A Fast Heuristic for Mapping Boolean Circuits to Functional Bootstrapping

Sergiu Carpov

Arcium, Baar, Switzerland, sergiu@arcium.com

Abstract. Functional bootstrapping in FHE schemes such as `FHEW` and `TFHE` allows the evaluation of arbitrary functions on encrypted data, while simultaneously reducing noise. Implementing programs that directly use functional bootstrapping is challenging and error-prone. In this paper, we propose a heuristic that automatically maps Boolean circuits to functional bootstrapping instructions. Unlike prior approaches, our method does not limit the encrypted data plaintext space to a power-of-two size, allowing the instantiation of functional bootstrapping with smaller parameters. Furthermore, the negacyclic property of functional bootstrapping is exploited to extend the plaintext effective space. Despite the inherently greedy nature of the heuristic, experimental results show that the mapped circuits exhibit a significant reduction in evaluation time. Our heuristic demonstrates a 45% reduction in evaluation time when compared to hand-optimized Trivium and Kreyvium implementations.

Keywords: Functional bootstrapping · Boolean circuit mapping · Fully Homomorphic Encryption

1 Introduction

Fully homomorphic encryption (FHE) is an encryption scheme that enables the direct execution of arbitrary computations on encrypted data. The first FHE scheme was introduced by Gentry in his seminal work [Gen09]. The construction relies on a technique called *bootstrapping*, which is used to reduce noise in FHE ciphertexts. This construction theoretically enables the execution of any computation directly over encrypted data but remains slow in practice. Many works [FV12, BGV14, DM15, CGGI16a, CGGI20] have built upon Gentry's initial proposal and have contributed to further improvements in the efficiency of FHE.

FHE schemes are typically classified into two main categories. The first category of FHE schemes is based on Gentry's initial proposal. While the bootstrapping procedure is relatively time-consuming, it enables the efficient packing of data through the use of batching techniques. Typically ciphertexts are bootstrapped as rarely as possible following the evaluation of numerous homomorphic operations. The second type of FHE schemes is based on the `GSW` somewhat homomorphic scheme, which was proposed in 2013 by Gentry [GSW13] and supports branching programs with polynomial noise overhead. These schemes are referred to as *fast bootstrapping* schemes. One limitation of these schemes is that they can only bootstrap one message at a time, but the bootstrapping procedure is relatively fast. One of the key benefits of these schemes is that they can be used to compute an arbitrary function while simultaneously reducing noise. We refer to it as *functional bootstrapping* (FBS). The `FHEW` scheme [DM15] introduced a FBS procedure which evaluates a `nand` gate in addition to noise reduction, and suggested an extension for other/larger gates. In a subsequent work [BR15] the `FHEW` scheme was adapted to accommodate arbitrary multi-input Boolean gates. The authors of [CGGI16a, CGGI20] further enhanced these

designs and introduced the `TFHE` library [CGGI16b]. `TFHE`'s bootstrapping implementation can execute any two-input Boolean gate in approximately 10 milliseconds. In [CIM19], the authors propose a bootstrapping method for evaluating several functions on the same inputs at once, which was further improved in [GBA21].

In order to evaluate functions with several inputs, it is necessary to linearly combine them into a single value before the functional bootstrap. We denote it as multi-input FBS. Linear combinations are fast and are implemented using scalar-ciphertext multiplications and ciphertext-ciphertext additions. Typically, the binary composition function $\sum_i x_i \cdot 2^i$ is used for evaluating any Boolean functions with n inputs. However, this approach has a significant drawback: the required plaintext space size is exponential in the number of function inputs. To address this limitation, we can use linear combinations with smaller plaintext space sizes for specific Boolean functions. One example are the *symmetric* Boolean functions, where the output depends only on the number of activated inputs and not on their position.

Motivation

Boolean circuits are evaluated in a gate-by-gate manner using fast bootstrapping schemes. Logic synthesis tools can be used to map circuits to a library of Boolean gates that are supported by a particular FHE scheme. As an example, in the case of `TFHE`, the library contains the complete set of 2-input gates. Another option is to map the Boolean circuit to lookup tables (LUT) and evaluate them homomorphically as generic n-input functions. Both of these solutions are straightforward to implement because Boolean circuit mapping is a well-studied problem in the field of logic synthesis, and there are plenty of performant tools available [Ber, SRH+22].

Limiting the evaluation to generic n-input gates is not the most efficient approach. As mentioned before, symmetric Boolean gates require smaller FHE parameters, resulting in faster processing times. A FBS parameterized for generic n-input gates can be used to evaluate any symmetric Boolean function with up to $2^n - 1$ inputs. Additionally, it should be noted that FBS with power-of-two plaintext space is not always needed.

As an example, the full-adder is a logic circuit which computes the sum of three input bits and outputs it on two bits. The minimal number of 2-input gates (or FBS with plaintext $\mathbb{Z}_4$) required for this circuit is 5. However, when mapping this circuit to 3-input LUTs (or FBS with plaintext $\mathbb{Z}_8$) only 2 are needed. Furthermore, this circuit can be implemented with 2 FBS with a plaintext space of only $\mathbb{Z}_3$ because full-adder outputs are symmetric functions, or as low as 1 FBS if multi-output technique from [CIM19, GBA21] is used.

Contribution

We propose a new heuristic method which automatically maps Boolean circuits to functional bootstrapping. The algorithm takes a circuit composed of two-input gates and the FBS parameters (plaintext space size and linear combination norm) as input. The nodes of the circuit are merged together into larger nodes as long as they can be evaluated using the given FBS procedure parameters. Nodes are visited only once in a topological order. The algorithm employs a greedy strategy which tries to merge nodes for as long as possible, with the goal, supposedly, of minimising the total number of bootstrappings. The size of the FBS plaintext is not limited to power-of-two values. The negacyclic property of the TFHE FBS implementation is used to enhance the performance of the mapped circuits.

We have implemented a proof of concept for this heuristic, which is publicly available. The heuristic has been tested on a number of circuit benchmark suites, including the EPFL combinational benchmarks, ISCAS 85 and ISCAS 89. It has also been applied to Boolean circuits implemented by hand, namely the Trivium/Kreyvium cipher and other

cryptographic primitives. The estimated evaluation time of the mapped circuits has been compared to the performance of non-mapped versions, specifically the original circuit evaluated with generic 2-input gates. Our heuristic consistently finds circuits that are more efficient than their non-mapped counterparts. In comparison to manual Kreyvium FBS implementations from [BOS23], our heuristic identified a Kreyvium implementation that uses 20% less FBSs and has a 45% lower evaluation cost.

The fast bootstrapping schemes `TFHE/FHEW` are particularly well suited for evaluating Boolean circuits and logic-based machine learning models, such as decision trees and random forests. Functional bootstrapping, combined with our heuristic, enables low-latency execution of such logic-heavy applications. In contrast, arithmetic FHE requires inefficient encoding for Boolean circuit evaluations, leading to substantial overheads.

Existing works

The AutoHoG method from [GMZ+24] presents an automated approach for mapping Boolean circuits to FBS. A procedure for optimising multi-input FBS linear combination coefficients is proposed. The objective is to maximise the number of inputs in each FBS, which should subsequently reduce execution time. The authors use `TFHE` FBS with $\mathbb{Z}_{32}$ in the benchmarks and do not consider other plaintext spaces. Another distinction from our work is that AutoHoG is more resource-intensive as it attempts to optimise the linear combination coefficients for multi-input sub-circuits. In comparison, in our work we restrict the search to linear combinations with two coefficients.

Helm [GMT25] is a framework for circuit synthesis, mapping and execution targeting `TFHE` gate or functional bootstrapping. The authors of [MKG23] introduce a circuit mapping to LUTs and consider the special case of full-adders being a symmetric Boolean function. Furthermore, a post-synthesis step is employed which groups several LUTs into one multi-output LUT, leveraging the results from [CIM19]. These works follow the standard approach to logic synthesis tools and do not take into account the specific characteristics of FBS.

Another line of research introduces gate libraries with either multi-input gates [MHSB21] or uses alternative plaintext space sizes as [CBS24]. In their work, the authors of [MHSB21] show how to evaluate 3-input gates using an extended plaintext space FBS. The Chocobo paper [CBS24] generalises binary logic gates to base-B gates which are computed as an FBS. A variety of approaches to computing two-input B-gates are presented. The chaining method corresponds to the multi-input FBS with plaintext space $\mathbb{Z}_{B^2}$. However the authors do not consider specific B-gates which require a much smaller plaintext space.

