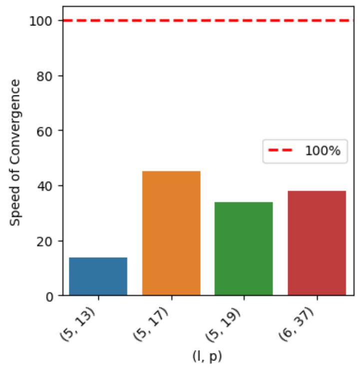
Let us consider a function $f : \mathbb{B}^\ell \mapsto \mathbb{B}$ of matrix of constraints $C = (C_j^{(i)})_{\substack{1 \le i \le n_j \\ 1 \le j \le \ell}}$ and its associated system of linear inequalities:

$$\begin{cases} c_1^{(1)} \times d_1 + c_2^{(1)} \times d_2 + \cdots + c_\ell^{(1)} \times d_\ell \neq 0 \mod p \\ c_1^{(2)} \times d_1 + c_2^{(2)} \times d_2 + \cdots + c_\ell^{(2)} \times d_\ell \neq 0 \mod p \\ \cdots \end{cases}$$

The principle is to sample random values in $\mathbb{Z}$ (with some large bound) and affect them to the d_j's. If all the corresponding values for all the $C_i = \sum_{j=1}^{\ell} c_j^{(i)} \times d_j$ are not divisible by a value p, then the vector $(d_j \mod p \mid j \in \{1, \ldots, \ell\})$ is a solution of the system of inequalities generated by C.



(a) Running time of the algorithm for different values of ℓ and p for random functions. Note that the scale is logarithmic.



(b) Ratio between the time to find a solution when it exists with the time to run the full algorithm when no solution exists.

Figure 6: Some metrics about running time.

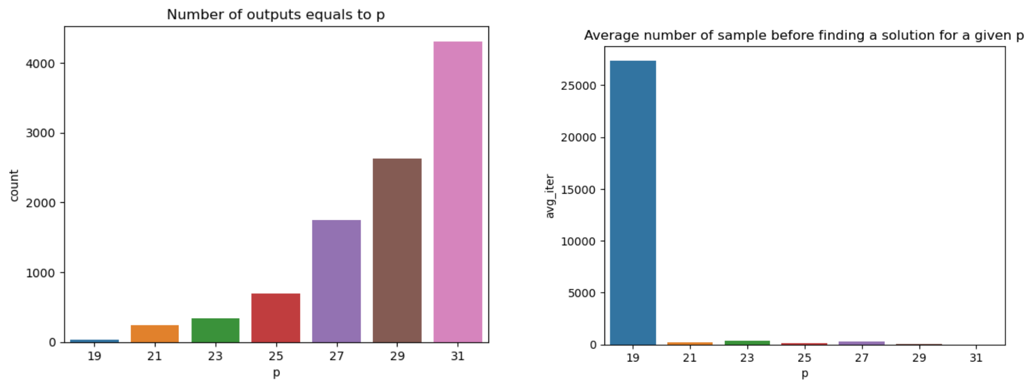
To reduce the amount of samples required to find a solution, we want to avoid sampling trivially wrong sets of d_j's. For example, if all the d_j's are themselves divisible by p, then the C_i's will all be divisible as well. To tackle this problem, we perform the sampling across *prime numbers in* $\mathbb{Z}$.

Algorithm 3 Sample a solution $\mathbf{d}$ in $\mathbb{Z}$ for a function f and returns a possible value for p.

Require:
  $\{C_i\}_{1\leq i\leq n}$ ▷ The lines of the matrix of constraints C of the function f
  P ▷ The sets of possible values for p to be tested
  D ▷ The sets of possible values in $\mathbb{Z}$ to assign to the d_i's. All these elements are big primes
Ensure: f is possible to evaluate using a modulus smaller or equal than p.
  $\mathbf{d} \xleftarrow{\$} D$ ▷ Sample random prime values in $\mathbb{Z}$ and assign it to $\mathbf{d} = (d_1, \ldots, d_l)$
  $\mathbf{r} = C \times \mathbf{d}$ ▷ $\mathbf{r}$ is the right member of the system
  for $p \in P$ **do**
    if $0 \in [\mathbf{r}]_p$ **then** ▷ If p divides one of the coordinates of $\mathbf{r}$
      $P \leftarrow P \setminus \{p\}$ ▷ This value of p is incorrect
    end if
  end for
  if $\mid P \mid > 0$ **then**
    return $\min(P)$ ▷ Returns the smallest possible value for p, if any.
  end if

Running this algorithm several times and keeping the smallest returned value for p, one gets an upper bound on the minimum p required to evaluate a function with our framework. Note that, on the contrary of the deterministic search algorithm, this heuristic does not require a prime p.

Example: Let us consider the s-box of the block cipher ASCON. We study this s-box in more details and provide an exact optimized solution for its homomorphic evaluation in Section 7.4. Here, we apply Algorithm 3 on the five functions generating the five output bits and monitor the results until we gather $N = 10000$ non-zero possible values for p.



(a) The outputs of 10000 runs of the Algorithm 3 for the first subfunction of the Ascon s-box

(b) Number of iterations required to get a solution for a given value of p

The figure 7a shows the repartition of the returned values of p by the algorithm during these N runs on the first subfunction. The optimal value of p found by the deterministic approach of Section 4.3 is 17 so the upper bound 19 is pretty close, despite being rarely found by the algorithm. Also, the figure 7b shows 21 (the second best solution found by the sieving) is almost instantly found by the algorithm.

In the process of finding the smallest p possible and a correct vector of p-encoding to evaluate a function f, this heuristic is really efficient to get a tight upper bound on the value of p.

5 Scaling our Approach to any Boolean Circuit

Our framework optimizes the homomorphic evaluation of single Boolean functions but suffers the following limitations:

1. For a Boolean function with a high number of inputs, the search algorithm may be very time-consuming.
2. Some functions simply do not have any solution for acceptable values for p ($p < 32$ for example) and thus are not efficiently evaluable in a single PBS.[1]

As a consequence, we need a solution to extend our framework to these cases. In this section, we propose a strategy to leverage the circuit representation of a "tough" function f to find a strategy of homomorphic evaluation with as few bootstrappings as possible.

5.1 Graph of Subcircuits

Let $f : \mathbb{B}^\ell \longrightarrow \mathbb{B}$ be a Boolean function, and let $\mathcal{F}$ be a Boolean circuit representing f (some preliminaries about Boolean circuits can be found in Section 2.6). Let us describe the layout of the circuit $\mathcal{F}$. It has ℓ input wires, denoted by $\{y_j\}_{1\leq j\leq \ell}$, and the output wire is denoted by z. The intermediary wires are denoted by $\{t_j\}_{1\leq j\leq \theta}$. The Boolean operation gates are of fan-out 1.

Our goal is to split the circuit into a directed acyclic graph $\mathcal{G}$, whose vertexes are subcircuits $\{\mathcal{F}_1, \ldots, \mathcal{F}_k\}$ and whose edges connect the outputs of a subcircuit with the input of another. Each subcircuit $\mathcal{F}_i$ represents a subfunction $f_i : \mathbb{B}^{l_i} \mapsto \mathbb{B}$ that is evaluable with a gadget with our framework.

[1]The PBS can be evaluated for larger values of p but it quickly becomes inefficient as p grows.

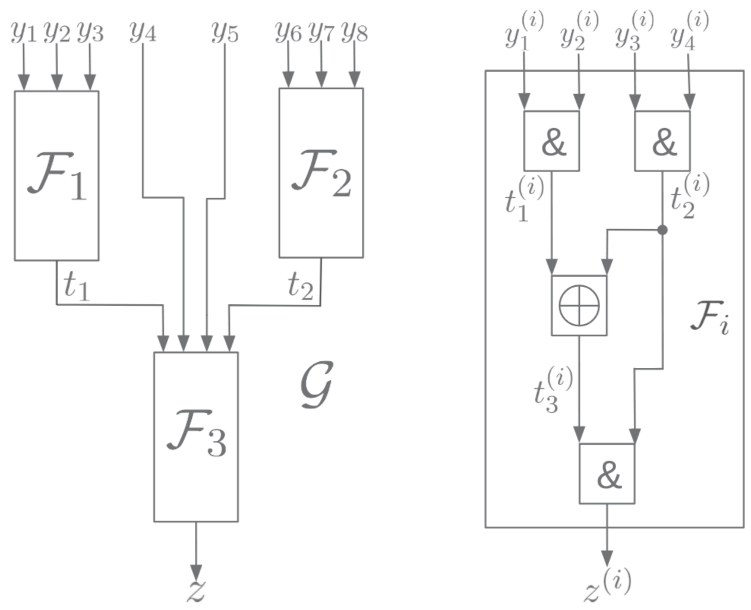


Figure 8: Example of graph of subcircuits (on left) and of a valid subcircuit (on right). Each subcircuit $\mathcal{F}_i$ is evaluated homomorphically with a gadget Γ_i.

