USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Efficient Batchable Secure Outsourced Computation: Depth-Aware Arithmetization of Common Primitives for BFV & BGV

Jelle Vos, *Delft University of Technology;* Mauro Conti, *University of Padua & Delft University of Technology;* Zekeriya Erkin, *Delft University of Technology*

https://www.usenix.org/conference/usenixsecurity25/presentation/vos

This paper is included in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the 34th USENIX Security Symposium is sponsored by USENIX.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

Efficient Batchable Secure Outsourced Computation: Depth-Aware Arithmetization of Common Primitives for BFV & BGV

Jelle Vos
Delft University of Technology

Mauro Conti
University of Padua & Delft University of Technology

Zekeriya Erkin
Delft University of Technology

Abstract

Homomorphic encryption enables secure outsourced computation, in which computations on sensitive data can be confidentially outsourced to another party. Homomorphic encryption cryptographically guarantees confidentiality while allowing an evaluator to manipulate the encrypted data using additions and multiplications. However, a remaining challenge is to translate complex computations into efficient circuits consisting of only additions and multiplications. We refer to this problem as arithmetization. The objective in arithmetization has typically been to minimize the number of multiplications (multiplicative size), as multiplications in most secure computation techniques are significantly more expensive than additions. However, the multiplicative depth of a circuit arguably plays an even more important role in deciding the computational cost: For homomorphic encryption schemes like BFV and BGV, it determines the choice of cryptographic parameters that allow evaluating the circuit without requiring expensive bootstrapping operations. We argue that arithmetization should be treated as a multi-objective minimization problem, in which a trade-off can be made between a circuit's multiplicative size and depth. We present efficient depth-aware arithmetization methods for many primitive operations such as exponentiation, univariate functions, equality checks, comparisons, and ANDs and ORs, which further take into account that squaring can be cheaper than multiplying, and we study how to compose these operations. We show that our circuits can outperform more recent homomorphic encryption schemes like TFHE, which can perform significantly faster homomorphic operations but only on one input at a time by batching several inputs into one ciphertext.

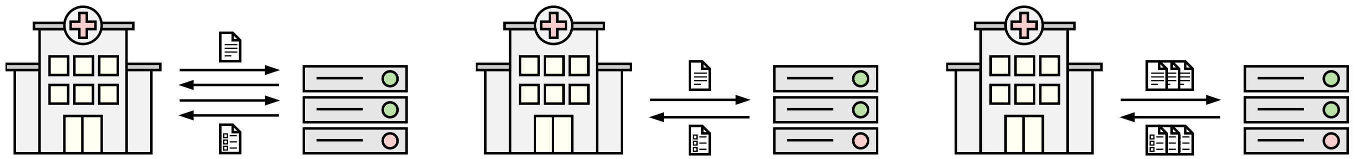
1 Introduction

Since the advent of cloud computing, outsourced computation has become a ubiquitous tool for organizations to fulfill their computational tasks without operating and maintaining a large computational infrastructure. However, classical outsourced computation does not provide any cryptographic guarantees to the confidentiality of data sent to the cloud. As a result, the sensitive nature of some of these data hinder outsourcing computations on them. Secure outsourced computation offers the benefits of outsourced computation while providing exactly those cryptographic guarantees.

Homomorphic encryption schemes like BFV [8, 17] and BGV [9] allow us to encrypt elements in $\mathbb{F}_{p^d}$ and perform non-interactive secure outsourced computation because they allow an evaluator to manipulate the encrypted data using additions and multiplications. However, a remaining challenge is to translate complex computations into efficient circuits comprised of only additions and multiplications. We refer to this sub-problem as *arithmetization*. In this paper, we show how to do so for BFV and BGV, by introducing a concept called depth-aware arithmetization. As shown in Figure 1, these techniques can be used to speed up secure outsourced computation of medical data.

One approach to arithmetization is to consider only Boolean circuits, so $p = 2$. This approach allows the use of existing Boolean circuit synthesis techniques, but it may result in circuits with a large number of multiplications. For example, an equality check between two 256-bit inputs requires just 8 squarings in $\mathbb{F}_{257}$ (see Sec. 5), but it takes 255 multiplications in $\mathbb{F}_2$.

Other works do consider arithmetization in circuits with $p > 3$, but these works either focus on minimizing the number of multiplications, known as the multiplicative size [14], or the multiplicative depth [18], which is the largest number of multiplications in any path through the circuit. Minimizing the multiplicative size stands to reason because multiplications are significantly more expensive to compute than additions in all of the techniques mentioned above. However, the multiplicative depth cannot be completely ignored, because it determines the size of the cryptographic parameters of the evaluated circuit: BFV and BGV ciphertexts contain noise that grows linearly during homomorphic additions and exponentially during multiplications, and for successful decryption, this noise must stay under some bound. Parameters



(a) Two-party computation techniques do not outsource the computation and they require multiple interactions.

(b) TFHE allows one medical file to be processed in 1 second after sharing a bootstrapping key.

(c) Our work using BGV allows 128 medical files to be processed in 44 seconds after sharing a key-switching key.

Figure 1: Secure outsourced computation of medical analyses. Our work allows batching files so they are processed more quickly.

for these schemes are therefore chosen to be large enough so that the noise has enough room to grow, remaining under the noise limit with high probability. Large parameters negatively impact the efficiency of each homomorphic multiplication.

Inspired by the depth-size trade-off in permutation circuits proposed by Halevi & Shoup [20], we propose a new type of arithmetization called depth-aware arithmetization, which considers *both* a circuit's multiplicative size and multiplicative depth in the arithmetization of every high-level operation. In doing so, depth-aware arithmetization allows one to reduce the size of the parameters needed in BFV and BGV, resulting in a lower computational cost. Specifically, we study the arithmetization of deterministic high-level functions while minimizing both the multiplicative size and depth of the generated circuit. We restrict these circuits to be deterministic (so constants are truly constant) and do not allow intermediate revealing of values. We also restrict the algebraic structure to a prime field $\mathbb{F}_p$, in which any function can be expressed as an arithmetic circuit.

As a second consideration, we take into account that squaring is typically a more efficient operation than performing arbitrary multiplications since some of the intermediate terms can be reused. We do so by defining a metric called the multiplicative cost, which is the number of multiplications between distinct non-constant inputs plus the total squaring cost, which is the number of squarings multiplied by $0.5 \leq \sigma \leq 1.0$.

Our work is not the first to reduce the depth of arithmetic circuits. Some works [4,10,25,35] take in arbitrary arithmetic circuits and reduce their depth while increasing their size. We do not consider these generic depth reduction methods in this work for two reasons. Firstly, these methods ignore the function that is being computed, but since we have this knowledge, we exploit it. Secondly, these methods reduce the depth by distributing products of sums, while increasing the multiplicative size. However, opportunities for distributing products of sums do not arise in the circuits generated in this paper (this is more common in Boolean circuits).

The rationale behind our work is to propose algorithms for generating efficient circuits for several common primitives. These primitives can be composed into more complex circuits. In each section, we first discuss how to obtain anchor points: the points that minimize the multiplicative cost (with the multiplicative depth as a secondary objective) or the multiplicative depth (with the multiplicative cost as a secondary objective). After that, we discuss how to obtain the other solutions in the depth-cost front. At the end of each section, we perform a case study, where we use the primitive for a common practical use case.

BFV and BGV offer performance gains because they allow packing multiple inputs into a single ciphertext, enabling computations on a single thread to implicitly parallelize across all inputs. An alternative that does not allow parallelization but allows for faster individual computations comes from schemes like TFHE [12]. However, BFV and BGV offer performance gains when amortized. For example, Iliashenko & Zucca [24] already showed that comparison circuits for BFV/BGV can outperform CKKS and TFHE when amortized. We show that the same holds for even more complex circuits, comparing to TFHE with optimized parameters. One might think that another alternative is the use of two-party computation, but this requires both parties to perform a significant amount of computations, so it does not successfully outsource the computation. We summarize this in Figure 1.

We note that our techniques may also be of use in other domains. For example, in some arithmetic garbling schemes, the multiplicative depth also plays an important role in the efficiency of a circuit [2]. Arithmetization is also a fundamental problems in these protocols.