A novel method of encoding Boolean values is introduced in [BPR24]. In contrast to the typical approach of using a fixed Boolean encoding scheme, namely the two-element set $\{0, 1\}$, the authors put forward a novel proposal: the use of a distinct Boolean encoding, denoted p-encoding, for each circuit wire. Each circuit wire has a unique set of potential values (from $\mathbb{Z}_p$) for each Boolean value. Our methodology is comparable to theirs. To provide some context, the sum of the p-encodings is equivalent to a linear combination of the two-element Boolean encoding used in our work, refer to Subsection 4.3 for more details. The authors present two methods for determining the p-encodings for the inputs of a Boolean function to be evaluated, one exact and one heuristic. A drawback of the proposed methods is the exponential complexity in the number of inputs ℓ of the Boolean function. In our work, we adopt an alternative approach to solving the problem for large Boolean functions. Rather than attempting to find a solution directly, we construct it in an iterative manner by examining a 2-input gate representation of the function in question. So, instead of searching for ℓ p-encodings (or linear combination coefficients), we only need to find 2.

The authors of [BOS23, TCBS23] present hand-optimised algorithms employing FBS. They demonstrate how to implement Trivium, Kreyvium and AES, showcasing various

optimisation techniques (negacyclic functions, larger than 2 plaintext spaces, etc.). However, they do not consider non-power-of-two plaintext spaces.

Paper organization

We begin with a comprehensive overview of functional bootstrapping in Section 2, followed by the proposed mapping heuristic in Section 3 and with experimental results in Section 4.

2 Multi-input functional bootstrapping

Notations

A vector of size n is denoted by $\boldsymbol{v}$, $\boldsymbol{v} = \{v_0, v_1, \ldots, v_{n-1}\}$, and the i-th vector element is v_i.

2.1 Functional bootstrapping

`TFHE` [CGGI16a, CGGI20] is a fully homomorphic encryption scheme with a fast bootstrapping procedure. The authors describe how to use functional bootstrapping to evaluate 2-input logic gate circuits. The bootstrapping procedure is implemented via a homomorphic accumulator. It evaluates the linear part of the decryption function, followed by the non-linear part. For this line of schemes, the structure of the bootstrapping can be divided in 4 steps:

1. The coefficients of an input `LWE` ciphertext $\boldsymbol{c} = (\boldsymbol{a}, b)$ are rounded to $\mathbb{Z}_{2N}$. A cyclic multiplicative group $\mathcal{G}$, where $\mathbb{Z}_{2N} \simeq \mathcal{G}$, is used for an equivalent representation of $\mathbb{Z}_{2N}$ elements. $\mathcal{G}$ contains all the powers $X^k \mod X^N + 1$, where $X^N + 1$ is the quotient polynomial defining the `RLWE` scheme.

2. The message phase encrypted in the input ciphertext $\boldsymbol{c}$ is transformed to a `RLWE` encryption of X^{φ}. The encryption X^{φ} is obtained by computing the linear transformation $b - \boldsymbol{a} \cdot \boldsymbol{s} \, (\approx \varphi)$ using `GSW` encryptions of X^{s_i} (i.e. bootstrapping key). We obtain the so-called accumulator ACC which contains an encryption of $X^{\varphi} \in \mathcal{G}$. This is the *linear step* of the `LWE` decryption algorithm.

3. The accumulator ACC is then multiplied with a *test polynomial* (or *test vector*) TV_F. The test polynomial encodes the output values of a function G for each possible input message phase $\varphi \in \mathbb{Z}_{2N}$, where G is a function from $\mathbb{Z}_{2N}$ to $\mathbb{Z}_p$. Function G is a composition of the "payload" function $F : \mathbb{Z}_p \to \mathbb{Z}_p$ and a rounding function $R_p : \mathbb{Z}_{2N} \to \mathbb{Z}_p$. The rounding function is needed because phase φ is a noised version of the actual message $m = R_p(\varphi)$ encrypted in $\boldsymbol{c} = (\boldsymbol{a}, b)$. The rounding function corresponds to the final *non-linear step* of ciphertext $\boldsymbol{c}$ decryption.

4. Finally, an `LWE` encryption of $F(m)$ (or $G(\varphi)$) is extracted from `RLWE` encryption $\mathsf{TV}_F \cdot X^{\varphi}$.

In what follows we consider the FBS as a generic method for evaluating functions $F : \mathbb{Z}_p \to \mathbb{Z}_p$. The input to this method is an `LWE` encryption of $m \in \mathbb{Z}_p$, and the output is also an `LWE` encryption of the function F applied on m. In the context of cyclotomic rings with modulus $X^N + 1$, the functional bootstrapping can be extended to negacyclic functions $F : \mathbb{Z}_{2p} \to \mathbb{Z}_p$, which verify the equality $F(x) = -F(x + p)$ for all $x \in \mathbb{Z}_p$.

2.2 Multi-input functional bootstrapping

The FBS procedure takes as input one encrypted message. In order to evaluate multi-input functions, the ciphertexts are linearly combined into a single ciphertext beforehand. We denote this bootstrapping procedure as *multi-input functional bootstrapping* due to its ability to evaluate generic multi-input functions.

The first step of multi-input FBS is a linear combination of inputs (LWE ciphertexts) with integer coefficients. The second step is a FBS procedure with a specially crafted test polynomial. The linear combination maps input Boolean values to an integer value, which is subsequently mapped back to a Boolean value by the single-input bootstrapping algorithm.

Let $\mathbb{Z}_p$ be the FBS message space (hereafter we ignore the fact that in the case of TFHE, messages are values on the torus and instead consider them scaled to $\mathbb{Z}_p$). Let f be an n-input Boolean function to be evaluated over Boolean values encrypted as LWE ciphertexts. Boolean values are encoded in LWE ciphertexts as either 0s or 1s. We denote the multi-input FBS as the composition of a linear function ϕ followed by a non-linear mapping $F : \mathbb{Z}_p \to \mathbb{Z}_2$, such that:

$$f = F \circ \phi.$$

The non-linear function F is embedded in the test vector. It maps each value of the image of the function ϕ to a Boolean value. The following sections will provide a more detailed overview of these two steps.

2.2.1 Linear combination

The function $\phi_{\boldsymbol{c}}(\boldsymbol{x})$ represents a linear combination, expressed as $\sum_{i=1}^{n} c_i \cdot x_i$, where the coefficients $\boldsymbol{c}$ are integers. A linear combination $\phi_{\boldsymbol{c}}$ can be used in a multi-input FBS to evaluate a logic function, f, if it is capable to distinguish the output values of f. In more formal terms, for any $\boldsymbol{x}$ and $\boldsymbol{x}'$ such that $f(\boldsymbol{x}) \neq f(\boldsymbol{x}')$ the linear combination must satisfy $\phi_{\boldsymbol{c}}(\boldsymbol{x}) \neq \phi_{\boldsymbol{c}}(\boldsymbol{x}')$.

As example, the binary composition function, $\sum_i x_i \cdot 2^i$, is a bijective linear combination that can be used to evaluate any Boolean function f. The mapping function is given by $F(z) = f(\boldsymbol{x})$ where $z = \sum_i x_i \cdot 2^i$. However, this approach requires an exponential FBS message space, $\mathbb{Z}_{2^n}$, where n is the number of function inputs.

Not all Boolean functions require a bijective linear combination. For example, the n-input majority function $\mathrm{MAJ}_n(\boldsymbol{x})$ (which outputs true if the majority of inputs are active) can be computed with the linear combination $\sum_i x_i$ and the mapping function $F(x) = x \geq n/2$. This linear combination is surjective and can only be used to evaluate a subclass of Boolean functions, in particular the symmetric functions. The required FBS message space, $\mathbb{Z}_{n+1}$, is significantly smaller when compared to the generic binary composition function.

Let us denote by $\mathrm{imsize}(\boldsymbol{c})$ the *image size* of the linear combination $\phi_{\boldsymbol{c}}$. It is defined as:

$$\mathrm{imsize}(\boldsymbol{c}) = \max_{\boldsymbol{x}} \phi_{\boldsymbol{c}}(\boldsymbol{x}) - \min_{\boldsymbol{x}} \phi_{\boldsymbol{c}}(\boldsymbol{x}) + 1$$

For the sake of simplicity, we assume that $\phi_{\boldsymbol{c}}(\boldsymbol{x}) \geq 0$. It is possible to transform any linear combination into an equivalent one that verifies the aforementioned relation by subtracting $\min_{\boldsymbol{x}} \phi_{\boldsymbol{c}}(\boldsymbol{x})$.