We use the same notations to refer to the elements of a subcircuit $\mathcal{F}_i$ and we index them with i. The output of $\mathcal{F}_i$ is denoted by $z^{(i)}$ and its inputs by $\{y_j^{(i)}\}_{1\leq j\leq \ell}$ and so on.

The graph is valid for f with respect to modulus p if the following properties are satisfied:

- Each subcircuit $\mathcal{F}_i$ has only one output $z^{(i)}$.

- For a subcircuit $\mathcal{F}_i$, all its inputs are either inputs of the whole circuit or outputs of other subcircuits of the graph. We can write this property as:

$$\{y_j^{(i)}\}_{1\leq j\leq l_i} \subset \left(\{y_j\}_{1\leq j\leq \ell} \cup \{z^{(j)}\}_{1\leq j<i}\right)$$

  Thus, the indexing of the $\mathcal{F}_i$'s respects the topological order of the graph, i.e. no gates of $\mathcal{F}_i$ has a child in any of the $\mathcal{F}_j$, with $j < i$.

- All the Boolean functions f_i represented by the subcircuits $\mathcal{F}_i$ are evaluable in a single bootstrapping with modulus p with our proposed method.

- The last subcircuit $\mathcal{F}_c$ of the graph has z (the output of the main circuit) for output: $z^{(c)} = z$.

To homomorphically evaluate the function f, we evaluate each subcircuits with one bootstrapping for each of them and get the final result. In order to reduce the cost of evaluation for a given p, the goal is hence to find the *smallest* valid graph possible in terms of number of subcircuits. Taking a greater value of p produces a different graph that may be smaller (as subcircuits might be larger), but the timings of bootstrapping in this graph might on the other hand be greater. One can therefore run the search for different values of p and keep the most efficient setup among the possible graphs.

5.2 Heuristics to Find a Small Graph

Finding such a graph can be done by exhaustively evaluating all the possible subcircuits with our method introduced in Section 4, and then find the more efficient one. However it is not really practical to evaluate *all* the possible subcircuits, so we develop some heuristics to reduce the search space. Let us start by defining a few bounds on the considered subcircuits, we will leave the other ones apart in our algorithm:

- The subcircuits have at most B inputs ($\forall i, l^{(i)} < B$). The purpose of this bound is to limit the running time of Algorithm 1. In practice, for our experiments, we took $B = 10$.
- The subcircuits are evaluable with one single bootstrapping with a maximum value p_{max}. This value ensures a bootstrapping with a reasonable timing. If the search algorithm fails for p_{max}, the subcircuit is dropped without trying to extend p. In our experiment, we took $p_{max} = 31$.

In order to decompose our Boolean circuit into a graph satisfying the above property for a modulus p, we would want to exhaustively search all the subcircuits of $\mathcal{F}$ compliant with the bounds we introduced earlier. However, all subcircuits are not equally worth to evaluate. In particular a wire incoming a copy gate is particularly worth evaluating because is costs one bootstrapping but produce several inputs for the next subcircuits.

We gather wires that precede a copy gate in the set $\mathcal{Z}$. We add to this set the global output z. We also gather the input wires of the global circuit $\mathcal{F}$ in the set $\mathcal{Y}$. We define the notion of *atomic subcircuit* that is a valid subcircuit whose all inputs belong to $\mathcal{Y} \cup \mathcal{Z}$ and whose output belongs to $\mathcal{Z}$. Note that the merge of two atomic subcircuits that respect the global circuit wiring is also an atomic subcircuit.

Our heuristic works as follows:

1. For each of these outputs $z_i \in \mathcal{Z}$, we exhaustively construct a set $\widehat{\mathcal{F}_{z_i}}$ that gathers all the atomic subcircuits whose output is z_i. We then filter out the subcircuits of $\widehat{\mathcal{F}_{z_i}}$ that do not comply with the bounds introduced at the beginning of the section or that are not evaluable with a gadget with the input modulus p (we use Algorithm 1 to decide that).

2. Now we want to construct the smallest valid graph evaluating $\mathcal{F}$ using subcircuits from the $\widehat{\mathcal{F}_{z_i}}$'s. While finding the smallest graph is hard, constructing any valid graph is easy. As a consequence, our strategy to find a small graph is to randomly create a lot of valid graphs and to take the smallest one. The procedure to create a valid graph is the following: we start from the output z and we randomly draw a subcircuit $\mathcal{F}_z$ from $\widehat{\mathcal{F}_z}$. The inputs of $\mathcal{F}_z$ can be sorted into two categories: the ones belonging to $\mathcal{Y}$ and the ones belonging to $\mathcal{Z}$. For each one of these latter wires $w \in \mathcal{Z}$, we repeat the procedure, i.e. we draw a subcircuit $\mathcal{F}_w$ from $\widehat{\mathcal{F}_w}$, and so on. When we have reached all the input wires of $\mathcal{F}$, we get a valid graph $\mathcal{G}$. This second step is run a large amount of times (the number of trials is a parameter of the method), and the smallest graph, i.e. the one with the fewest subcircuits, is returned.

We carried on this method on the s-box of AES in Section 7.5.

5.3 Parallelization of the Execution of the Graph

Once we have our graph $\mathcal{G}$, we can identify its $n_{\mathcal{L}}$ *layers*. Formally, they are defined as:

Definition 9. A layer $\mathcal{L}$ of a graph $\mathcal{G}$ is a set of subcircuit $\{\mathcal{F}_\alpha, \dots, \mathcal{F}_\omega\}$ of $\mathcal{G}$ that verifies: $\forall \mathcal{F}_i, \mathcal{F}_j \in \mathcal{L}, \mathcal{F}_i$ is not an ancestor node of $\mathcal{F}_j$.

By construction, all the subcircuits belonging to the same layer can be evaluated in parallel. This reduces the number of bootstrapping steps from k (the number of subcircuits in the graph $\mathcal{G}$) to $n_\mathcal{L}$ (the number of layers). Our graph-finding heuristic can be tweaked to select the graph with minimum number of layers instead of minimum number of subcircuits to optimize parallelization.

6 Adaptation of TFHE and the `tfhe-rs` Library

From a high level point of view, our technique can be seen as adding an additional layer of abstraction on top of TFHE. However things are not that simple: picking odd values for p leads to some changes in the inner working of the programmable bootstrapping (PBS), and the choice of parameters is also affected by this change. Moreover, we implemented our framework by forking the `tfhe-rs` library [Zam22b] written in Rust. The following section covers the adaptation of the PBS and the choice of new parameters. The adaptation of the library is treated in Section 6.4.

6.1 Dealing with the Negacyclicity Problem for an Odd p

In the following, we explain the negacyclicity problem and how we propose to solve it. To do so, we need to dig into the details of the `BlindRotate` step of the PBS, that we have introduced in Section 2.5.

Let $v(X)$ be a polynomial of the ring $\mathbb{Z}_{q,N}[X]/(X^N+1)$, denoted by $v(X) = \sum_{k=0}^{N-1} v_k X^k$. Observe that a multiplication by X in this ring "rotates" the coefficients of the polynomial:

$$X \cdot v(X) = -v_{N-1} + v_0 \cdot X \cdots + v_{N-2} X^{N-1} \ .$$

In TFHE, the polynomial multiplication in the blind rotation is actually done by $X^{-\tilde{\mu}}$, with $\tilde{\mu} = \left\lfloor \frac{\mu \cdot 2N}{q} \right\rceil$, which lives in $\{0, \dots, 2N-1\}$. This leads to two problems:

- A coefficient v_j can be brought in first place by two differents rotations: the one induced by the polynomial multiplication by $X^{[-j]_{2N}}$ and the one by $X^{[-j+N]_{2N}}$.
- Each time a coefficient goes last to first, it gets negated (because $X^N = -1$ in the ring). So actually, the multiplication by $X^{[-j]_{2N}}$ yields correctly v_j, but the one by $X^{[-j+N]_{2N}}$ yields $-v_j$.