Concretely, our contributions are as follows. We propose:

- A linear-time algorithm for optimal depth-aware arithmetization of products of independent inputs.
- Efficient generation of Pareto-optimal exponentiation circuits via multiple MaxSAT calls.
- A new univariate polynomial evaluation technique that achieves lower-depth circuits.
- An algorithm for hybrid circuits for ANDs/ORs trading off depth and cost, outperforming non-hybrid circuits.
- Depth-aware arithmetization for high-level circuits composed of multiple primitives.

We also propose two incremental contributions:

- A translation from MILP to a more efficiently solvable MaxSAT formulation for depth-constrained minimal-cost exponentiation circuits.
- The observation that varying k, the amount of precomputations, in existing polynomial evaluation techniques, enables a depth-cost tradeoff.

Together, the primitives we discuss in this paper are sufficient to describe any circuit (using equality checks and lookup tables), and they can be used to implement high-level number representations. Other operations like if-statements are easy to arithmetize. We do not consider bivariate polynomial evaluation because univariate evaluation is sufficient for many operations, including comparisons and constant-division.

The structure of our paper is as follows. In Section 2, we describe our notation. In Section 3, we briefly review related work. In Section 4, we discuss the depth-aware arithmetization of sums and products. After that, in Section 5, we provide a MaxSAT formulation for generating exponentiation circuits. We use the exponentiation circuits to arithmetize the equality operator. In Section 6, we vary the number of precomputations k in two existing techniques to generate circuits for univariate polynomial evaluation. We use these circuits for arithmetizating the comparison operator. We present the last primitive in Section 7, where we generate circuits for AND and OR operations. We study veto voting circuits, which are essentially OR operations. In Section 8 & 9, we compose these primitives into larger circuits. We conclude in Section 10.

2 Notation and conventions

In this work, we typically denote circuits by upper-case letters and symbolic variables by lower-case letters. Since multiplications with constants are much cheaper to compute than other multiplications, we denote the former as e.g. $3C$ or $3 \cdot 5$, while we denote the latter using a $\times$ operator.

An arithmetic circuit $C = (V, E)$ is a directed acyclic graph consisting of variable & constant nodes, which form the leaves of the graph, and arithmetic operations. The roots of the graph are the outputs of the circuit. In this work, we consider only addition and multiplication operations, but we note that arithmetic circuits are used in various different contexts, which may allow for a larger set of arithmetic operations such as subtraction.

In many cases, we will write e.g. $C = X + Y$ when we know that C only has a single root, and it is an addition node. In this work, we do not work with the set of edges E, so we use $X \in C$ to actually denote $X \in V$. In other words, we only consider C's nodes. Putting these two shorthands together, we write $[X \times Y \in C] = \{Z \in C \mid Z = X \times Y\}$ to denote the set of all multiplications in C.

We define several metrics for arithmetic circuits below. These metrics only consider multiplications. For this reason, arithmetic circuits that also allow subtraction do not affect the results in this work.

Definition 2.1 (Multiplicative size)**.** The multiplicative size of a circuit or several subcircuits is the number of multiplications in these potentially-overlapping (sub)circuits:

$$\text{size}(C_1, \ldots, C_n) = |[X \times Y \in C_1] \cup \cdots \cup [X \times Y \in C_n]| \,.$$

Definition 2.2 (Multiplicative cost)**.** The multiplicative cost of a circuit is a weighted sum of the cost of all multiplications in a circuit. We consider that the cost of squaring relates to that of arbitary multiplications as the ratio $\sigma : 1$, which yields:

$$\begin{aligned}&\text{cost}(C_1, \ldots, C_n) = \sigma|[X \times X \in C_1] \cup \cdots \cup [X \times X \in C_n]| \\ &+ |[X \times Y \in C_1 \mid X \neq Y] \cup \ldots \cup [X \times Y \in C_n \mid X \neq Y]| \,.\end{aligned}$$

Definition 2.3 (Multiplicative depth)**.** The multiplicative depth of a circuit C is the largest number of multiplications in any path through the circuit:

$$\text{depth}(C) = \begin{cases} 0 & \text{If } C \text{ is a leaf} \\ \max(\text{depth}(X), \text{depth}(Y)) & \text{If } C = X + Y \\ 1 + \max(\text{depth}(X), \text{depth}(Y)) & \text{If } C = X \times Y \end{cases}$$

For any circuit C, there exist an infinite number of different circuits C' that perform the same computation. We denote such an equivalence as $C = C'$. An interesting question to answer is for some circuit C, what is an equivalent circuit C' that minimizes some metric (such as the ones defined above). We denote the minimal multiplicative size, cost, or depth, that can be achieved by any equivalent circuit to some circuit C as $\text{size}^*(C)$, $\text{cost}^*(C)$, and $\text{depth}^*(C)$, respectively.

3 Related work

We briefly go over previous works in the same order as this work, and describe their relation.

3.1 Arithmetization of Sums & Products

Products can be trivially expressed in an arithmetic circuit. While the multiplicative size of a product cannot be reduced, depth-aware arithmetization may rebalance a multiplication tree to reduce the multiplicative depth. This has been studied before, such as in the EVA and Ramparts compilers [3, 13]. However, these compilers rebalance multiplication trees without regard for the multiplicative depths of the operands, so the result is suboptimal. We provide a simple algorithm for optimally rebalancing multiplication trees and a closed-form expression for the resulting multiplicative depth. There are also works [4, 10, 25, 35] that show how to reduce the multiplicative depth of a circuit beyond multiplication trees by distributing products. We note that these techniques are less

powerful than depth-aware arithmetization, as they do not alter *what* is computed, but only *how*. For example, given an arithmetized equality check, they cannot generate all other circuits that optimally trade off depth and size, as there are no products to distribute. The same holds for the polynomial evaluation circuits and the ORs & ANDs we propose in this work. These techniques are more powerful when applied to Boolean circuits, in which doubling and squaring are no-ops. On the other hand, in Section 7, we show that our arithmetic circuits require fewer multiplications than Boolean circuits do for large ORs & ANDs.

3.2 Arithmetization of Exponentiations

The problem of arithmetizing exponentiations (repeated multiplication) is equivalent to the problem of arithmetization of repeated additions. In cryptography, exponentiation circuits have been studied extensively. As a result, methods like square & multiply (also known as double & add) [21], window methods [21], and ones based on heuristics [6] are widely deployed. While these methods are highly efficient in generating circuits, they only optimize for the multiplicative size, meaning that the circuits themselves are not necessarily efficient. Besides that, these methods do not consider that squaring can be cheaper than arbitrary multiplications, and they ignore the cyclic nature of $\mathbb{F}_p$. For example, in the BFV & BGV cryptosystems, a multiplication requires a tensor product involving four distinct multiplication terms, whereas a squaring operation contains only three distinct terms.

Abbas & Gustafsson [1] propose a depth-aware arithmetization method for exponentiations based on a mixed-integer linear program (MILP) formulation. They also show how to adapt the formulation to consider that squaring is cheaper than arbitrary multiplications. The formulation finds optimal arithmetic circuits, but it is slow (see Table 1). In Section 5, we translate this MILP to a MaxSAT formulation that is significantly faster to solve, along with several optimizations.

3.3 Arithmetization of Polynomial Evaluation

The arithmetization of polynomial evaluation has been studied in many previous works, but the work by Paterson & Stockmeyer [29] is of particular interest because it specifically considers minimizing the number of non-scalar multiplications (i.e. the multiplicative size). Paterson & Stockmeyer provide two methods, which we discuss in detail in Section 6, and we show how to tweak them to obtain a depth-size trade-off.

Iliashenko et al. [23, 24] show that for many common integer functions, it is possible to choose a convenient p such that the polynomial is efficiently computable. The key idea is that the polynomial has a sparse structure of equally-spaced monomials apart from the leading term. This choice of p is quite restrictive. For example, for some of the functions, p must be a Mersenne prime. We allow a broader choice of p.

Comparisons between two elements in $\mathbb{F}_p$ have also been studied in other works. Let us focus on $x < y$, from which the other comparisons follow easily. The approach taken by the T2 compiler [18] performs an equality check for each positive case of the comparison. In other words, $\sum_{x'=0}^{p-1} \sum_{y'=x'+1}^{p-1} (x = x' \cdot y = y')$, which has optimal depth but requires a large amount on non-scalar multiplications. Iliashenko & Zucca [24] show how to generate efficient circuits that only work for half of the elements in $\mathbb{F}_p$. These circuits have significantly lower multiplicative size, but a higher depth. In this work, we show how to trade off multiplicative cost and depth. We also use our formulation for finding efficient exponentiation circuits to reuse the powers that must be precomputed for polynomial evaluation, which allows us to find slightly smaller circuits.