Let us suppose that $\mathrm{imsize}(\boldsymbol{c}) \leq p$. In this case function $f = F \circ \phi_{\boldsymbol{c}}$ can be evaluated by a multi-input FBS with message space $\mathbb{Z}_p$. The noise of the input LWE ciphertexts is inversely proportional to the Euclidean norm $\|\boldsymbol{c}\|_2$ and it should be chosen in such a way that the error amplitude of the linear combination over the LWE ciphertexts is smaller than $1/2$. This implies that the output noise of the linear combination should remain within one message space segment with overwhelming probability.

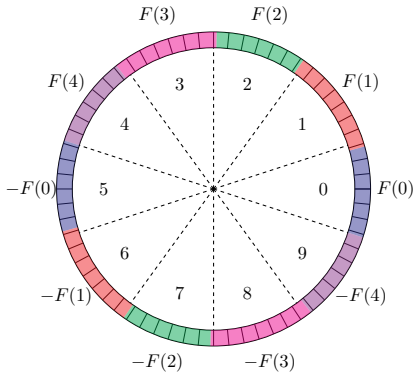


Figure 1: Example of FBS function encoding (colored segments) and message space (dashed lines separators).

To summarize, the precision of FBS depends on two factors: the plaintext space size, p, and the maximal Euclidean norm, l, of supported linear combinations. A FBS parameterised with p and l can be used to evaluate any other Boolean function (regardless of input count) whose linear combination, $\phi_{\boldsymbol{c}}$, verifies $\text{imsize}(\boldsymbol{c}) \leq p$ and $\|\boldsymbol{c}\|_2 \leq l$.

2.2.2 Blind rotation

The output of the linear combination evaluation over the encrypted values $\boldsymbol{x}$ is a `LWE` encryption of $\phi_{\boldsymbol{c}}(\boldsymbol{x})$. Each value of linear combination, $\phi_{\boldsymbol{c}}$, image is mapped to the corresponding value of the Boolean function f by the FBS procedure using a specific test vector TV. TV maps the integer value $\phi_{\boldsymbol{c}}(\boldsymbol{x})$ to the Boolean value $f(\boldsymbol{x})$ for every $\boldsymbol{x}$. The TV is defined as follows:

$$\mathsf{TV} = \sum_{0 \leq k < K} F(k) \sum_{\left\lfloor \frac{k \cdot N}{p} \right\rceil \leq i < \left\lfloor \frac{(k+1) \cdot N}{p} \right\rceil} X^i \mod X^N + 1.$$

In addition to the function F, this test vector encodes the rounding to Z_p function of the `LWE` decryption.

Figure 1 illustrates the message space partition for $p = 5$ and the function F encoded in the TV. For illustration purposes, a small `RLWE` ring size ($N = 32$) was selected. It should be noted that the encoding of the function and message space do not match exactly, as the test vector is discretised to $2 \cdot N$ values and message space elements are not. It is possible to extend the message space to $\mathbb{Z}_{2p}$ without incurring any additional cost for *negacyclic* functions F. Refer to the lower half of Figure 1 for negacyclic function encoding illustration.

Negacyclic function evaluation. A FBS parameterised for message space $\mathbb{Z}_p$ can be employed for negacyclic functions over $\mathbb{Z}_{2p}$. A function F is negacyclic if $F(x) = -F(x + p)$ for any $x \in \mathbb{Z}_p$.

Consider a Boolean function, $f = F \circ \phi_{\boldsymbol{c}}$, where the image size of $\phi_{\boldsymbol{c}}$ is larger than FBS parameter p, i.e. $\text{imsize}(\boldsymbol{c}) > p$. Three distinct types of negacyclic Boolean functions exist:

1. $F(x) = \neg F(x + p)$ – first and last function values are negated,
2. $F(x) = F(x + p) = 1$ – first and last function values are ones,
3. $F(x) = F(x + p) = 0$ – first and last function values are zeros.

These functions are evaluated as a FBS of function $F'(x) = F(x) - \mu$ for $x \in \mathbb{Z}_p$ followed by an addition of a constant μ. Here μ equals to $1/2$, 1 and 0 for each negacyclic function type respectively. It is straightforward to see that function F' is negacyclic. Furthermore, it can be shown that after the constant μ has been added, the original function F is restored.

3 Mapping Boolean circuits to functional bootstrapping

A Boolean circuit is a directed acyclic graph, denoted by $G = (V, E)$, where V is the set of nodes and E is the set of edges. A node $v \in V$ can be either a circuit input, a circuit output, or a logic gate. An edge, $(w, v) \in E$, is a directed connection from a source node w to a destination node v. The function $\text{pred}(v)$ returns the predecessors of a given node v, and is defined as $\text{pred}(v) = \{u \mid (u, v) \in E\}$. A gate node is associated with a Boolean function, $f_v(U)$, where $U = \text{pred}(v)$ represents the set of predecessors of v. A cone, denoted by C_v, is defined as a sub-set of the node v ancestors, including the node itself, such that for any $w \in C_v$ every path from w to v must lie entirely within C_v. The support of a cone, denoted by $sup(C_v)$, is a set of nodes that feed into the nodes in the cone but do not belong to it. Formally, this is expressed as $sup(C_v) = \{u \mid (u, w) \in E, w \in C_v\}$. The Boolean function f_{C_v} with inputs $sup(C_v)$ is defined as the logic function of cone C_v.

Our goal is to partition a Boolean circuit into a set of sub-circuits such that each sub-circuit can be executed by one functional bootstrapping. We call this problem *Boolean circuit mapping to functional bootstrappings*. Let p be the plaintext space size and l the linear combination Euclidean norm for which FBS has been parameterised. A solution to this problem is a set B of circuit nodes to bootstrap where for each node $v \in B$ we have a cone C_v and a vector of integer coefficients $\boldsymbol{c}_v$ (a coefficient per node in $sup(C_v)$) such that:

- B contains all circuit outputs,
- any circuit node belongs to at least one cone: $\bigcup_{v \in B} C_v = V$,
- linear mapping $\phi_{\boldsymbol{c}_v}$ is valid for cone logic function f_{C_v}: for any $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{Z}_2^{|\boldsymbol{c}_v|}$ such that $f_{C_v}(\boldsymbol{x}) \neq f_{C_v}(\boldsymbol{x}')$ we have $\phi_{\boldsymbol{c}_v}(\boldsymbol{x}) \neq \phi_{\boldsymbol{c}_v}(\boldsymbol{x}')$,
- FBS parameters are valid, i.e. $\text{imsize}(\boldsymbol{c}_v) \leq p$ and $\|\boldsymbol{c}_v\|_2 \leq l$.

Given a Boolean circuit the optimization problem is to find a mapping which minimizes circuit evaluation time. The evaluation time of a circuit depends on the input FBS parameters and on the number of bootstrappings in the mapped circuit.

3.1 Heuristic mapping

The following section introduces a heuristic that maps a Boolean circuit to FBSs with fixed parameters. The heuristic algorithm is given in Algorithm 1. The algorithm takes as inputs a Boolean circuit G, a cone composition function FindParams which returns a valid linear combination for a node v and functional bootstrapping parameters p and l. The functional bootstrapping parameters must support any 2-input Boolean gate, at least. The algorithm output is a solution to the Boolean circuit mapping to functional bootstrappings problem. The heuristic traverses the circuit in a topological order, starting from the input nodes. At each gate node, it attempts to merge cones rooted at this node. In case the merging is not possible, due to unsatisfied functional bootstrapping parameters, the heuristic bootstraps one or both node inputs.

An empty cone (line 3) is assigned to each circuit input. By convention, the logic function of an empty cone is the identity function, $f_{\{\}}(x) = x$ and it has an image size 2.

The circuit output nodes are added to the set B of nodes to bootstrap. For each gate node v, function FindParams returns a valid linear combination for the merged cone $\{v\} \cup C_u \cup C_w$ where u, w are the predecessors of v (line 9). In case a linear combination, which verifies FBS parameters p and l, is found (function IsValidSize) the linear combination coefficients $\boldsymbol{c}_v$ and cone C_v for node v are added to solution. Otherwise, the algorithm bootstraps predecessor node with largest linear combination image size (line 11) and resets its cone C_u to single node (line 12). The process is repeated, i.e. the second predecessor w is bootstrapped, in the event that no valid linear combination is identified (line 13). After the third FindParams call (line 17), both the predecessors of node v are bootstrapped, and the new linear combination is certainly valid.