However, these problems can be circumvented for even and odd values of p. Recall that $\mu = m + e \in \mathbb{Z}_q$, with e sampled from a small centered Gaussian. The use of a small error makes that μ does not take all the values of $\mathbb{Z}_q$ with the same probability: in particular, the densest parts in terms of probability over $\mathbb{Z}_q$ are the one close to the "unscrambled" values of m, namely $\left\{ \left\lfloor \frac{kq}{p} \right\rceil \mid k \in \mathbb{Z}_p \right\}$. We illustrate this distribution on Figure 9. We call these sections of the torus the *dense spots*.

When we transpose these dense spots into $\mathbb{Z}_{2N}$, they become the sectors close to $\left\{ \left\lfloor \frac{k \cdot 2N}{p} \right\rceil \mid k \in \mathbb{Z}_p \right\}$. Let us note that the noises in $\mathbb{Z}_q$ and $\mathbb{Z}_{2N}$ are fundamentally different: the former is the one added at encryption that may have grew during the homomorphic computations, and the latter is called "drift" and is caused by the accumulation of the rounding errors on each coefficient of the ciphertext during the modulus switching (but this difference in nature does not impact our purpose). Let $k \in \mathbb{Z}_p$, the multiplication $X^{-\frac{k \cdot 2N}{p}} \cdot v(X)$ yields the same degree-zero coefficient as the multiplication $X^{\left[-\frac{k \cdot 2N}{p} + N\right]_{2N}}$.

$p = 6$ $\qquad$ $p = 5$

Figure 9: Distribution of the values of μ across $\mathbb{Z}_q$ for $p = 6$ and $p = 5$: the colored parts show the dense spots where the value has a high probability to lie in. The width of these sectors depends on σ (the standard deviation of the error distribution χ of TFHE). Note that this repartition looks the same for $\tilde{\mu}$ in $\mathbb{Z}_{2N}$.

$p = 6$ $\qquad$ $p = 5$

(a) With p even, the dense spots of each half of the torus are aligned.

(b) With p odd, the dense spots face empty spots, close to the bounds of the p-sectors.

Figure 10

$v(X)$, up to the minus sign. For the sake of clarity, we write the exponent of the latter in a slightly different manner:

$$\left[\frac{-k \cdot 2N}{p} + N\right]_{2N} = \left[\frac{(-k + \frac{p}{2}) \cdot 2N}{p}\right]_{2N}$$

This is where the parity of p plays a part: if p is even, then $\left[\frac{(-k+\frac{p}{2})\cdot 2N}{p}\right]_{2N}$ is a dense spot as well. So, the rotations by these two values will happen with high probability and they will both yield the same coefficient $v_{\frac{k\cdot 2N}{p}}$ (up to the minus sign for one of them). Thus, when evaluating a function f with a PBS, the calls $f(k)$ and $f(k+\frac{p}{2})$ will produce the same output (one again, up to the minus sign), which is a collision constraining the definition of f. On the other hand, let us consider an odd value for p. Then, $\left[\frac{(-k+\frac{p}{2})\cdot 2N}{p}\right]_{2N}$ is no longer a dense spot, as it lies exactly halfway between the two dense spots $\left[\frac{(-k+\frac{p-1}{2})\cdot 2N}{p}\right]_{2N}$ and $\left[\frac{(-k+\frac{p+1}{2})\cdot 2N}{p}\right]_{2N}$. As a consequence, collision never occurs. Figure 10 illustrates this phenomenon.

That is why we select only odd values for p in our framework. We will see in Section 6.3 how this change impacts the parametrization of the scheme.

Exception for $p = 2$: We just said that only odd values can be selected for p in our framework, however p-encodings with even values of p exist as well: nonetheless they need to achieve the relaxed negacyclicity property introduced in Definition 5. This restriction makes them basically useless, as using only odd p-encodings is sufficient to evaluate all possible Boolean functions without having to bother with the negacyclicity property. However, the case $p = 2$ is an exception: the valid 2-encodings are automatically negacyclic and allow to evaluate the `XOR` operation by simply performing an homomorphic sum (so without bootstrapping). So it might be efficient to switch between 2-encodings for `XOR` operations and p-encodings (with odd p) for non-linear Boolean functions. We make use of this strategy in our implementation of the Keccak permutation in Section 7.3 and for the AES in Section 7.5.

6.2 Construction of the Accumulator for an Odd p

The accumulator is the polynomial $v(X)$ used in the `BlindRotate` step of the PBS. In the Section 6.1, we showed how the values are spread over the torus after bootstrapping. To actually make that works, we need to explicitly characterize this polynomial. In the following presentation, we neglect roundings to keep notations light (as if p would divide N), or, equivalently, the division operator is assumed to include rounding.

Definition 10. If p is an odd modulus, and $f : \mathbb{Z}_p \mapsto \mathbb{Z}_{p'}$ a function, then the accumulator $v(X) \in \mathbb{Z}_{N,q}[X]/(X^N + 1)$ has the form:

$$v(X) = X^{-\frac{N}{2p}} \cdot \sum_{j=0}^{N/p-1} X^j \cdot \left(\sum_{i=0}^{\frac{p-1}{2}} f(i) X^{i\frac{2N}{p}} + \sum_{i=0}^{\frac{p-1}{2}-1} -f\left(i + \frac{p+1}{2}\right) X^{i\frac{2N}{p}+\frac{N}{p}} \right)$$

Let us explain the structure of this accumulator. The polynomial has degree N and is made of p distinct windows of width $\frac{N}{p}$. Each of these windows has constant coefficient value $f(k)$, for $k \in \{0, \dots, p-1\}$. For $0 \leq \alpha \leq \frac{p-1}{2}$, the range of degrees whose coefficients are $f(\alpha)$ is $\left[\alpha\frac{2N}{p} - \frac{N}{2p} \,;\, \alpha\frac{2N}{p} + \frac{N}{2p}\right]$. Now, for $\frac{p+1}{2} \leq \beta \leq p-1$, we can write $\beta = \alpha + \frac{p+1}{2}$, with $0 \leq \alpha < \frac{p-1}{2}$. This time, the range of spanned degrees is $\left[\alpha\frac{2N}{p} + \frac{N}{2p} \,;\, (\alpha+1)\frac{2N}{p} - \frac{N}{2p}\right]$. Thus, the values $k \in \{0, \dots, p-1\}$ spans the entire space $[0; N)$ without overlap. The values over $\frac{p+1}{2}$ gets negated by the negacyclicity, so the underlying coefficient is also negated to compensate this effect. We illustrate this construction on Figure 11.

6.3 Crafting of Parameters

The instances of the TFHE scheme are defined by a set of parameters. These parameters should simultaneously ensure the security of the scheme and the correctness of the homomorphic computations. They also determine the time of execution of one PBS. Here we define a framework to dimension the parameters required to optimally execute a given gadget.

Finding an optimal set of parameters for a given application is a hard problem and has been studied in particular in [BBB+23]. The parameters need to ensure three properties: security, correctness and efficiency.

Let us start by an overview of the different parameters at play in an instance of the TFHE bootstrapping:

- n: the dimension of the LWE samples. Namely, the TLWE ciphertexts are vectors of length $n + 1$.
- q: the modulus of the ring the encrypted values live on. In `tfhe-rs` those values are stored on `u32` values, making $q = 2^{32}$. We treat this as an immutable platform-dependent value.