3.4 ORs & ANDs

ANDs are typically arithmetized using a product $x_1 \wedge x_2 \wedge \cdots \wedge x_k = x_1 \times x_2 \times \cdots \times x_k$, which leads to a circuit of depth $\lceil \log_2 k \rceil$. The OR operation allows using DeMorgan's law, which does not introduce further non-scalar multiplications. An alternative arithmetization [7] uses a summation and an IsNonZero check to compute such operations on many inputs. It turns out that by combining both arithmetizations in $\mathbb{F}_5$, one can find circuits on the depth-size front. The resulting circuits can outperform equivalent Boolean circuits, requiring fewer non-scalar multiplications.

4 Arithmetization of Sums & Products

Let us consider the class of arithmetic circuits consisting of only multiplications. In such a circuit, one can only reduce the number of multiplications by eliminating common subexpressions, possibly introducing a trade-off between the circuit's multiplicative depth and size. When such an arithmetic circuit does not contain common subexpressions, we cannot reduce its multiplicative size, but we may still reduce its multiplicative depth. An example can be seen in Figure 2. The left subfigure shows a depth-3 product, whereas the right subfigure shows a rearranged product of depth 2. This is the minimal depth that such a circuit can achieve, because a binary tree of depth d can only contain $2^d - 1$ operations, so a product of $n = 4$ distinct inputs requiring $n - 1 = 3$ binary multiplications requires $d \geq \log_2 n = 2$. This simple optimization called rebalancing has been implemented in multiple homomorphic encryption compilers [3, 13].

General arithmetic circuits which also contain additions are harder to analyze. In those cases, reducing the depth beyond rebalancing requires distributing multiplications of sums. It is still possible to determine the minimal depth of such a circuit by relating it to the number of multiplicands. For this reason, we define the multiplicative breadth:

Definition 4.1 (Multiplicative breadth)**.** The multiplicative

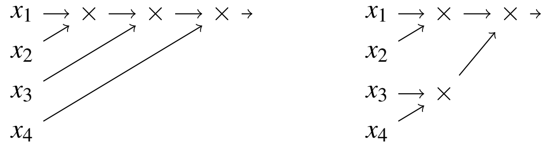


Figure 2: Two circuits that compute $x_1 \times x_2 \times x_3 \times x_4$. Left, an inefficient circuit of depth 3. Right, an optimal circuit that uses a binary tree to compute the product in depth 2.

breadth of a node in an arithmetic circuit is the largest number of multiplicands in any path of the circuit up to that node. The breadth of a node is given by:

$$\text{breadth}(C) = \begin{cases} 1 & \text{If } C \text{ is a leaf} \\ \max(\text{breadth}(X), \text{breadth}(Y)) & \text{If } C = X + Y \\ \text{breadth}(X) + \text{breadth}(Y) & \text{If } C = X \times Y \end{cases}$$

The breadth of an arithmetic circuit does not change when the circuit is rebalanced, therefore it relates to the circuit's minimal multiplicative depth. Since each multiplication can only take two operands, we have that:

$$\text{depth}^*(C) = \lceil \log_2 \text{breadth}(C) \rceil \ . \tag{1}$$

Conversely, it always holds that $\text{breadth}(C) \leq 2^{\text{depth}(C)}$.

In our work we do not consider depth reduction of general arithmetic circuits, but we rather study how to arithmetize several high-level operations. For this reason, we do not consider distributing multiplications of sums. As such, we can consider additions as 'optimization fences' beyond which we do not change the circuit. Even in this limited model, we show that the rebalancing operation described above can be improved by taking into account the depth of the operands. Algorithm 1 shows how to perform depth-aware rebalancing, effectively answering the question of how to optimally perform depth-aware products of distinct multiplicands.

Algorithm 1 Depth-aware product of distinct multiplicands

```
 1: procedure PRODUCT(C_1, ..., C_n)
 2:     Let Q be an empty priority queue
 3:     for i = 1, ..., n do
 4:         Insert C_i into Q with priority depth(C_i)
 5:     while |Q| ≥ 2 do
 6:         Pop X and Y from Q          ▷ Returns lowest depth
 7:         d ← 1 + max(depth(X), depth(Y))
 8:         Insert X × Y into Q with priority d
 9:     Pop C from Q                    ▷ There is only one C in Q
10:     return C
```

We can now derive a closed-form expression for the depth of the circuit resulting from depth-aware arithmetization of a product. Since we do not modify the subcircuits, we model them as having maximal breadth for their depth, yielding:

$$\text{depth}(\text{PRODUCT}(C_1, \ldots, C_n)) = \left\lceil \log_2 \sum_{i=1}^{n} 2^{\text{depth}(C_i)} \right\rceil \ . \tag{2}$$

Since the multiplicative size (and the cost) of such a product is $n-1$, there is no depth-cost trade-off.

5 Arithmetization of Exponentiations

Exponentiations are a crucial primitive in many high-level operations. In this section, we show how to perform optimal depth-aware arithmetization of the map x^t, for a constant exponent t. Our main tool is a MaxSAT solver [28], which we use to solve a reformulation of the mixed-integer linear programming (MILP) formulation by Abas & Gustafsson [1]. Such a solver attempts to find a variable assignment that satisfies a set of clauses called *hard clauses*, and as many additional clauses as possible from a set of *soft clauses*. More precisely, the solver maximizes the total weight of satisfied soft clauses.

We first describe how to generate a minimum-cost circuit, after which we use an adapted formulation to find a minimum-depth anchor point. Having this anchor point and a lower bound on the cost of the exponentiation circuit allows us to efficiently generate the entire front. We conclude by applying our exponentiation circuits for performing equality checks.

5.1 Finding a Minimum-Cost Circuit

Finding minimum-cost exponentiation circuits has been studied under the name of 'addition chains' (as multiplication chains are effectively addition chains in the exponent). The aim is typically to find minimum-length chains, which correspond to minimizing the multiplicative size of exponentiation circuits, but some works also consider the multiplicative cost [1, 26]. Much theoretical work has been done [30] and many heuristics have been proposed [6, 26]. Variants of the problem have also been studied, such as addition sequences [16], which compute multiple exponentiations, reusing intermediate computations. Because exponentiations are so crucial in determining the efficiency of other high-level operations, we are looking for optimal solutions. We propose a MaxSAT formulation that is amenable to computing addition sequences and to consider precomputations provided by other computations (see Section 6.2).

We adapt the MILP formulation by Abbas & Gustafsson [1] into a MaxSAT formulation that is significantly more efficient to solve in practice. Let Boolean variables x_i represent that number i is covered in the addition chain, and let $y_{i,j}$ represent that the chain computes $i + j$. Abbas & Gustafsson define the following constraints for a target exponent t:

1. If $y_{i,j} = 1$, then $x_i = 1$, $x_j = 1$, and $x_{i+j} = 1$.

2. Cutting away: For $k \in [2,t]$, $x_{\lceil \frac{k}{2} \rceil} \vee \cdots \vee x_{k-1} = 1$.

To minimize the size of the addition chain, we want to maximize the number of x_i that are 0. I.e. we want to maximize $\bigwedge_{i \in I} \neg x_i$. The authors also suggest replacing this objective with an objective that maximizes the number of $y_{i,j}$ that are 0, which takes into account that squaring is cheaper than multiplying. In other words, it allows us to minimize the multiplicative cost.