Algorithm 1 Generic mapping algorithm

Input: Boolean circuit $G = (V, E)$ with 2-input gates
Input: FindParams$(f_{C_u}, \boldsymbol{c}_u, f_{C_w}, \boldsymbol{c}_w, f_v)$ - a function that returns a valid linear combination for cone $v \cup C_u \cup C_w$.
Input: p, l - FBS message space size and maximal norm of linear combination
Output: B - A set of gates to bootstrap
Output: C_v - A cone for $v \in B$
Output: $\boldsymbol{c}_v$ - A vector of coefficients for $v \in B$

```
1:  for all node v ∈ V in topological order do
2:      if v is input then
3:          C_v, c_v ← {}, [1]
4:      else if v is output then
5:          B ← B ∪ {v}
6:          C_v, c_v ← {}, [1]
7:      else
8:          u, w ← pred(v) such that imsize(c_u) ≥ imsize(c_w)
9:          c_v ← FindParams(f_{C_u}, c_u, f_{C_w}, c_w, f_v)
10:         if not IsValidSize(c_v) then
11:             B ← B ∪ {u}
12:             C_u, c_u ← {}, [1]
13:             c_v ← FindParams(f_{C_u}, c_u, f_{C_w}, c_w, f_v)
14:             if not IsValidSize(c_v) then
15:                 B ← B ∪ {w}
16:                 C_w, c_w ← {}, [1]
17:                 c_v ← FindParams(f_{C_u}, c_u, f_{C_w}, c_w, f_v)
18:             end if
19:         end if
20:         C_v ← {v} ∪ C_u ∪ C_w
21:     end if
22: end for
23: function IsValidSize(c)
24:     return imsize(c) ≤ p and ‖c_v‖_2 ≤ l
25: end function
```

We introduce two algorithms for the cone composition function. The objective of FindParams is to identify a linear combination that describes the logic function of the merged cone $\{v\} \cup C_u \cup C_w$. Let $f(\boldsymbol{x} \| \boldsymbol{y}) = f_v(f_{C_u}(\boldsymbol{x}), f_{C_w}(\boldsymbol{y}))$ denote the Boolean function of the merged cone. The composition algorithm returns a vector of coefficients, $\boldsymbol{c}$, such that $\phi_{\boldsymbol{c}}$ is a valid linear combination for the function f. Note that these functions are agnostic about the FBS parameters and can return a vector $\boldsymbol{c}$ whose image size or Euclidean norm is larger than p or l, respectively.

The first function is illustrated in Algorithm 2. A scaled version of coefficient vector $\boldsymbol{c}_u$ is concatenated with vector $\boldsymbol{c}_w$ (see line 3). The coefficient vector $\boldsymbol{c}_u$ is scaled by the image size $\text{imsize}(\boldsymbol{c}_w)$ of the second vector. The output linear combination function is:

$$\text{imsize}(\boldsymbol{c}_w) \cdot \phi_{\boldsymbol{c}_u}(\boldsymbol{x}) + \phi_{\boldsymbol{c}_w}(\boldsymbol{y})$$

for $\boldsymbol{x} \in \mathbb{Z}_2^{|\boldsymbol{c}_u|}$ and $\boldsymbol{y} \in \mathbb{Z}_2^{|\boldsymbol{c}_w|}$.

Algorithm 2 Naive cone composition

1: **function** FindParamsNaive($f_{C_u}, \boldsymbol{c}_u, f_{C_w}, \boldsymbol{c}_w, \cdot$)
2: $a, b \leftarrow \text{imsize}(\boldsymbol{c}_w), 1$
3: $\boldsymbol{c}_v \leftarrow \left[a \cdot \boldsymbol{c}_u \,\middle\|\, b \cdot \boldsymbol{c}_w\right]$
4: **return** $\boldsymbol{c}_v$
5: **end function**

The second cone composition function, outlined in Algorithm 3, also concatenates scaled versions of the vectors $\boldsymbol{c}_u$ and $\boldsymbol{c}_w$ (line 5). However, this function exhaustively searches for scaling coefficients a and b and returns the coefficient vector with the smallest image size. The output linear combination function is:

$$a \cdot \phi_{\boldsymbol{c}_u}(\boldsymbol{x}) + b \cdot \phi_{\boldsymbol{c}_w}(\boldsymbol{y})$$

for $\boldsymbol{x} \in \mathbb{Z}_2^{|\boldsymbol{c}_u|}$ and $\boldsymbol{y} \in \mathbb{Z}_2^{|\boldsymbol{c}_w|}$. The possible ranges for these coefficients are $|a| \leq \text{imsize}(\boldsymbol{c}_w)$ and $|b| \leq \text{imsize}(\boldsymbol{c}_u)$. Since the linear combination functions $\phi_{\boldsymbol{c}}$ and $\phi_{-\boldsymbol{c}}$ are equivalent, only positive values for the coefficient a are considered. This effectively reduces the search space by 2.

Heuristic termination, correctness and complexity. Algorithm termination is guaranteed by construction. Each circuit node is visited exactly once in topological order. At each node, the heuristic either merges the node into an existing cone (Algorithm 1, line 20) or triggers bootstrapping on one (line 11) or both of its inputs (line 15).

The output circuit remains functionally equivalent to the input. The function FindParams constructs a valid linear combination based on the linear combinations of a node's inputs. In particular, FindParamsNaive guarantees correctness by design, while FindParamsSearch verifies validity explicitly using the IsValid function.

If no valid linear combination can be found for the FBS parameter constraints, message space size p and maximum linear combination norm l, then one or both of the node's inputs are bootstrapped. Circuit output nodes are always bootstrapped to ensure outputs are Boolean.

The time complexity of the heuristic is $O(n)$ when using FindParamsNaive cone merge function, and $O(n \cdot p^2)$ when using FindParamsSearch, where n is the number of nodes in the circuit and p is the plaintext space size.

Limitations. The heuristic uses a greedy strategy for deciding when to bootstrap nodes and is sensitive to the order (topological) in which nodes are processed. As a result, it may produce suboptimal mappings, especially in circuits with multiple topological orders. One of our goals was to develop a lightweight, on-the-fly heuristic which can be used in a homomorphic execution runtime which does not require a complete circuit compilation phase.

The execution time of the heuristic is pseudo-polynomial (FindParamsSearch function), as it depends on the FBS plaintext space size p. However, our experiments show that little benefit is gained for values $p > 9$, as the increased cost of bootstrapping outweighs the marginal reduction in bootstrap count.