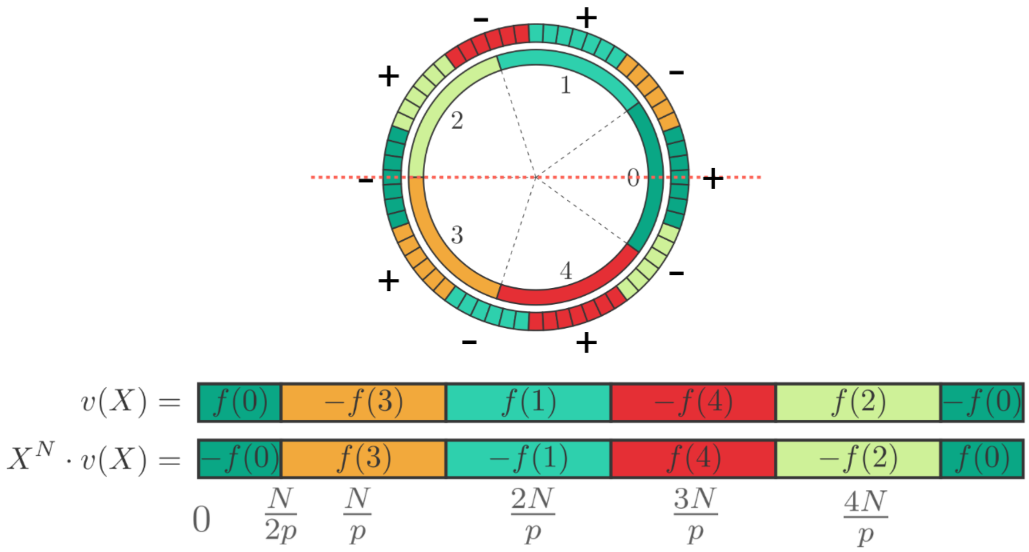


Figure 11: Illustration of the construction of the accumulator. On top is the ring $\mathbb{Z}_{2N}$ splitted in windows. Below is a representation of the polynomial v, with its version once rotated by a multiplication by X^N. On the figure, $p = 5$.

- σ: the standard deviation of the Gaussian distribution of error in LWE samples.
- k: the dimension of the GLWE samples. If $k = 1$, we talk about RLWE samples.
- σ': the standard deviation of the Gaussian distribution of error in GLWE samples.
- A few more parameters dimensioning some inner algorithms of the bootstrapping. A detailed description and an analysis of their impact on performances and noise level can be found in [BBB+23]. In this work, they are denoted as *micro-parameters*.

In [BBB+23], authors elaborate a strategy where they define an *atomic pattern* of FHE operators, that is to say a subgraph of FHE operators in which the noise of the output is independent from the one in the inputs. Then, they develop an optimization framework to derive the best set of parameters for a given atomic pattern.

In particular, the first atomic pattern they study, that they denote by $\mathcal{A}^{(CJP21)}$, is a subgraph composed of a linear combination of ciphertexts with clear constants, then a `Keyswitch` and then a `BlindRotate` followed by a `SampleExtract` (`ModulusSwitch` is seen as a part of `BlindRotate`). Note that in Section 2.5 we introduced the bootstrapping of TFHE by putting the `BlindRotate` before the `Keyswitch`, but the other way around is also doable. To dimension the parameters of TFHE to evaluate such an atomic pattern, their framework takes as input the 2-norm of the vector of constants of the linear combination (denoted by ν) and a noise bound t on the standard deviation of the distribution of error in a ciphertext that ensures a correct decryption with a good probability $(1 - \epsilon)$. We elaborate further on how this bound is constructed below in this section.

If we look closely, the evaluation of a gadget we introduced in Definition 8 can be seen as a $\mathcal{A}^{(CJP21)}$ with a few differences. Thus, we slightly modified the tool `concrete-optimizer` [Zam22a], that allows to generate parameters for different types of atomic patterns, to support our gadget as a new atomic pattern. Let us dive into the differences between a gadget and a $\mathcal{A}^{(CJP21)}$:

Support of odd values for p: Using an odd value for p changes the bootstrapping procedure, and in particular the definition of the accumulator for the `BlindRotate` (as explained in Section 6.2). With our construction, the windows in the polynomial are half the size of the ones for an even p, which impacts the noise bound t. As this bound depends of the failure probability α that the user is ready to tolerate, we shall denote it t_α hereafter, which satisfies: $t_\alpha = \frac{\Delta}{2z^*(1-\sqrt[N]{1-\alpha})}$ where z^* is the *standard score* and Δ is the scaling factor (see [BBB+23] for more explanations). The impact of our adaptation on this formula is solely with respect to the scaling factor. In the context of an $\mathcal{A}^{(CJP21)}$, we have $\Delta = \frac{q}{2^\pi p}$ with π the number of MSB for padding. As explained in Section 6.1, we do not need any padding mechanism anymore, so the 2^π vanishes. However, the length of a window is divided by 2, and p does not divide q anymore so we need to add a rounding. We finally get $\Delta = \left\lfloor \frac{q}{2p} \right\rceil$.

Link between input encodings and ν: In a scenario where only one gadget has to be evaluated, its inputs are freshly encrypted ciphertexts. Then, there is no need to perform any encoding switching before evaluating the gadget, and so we are in the context of a $\mathcal{A}^{(CJP21)}$ with $\nu = 1$. However, if we are in a context of a *graph* of gadgets like in Section 5, the output of a gadget can be used as input of subsequent gadgets under different encodings. In this case, some encoding switchings are necessary. If these encoding switching are made using a mutiplication by a constant (Property 3), we are still in the context of a $\mathcal{A}^{CJP21}$ but with $\nu \neq 1$. To formalize that, we first recall that Algorithm 1 produces gadgets of the form $\Gamma = (\mathcal{E}_{\mathbf{in}}, \mathcal{E}_{out}, p_{in}, p_{out}, f)$, with $\mathcal{E}_{in}^{(i)} = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{d_i\} \end{cases}$. Thus, if we fix that all gadget output ciphertexts are encoded under $\mathcal{E}_{out} = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{1\} \end{cases}$, then the encoding switchings needed before an evaluation of Γ corresponds to a linear combination of the inputs with the vector $\mathbf{d} = (d_i \mid i \in [1, \ell])$, so we fall back on a $\mathcal{A}^{(CJP21)}$ with $\nu = \|\mathbf{d}\|$.

We implemented these changes in `concrete-optimizer` and uses it to generate sets of parameters for our implementations detailed in Section 7.

6.4 Concrete Implementations of p-Encodings and Homomorphic Functions in `tfhe-rs`

To implement our framework, we relied on the `tfhe-rs` library [Zam22b]. Here is a list of the major changes we applied to the code:

Addition of the notion of p-encoding: An encoding $\mathcal{E}$ is simply implemented with a structure `Encoding` storing two `HashSets` and the modulus p. The `HashSets` represent both sets $\mathcal{E}(0)$ and $\mathcal{E}(1)$. When creating an `Encoding`, the code checks whether the two underlying sets are disjoint or not. Moreover, the operation of encryption and decryption are modified as well. The signatures change from:

```
encrypt(Boolean, ClientKey) -> Ciphertext
```

to:

```
encrypt(Boolean, ClientKey, Encoding) -> Ciphertext
```

(same for `decrypt`). The functions also perform the mapping $\mathbb{B} \mapsto \mathbb{Z}_p$ before encryption and the other way around after decryption.

Support of odd moduli: The native `tfhe-rs` only support power-of-two-moduli p. We extended the library to handle odd values for p. This required modifying the encryption and decryption algorithm, and to compute the sets of parameters with the method of Section 6.3.

Definition of the new structure `Gadget`: According to the evaluation strategy we introduced in Section 3.2, we wrote a new structure `Gadget`, associated to a Boolean function $f : \mathbb{B}^\ell \mapsto \mathbb{B}$, carrying:

- A list of the `Encoding` objects for the inputs: $\mathcal{E}_{in} = (\mathcal{E}_1, \dots, \mathcal{E}_l)$, with the input modulus p_{in} they encoded on.
- The output `Encoding` object $\mathcal{E}_{out}$, with the output modulus p_{out} it is encoded on.
- The clear function f.

When such a structure is constructed, it self-checks whether $f(\mathcal{E}_{in})$ is valid. Then, when provided ℓ `Ciphertexts` objects encoded under their respective p-encoding, it executes the homomorphic sum and the PBS and outputs the results encoded under $\mathcal{E}_{out}$. Some utilitary functions performing encoding-switching are also available, allowing the chaining of several `Gadget`.

Implementation of the accumulator: The procedure of bootstrapping of `tfhe-rs` is slightly modified to support the new version of the accumulator we introduced in Section 6.2.

Parsing of graphs: We implemented a Python script that produces graphs to represent more complex functions that requires several PBS, as described in Section 5. These graphs are stored with a comprehensive file format and our Rust implementation has a module of parsing allowing to load these graphs and automatically generate the corresponding graph of `Gadget`.

7 Application to Cryptographic Primitives

In this section, we apply our approach on some cryptographic primitives. For each primitive, we first explain the construction of the gadgets required and report the concrete performances of our implementation. We detailed all the timings of our experimentations along with the sets of parameters we used in Section 7.6.