We define $P = \{(i,j) \in [1,t]^2 : i \leq \min(j, t-j)\}$, which is the set of all ordered pairs (i,j) such that $i + j \leq t$. Our basic MaxSAT formulation is as follows:

Hard clauses:

$$\begin{aligned} &(x_t), \\ &(\neg y_{i,j} \vee x_i), \quad \forall (i,j) \in P \\ &(\neg y_{i,j} \vee x_j), \quad \forall (i,j) \in P \\ &\left(\neg x_k \vee \bigvee_{(i,j) \in P : i+j=k} y_{i,j} \right), \quad \forall k \in [2,t] \end{aligned}$$

Soft clauses:

$$\begin{aligned} &\text{weight } 1\ (\neg y_{i,j}), \quad \forall (i,j) \in P : i \neq j \\ &\text{weight } \sigma\ (\neg y_{i,j}). \quad \forall (i,j) \in P : i = j \end{aligned}$$

We can add several cuts to this formulation to reduce the search space. We add three kinds of cuts:

- Bounds from original [1]
- The bounds derived by Thurber & Clift [32]. Given an upper bound on the cost of the chain, we can use these to find lower bounds for the ith element in the chain. For our MaxSAT formulation, let $T_t(c_{\max})$ return a set of pairs (l,u) such that the ith element is bounded from below by l and from above by u for a chain with cost at most $c_{\max}$. We also have that $c_{\max} \geq \sigma s_{\min}$.
- Knowing a lower bound $s_{\min}$ on the size of the chain, we can add a cardinality constraint that $\sum_{i=2}^{t} x_i \geq s_{\min}$. This constraint can be turned into a set of clauses using multiple different techniques. We find a sequential counter approach [31] to work well in practice.[1]

We can add these cuts using the following hard clauses:

$$\begin{aligned} &\left(\bigvee_{m=\lceil \frac{k}{2} \rceil}^{k} x_m \right), \quad \forall k \in [2,t] \\ &\left(\bigvee_{m=l}^{u} x_m \right), \quad \forall (l,u) \in T_t(c) \\ &\left(\sum_{i=2}^{t} x_i \geq s_{\min} \right), \quad \text{encoded with [31]} \end{aligned}$$

[1]Our implementation supports the choices offered by PySAT [22].

To determine $s_{\min}$ we combine three lower bounds reported by Schönhage [30], where $\nu(t)$ is the Hamming weight of t:

$$s_{\min}(t) \geq \lceil \log_2(t) \rceil\,, \tag{3}$$
$$s_{\min}(t) \geq \lceil \log_2(t) + \log_2(\nu(t)) - 2.13 \rceil\,, \tag{4}$$
$$s_{\min}(t) \geq \lceil \log_2(t) + \log_3(\nu(t)) - 1 \rceil\,. \tag{5}$$

Finally, in $\mathbb{F}_p$, we must take into account its cyclic nature (or the resulting exponentiation circuit cannot be considered optimal). For example, $x^{62} \equiv x^{128} \pmod{67}$, but the shortest addition chain for 62 has 8 multiplications, while 128 requires 7 multiplications. We address this by generating an exponentiation circuit for several $t' = t + i\phi(p)$, where $\phi()$ is the totient function, with $i = 1, 2, \ldots$, and selecting the most efficient t'.

The challenge in the solution provided above is in determining when to stop increasing i. To do so, we use monotonically growing lower bound c_{mono} on the multiplicative cost of the exponentiation circuit:

$$c_{\text{mono}}(t') = \sigma \lceil \log_2 t' \rceil\,. \tag{6}$$

If $c_{\text{mono}}(t')$ is greater or equal to the current best cost, we can terminate the search. Next to that, when we find a circuit with a lower multiplicative cost than before, we can lower $c_{\max}(t')$, making the formulation faster to solve and allowing us to skip targets t' for which $\sigma s_{\min}(t) \geq c_{\max}(t')$.

5.2 Finding a Minimum-Depth Anchor Point

One very common method for arithmetizing exponentiations is the square & multiply method, which produces a circuit as shown in Figure 3. As seen in the figure, this method actually produces minimum-depth circuits, seeing as a multiplication can at most double the exponent in either of its inputs, so:

$$\text{depth}^*(X^t) = \lceil \log_2 t \rceil\,. \tag{7}$$

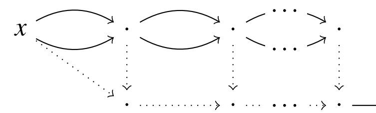


Figure 3: Square & multiply method for computing x^t.

While square & multiply produces a minimum-depth circuit, it does not necessarily produce a minimum-depth anchor point (i.e. a circuit that is minimal in depth and secondarily minimal in cost). To find such an anchor point, we make another call to the MaxSAT formulation, but this time we provide the following constraints:

- The maximum depth is $\lceil \log_2 t \rceil$.
- The maximum cost is $c_{\max} = \sigma \lfloor \log_2 t \rfloor + \nu(t) - 1$, corresponding to the cost of square & multiply.

Table 1: Solver time in seconds and standard deviations on an Apple M1 CPU averaged over 10 runs. MILP w/ cuts reports Abbas & Gustafsson's [1] with our additional cuts. The MILP formulations were solved by the commercial Gurobi solver [19] on 8 threads whereas the MaxSAT formulations used the open-source single-threaded Pysat RC2 solver with the Glucose 4.2.1 SAT solver [5]. The MaxSAT formulation is at least an order of magnitude faster with only one thread.

Exp.	No depth constraint			Minimum depth		
	MILP	MILP w/ cuts	MaxSAT	MILP	MILP w/ cuts	MaxSAT
31	0.02 ±0.01	0.02 ±0.00	0.00 ±0.00	0.06 ±0.01	0.06 ±0.01	0.00 ±0.00
71	0.88 ±0.07	0.89 ±0.02	0.01 ±0.02	1.97 ±0.26	1.83 ±0.05	0.01 ±0.03
111	2.19 ±0.13	2.18 ±0.07	0.01 ±0.02	5.07 ±0.12	5.03 ±0.09	0.62 ±1.96
151	7.48 ±0.27	7.74 ±0.46	0.07 ±0.23	15.83 ±0.22	16.06 ±0.14	0.25 ±0.78
191	140.70 ±3.70	141.02 ±3.57	0.94 ±2.97	113.75 ±3.59	112.00 ±4.60	2.24 ±7.07
231	4.92 ±0.11	5.04 ±0.19	0.24 ±0.75	205.17 ±4.45	212.01 ±7.68	4.02 ±12.73

What remains, is to modify the MaxSAT formulation to incorporate a bound on the depth $d_{\max}$ of the exponentiation circuit. We introduce the following sets of hard clauses:

$$\begin{aligned}(d_{k,m+1} \vee \neg d_{i,m} \vee \neg y_{i,j}), &\quad \forall (i,j) \in P, \forall m \in [0, d_{\max}]\\ (d_{k,m+1} \vee \neg d_{j,m} \vee \neg y_{i,j}), &\quad \forall (i,j) \in P, \forall m \in [0, d_{\max}]\\ (\neg d_{k,d_{\max}+1}), &\quad \forall k \in [2,t]\\ (d_{1,0}). &\end{aligned}$$

These clauses encode the depth of an exponent as a Boolean vector, such that the highest-index Boolean that is true represents the depth of that exponent. By forcing the $d_{\max}+1$th Boolean to be false, we ensure that the depth limit is not exceeded. This is a different encoding than the one used by Abbas & Gustafsson [1], which uses integers to encode depth.

It is tough to say with certainty *why* the MaxSAT formulation outperforms the MILP, and if this remains the case for large exponents. Of course, the approach taken by these solvers differs significantly. A MILP formulation may be easily solvable when the linear programming relaxation is a (close) solution to the MILP. We conjecture that this is not the case for this formulation, where almost all variables are Boolean. Moreover, it is not easy to go from a suboptimal solution to an optimal one. The MaxSAT solver takes a radically different approach; simplifying clauses and guessing variables. The solver tries to find chains that are increasingly more costly, until it finds one. Many of these iterations can be trivially skipped due to the cuts we add to the formulation.

5.3 Finding Circuits on the Depth-Cost Front

We can generate circuits on the depth-cost front using the same method that we described for finding an anchor point given a minimal-depth circuit with suboptimal cost. We do so by incrementally going through all such circuits, from least to highest depth. For the the maximum cost, we can use the current best cost. We present our approach in Algorithm 2, in which we call our MaxSAT formulation as $\textsc{AddChain}(t, d_{\max}, c_{\max}, \sigma, s_{\min})$, which returns a circuit satisfying the constraints or $\perp$ if no circuit could be found.