Algorithm 3 Search cone composition

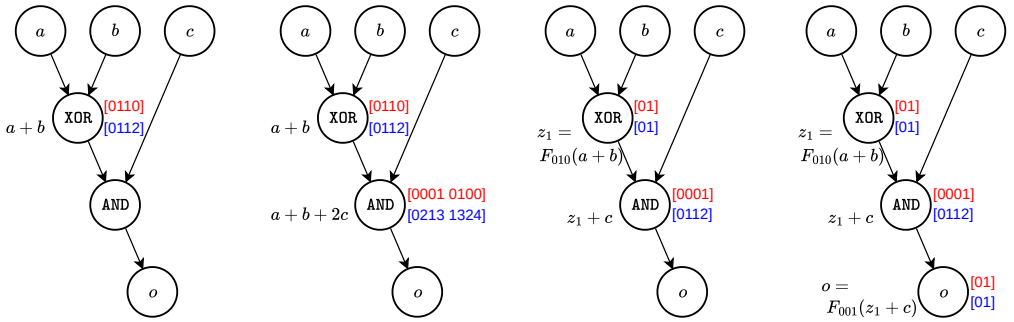
function FINDPARAMSSEARCH($f_{C_u}, \boldsymbol{c}_u, f_{C_w}, \boldsymbol{c}_w, f_v$)
  $\boldsymbol{c}_v^{\min} \leftarrow \varnothing$
  for all $a = 1, \ldots, \text{imsize}(\boldsymbol{c}_w)$ **do**
    for all $b = -\text{imsize}(\boldsymbol{c}_u), \ldots, -1, 1, \ldots, \text{imsize}(\boldsymbol{c}_u)$ **do**
      $\boldsymbol{c}_v \leftarrow \left[a \cdot \boldsymbol{c}_u \parallel b \cdot \boldsymbol{c}_w\right]$
      Let $f(\boldsymbol{x} \| \boldsymbol{y}) = f_v\left(f_{C_u}(\boldsymbol{x}), f_{C_w}(\boldsymbol{y})\right)$
        ▷ f is the logic function of cone $C_u \cup C_w \cup \{v\}$
      if ISVALID($f, \boldsymbol{c}_v$) **then**
        if $\text{imsize}(\boldsymbol{c}_v) \leq \text{imsize}\left(\boldsymbol{c}_v^{\min}\right)$ **then**
          if $\|\boldsymbol{c}_v\|_2 < \left\|\boldsymbol{c}_v^{\min}\right\|_2$ **then**
            $\boldsymbol{c}_v^{\min} \leftarrow \boldsymbol{c}_v$
          end if
        end if
      end if
    end for
  end for
  return $\boldsymbol{c}_v^{\min}$
end function
function ISVALID($f, \boldsymbol{c}$)
  $V_0 \leftarrow \left\{\phi_{\boldsymbol{c}}(\boldsymbol{x}) \mid \boldsymbol{x} \in \mathbb{Z}_2^{|\boldsymbol{c}|}, f(\boldsymbol{x}) = 0\right\}$
  $V_1 \leftarrow \left\{\phi_{\boldsymbol{c}}(\boldsymbol{x}) \mid \boldsymbol{x} \in \mathbb{Z}_2^{|\boldsymbol{c}|}, f(\boldsymbol{x}) = 1\right\}$
  return $V_0 \cap V_1 \equiv \emptyset$
end function

Cone composition example. Let us consider a node, v, with a logic function $f_v(x, y) = x$ `and` y. Additionally, we assume that the gate predecessor cones are empty ($C_x = C_y = \{\}$). The functions of the predecessors are identities, we have $f_{C_x}(x) = x$ and $f_{C_y}(y) = y$. The coefficients vectors are equal, $\boldsymbol{c}_x = \boldsymbol{c}_y = [1]$. The naive composition function returns the coefficients $\boldsymbol{c}^{naive} = [2, 1]$, where 2 is the image size of $\boldsymbol{c}_x$, and the search composition function returns $\boldsymbol{c}^{search} = [1, 1]$. The image size of $\boldsymbol{c}^{naive}$ is 4, whereas the image size of $\boldsymbol{c}^{search}$ is only 3. The truth-table and linear combination outputs for function f_v are given following table:

$\boldsymbol{x}$	$f_v(\boldsymbol{x})$	$\phi_{\boldsymbol{c}^{naive}}(\boldsymbol{x})$	$\phi_{\boldsymbol{c}^{search}}(\boldsymbol{x})$
$[0, 0]$	0	0	0
$[0, 1]$	0	1	1
$[1, 0]$	0	2	1
$[1, 1]$	1	3	2

Observe that the functions $\phi_{\boldsymbol{c}^{naive}}$ and $\phi_{\boldsymbol{c}^{search}}$ are valid linear combinations for the node v logic function as the corresponding linear combination values for $f_v(x) \equiv 0$ and $f_v(x) \equiv 1$ are different.

Execution example. Consider the Boolean circuit with three inputs and two gates, as illustrated in Figure 2, and suppose we want to map it to an FBS with $\mathbb{Z}_3$. The first node visited is the `XOR` gate. A valid linear combination for this node is $a + b$, which has image [0..2] (see Figure 2a). The next visited node is the `AND` gate. A linear combination for this node is $a + b + 2 \cdot c$, but its image, [0..4], overflows the $\mathbb{Z}_3$ space (see Figure 2b). The `AND` gate's input (the `XOR` gate) is bootstrapped, and a valid linear combination, $z_1 + c$,



(a) `XOR` (b) `AND` – linear combination overflows (c) `AND` – bootstrap `XOR` (d) Output – bootstrap `AND`

Figure 2: An example of circuit traversal for an FBS over $\mathbb{Z}_3$. The vectors in red (top) represent the node's cone truth tables, and the vectors in blue (bottom) represent the evaluations of the node's cone linear combination. $F_{010}(a+b)$ denotes the FBS which maps consecutive values of $a+b$ image to Boolean values $0, 1, 0$, i.e. $F_{010}(0) = 0$, $F_{010}(1) = 1$ and $F_{010}(2) = 0$.

is found for this node (see Figure 2c). Finally, the output node o is mapped to Boolean space using another FBS (see Figure 2d).

Implementation details. The Algorithm 1 listing has been simplified by omitting several implementation details. Gates with a single-input $f(v)$, same-input gates $f(v, v)$, and constant $f(\boldsymbol{v}) = cst$ gates are ignored because the same linear combination $\boldsymbol{c}_v$ of gate input is valid for gate output also. Furthermore, linear combinations with image sizes larger than 2^{16} are pruned. In the search composition function, Algorithm 3, common nodes in the supports of C_u and C_v are only considered once. In this way obtained linear combination coefficients are smaller.

4 Implementation and performance

A proof of concept of the proposed mapping heuristic has been implemented in python and is publicly available[1]. The algorithm is parameterised by the two cone composition methods that were previously presented. The mapping heuristics are referred to as either as the *naive heuristic* or as the *search heuristic*. A minor discrepancy between Algorithm 1 and the implemented version is that the latter does not consider the Euclidean norm l when evaluating the validity of a linear combination (function IsValidSize). This was done because the Euclidean norm l exerts a lesser influence on FBS parameters when compared to the number of plaintext divisions p. For instance, according to `concrete`[2] estimator, increasing the norm l from 4 to 24 for $p = 16$ increases bootstrapping cost by less than 2%.

In order to assess the efficacy of the proposed heuristic, we have employed Boolean circuits from a range of benchmark suites, as well as manually generated circuits. Both heuristics have been executed on each benchmark circuit for different values of the FBS parameter p. For each execution, the elapsed time has been recorded, along with the characteristics of the mapped circuit, including the number of bootstrappings, linear combinations, the maximal Euclidean norm of the linear combinations, and so forth.

[1] https://github.com/ssmiler/tfhe_fbs_map
[2] https://github.com/zama-ai/concrete

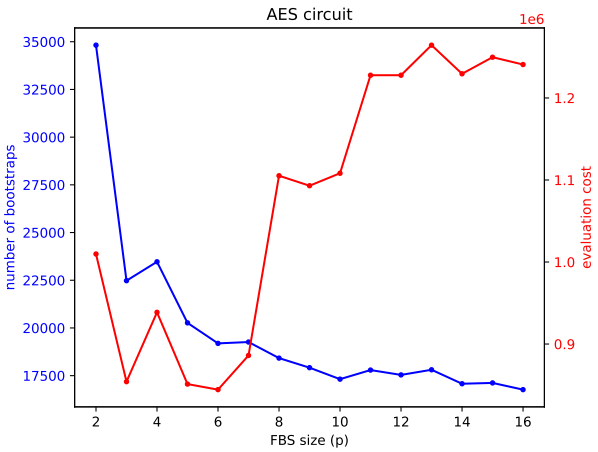


Figure 3: Search heuristic applied to AES circuit. Mapped circuit number of bootstrappings and estimated evaluation cost as a function of FBS size.

The experimental results demonstrate that the number of output bootstrappings does not exhibit a monotonically decreasing trend with an increase in the value of p. To illustrate, the blue line in Figure 3 depicts the output bootstrap count as a function of the FBS size, p, for the search heuristic applied to the AES 128 circuit. A similar phenomenon is observed for the naive heuristic. We think this phenomenon occurs due to the greedy nature of the heuristics, which aim to maximise the image size of linear combinations and consequently FBS are added too late by the heuristic. In the conducted tests, a maximal value P was assigned for the FBS size. Subsequently, the heuristic was executed for each value of $2 \leq p \leq P$. The mapped circuit with the lowest metric (either evaluation cost or bootstrapping count) value is kept as output.