For performance measurement, we implemented our framework in our fork of the library `tfhe-rs` [Zam22b] adapted as discussed in Section 6 and we generated the sets of parameters thank to our version of `concrete-optimizer` [Zam22a]. By default, we tailored the sets of parameters to limit the probability of failure ϵ of a bootstrapping to 2^{-40}, and a security level of $\lambda = 128$ bits. All experiments have been carried out on a laptop with a 12th Gen Intel(R) Core(TM) i5-1245U CPU with 10 cores and a frequency of 4.4 GHz, and 16 GB of RAM.

7.1 SIMON Block Cipher

SIMON is a hardware-oriented block cipher developed in [BSS+15], which relies only on the following operations: `AND`, rotation, `XOR`. It is a classical Feistel network for which the Feistel function consists in applying basic operations on the branch, xoring the subkey and then xoring the result with the other branch as depicted in the Figure 12 (on this figure, S^i denotes the left circular shift by i bits.). We use one ciphertext per bit so the rotation

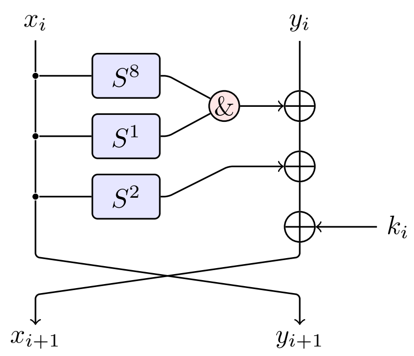


Figure 12: One Feistel round of SIMON.

operation is essentially free. Note that the key is considered as a plaintext, which does not change anything in the framework. In our implementation, we considered a (128-128) instance of SIMON (i.e. the whole state and the key are of size 128).

The Boolean function to evaluate can be defined as

$$f(b_0, b_1, b_2, b_3, b_4) = b_0 \cdot b_1 \oplus b_2 \oplus b_3 \oplus b_4 \ .$$

Using Algorithm 1, we found the smallest possible p ($p = 9$) and the following 9-encodings to evaluate each bit of the Feistel function with one single bootstrapping (i.e. totalling 64 PBS per round).

$$\mathcal{E}_0 = \mathcal{E}_1 = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{1\} \end{cases} \quad \text{and } \mathcal{E}_2 = \mathcal{E}_3 = \mathcal{E}_4 = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{2\} \end{cases} \quad \text{with } p = 9.$$

The sum of these p-encodings yields the output encoding:

$$\mathcal{E}_{out} = \begin{cases} 0 \mapsto \{0, 1, 4, 5, 8\} \\ 1 \mapsto \{2, 3, 6, 7\} \end{cases} \quad \text{with } p = 9$$

which is valid for f. After the PBS, all the bits of the state are encrypted under the encoding $\mathcal{E}_0$. We formalize that with the gadget $\Gamma = ((\mathcal{E}_0, \mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \mathcal{E}_4), \mathcal{E}_0, 9, 9)$

To perform a Feistel round on a state of size k, the gadget Γ is applied in parallel $k/2$ times. Note that one bit may be used in several evaluation as b_0, b_1 and b_2. So we sometimes have to switch from $\mathcal{E}_0$ to $\mathcal{E}_1$ by a simple external multiplication by 2, which is negligible in terms of performances.

Using our version of `concrete-optimizer` [Zam22a], we crafted a set of parameters suitable for this modulus and these encodings. On our machine, one PBS with such parameters takes about 9.5 ms. The theoretical timings achieved on one full block without any parallelization is 41 seconds (68 rounds $\times$ 64 bits $\times$ 9.5 ms) which we confirmed experimentally.

Nonetheless, this setting is intrinsically parallelizable: the 64 gadgets of each round can be performed in parallel. We implemented parallelization using the module `Rayon` of Rust, which made the total timings drop to 13 seconds on our machine.

Compared to [BSS+23] that implemented the same block cipher on an equivalent hardware with parallelism, our implementation is about 10 times faster. Table 6 shows the comparison. Note that in this paper, the probability of failure is not specified. As ours is pretty conservative, this is a good argument in favor of our framework.

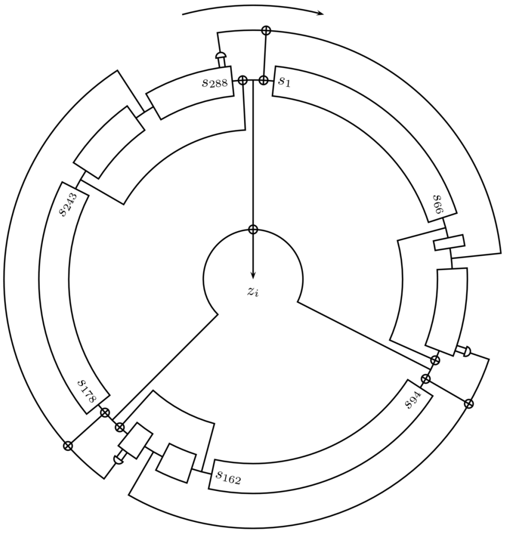


Figure 13: The trivium stream cipher. Figure extracted from [Can06]

7.2 The Trivium Stream Cipher

Trivium [Can06] is a stream cipher that uses a circular state. At each round, the bits are rotated within the state, except for three of them that are refreshed using the Boolean function of Section 7.1. The outer stream is generated by xoring three bits of the state each round once a "warming-up" phase is achieved.

For each generated key bit, it requires performing this function three times and aggregating five `XOR` operations in the center. Our strategy is to evaluate the refreshing function three times per round with one PBS for each of them, then get the result in $\mathbb{Z}_2$ and chain the five `XOR` operations to get the output. Figure 13 illustrates the layout of the cipher.

In [BOS23], the authors implement Trivium using the original `tfhe-rs` library, with 2 bits of message and 2 bits of carry for a total of 4 significative bits out of the 32 of a ciphertext component. They call this mode the `shortint` mode. The use-case they target is transciphering.

To compare our implementation with the one of [BOS23], timings are not a good metric as in their work they are provided on a massive AWS instance with a significant amount of parallelism. A better metric is to count the number of PBS and compare the parameter sets.

We reproduced the PBS operation with their parameter set on our machine and then simply estimated the timings of one round of Trivium with their approach with no parallelism. The results are summed up in Table 1. Note that in our implementation we do not refresh the output bits with a PBS after the chain of `XOR`, because in the use-case of transciphering one more `XOR` has to be performed with the message. We take advantage of this and move the last PBS into the transciphering phase.

Table 1: Comparison of timings of one round of Trivium between our work and [BOS23], with $\epsilon = 2^{-40}$.

Instance	Timing PBS	Number of PBS per round	Estimated timings
[BOS23]	6.6	7	46.2 ms
Our work	9.5	3	28.5 ms

7.3 Keccak Permutation

Keccak is a hash function standardized by NIST under the name *SHA-3* [NIS15]. It is a sponge function, whose transformation is called the *Keccak permutation*. It consists of five sub-functions: θ, ρ, π, χ, and ι.

Let us recall that our approach encrypts each bit in one TFHE ciphertext. Let us explain the stategies of homomorphization of these sub-functions:

- ρ and π simply reorder the bits within the state, so they are not impacted by the homomorphization.
- θ is just a serie of `XOR` operations, so it can be performed with a serie of homomorphic additions and without any PBS provided that the input ciphertexts are defined over $\mathbb{Z}_p$ with $p = 2$.
- χ is the only non-linear function of the permutation, and has to be performed with a PBS. It is the transformation that applies the function defined by

$$f_\chi(a, b, c) = a \oplus c \oplus b\&c$$

  to get each bit of the output state.
- Finally, ι performs a simple `xor` with a constant, so it can be handled in a similar manner that θ. The difference is that the constant is in clear this time.

The p-encodings we use are:

- $\mathcal{E}_\& = \begin{cases} 0 \mapsto \{1\} \\ 1 \mapsto \{2\} \end{cases}$ with $p_\& = 3$ to evaluate the & operator in the alternative formula of χ.
- $\mathcal{E}_\oplus = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{1\} \end{cases}$ with $p_\oplus = 2$ for the other operations of $\oplus$.