Algorithm 2 Depth-aware product of distinct multiplicands

```
1:  procedure GenExpFront(C)
2:      Find and yield C such that cost(C) = cost*(C)
3:      d ← ⌈log2 t⌉
4:      c ← σ⌊log2 t⌋ + ν(t) − 1
5:      while c < cost*(C) and d < depth(C) do
6:          Use largest s_min that satisfies (3), (4), and (5)
7:          C′ ← AddChain(t, d, c, σ, s_min)
8:          if C′ ≠ ⊥
9:              yield C′
10:             c ← cost(C′)
11:         d ← d + 1
```

5.4 Case Study: Equality Checks

As explained by Iliashenko & Zucca [24], equality checks can be arithmetized as $[x = y] = 1 - (x-y)^{p-1}$. The cost of such an operation is almost exclusively determined by the exponentiation circuit, as it is the only operation requiring multiplications. In Figure 4, we plot the multiplicative cost of the optimal exponentiation circuits we found using our MaxSAT formulation for different prime moduli p and for fixed $\sigma = 0.75$. We also show how long it took to generate these circuits, with and without consideration of the cyclic nature of $\mathbb{F}_p$. For the moduli in Figure 4, the circuits generated by ignoring or considering the modulus are the same, but it is significantly more efficient to ignore the modulus. One can interpret the 'considering modulus' generation time, which is when we consider the cyclic nature of $\mathbb{F}_p$, as the time it takes to prove optimality.

6 Arithmetization of Polynomial Evaluation

For many high-level operations there is not a straightforward arithmetization. For example, checking if a field element is within a given range can be expressed as a large number of equality operations but this is inefficient. In these situations, it is typical to interpolate a polynomial and to find an efficient circuit to evaluate it. In this section, we show how to perform depth-aware arithmetization for univariate polynomial evaluation. These cover many common operations including

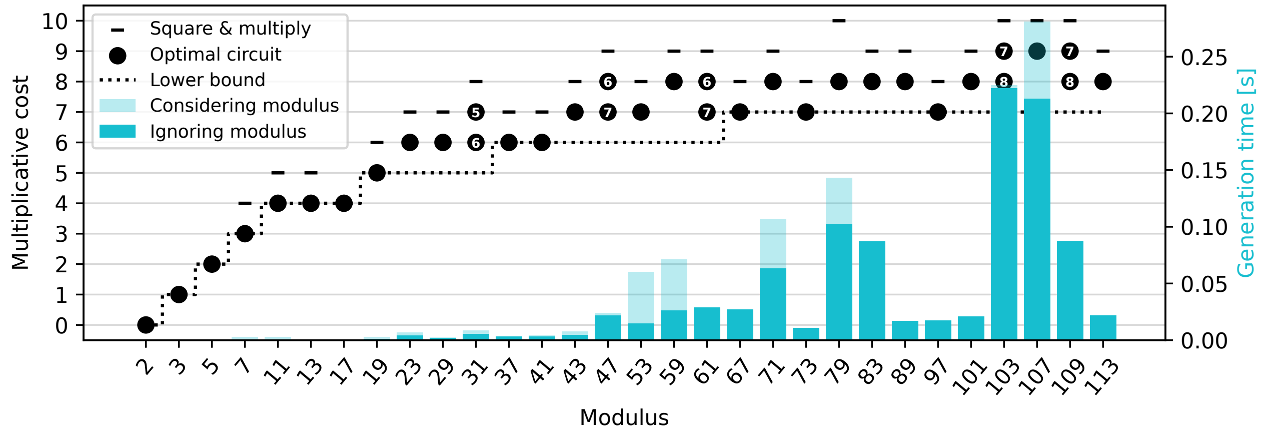


Figure 4: Equality circuits generated using square & multiply and our MaxSAT formulation, where $\sigma = 1.0$. Square & multiply is only optimal when p is of the form $2^k + 1$. When we find a depth-cost trade-off, we denote the depth in the markers. The run time of our algorithm is hard to predict, but it increases with the modulus p. In some cases, ignoring the modulus makes a large difference in generation time, but the result is not guaranteed to be optimal.

comparisons, which we highlight in our case study at the end of this section.

When it comes to the multiplicative cost of polynomial evaluation circuits, we know that the multiplicative cost of a degree-d polynomial is at least as high as that of an exponentiation circuit with target t. Next to that, Paterson & Stockmeyer [29] provide an asymptotic bound:

$$\text{cost}^*\left(x^d\right) \leq \text{cost}^*\left(\sum_{i=0}^{d} c_i x^i\right) \leq O(\sqrt{d}) \,. \tag{8}$$

In fact, Paterson & Stockmeyer already provide two algorithms that generate circuits with the same asymptotic complexity. We discuss these two algorithms later on.

The multiplicative depth of polynomial evaluation circuits can also be bounded. To achieve the minimal depth, we can simply compute all monomials and evaluate the polynomial using a linear combination. So:

$$\text{depth}^*\left(\sum_{i=0}^{d} c_i x^i\right) = \lceil \log_2(d) \rceil \,. \tag{9}$$

This is an equality because we cannot evaluate x^d with fewer multiplications. In Paterson & Stockmeyers's methods, this is equivalent to choosing $k = d$. Our key idea for generating circuits that trade off multiplicative depth and cost is to vary this parameter k.

6.1 Baby-Step Giant-Step

The baby-step giant-step method was one of the two algorithm proposed by Paterson & Stockmeyer [29], but we refer to it with this name because it is colloquially known as such in the cryptography community. It is also known as the two-level evaluation method [15].

The algorithm, parameterized by an integer $1 \leq k \leq d$, starts by precomputing the monomials $X^2, X^3, \ldots, X^k$. It will later use these precomputed powers to evaluate a $k-1$-degree polynomial without performing any more multiplications. In this work, we also want to minimize the multiplicative depth, so we do not use sequential multiplications to compute these powers. Instead, we start by computing X^2 and use it to compute X^3 and X^4. We then use X^4 to compute $X \times X^4 = X^5, X^2 \times X^4 = X^6, \ldots, X^4 \times X^4 = X^8$, etc. Given these precomputed powers, the key idea behind this algorithm is the following identity:

$$\left[\sum_{i=0}^{d} c_i X^i\right] \leftarrow X^k \left[\sum_{i=0}^{d-k} q_i X^i\right] + \left[\sum_{i=0}^{k-1} r_i X^i\right] , \tag{10}$$

where the rightmost polynomial can be evaluated using only additions and constant multiplications. In other words, the polynomial can be evaluated by taking approximately $\frac{d}{k}$ giant steps after computing k baby steps. Paterson & Stockmeyer show that this method requires approximately $2\sqrt{d}$ multiplications for the right choice of k. This makes it asymptotically optimal in terms of the multiplicative cost and size. Due to its sequential nature, the circuits generated by this method are typically larger in depth than the circuits generated by the other two methods that we discuss.

6.2 Paterson & Stockmeyer's method

Paterson & Stockmeyer also propose a method that evaluates polynomials of a specific degree in $\sqrt{2d} + O(\log d)$ non-constant multiplications for the right choice of k. This method is defined for monic polynomials (i.e. the leading coefficient

is 1) of degree $d = (2^n - 1)k$, but it can be adapted to evaluate any polynomial by extending it to the next monic polynomial of the correct degree (or using a constant multiplication if it is a non-monic polynomial of the correct degree). We can then remove this added monomial from the final result by computing it and subtracting it or by adapting the coefficients.

Paterson & Stockmeyer's method [29] works by reducing the evaluation of a degree-$(2^n - 1)k$ monic polynomial to the evaluation of two monic polynomials of degree $(2^{n-1} - 1)k$ and a polynomial of degree $k - 1$ using the following identity:

$$\left[X^{(2^n-1)k} + \sum_{i=0}^{(2^n-1)k-1} c_i X^i\right] \leftarrow \left(X^{2^{n-1}k} + \left[\sum_{i=0}^{k-1} c'_i X^i\right]\right) \left[X^{(2^{n-1}-1)k} + \sum_{i=0}^{(2^{n-1}-1)k-1} q_i X^i\right] + \left[X^{(2^{n-1}-1)k} + \sum_{i=0}^{(2^{n-1}-1)k-1} r_i X^i\right], \quad (11)$$

where the square brackets group together the terms of a polynomial. The coefficients of these smaller polynomials can be obtained using a Euclidean division. Note that the polynomial of degree $k - 1$ can be computed using the precomputed powers without any multiplications. Note that where the previous method only precomputes monomials $X^2, X^3, \ldots, X^k$, this method must also precompute monomials $X^{2k}, X^{4k}, X^{2^{n-1}k}$, which requires $n - 1$ squarings.