The evaluation cost of a circuit is estimated as the number of circuit bootstrappings multiplied by the cost of a single bootstrapping. The `concrete` compiler is used to estimate the execution cost of a single multi-input FBS with given parameters. In order to facilitate the utilisation of non power-of-two values for the FBS size and Euclidean norm, the compiler code has been patched. As an example, the red line in Figure 3 illustrates the evaluation cost of the AES circuit as a function of FBS size. It is important to note that while the number of bootstrappings may continue to decrease with FBS size, the evaluation cost of the circuit consistently increases for $p > 6$. Furthermore, starting from $p = 8$, the evaluation cost will exceed that of the non-optimised circuit (i.e. the mapped circuit for $p = 2$). This is due to the fact that the reduction in bootstrapping count is no longer sufficient to compensate for the increase of a FBS cost.

4.1 EPFL combinational benchmark suite

The EPFL combinational benchmark suite [AGDM15] comprises 10 arithmetic, 10 random/control circuits and 3 multi-million gate designs. In our experiments, we have used the first two types of benchmarks, a total of 20 Boolean circuits. The naive and the search heuristics are used to map each benchmark circuit to FBS. The mapping heuristic is executed with FBS sizes varying from 2 to 15. The mapped circuit with the smallest cost and smallest the FBS size in case of a tie is kept as the output. Furthermore, we estimate the cost of executing a *reference mapping* where each Boolean circuit gate is executed as a `TFHE` gate bootstrapping (i.e. FBS with size 2).

Refer to Table 1 for a comparison of the evaluation speedups of the circuits mapped with the proposed heuristics compared to the reference mapping (column “cost”). The

Table 1: EPFL benchmark. Evaluation cost improvements for FBS mapped circuits (column "cost"), decrease in bootstrappings count (column "#boots.") and corresponding FBS sizes. Cells where no gain has been obtained are not included in the presentation.

bench	naive			search		
	cost	#boots.	FBS size	cost	#boots.	FBS size
adder				−64%	−75%	5 (7)
bar				−24%	−50%	7 (10)
div				−30%	−52%	5 (10)
hyp				−39%	−62%	7 (14)
log2				−37%	−56%	5 (10)
max	−6%	−38%	7 (8)	−34%	−68%	9 (15)
multiplier				−49%	−68%	7 (14)
sin				−36%	−58%	6 (11)
sqrt				−54%	−70%	6 (11)
square				−32%	−53%	5 (10)
arbiter	−20%	−45%	5 (8)	−48%	−64%	5 (8)
cavlc				−36%	−59%	7 (13)
ctrl	−3%	−37%	7 (8)	−40%	−61%	7 (12)
dec						
i2c	−1%	−35%	7 (8)	−34%	−57%	7 (14)
int2float	−8%	−40%	7 (8)	−49%	−67%	7 (13)
mem_ctrl	−4%	−38%	7 (8)	−31%	−55%	7 (13)
priority				−40%	−60%	6 (11)
router	−9%	−40%	7 (8)	−42%	−63%	7 (14)
voter				−38%	−57%	5 (10)
avg.	−3%	−15%		−38%	−58%	

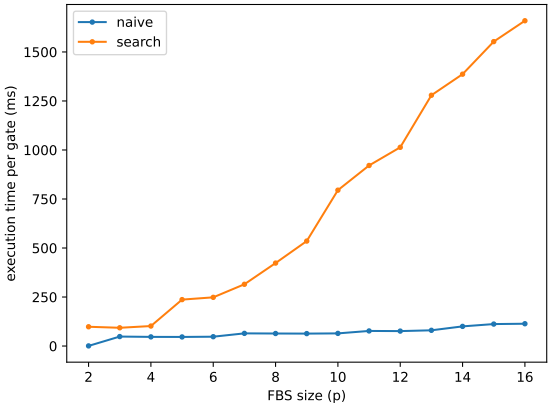


Figure 4: The average of the mapping heuristics execution time divided by the number of circuit nodes.

two column groups (denoted as "naive" and "search") represent the two cone composition functions. Furthermore, two additional columns are given for the mapping with the lowest execution cost:

- "#boots." – difference in number of bootstrappings.
- "FBS size" – FBS size and linear combination image size (in brackets) for which this mapping was obtained. We remind that the linear combination image size can be larger than the FBS size in the case of negacyclic functions.

The last table row represents the average of the respective columns.

The mapping heuristic with search cone composition consistently gives better execution cost and a lower bootstrapping number when compared to the naive cone composition. On average, the search heuristic results in 38% reduction in execution cost and a 58% reduction in the number of bootstrappings. The functional bootstrapping size, which yields the lowest execution cost, is almost always smaller or equal to 9. This aligns with the earlier observation made for the AES circuit.

Figure 4 illustrates heuristics average execution time divided by the input circuit gate count, for each FBS size. As anticipated, the search heuristic is slower than the naive one and scales non-linearly with FBS size due to the exhaustive search in Algorithm 3.

4.2 Trivium and Kreyvium stream ciphers

The authors of [BOS23] introduce hand-optimised implementations for Trivium/Kreyvium stream ciphers [DC06, CCF+18] using `TFHE` FBS. Several approaches to implementing a single iteration of stream ciphers are presented, beginning with gate bootstrapping and concluding with functional bootstrapping. The most efficient solution uses a 2-bit message space (or 3-bit in case of negacyclic functions).

The heuristics we introduce process circuit gates in the order in which they appear in the input circuit file. This is just one of the many possible topological orders and it has an impact on the quality of the mapped circuit. Two versions of each stream cipher have been implemented, resulting in a total of four circuits. The code used to generate these circuits is given in Listing 1. The difference between the two versions is in how the last 3 instructions are expressed. The second version groups together the XOR gates when variables `out_t1`, `out_t2` and `out_t3` are computed.

```
// version 1
t1 = s66 ^ s93
t2 = s162 ^ s177
t3 = s243 ^ s288 ^ k127

out = t1 ^ t2 ^ t3

out_t1 = t1 ^ (s91 & s92) ^ s171 ^ iv127
out_t2 = t2 ^ (s175 & s176) ^ s264
out_t3 = t3 ^ (s286 & s287) ^ s69

// version 2
t1 = s66 ^ s93
t2 = s162 ^ s177
t3 = s243 ^ s288 ^ k127

out = t1 ^ t2 ^ t3

out_t1 = (t1 ^ s171 ^ iv127) ^ (s91 & s92)
out_t2 = (t2 ^ s264) ^ (s175 & s176)
out_t3 = (t3 ^ s69) ^ (s286 & s287)
```

Listing 1: The two versions of an iteration of Trivium, underlined are Kreyvium differences. The circuits have 15 inputs, respectively 17 for Kreyvium, and 4 outputs (`out`, `out_t1`, `out_t2` and `out_t3`).

The search heuristic has been applied to the four circuits with FBS sizes varying from 2 to 12. Figure 5 and Figure 6 plot the bootstrapping count and the estimated evaluation cost for the mapped circuits. The second version demonstrates a faster convergence rate and a lower number of required bootstrappings, with the exception of $p = 7$. The minimal number of bootstrappings for these circuits is four, which is the number of outputs. This is reached at $p = 6$ for Trivium and at $p = 9$ for Kreyvium.

Our heuristic maps the Trivium circuit to the same number of bootstrappings (8) and the Kreyvium circuit to 20% less bootstrappings (8 instead of 10) using the same message space $\mathbb{Z}_4$ as in [BOS23]. Furthermore, a solution with the same number of bootstrappings is found for $p = 3$. The evaluation cost is reduced in this case due to the smaller `TFHE` FBS parameters.

The mapped circuit with the lowest evaluation cost is obtained with $p = 6$ for Trivium (4 bootstrappings) and for Kreyvium (5 bootstrappings). The evaluation cost is 45% less than that of the solutions presented in [BOS23]. One significant advantage of the heuristic proposed in this paper over [BOS23] is that circuits are automatically mapped.

Listing 2 illustrates version 2 of ciphers from Listing 1 which have been mapped to FBS with size $p = 3$. The mapped circuit has 8 bootstrappings and is not dependent on the implemented stream cipher. The first and the last 4 bootstrappings are independent of each other. In the scenario of parallel execution, the mapped circuit has a latency equivalent of 2 bootstrappings. When compared to [BOS23], the Kreyvium latency is reduced by 50% (from 3 to 2) by our heuristic.