Our strategy of homomorphic evaluation of the Keccak permutation is as follows:

1. Encrypt the input state under the encoding $\mathcal{E}_\oplus$.
2. Evaluate the subfuctions θ, ρ, and π. Theses functions being purely linear, they can be performed only with sums under $\mathcal{E}_\oplus$.
3. Change the encoding from $\mathcal{E}_\oplus$ to $\mathcal{E}_\&$ with one PBS per bit of the state (Property 5).
4. Evaluate the `AND` operator of the subfunction χ with the gadget

$$\Gamma_\& = ((\mathcal{E}_\&, \mathcal{E}_\&), \mathcal{E}_\oplus, 3, 2)$$

   associated to function $f_\& : (x, y) \mapsto x\&y$. This gadget is applied once per bit of the state.
5. Evaluate the remaining $\oplus$ operators of χ and the ι subfunctions, then jump back Step 2. for the next loop iteration.

Casting a ciphertext from $\mathcal{E}_\oplus$ to $\mathcal{E}_\&$ (Step 3) is a bit tricky because $p_\oplus = 2$ is even. Because of the negacyclicity problem, one needs $\mathcal{E}_\&(0) = [-\mathcal{E}_\&(1)]_{p_\&}$. With $p_\& = 3$, the only candidate is the encoding $\mathcal{E}_\&$ defined above.

As a result, each round takes two programmable bootstrappings per bit. An implementation with our tweaked version of `tfhe-rs` takes 16.5 seconds (without any parallelism) on our hardware to perform one Keccak round on a state of 1600 bits in spite of the two PBS required per round and per bit. Those timings are possible because of the small values of p allowing the use of a set of small parameters, which speeds up the computation. A full run of Keccak counting 24 rounds, we can then estimate the timings without parallelism to 6.6 minutes. For the sake of simplicity, we use the same set of parameters for both types of PBS, avoiding the hassle of using two different server keys.

This strategy of implementation complies with the more generic one that we introduce in Section 7.4 and that is illustrated on Figure 15. It suits very well the use-cases where linear and non-linear operations are alternating.

7.4 Ascon

Ascon [DEMS21] is a lightweight block cipher algorithm that was designed to provide efficient and secure encryption and authentication for a wide range of applications, particularly in resource-constrained environments such as embedded systems and IoT devices. The name "Ascon" stands for "Authenticated encryption for Small Constrained Devices". We implemented its s-box, whose circuit is represented on Figure 14.

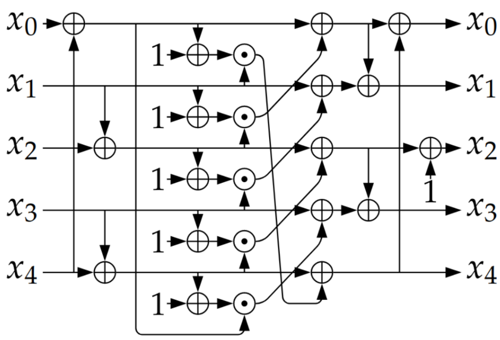


Figure 14: The 5-bits look-up table of ASCON. Figure extracted from [DEMS21]

This layout is a bit different from the others: the s-box takes five bits as input and outputs five bits. We denote $f_0, \ldots, f_4$ the five functions of $\mathbb{B}^5 \mapsto \mathbb{B}$ that generate the 5 output bits $x_0, \ldots, x_4$. Thus, we need to define five gadgets (one per function).

These functions, once analyzed by the algorithm, can be computed in one single bootstrapping each, but for different values of p (respectively $p = 17, 7, 7, 15, 11$ that are the smallest possible values). We could implement the gadgets $\Gamma_0, \ldots, \Gamma_4$ (associated to $f_0, \ldots, f_4$) with different values for p_{in}, but this would imply to introduce some encoding switchings before each round of hashing. To keep things simpler we generated only encodings with $p = 17$, making the implementation more straightforward as no encoding switching is required. For each subfunction f_i, five canonical 17-encodings $(\mathcal{E}_{i,0}, \ldots, \mathcal{E}_{i,4})$ of form

$$\mathcal{E}_{i,j} = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{d_{i,j}\} \end{cases}$$

are computed. The results are displayed in the Table 2. Note the zero values in some cases, they show that the variable is not used in the subfunction.

The s-box layer is followed by a linear layer, where the bits of the states are shifted and combined with `XOR` operations. This can be trivially done with $p = 2$. Finally, to prepare the next round, an encoding switching is performed to send back the ciphertexts on 17-encodings. This is summed up in Figure 15. Note that there is no encoding switching

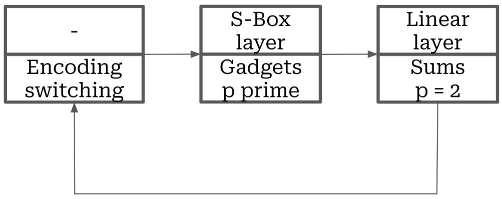


Figure 15: A common layout to evaluate cryptographic primitives. The upper part of the boxes represents what happens in the clear, while the lower part shows the encrypted operations.

Table 2: Parameters $d_{i,j}$ for Ascon, with $p = 17$ for every subfunction.

subfunction	$d_{i,0}$	$d_{i,1}$	$d_{i,2}$	$d_{i,3}$	$d_{i,4}$
f_0	1	2	3	7	14
f_1	1	2	2	2	4
f_2	1	2	2	4	0
f_3	1	1	5	5	3
f_4	1	2	0	4	3

from non-linear layer to linear layer because the gadgets can directly outputs ciphertexts under $\mathcal{E}_{\oplus} = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{1\} \end{cases}$ with $p = 2$.

To wrap up, we construct the five gadgets $\Gamma_i = ((\mathcal{E}_{i,0}, \ldots, \mathcal{E}_{i,4}), \mathcal{E}_{\oplus}, 17, 2, f_i)$. They will carry the evaluation of the s-boxes and output ciphertexts encrypted under $\mathcal{E}_{\oplus}$. Then, the linear layer is trivially evaluated with homomorphic sums. An encoding switching from $\mathcal{E}_{\oplus}$ to $\mathcal{E}_{i,j}$ allows to come back to non-linear operations.

Using this solution, the s-box is evaluated in 92 ms. Note that the 5 different PBS described in Table 2 have different norms of vector $\mathbf{d}$ so they may have a different set of parameters for each. We use the more restrictive one (i.e. the one with greater $\|\nu\|$) for the 5. Estimating the timings of a full run of Ascon is not trivial because it depends a lot of the parameters. To give a rough idea, in hashing mode, 64 s-boxes are required per round, with 12 rounds recommended. The outputs of the s-boxes are in $\mathbb{Z}_2$ to allow the evaluation of the linear layer of Ascon. At the end of this linear layer, the encoding of each of the 320 bits of the state must be switched back to $\mathbb{Z}_{17}$ with a PBS. To do so, we use the same set of parameters as for the encoding switching in Step 3 of the Keccak evaluation in Section 7.3.

This gives an estimation of 89 seconds for one Ascon hash.

7.5 AES

AES [DR00], or Advanced Encryption Standard, stands as one of the most widely used and trusted encryption algorithms in the world of computer security. Its standardization occured in 2001 when it was adopted by NIST to replace the obsolete DES (Data Encryption Standard). Implementing this primitive in FHE is known as particularly tricky and only few attempts have been made [GHS12], [CLT14], [TCBS23].

A round of AES can be decomposed into 4 steps:

1. `SubBytes`: a non-linear substitution step where each byte is replaced by another according to a lookup table. This step concentrates all the challenge for homomorphization, the other one being trivial with our framework.

2. `ShiftRows`: a transposition step where the last three rows of the state are shifted cyclically a certain number of times. As our framework encrypts each bit in a distinct ciphertext, this step is for free.

3. `MixColumns`: a linear mixing operation which operates on the columns of the state, combining the four bytes in each column. This step can be implemented using only `XOR` operations and bit-shiftings. The former are trivial with our framework using $p = 2$ and the latter are for free as the ones in the previous step.

4. `AddRoundKey`: each byte of the state is combined with a byte of the key from the key schedule using a `XOR`. Still using $p = 2$, this can be carried out easily.