As described previously, the method can be extended to any polynomial of degree-d by padding it with a monomial $(2^n - 1)k \geq d$, which is of the correct degree. However, we must compensate for this added monomial in the final result. If it holds that $i = (2^n - 1)k \mod \phi(p) \leq d$, then we can easily compensate for it by decrementing the i-th coefficient. Otherwise, we must compute the monomial separately and subtract it at the end.

For the case that we must compute the padding monomial separately, we slightly modify the MaxSAT formulation described in Section 5 to take into account that the polynomial evaluation circuit already precomputes a large number of monomials. We ensure that these monomials count for free towards the cost of the addition chain, while still considering their depth. We do so by adding new variables z_k that represent using previously-computed power X^k. When they are enabled, they incorporate the fixed depth of the precomputed power. Given precomputed powers $t_1, \ldots, t_n$ with depths $d_1, \ldots, d_n$, we add the following hard clauses:

$$(d_{t_i,d_i}, \neg z_{t_i}), \quad \forall i \in [1, n]$$

Next to that, we adapt the following hard clause in the original formulation to allow x_k to be true when z_k is:

$$\left(\neg x_k \vee z_k \vee \bigvee_{(i,j)\in P: i+j=k} y_{i,j}\right). \quad \forall k \in \{t_1, \ldots, t_n\}$$

We also have to remove the cuts described in Section 5.1 from the formulation, as they do not apply to depth-constrained circuits.

6.3 Our work: Divide & conquer

We propose a new method for evaluating univariate polynomials of any degree inspired by Paterson & Stockmeyer's method. While our method does not achieve as small of a multiplicative cost, it achieves a low multiplicative depth. It is essentially a simplified version of Paterson & Stockmeyer's method that retains the divide & conquer strategy. The key idea is to split evaluation of a degree-$2^n k$ polynomial into the evaluation of two degree-$2^{n-1}k$ polynomials:

$$\left[\sum_{i=0}^{d} c_i X^i\right] \leftarrow X^{2^{n-1}k} \left[\sum_{i=0}^{d-(2^{n-1}k-1)} q_i X^i\right] + \left[\sum_{i=0}^{2^{n-1}k-1} r_i X^i\right], \quad (12)$$

where $d \leq 2^n k$. This method requires the same precomputations as Paterson & Stockmeyer's method.

We briefly analyze the cost and depth of the circuits generated by our method. Let $N(d)$ denote the cost of computing a degree-d polynomial using our method when we have already computed the precomputations. We have:

$$N(2^n k) \leq \begin{cases} 0 & \text{If } n = 0 \\ 1 + 2N(2^{n-1}k) & \text{If } n > 0 \end{cases}, \quad (13)$$

$$\leq 1 + 2(1 + 2N(2^{n-2}k)), \quad (14)$$

$$= 3 + 4N(2^{n-2}k), \quad (15)$$

$$\leq 2^i - 1 + 2^i N(2^{n-i}k), \quad (16)$$

$$\leq 2^n - 1 + 2^n 0, \quad (17)$$

$$= 2^n - 1. \quad (18)$$

If it takes $k - 1$ multiplications to compute $X^2, \ldots, X^k$ and $n - 1$ squarings to compute $X^{2k}, X^{4k}, \ldots, X^{2^{n-1}k}$, then the total cost of our circuit C is:

$$\text{cost}(C) \leq k + n + 2^n - 3 \leq k + \log_2\left(\left\lceil \frac{d}{k} \right\rceil\right) + \left\lceil \frac{d}{k} \right\rceil. \quad (19)$$

The depths of precomputations X^i for $i = 2, \ldots, k$ are $\lceil \log_2 i \rceil$, and the depths of $X^{2^i k}$ for $i = 1, \ldots, n - 1$ are $\lceil \log_2 k \rceil + i$. As a result, the depth of the circuit is:

$$\text{depth}(C) \leq \lceil \log_2 k \rceil + n \leq \lceil \log_2 k \rceil + \left\lceil \frac{d}{k} \right\rceil. \quad (20)$$

From this analysis it is clear that choosing a large value of k reduces the depth significantly.

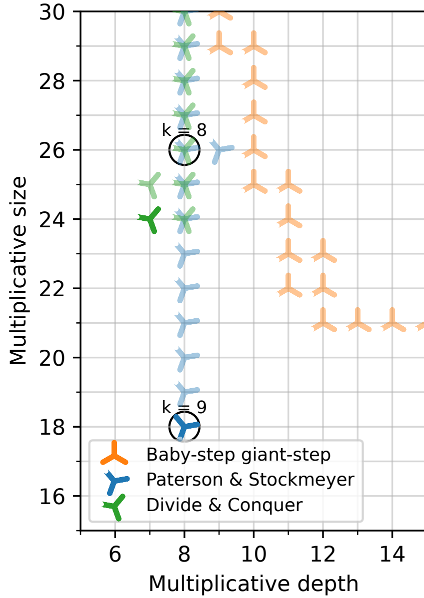


Figure 5: Polynomial evaluation circuits for computing $x \pmod 7$ in $\mathbb{F}_{127}$. This is a degree-126 polynomial, so Paterson & Stockmeyer's parameter for minimizing multiplicative size is $k = \sqrt{0.5 \cdot 126} \approx 8$. However, the optimum occurs when $k = 9$. Divide & conquer leads to a lower depth.

6.4 Finding Circuits on the Depth-Cost Front

The three methods described above all achieve a different depth-cost trade-off when varying k. It turns out that, while it is possible to compute the optimal k for reducing the multiplicative cost, there are cases where other values of k achieve a lower cost. In Figure 5 we highlight such a situation. In this figure, we show all circuits computing $x \pmod 7$ in $\mathbb{F}_{127}$ that we can generate by varying k. While Paterson & Stockmeyer show that $k = 8$ minimizes the multiplicative cost, it turns out that we can achieve a significantly better circuit using $k = 9$. Instead of searching the entire range of $k \in [1,d]$, we use the theoretical optimum for the minimum cost circuit k^* to limit the search to $k \in [1,2k^*]$. For Paterson & Stockmeyer's method, $k^* \approx \sqrt{2d}$, whereas $k^* \approx \sqrt{d}$ for the other two methods. Moreover, we use the estimated costs and depths (e.g. from (19) and (20)) to prevent generating circuits that cannot be in the Pareto front.

6.5 Case Study: Comparisons

We show that our depth-aware arithmetization method allows to generate a front of circuits that trade off multiplicative depth and cost, even for complex operations such as comparisons. We use the technique proposed by Iliashenko & Zucca [24] for performing comparisons between half of the elements in the field $\mathbb{F}_p$ using a univariate polynomial evaluation. By computing the leading term of the polynomial separately, the remainder of the polynomial can be decomposed so that its degree is only $\frac{p-1}{2}$.

Another method for generating such circuit is implemented in the T2 compiler [18], in which the comparison is implemented as a number of equality checks:

$$[X < Y] = \sum_{a=\frac{p+1}{2}}^{p} [(X-Y)=a] = \sum_{a=\frac{p+1}{2}}^{p} 1-(X-Y-a)^{p-1} \,. \tag{21}$$

We provide an optimistic implementation of this technique in which we use the minimal-cost exponentiation circuit to implement the equality checks.

We also provide an optimistic implementation of the work by Iliashenko & Zucca [24], in which we only use the Paterson & Stockmeyer method with their choice of k with the intent of minimizing the multiplicative cost. One problem is that their proposed way to compute the final term requires a certain polynomial degree, but it is not possible for all p to find a certain k. Instead, we use our method for finding the optimal addition chain given precomputed powers to compute the leading term of the univariate polynomial.

In Table 2 we provide an overview of different methods for generating comparison circuits. We find that our work consistently finds circuits in the depth-size front, but the other methods do so too. We mark values on the front in bold. For example, while the T2 compiler finds circuits with large size, their depth is minimal. We find that the method by Iliashenko & Zucca does not outperform ours, unless we apply common subexpression elimination (CSE). In some cases, this allows the method to find circuits on the front. We evaluate the run time of these circuits using fhegen [27] to generate parameters and execute the circuits using HElib on a Threadripper 7970X CPU, using only one thread to compile and evaluate the circuits. The machine has 4x64GB DDR5 RAM, but only a fraction was used.