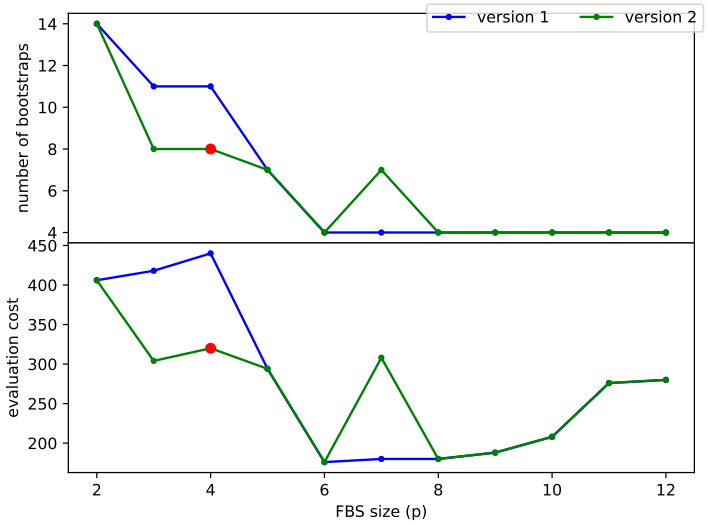


Figure 5: Trivium stream cipher. Bootstrapping count and evaluation cost for the 2 circuit versions. The result from [BOS23] is shown with red dots.

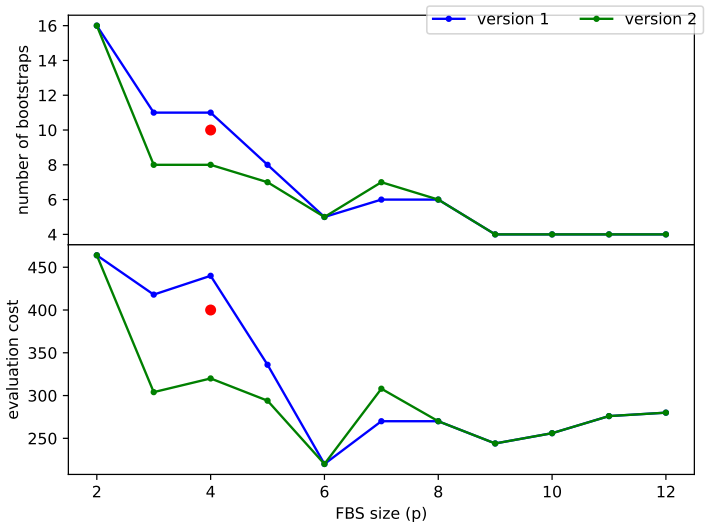


Figure 6: Kreyvium stream cipher. Bootstrapping count and evaluation cost for the 2 circuit versions. The result from [BOS23] is shown with red dots.

```
m1 = 2 - s66 + s93 - s162 + s177
m2 = Bootstrap(m1, [0, 1, 0, 1, 0])
m3 = 1 - s66 + s93 + s171 + iv127
m4 = Bootstrap(m3, [1, 0, 1, 0, 1])
m5 = 1 - s162 + s177 + s264
m6 = Bootstrap(m5, [1, 0, 1, 0])
m7 = 1 - s243 + s288 + k127 + s69
m8 = Bootstrap(m7, [1, 0, 1, 0, 1])
m9 = 1 + m2 - s243 + s288 + k127
out = Bootstrap(m9, [1, 0, 1, 0, 1])
m10 = 3 * m4 + s91 + s92
out_t1 = Bootstrap(m10, [0, 0, 1, 1, 1, 0])
m11 = 3 * m6 + s175 + s176
out_t2 = Bootstrap(m11, [0, 0, 1, 1, 1, 0])
m12 = 3 * m8 + s286 + s287
out_t3 = Bootstrap(m12, [0, 0, 1, 1, 1, 0])
```

Listing 2: Version 2 of Listing 1 which was mapped to FBS with size $p = 3$.

4.3 Comparison to other cryptographic primitives

In the paper [BPR24], the authors introduce a novel $p-$encoding of Booleans in `TFHE` plaintext space $\mathbb{Z}_p$. In contrast to the conventional approach of encoding Booleans as two distinct values, namely 0 and 1, the authors propose to encode them as distinct sets of values from $\mathbb{Z}_p$. A $p-$encoding is defined as a pair $\{\mathcal{E}_0, \mathcal{E}_1\}$, where $\mathcal{E}_0, \mathcal{E}_1 \subset 2^{\mathbb{Z}_p}$ and $\mathcal{E}_0 \cap \mathcal{E}_1 = \emptyset$. The inputs of a Boolean gate to be evaluated are encoded as singleton sets. The $p-$encoding for gate inputs are chosen such that their sum is a valid $p-$encoding representing gate functionality. Let $\{\{k_0\}, \{k_1\}\}$ be a $p-$encoding of an input. This encoding is as an affine transformation of the Boolean encoding we use. The $p-$encoding of x is $\mathcal{E}_x = \{(k_1 - k_0) \cdot x + k_0\}$ for $x \in \{0, 1\}$. So in this context, the sum of the p-encodings is equivalent to a linear combination of the two-element Boolean encoding used in our work.

There is a major difference in the manner the authors of [BPR24] encode plaintext messages. The authors chose to split the plaintext message space into $2 \cdot p$ segments for odd p-s and to use half of the message space values, i.e. values $2 \cdot k$ for $0 \leq k < p$. This trick allows to obtain a $\mathbb{Z}_p$ message space and to completely ignore the negacyclic property of `TFHE`. In our case, we use a larger but more constrained message space, namely $\mathbb{Z}_{2p}$, however the evaluated functions must be negacyclic.

We have implemented the cryptographic primitives described in [BPR24] and mapped them to FBS using the proposed search heuristic. As before, we have executed the heuristic with different FBS sizes and kept the best solution in terms of estimated execution time. Our heuristic found the same or better solutions for all the circuits. Table 2 gives the circuits for which a better solution is found. The search heuristic proposed in this work

Table 2: Comparison of search heuristic results with [BPR24].

bench	[BPR24]		search		speedup
	#boots.	FBS size	#boots.	FBS size	
SIMON	1	9	1	6 (9)	1.07×
ASCON	5	17	7	6 (9)	1.22×
AES s-box	36	11	39	6 (12)	1.45×

Interestingly enough the search heuristic found an equivalent solution for SIMON function. Due to different plaintext encoding and to the use of the negacyclic property our solution uses a smaller plaintext space $\mathbb{Z}_6$ instead of $\mathbb{Z}_9$. Listing 3 gives the solution the search heuristic found. Observe that linear combination coefficients have the same values as inputs encoding from [BPR24]: $\mathcal{E}_0 = \mathcal{E}_1 = \{0, 1\}$ and $\mathcal{E}_2 = \mathcal{E}_3 = \mathcal{E}_4 = \{0, 2\}$. And respectively, the output encoding $\mathcal{E}_{out} = \{\{0, 1, 4, 5, 8\}, \{2, 3, 6, 7\}\}$ is also the same as the test vector Boolean values positions from Listing 3. Note that in our case because the test vector is negacylic a FBS of size 6 is sufficient even if test vector length is 9.

```
m1 = 1 * b0 + 1 * b1 + 2 * b2 + 2 * b3 + 2 * b4
out = Bootstrap(m1, [0, 0, 1, 1, 0, 0, 1, 1, 0])
```

Listing 3: SIMON function which was mapped to FBS with size $\mathbb{Z}_6$.

In case of ASCON and the AES s-box the heuristic proposed in this paper obtains solutions with smaller FBS sizes and by consequence faster. For example, for AES s-box a 45% speedup in the evaluation time is obtained.

4.4 Comparison to AutoHoG

In paper [GMZ+24] the authors proposed a circuit mapping procedure, designated AutoHoG, for `TFHE` functional bootstrapping. AutoHoG takes a Boolean circuit as input and generates a circuit with "compound" gates, which is similar to our FBS mapping. In addition to single-output gates, the AutoHoG authors employ the multi-output evaluation method from [CIM19] to factor out bootstrappings with the same inputs.

In their experiments, the authors use the ISCAS'85 circuit benchmark [BF85] and the ISCAS'89 sequential circuit benchmark [BBK89] in their experiments. We use the same techniques to transform ISCAS'89 sequential circuits with flip-flops into combinational circuits. The sequential circuits are unrolled for 10 clock cycles using the ABC logic synthesis tool [Ber] (command `frames -F 10 -i`). Both ISCAS'85 and ISCAS'89 (after unrolling) are mapped to 2-input gates using a complete gate library that has been generated manually.

In AutoHoG, a single parameterisation of `TFHE` is employed, enabling the evaluation of a multi-output FBS with $p = 32$. The authors compare the execution times of mapped circuits with those of input circuits using the same `TFHE` parameters. However, the presented speedup results may be overly optimistic, given that the input circuit can be executed with significantly smaller `TFHE` parameters.