The s-box of `SubBytes` takes 8 bits in input and yields 8 bits of output. It is defined by two substeps: an inversion in $GF(2^8)$ followed by an affine transformation. While the latter is trivial to compute with TFHE, the former is much trickier and thus we did not take advantage of this representation. Using our framework, the obvious-looking solution is to split the full s-box $\mathbb{B}^8 \mapsto \mathbb{B}^8$ into 8 subfunctions $f_0, \ldots, f_7 : \mathbb{B}^8 \mapsto \mathbb{B}$. We could then give them to the search algorithm of Section 4. If this would work, we could evaluate the Rjindael s-box in 8 PBS. Unfortunately, the algorithm does not converge for values of p "reasonable", that is to say less than 7 bits.

We thus need to leverage an alternative representation of the s-box. A well known efficient Boolean representation of the AES s-box is given in [BP10]. In this work, authors applied logic minimization techniques to produce an optimized Boolean circuit (in terms of number of gates) of the s-box splitted in 3 phases:

1. A purely linear layer mapping the 8 input bits onto 22 bits.

2. A middle non-linear layer, represented by a circuit with exclusively `AND` and `XOR` logic gates, mapping the previous 22 bits onto 18 bits.

3. A final purely linear layer mapping the 18 bits on the 8 output bits of the s-box.

To design our implementation of AES, we will use the strategy we introduced for Keccak (Section 7.3) and ASCON (Section 7.4) and that is illustrated on Figure 15. The steps `ShiftRows`, `MixColumns`, `AddRoundKeys` only involves `XOR` operators, so we will carry them out with $p = 2$. Same things with the steps 1. and 3. of the circuit of `SubBytes` of [BP10]. The only part remaining is the Step 2. of the `SubBytes`, that is a non-linear circuit. We evaluate this circuit using gadgets and the approach introduced in Section 5. A layer of encoding switching allows to link both parts.

In particular, `MixColumns` can be reduced to a serie of `XOR` (in our implementation, we use the circuit designed in [Max19]).

In the following, we focus on the implementation of the non-linear layer using the approach by graphs of Section 5.

7.5.1 Homomorphization of the S-box

We start from the circuit representation given in the work of [BP10]. This set of instructions is compiled into a circuit $\mathcal{A}$, compliant with the definitions introduced in Section 5.1.

Each of the 18 outputs $(z_0, \ldots, z_{17})$ are isolated from each other and the circuits $(\mathcal{A}_0, \ldots, \mathcal{A}_{17})$ generating them are separated. Of course, some intermediary values are used in several circuits, but for now we ignore this and we considerate the 18 problems as independent from each other.

Then, for each circuit $\mathcal{A}_i$, we run the algorithm explained in Section 5 to produce an efficient graph. We merge all those graphs and run everything for a total of 36 PBS to evaluate the full circuit $\mathcal{A}$, with a global $p = 11$. This allows a relatively quick bootstrapping.

Recall that the `SubBytes` step is made of 16 s-boxes. So, we can derive that one execution of the `SubBytes` step takes $16 \times 36 = 576$ PBS.

The outputs of this step would be encoded with $p = 2$, allowing the `XOR` operations of the following steps to be performed efficiently. We also need to take into account the encoding switching to come back to $p = 11$ before each `SubBytes`. It costs one PBS per bit, so 128 PBS. Finally, this gives a total of 704 PBS per round. For `AES-128`, which takes 10 rounds, we estimate a full run to 7040 PBS.

7.5.2 Performances

In terms of performances, with a set of parameters ensuring a security level of $\lambda = 128$ bits and an error probability $\epsilon = 2^{-40}$, a PBS takes 17 ms on our hardware. The total runtime of the whole implementation on one thread is 135 s. We note that the 16 evaluations of s-boxes in `SubBytes` can be parallelized, as well as each of the 128 encoding switchings before `SubBytes`. Moreover, within each s-box, we can locally apply our strategy of parallelization introduced in Section 5.3.

We compare favorably to previous works of [GHS12] and [CLT14], who report timings of respectively 18 minutes and 5 minutes for a full AES, Once again, authors do not mention the value of ϵ. The more recent work of [TCBS23], also proposes an implementation of `AES-128` using a completely different technique called the *tree-bootstrapping*. On a similar experimental setup, but with a failure probability $\epsilon = 2^{-23}$, they claim an execution in 270 s on one thread. We ran again our code with an other set of parameters tailored for the same ϵ and obtained a full run in 103 s. Note that in our implementation, we used the mode restrictive set of parameters $\mathtt{PBS}_{(11,4)}$ for every bootstrapping (even the ones that should be performed with $\mathtt{PBS}_{(2,1}$. We also derived the theoretical timing that could have been achieved if we had implemented this with two server keys (one for each set of parameters). This theoretical timing should be of 105 s with $\epsilon = 2^{-40}$, we added it in Table 6.

7.6 Summary of Applications

We summarize hereafter the parameters and performances of our implementations of cryptographic primitives. Table 3 gives an overview of the TFHE parameters used for each value of p in these examples. They all meet the required level of security of 2^{128} and the error probability $\epsilon = 2^{-40}$. It also shows the associated p and the norm of $\mathbf{d}$, denoted by $\mathcal{N}_{\mathbf{d}}$ (that corresponds to $\mathcal{N}_{\mathbf{d}} = \lceil \log_2(\|\mathbf{d}\|) \rceil$) that are the input of the parameter selection algorithm. To allow the comparison with the strategy of gate bootstrapping, we also included the set of parameters hardcoded in `tfhe-rs` to evaluate boolean operators. Table 4 shows the complexity of the cryptographic primitives expressed in PBS with our framework. It can be compared with Table 5, that illustrates the number of PBS required with the naive strategy of gate bootstrapping. Finally, Table 6 shows the concrete performance achieved by our implementations on our machine, as well as the comparison with other works and with the gate bootstrapping. For more information about an implementation or a comparison, the reader is referred to the related section.

8 Conclusion

In this paper, we have proposed a new strategy to evaluate Boolean functions homomorphically using TFHE. Our technique relies on constructing an intermediate homomorphic layer between the Boolean space $\mathbb{B}$ of the plaintexts and the torus $\mathbb{T}_q$ on which ciphertexts live. We introduced a formal model for our technique and detailed algorithms to efficiently construct such layers and select appropriate parameters. We further extended our strategy

Table 3: Sets of TFHE parameters for each PBS used in our implementations, with the constraints used to generate the sets, and the performances. Each setting is referenced as $\mathsf{PBS}_{(p,\mathcal{N}_d)}$ with $\mathcal{N}_d = \lceil \log_2(\|d\|) \rceil$. All this parameters ensure a level of security $\lambda = 128$ bits and a failure probability of bootstrapping of $\epsilon = 2^{-40}$. q is always fixed to 2^{32}. PBS_{gate} refers to the naive case of the gate bootstrapping implemented in [Zam22b] and is used to estimate the timings of the naive strategy in Table 6.

Identification		**TFHE parameters**									**Timings**
Ref.	Sections	n	k	N	σ_{LWE}	σ_{GLWE}	B_g	ℓ_g	B_{KS}	ℓ_{KS}	PBS
PBS_{gate}	Table 5	722	2	512	$2^{16.2}$	$2^{7.8}$	2^6	3	2^3	4	10 ms
$\mathsf{PBS}_{(9,2)}$	7.1, 7.2	684	3	512	2^{16}	$2^{2}2$	2^{10}	2	2^3	4	9.5 ms
$\mathsf{PBS}_{(3,2)}$	7.3	676	5	256	2^{22}	$2^{7}7$	2^{18}	1	2^4	3	5.25 ms
$\mathsf{PBS}_{(2,1)}$	7.3, 7.4	676	5	256	2^{22}	$2^{7}7$	2^{18}	1	2^4	3	5.25 ms
$\mathsf{PBS}_{(17,5)}$	7.4	740	2	1024	2^{13}	$2^{2}2$	2^7	3	2^5	3	18 ms
$\mathsf{PBS}_{(11,4)}$	7.5	708	3	512	2^{15}	$2^{2}2$	2^6	4	2^2	7	17 ms

Table 4: Complexity of the different primitives we implemented with respect to the PBS of Table 3. The primitives indicated by a $(*)$ are estimations while the others have been fully implemented.