7 Arithmetization of ANDs and ORs

Finally, we study the depth-aware arithmetization of AND and OR operations. The typical arithmetization of an AND operation is to treat it as a product:

$$X_1 \wedge \cdots \wedge X_k = X_1 \times \cdots \times X_k \,. \tag{22}$$

As shown in Section 4, there is a single optimal circuit C_1 to compute this product. It has the following properties:

$$\mathrm{cost}(C_1) = k - 1 + \mathrm{cost}(X_1, \ldots, X_k) \,, \tag{23}$$

$$\mathrm{depth}(C_1) = \left\lceil \log_2 \sum_{i=1}^{k} 2^{\mathrm{depth}(X_i)} \right\rceil \,. \tag{24}$$

OR operations are sometimes arithmetized as follows:

$$X_1 \vee \cdots \vee X_k = (X_1 + \cdots + X_k)^{p-1} \,, \tag{25}$$

Table 2: Comparison circuits for different moduli p with squaring cost $\sigma = 1.0$. Run times were averaged over 10 iterations. Our circuits often outperform previous techniques. For T2 & IZ21, we also report results after common subexpression elimination.

Method	$p=29$			$p=43$			$p=61$			$p=101$			$p=131$		
	Depth	Size	Run time (s)	Depth	Size	Run time (s)	Depth	Size	Run time (s)	Depth	Size	Run time (s)	Depth	Size	Run time (s)
T2 [18] + CSE	**5**	**84**	1.16	**6**	**147**	4.86	7	210	6.79	**7**	**400**	11.71	8	520	17.24
IZ21 [24]	7	12	0.44	8	18	0.68	9	14	0.55	**8**	**16**	**0.60**	9	30	1.22
IZ21 [24] + CSE	7	12	0.44	8	16	0.59	**9**	**13**	0.52	**8**	**16**	0.63	9	23	0.97
Our work	**6**	**11**	0.41	**7**	**12**	**0.44**	**7**	**15**	0.59	**8**	**16**	0.62	**8**	**20**	**0.86**
	7	**10**	**0.38**				**8**	**14**	**0.51**						

where x^{p-1} maps $0 \mapsto 0$ and $\{1,\dots,p-1\} \mapsto 1$. Note that this arithmetization is only guaranteed to work when $k < p$, otherwise the result of the summation might wrap around the modulus. Let circuit C_2 be a circuit that evaluates this arithmetization, which first sums the operands and then uses another circuit C_{exp} for exponentiation by $p-1$. Then, $C_2(C_{\text{exp}})$ has the following properties:

$$\operatorname{cost}(C_2(C_{\text{exp}})) = \operatorname{cost}(C_{\text{exp}}) + \operatorname{cost}(X_1,\dots,X_k)\ , \tag{26}$$

$$\operatorname{depth}(C_2(C_{\text{exp}})) = \operatorname{depth}(C_{\text{exp}}) + \max_{i=1,\dots,k} \operatorname{depth}(X_i)\ . \tag{27}$$

While this method allows varying the depth and size using different circuits for C_{exp}, this only provides minimal variance.

DeMorgan's law provides a bidirectional transformation between AND and OR circuits that does not increase the size or depth because it only requires negation, which does not require non-scalar multiplications:

$$X_1 \wedge \cdots \wedge X_k = \overline{\overline{X_1} \vee \cdots \vee \overline{X_k}}\ . \tag{28}$$

So, either of the two arithmetizations above can be used for AND and OR operations at the same depth and size cost. In fact, they can be composed to achieve a hybrid arithmetization. This allows one to trade off depth and size. It also allows reaching smaller sizes than what could be reached by a non-hybrid arithmetization.

We cannot prove that minimizing the depth and size of the hybrid arithmetization described above coincides with minimizing the depth and size of all potential arithmetic circuits for ANDs and ORs. That said, we argue that our method is a useful heuristic.

7.1 Finding a Minimum-Cost Circuit

It is easy to see that if $k < p$, then $\operatorname{cost}(C_2(C_{\text{exp}})) < \operatorname{cost}(C_1) \iff \operatorname{cost}(C_{\text{exp}}) < k-1$. So in this case, it is easy to decide the minimum-cost circuit. Let $N(k)$ represent the minimal multiplicative cost of a circuit for the hybrid arithmetization of an AND or OR operation with k operands, and let c denote the multiplicative cost of C_{exp}. We have:

$$N(k) = \min(c, k-1) \quad \text{if } k \le p-1\ . \tag{29}$$

When $k \ge p$, we must consider a hybrid arithmetization. Notice that the cost of the smallest hybrid circuit $C_3(C_{\text{exp}})$ grows monotonically with k. So, if it holds that $\operatorname{cost}(C_{\text{exp}}) \le p-1$, we can perform $C_2(C_{\text{exp}})$ on $p-1$ operands (e.g. $X_1,\dots,X_{p-1}$) to obtain a new problem with $k-(p-1)$ operands. It turns out that $\operatorname{cost}^*(C_{\text{exp}}) \le p-1$ always holds.

Using the strategy described above, we get that:

$$\begin{aligned} N(k) &= c + N(k-(p-1)+1)\ , && (30)\\ &= 2c + N(k-p-(p-1)+1)\ , && (31)\\ &\ \vdots && (32)\\ &= rc + N(k - r(p-1) + r)\ , && (33)\\ &= rc + N(k + r(2-p))\ . && (34) \end{aligned}$$

We reach the base case when $k + r(2-p) \le p-1$. This happens when $r = \lceil \frac{p-1-k}{2-p} \rceil$, so we have:

$$\operatorname{cost}^*(C_3(C_{\text{exp}})) = N(k) = \left\lfloor \frac{k}{p} \right\rfloor c + \min\left(c, k - \left\lfloor \frac{k}{p} \right\rfloor p - 1\right) \tag{35}$$

Notice that increasing c always increases the total multiplicative cost, apart from the case where $k < p$ and $c \ge k-1$, in which case c does not influence the result. We conclude that to minimize $N(k)$, c needs to be minimal.

7.2 Finding Circuits on the Depth-Cost Front

In minimizing the multiplicative depth of the circuit, we define a useful metric called fullness. This metric captures both the depth of the circuit and how many multiplications can still be absorbed by the multiplication tree in the outer layer of the circuit without increasing the circuit's depth.

Definition 7.1 (Fullness)**.** The fullness is defined as:

$$\begin{aligned} \operatorname{fuln}(X+Y) &= 2^{\max(\operatorname{depth}(X),\operatorname{depth}(Y))} \\ \operatorname{fuln}(X \times Y) &= \operatorname{fuln}(X) + \operatorname{fuln}(Y) \\ \operatorname{fuln}(v) &= 1 \end{aligned}$$

Notice that:

$$\operatorname{depth}(C) = \lceil \log_2 \operatorname{fuln}(C) \rceil\ .$$

To find a minimum-depth anchor point, we put forward a recursive algorithm that finds a circuit for performing an AND operation while satisfying the constraint that the fullness is at most f, and the cost is less than c. We present it in Algorithm 3 (see Appendix A), in which $\mathrm{cost}(C)$ ignores the cost of subcircuits $X_1,\dots,X_k$. The algorithm also inputs E, which is a collection of exponentiation circuits that are on the Pareto front, and p, the order of the prime field.

Our recursive algorithm is essentially a bounded search. We use the bounds derived above to decide whether certain branches are not worth exploring. By starting with $f = 2^d$ for $d = \lceil \log_2 \mathrm{fuln}(X_1) \rceil$, where X_1 is the operand with the highest fullness, we can iteratively increment d until the algorithm finds a circuit. This first circuit is a minimum-depth anchor point because the algorithm outputs the minimal cost circuit for this fullness bound f.

We can keep going in the fashion described above, incrementing d, to generate the entire depth-cost front. Since it is easy to compute $\mathrm{cost}^*(C_3(C_{\mathrm{exp}}))$, we know when to stop the search. Note that while we describe the algorithm to compute a circuit for an AND operation, the algorithm for OR operations follows almost identically: For OR operations, one must apply DeMorgan's law.