To ensure a fair comparison with AutoHoG, we execute our mapping heuristic with FBS size varying from 2 to 32 and keep the circuit with smallest number of bootstrappings as output. The speedup is approximated as the ratio between the number of input circuit gates and the number of FBSs in the mapped circuit. This is equivalent to the method used by AutoHoG to compute speedup. In this approximation, the overhead due to linear combinations in multi-input FBS evaluation is ignored. The linear combination has a much smaller impact on execution time than the bootstrapping part.

Table 3 and Table 4 depict the speedups of the search heuristic (column "search") and the speedup from AutoHoG paper. As previously, we provide the FBS size (column "FBS size") for which the circuit with the fewest number of bootstrappings is obtained. Observe that the FBS size is not always equal to the maximum value 32. This indicates that fixing the FBS size in advance is not always advantageous. The speedup of AutoHoG results has been computed by dividing the available numeric values in [GMZ+24] (Fig. 7 and TABLE IV). AutoHoG demonstrates enhanced speedups across the majority of benchmarks in case of ISCAS'85 and for ISCAS'89 benchmarks our heuristic results get closer to AutoHoG

Table 3: ISCAS'85 speedup and comparison with AutoHoG. The best benchmark-wise speedup is presented in bold, with the exception of benchmarks for which an AutoHoG speedup is not available.

bench	search		AutoHoG
	speedup	FBS size	speedup
c17	**3.00**×	8 (13)	2.50×
c432	**3.05**×	28 (41)	2.16×
c499	3.86×	14 (27)	–
c880	2.85×	15 (26)	–
c1355	3.86×	14 (27)	**6.03**×
c1908	2.65×	32 (45)	–
c2670	4.41×	29 (48)	–
c3540	2.54×	31 (53)	**3.90**×
c5315	3.31×	25 (46)	–
c6288	2.96×	26 (37)	–
c7552	2.58×	23 (46)	**5.68**×
avg.	3.19×		**4.05**×

ones. This outcome was expected, given that our approach does not use the multi-output FBS technique.

Conclusion and Perspectives

The heuristic presented in this paper has several limitations that require further attention and need to be tackled in future works. The first issue is that the heuristic performance depends on the order of visit of circuit nodes. As an example, two implementations of Trivium/Kreyvium have been used (each resulting in different visit orders) to ensure the best solution is found. Another shortcoming is to not use the multi-output FBS techniques from [CIM19], which could potentially result in mapped circuits with a smaller number of FBS. It would be beneficial to consider the reduction modulo p property of the input plaintext space $\mathbb{Z}_p$. The authors of [BPR24] use a plaintext space $\mathbb{Z}_2$ to evaluate two-input XOR gates for free, in contrast to a plaintext space $\mathbb{Z}_3$ and one FBS in our case.

References

[AGDM15] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. The epfl combinational benchmark suite. In *Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)*, 2015.

[BBK89] Franc Brglez, David Bryan, and Krzysztof Kozminski. Combinational profiles of sequential benchmark circuits. In *1989 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1929–1934. IEEE, 1989.

[Ber] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification, commit 2d70deb. http://www.eecs.berkeley.edu/~alanmi/abc/.

[BF85] Franc Brglez and Hideo Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator. In *Fortran. ISCAS'85*, 1985.

Table 4: ISCAS'89 speedup and comparison with AutoHoG. The best benchmark-wise speedup is presented in bold, with the exception of benchmarks for which an AutoHoG speedup is not available.

bench	search		AutoHoG
	speedup	FBS size	speedup
s27	**4.11**×	27 (40)	1.27×
s208	3.74×	29 (50)	–
s298	3.14×	31 (50)	**3.43**×
s344	**3.26**×	27 (52)	3.05×
s349	**3.24**×	32 (54)	2.79×
s382	3.23×	25 (48)	**4.46**×
s386	3.06×	32 (64)	**5.85**×
s400	3.34×	30 (55)	**4.73**×
s420	**5.31**×	29 (53)	2.94×
s444	3.54×	32 (57)	**4.73**×
s510	2.83×	28 (56)	**3.43**×
s526	3.04×	28 (52)	**4.19**×
s641	**3.04**×	21 (34)	2.14×
s713	**3.23**×	27 (45)	2.45×
s820	4.59×	26 (39)	**4.75**×
s832	**4.93**×	27 (39)	4.79×
s838	**5.59**×	32 (56)	3.01×
s953	2.76×	32 (61)	**3.51**×
s1196	3.67×	32 (50)	**4.15**×
s1238	**3.75**×	32 (50)	3.66×
s1423	**3.16**×	29 (55)	3.01×
s1488	3.66×	31 (59)	**7.45**×
s1494	3.61×	30 (51)	–
s5378	3.73×	29 (57)	**7.35**×
s9234	3.12×	21 (40)	**3.60**×
s13207	**3.54**×	29 (52)	2.33×
s15850	**3.24**×	31 (61)	2.22×
s35932	**5.71**×	28 (46)	3.19×
s38417	3.82×	26 (52)	–
s38584	**3.41**×	30 (60)	2.51×
avg.	3.68×		**3.74**×

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[BOS23] Thibault Balenbois, Jean-Baptiste Orfila, and Nigel Smart. Trivial transciphering with trivium and tfhe. In *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 69–78, 2023.

[BPR24] Nicolas Bon, David Pointcheval, and Matthieu Rivain. Optimized homomorphic evaluation of boolean functions. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(3):302–341, 2024.

[BR15] Jean-François Biasse and Luis Ruiz. Fhew with efficient multibit bootstrapping. In *Progress in Cryptology – LATINCRYPT 2015: Proceedings 4*, pages 119–135. Springer, 2015.

[CBS24] Pierre-Emmanuel Clet, Aymen Boudguiga, and Renaud Sirdey. Chocobo: Creating homomorphic circuit operating with functional bootstrapping in basis b. *Cryptology ePrint Archive*, 2024.

[CCF+18] Anne Canteaut, Sergiu Carpov, Caroline Fontaine, Tancrède Lepoint, María Naya-Plasencia, Pascal Paillier, and Renaud Sirdey. Stream ciphers: A practical solution for efficient homomorphic-ciphertext compression. *Journal of Cryptology*, 31(3):885–916, 2018.

[CGGI16a] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachene. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Advances in Cryptology – ASIACRYPT 2016*, pages 3–33. Springer, 2016.

[CGGI16b] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption library, August 2016. https://tfhe.github.io/tfhe/.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.

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

[DC06] Christophe De Canniere. Trivium: A stream cipher construction inspired by block cipher design principles. In *International Conference on Information Security*, pages 171–186. Springer, 2006.

[DM15] Léo Ducas and Daniele Micciancio. FHEW: bootstrapping homomorphic encryption in less than a second. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 617–640. Springer, 2015.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive*, 2012.

[GBA21] Antonio Guimarães, Edson Borin, and Diego F Aranha. Revisiting the functional bootstrap in TFHE. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 229–253, 2021.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[GMT25] Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. Helm: Navigating homomorphic encryption through gates and lookup tables. *IEEE Transactions on Information Forensics and Security*, 2025.

[GMZ+24] Zhenyu Guan, Ran Mao, Qianyun Zhang, Zhou Zhang, Zian Zhao, and Song Bian. AutoHoG: Automating Homomorphic Gate Design for Large-Scale Logic Circuit Evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology – CRYPTO 2013*, pages 75–92. Springer, 2013.

[MHSB21] Kotaro Matsuoka, Yusuke Hoshizuki, Takashi Sato, and Song Bian. Towards better standard cell library: Optimizing compound logic gates for tfhe. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 63–68, 2021.

[MKG23] Johannes Mono, Kamil Kluczniak, and Tim Güneysu. Improved Circuit Synthesis with Amortized Bootstrapping for FHEW-like Schemes. *Cryptology ePrint Archive*, 2023.

[SRH+22] Mathias Soeken, Heinz Riener, Winston Haaswijk, Eleonora Testa, Bruno Schmitt, Giulia Meuli, Fereshte Mozafari, Siang-Yun Lee, Alessandro Tempia Calvino, and Giovanni Marakkalage, Dewmini Sudara De Micheli. The EPFL logic synthesis libraries, June 2022. arXiv:1805.05121v3.

[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 *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 79–90, 2023.