Section	**Primitive**	**Complexity in PBS**
7.1	One round of SIMON-128	64 $\mathsf{PBS}_{(9,2)}$
	One full run of SIMON-128	4352 $\mathsf{PBS}_{(9,2)}$
7.2	One round of Trivium	3 $\mathsf{PBS}_{(9,2)}$
	One warm-up phase of Trivium $(*)$	3456 $\mathsf{PBS}_{(9,2)}$
7.3	One round of Keccak	1600 $\mathsf{PBS}_{(3,2)}$ + 1600 $\mathsf{PBS}_{(2,1)}$
	A full Keccak permutation $(*)$	38400 $\mathsf{PBS}_{(3,2)}$ + 38400 $\mathsf{PBS}_{(2,1)}$
7.4	One evaluation of Ascon's s-box	5 $\mathsf{PBS}_{(17,5)}$
	One full Ascon hashing run $(*)$	3840 $\mathsf{PBS}_{(17,5)}$ + 3840$\mathsf{PBS}_{(2,1)}$
7.5	One evaluation of the AES s-box	36 $\mathsf{PBS}_{(11,4)}$
	A full run of `AES-128`	5760 $\mathsf{PBS}_{(11,4)}$ + 1280 $\mathsf{PBS}_{(2,1)}$

Table 5: Number of logic gates in the circuit of each primitive. This shows the heavy cost of the naive method of performing one bootstrapping per gate (except the `NOT` ones).

Section	**Primitive**	**Number of logic gates**
7.1	One round of SIMON-128	256
	One full run of SIMON-128	17408
7.2	One round of Trivium	13
	One warm-up phase of Trivium $(*)$	14976
7.3	One round of Keccak	7687
	A full Keccak permutation $(*)$	184488
7.4	One evaluation of Ascon's s-box	16
	One full Ascon hashing run $(*)$	19968
7.5	One evaluation of the AES s-box	115 ([BP10])
	A full run of `AES-128`	23360 ([BP10], [Max19])

Table 6: Timings of evaluation of full primitives, and comparison with previous works when they exist. Like on Table 4, a star ($*$) is added in the cells if our timing is not obtained from a full implementation but estimated from an implemented building block. Also, the security level of each implementation is $\lambda = 128$ and the default error probability is $\epsilon = 2^{-40}$. The concurrent works that do not indicates their ϵ are marked with †.

Primitive	**Section or Other work**	**Performances**
One full run of SIMON	Gate Bootstrapping	174 s
	[BSS+23] †	128 s
	Our work (Section 7.1)	10 s
One warm-up phase of Trivium (*)	Gate Bootstrapping	1498 s
	[BOS23] (estimation on our machine)	53 s
	Our work (Section 7.2)	32.8 s
One Full Keccak permutation ($*$)	Gate Bootstrapping	30.7 min
	Our work (Section 7.3)	8.8 min
One Ascon hashing ($*$)	Gate Bootstrapping	200s
	Our work (Section 7.4)	92 s
One full evaluation of AES-128 ($\epsilon = 2^{-23}$) on one thread	[GHS12] †	18 min
	[CLT14] †	5 min
	[TCBS23]	270 s
	Our work (Section 7.5)	103 s
One full evaluation of AES-128 ($\epsilon = 2^{-40}$) on one thread	Gate Bootstrapping	234 s
	Our work (Real implementation)	135 s
	Our work (Theoretical timing with two keys)	105 s

to the case of arbitrary Boolean circuits by developing some heuristic to decompose a circuit into Boolean functions efficiently evaluable with our framework. We applied our framework to various cryptographic primitives, in particular to the challenging AES cipher. All the reported implementations outperform the state of the art.

We are currently working on a generalization of the ideas developed in this paper to the arithmetic case. We would also like to experiment with some more sophisticated bootstrapping techniques, such that the multi-value bootstrapping introduced in [CIM19] that would allow to evaluate several gadgets at once. Moreover, future work may be focused on the search algorithm (thet can probably be enhanced to scale better with the arity of the input function). Finally, more work on the efficient decomposition of Boolean circuits would be welcome: especially if one wants to evaluate deeper circuits.

References

[BBB+23] Loris Bergerat, Anas Boudi, Quentin Bourgerie, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Parameter optimization and larger precision for (T)FHE. *J. Cryptol.*, 36(3):28, 2023.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012*, pages 309–325. ACM, January 2012.

[BOS23] Thibault Balenbois, Jean-Baptiste Orfila, and Nigel P. Smart. Trivial transciphering with trivium and TFHE. In Michael Brenner, Anamaria Costache, and Kurt Rohloff, editors, *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Copenhagen, Denmark, 26 November 2023*, pages 69–78. ACM, 2023.

[BP10] Joan Boyar and René Peralta. A new combinational logic minimization technique with applications to cryptology. In Paola Festa, editor, *Experimental Algorithms, 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings*, volume 6049 of *Lecture Notes in Computer Science*, pages 178–189. Springer, 2010.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 868–886. Springer, Heidelberg, August 2012.

[BSS+15] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK lightweight block ciphers. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 175:1–175:6. ACM, 2015.

[BSS+23] Adda-Akram Bendoukha, Oana Stan, Renaud Sirdey, Nicolas Quero, and Luciano Freitas. Practical homomorphic evaluation of block-cipher-based hash functions with applications. Cryptology ePrint Archive, Paper 2023/480, 2023. https://eprint.iacr.org/2023/480.

[Can06] Christophe De Cannière. Trivium: A stream cipher construction inspired by block cipher design principles. In Sokratis K. Katsikas, Javier López, Michael Backes, Stefanos Gritzalis, and Bart Preneel, editors, *Information Security, 9th International Conference, ISC 2006, Samos Island, Greece, August 30 - September 2, 2006, Proceedings*, volume 4176 of *Lecture Notes in Computer Science*, pages 171–186. Springer, 2006.

[CCS19] Hao Chen, Ilaria Chillotti, and Yongsoo Song. Improved bootstrapping for approximate homomorphic encryption. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 34–54. Springer, Heidelberg, May 2019.

[CGGI18] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: Fast fully homomorphic encryption over the torus. Cryptology ePrint Archive, Paper 2018/421, 2018. https://eprint.iacr.org/2018/421.

[CHK+18] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 360–384. Springer, Heidelberg, April / May 2018.

[CIM19] Sergiu Carpov, Malika Izabachène, and Victor Mollimard. New techniques for multi-value input homomorphic evaluation and applications. In Mitsuru Matsui, editor, *Topics in Cryptology – CT-RSA 2019*, pages 106–126, Cham, 2019. Springer International Publishing.

[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, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 409–437. Springer, Heidelberg, December 2017.

[CLOT21] Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part III*, volume 13092 of *LNCS*, pages 670–699. Springer, Heidelberg, December 2021.

[CLT14] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. Scale-invariant fully homomorphic encryption over the integers. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 311–328. Springer, Heidelberg, March 2014.

[DEMS21] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2: Lightweight authenticated encryption and hashing. *J. Cryptol.*, 34(3):33, 2021.

[DR00] Joan Daemen and Vincent Rijmen. The block cipher rijndael. In Jean-Jacques Quisquater and Bruce Schneier, editors, *Smart Card Research and Applications*, pages 277–284, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. volume 9, pages 169–178, 05 2009.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 850–867. Springer, Heidelberg, August 2012.

[Joy22] Marc Joye. SoK: Fully homomorphic encryption over the [discretized] torus. *IACR TCHES*, 2022(4):661–692, 2022.

[LLL+20] Joon-Woo Lee, Eunsang Lee, Yongwoo Lee, Young-Sik Kim, and Jong-Seon No. High-precision bootstrapping of rns-ckks homomorphic encryption using optimal minimax polynomial approximation and inverse sine function. Cryptology ePrint Archive, Paper 2020/552, 2020. https://eprint.iacr.org/2020/552.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23. Springer, Heidelberg, May / June 2010.

[Max19] Alexander Maximov. Aes mixcolumn with 92 xor gates. Cryptology ePrint Archive, Paper 2019/833, 2019. https://eprint.iacr.org/2019/833.

[NIS15] NIST. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, 2015. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 84–93. ACM, 2005.

[TCBS23] Daphné Trama, Pierre-Emmanuel Clet, Aymen Boudguiga, and Renaud Sirdey. A homomorphic AES evaluation in less than 30 seconds by means of TFHE. In Michael Brenner, Anamaria Costache, and Kurt Rohloff, editors, *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Copenhagen, Denmark, 26 November 2023*, pages 79–90. ACM, 2023.

[Zam22a] Zama. concrete-optimizer : Concrete Optimizer is a Rust library that find the best cryptographic parameters for a given TFHE homomorphic circuit., 2022. https://github.com/zama-ai/concrete/tree/main/compilers/concrete-optimizer.

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