7.3 Case Study: Veto Voting

We study the problem of veto voting, where multiple parties submit a Boolean value, indicating whether they veto or not. If no one vetoes, the result should be false. If anyone vetoes, the result should be true. This is exactly an OR operation. We consider the setting where we do not know a bound on the possible number of vetoes.

In Figure 6, we demonstrate the circuits that our algorithm generates for two values of p when the number of operands grows. It is clear that for almost every number of operands, there exists a cost-depth trade-off. What is more, there is also a trade-off between different values of p. Whereas a larger value of p allows one to find circuits with fewer multiplications when the number of operands grows, there are still cases where one might favor a smaller p as it provides a better depth-cost trade-off. For example, when there are 13 operands, $p = 7$ permits a depth-4 circuit at 10 multiplications, while $p = 13$ requires 12 multiplications. Finally, notice that there are only a few cases where computing an OR operation using a C_1 circuit is necessary to achieve a minimum depth. In many other cases, we can achieve the same minimum depth with far fewer multiplications.

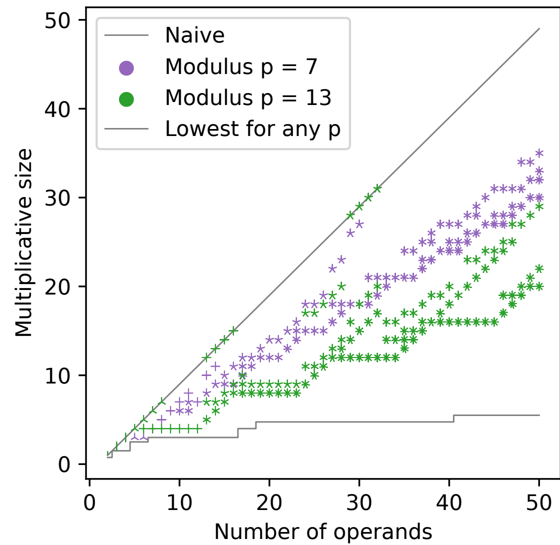


Figure 6: Circuits computing an OR operation with $\sigma = 1.0$, for a growing number of operands. The number of ticks on a marker indicates the depth of the circuit. Depending on the depth that one wants to achieve and the number of operands, it is better to choose $p = 7$ or $p = 13$.

8 Depth-Aware Composition

In the previous sections, we put forward methods for depth-aware arithmetization of several common primitives, but many interesting circuits emerge as the composition of these primitives. In this section, we discuss depth-aware arithmetization of high-level circuits that compose multiple primitives.

Suppose we have a circuit $X^{31} < Y^{31}$. We can generate a front for the exponentiation circuits of X^{31} and Y^{31}, but at that point we are stuck, because our method for arithmetizing comparisons inputs subcircuits rather than two fronts of circuits. For composition, we propose the following heuristic: we generate a new Pareto front in which we try all possible combinations of input arithmetizations. This is a heuristic because we do not change the arithmetizations of the inputs, even when this could lead to a lower cost or depth. In short, we apply depth-aware arithmetization to all operands and extract the Pareto front from the resulting set of all combinations:

$$\mathrm{Arith}(C(C_1,\dots,C_\ell)) = \{\mathrm{Arith}(C_1) \times \cdots \times \mathrm{Arith}(C_\ell)\}. \quad (36)$$

Because of this heuristic, composing two optimal subcircuits does not necessarily lead to an optimal composition. For example, the condition $X^{10} = 0$ is equivalent to $X = 0$, meaning that straightforward composition leads to a circuit that is unnecessarily large. On the other hand, it often infeasible to arithmetize a complex high-level circuit to optimality because its search space is astronomical.

While the method described above offers a generic solution of dealing with composition, it can be inefficient. For example, when we want to arithmetize $X^{31} + Y^{31}$, we know that it is never better to choose a subcircuit for X^{31} with a lower depth as the subcircuit for Y^{31} and vice versa. In those cases, we do not exhaustively try all combinations, but we iteratively increment a depth limit and choose the lowest-cost subcircuits that

Table 3: Fronts generated by our methods for the *cardio* and *cardio-elevated* circuits, displaying the cost-depth trade-off.

Cardio risk assessment				Cardio elevated risk			
Depth	$\sigma = 1.0$	$\sigma = 0.75$	$\sigma = 0.5$	Depth	$\sigma = 1.0$	$\sigma = 0.75$	$\sigma = 0.5$
Gen.(s)	*54*	*55*	*55*		*56*	*56*	*55*
11	419.0	389.0	359.0	14	428.0	396.0	364.0
12		384.0	349.0	15	·	389.0	350.0
13		374.0	329.0	16	·	383.0	338.0
				·	·	·	·
				19	427.0	·	·
				·		·	·
				21		380.0	333.0

still satisfy the depth limit. Consider the following example. If arithmetization of C_1 produces a front with circuits with depths $\{2,3,6\}$ and C_2 produces depths $\{3,5\}$, we do not have to consider the product of these sets to arithmetize $C_1 + C_2$. Instead, we only need to consider $\{(3,3),(3,5),(6,5)\}$. After all, when the depth of C_2 is at least 3, it is always better to choose C_1 with depth 3 as opposed to depth 2.

Finally, one might consider heuristics that cut away even more solutions. For example, increasing a circuit's depth by one layer while saving one multiplication may not be worth it in practice. We do not implement such a heuristic.

To highlight the effectiveness of our methods, we apply them to a practical example that composes all primitives described in this work: We evaluate them on the *cardio* circuit as proposed by Carpov et al. [11] and used as a benchmark in other works [33]. The circuit computes predicates relating to a person's cardiac health and returns how many evaluate to true. These predicates involve comparisons, such as checking whether a person's weight is smaller than its height - 90. We also consider a variant of this circuit that we call *cardio-elevated*, which only returns if any of the risk factors were true. In other words, we compute an OR over all the predicates.

In Table 3 we present the results of our methods applied to the *cardio* and *cardio-elevated* circuits for a fixed value $p = 257$, since all values fit under this modulus. We report the fronts that our methods generated for different costs of squaring σ (which is an estimate used to prioritize squaring over generic multiplication), and how long these fronts took to generate (after implementing several run time optimizations). We do not take the cyclic nature of $\mathbb{F}_p$ into account for the exponentiations in padding the polynomials to make it run in reasonable time, and since these are unlikely to produce significantly better results for $p = 257$. We show that the *cardio* circuit can be evaluated in 419 multiplications.

Notice that if we solely optimize multiplicative cost/size, the resulting circuits may be wasteful in terms of the multiplicative depth. A good example is in the *cardio-elevated* circuit when $\sigma = 1.0$: If we would only focus on multiplicative cost, we would save 1 multiplication at the cost of 5 layers of depth. Moreover, optimizing both multiplicative cost *and* depth allows one to save multiplicative cost on branches of the circuit that do not contribute to the multiplicative depth, unlike what happens when optimizing for depth in isolation.

Table 4: Run times averaged over 10 iterations; $\sigma = 0.75$. Our circuits outperform TFHE when the run time is amortized.

Work	Circuit	Slots	Time (s)	Amortized (s)
	Cardio risk assessment			
Ours	Depth-11	128	43.98	**0.34**
	Depth-12	128	50.14	0.39
	Depth-13	128	50.87	0.40
Previous	-	128	50.71	0.40
TFHE	-	1	0.97	0.97
	Cardio elevated risk			
Ours	Depth-14	128	57.26	0.45
	Depth-15	128	57.56	0.45
	Depth-16	128	56.53	**0.44**
	Depth-21	128	73.78	0.58
Previous	-	128	156.65	1.22
TFHE	-	1	0.49	0.49

Finally, we measure the run times of our circuits using the same machine described in Section 6, and compare these to the equivalent circuits when generated using naive arithmetization or when implemented in TFHE-rs[2] in Table 4. We use unsigned 256-bit high-level integers to generate these circuits in TFHE. For the *Previous* arithmetization, we use products for ANDs, IZ21 for comparisons, and square & multiply for exponentiation. We conclude that the lower-cost circuits are not always better, and the trade-off between cost and depth allows for more efficient circuits in practice.

9 Results

We apply depth-aritmetization to two practical applications in the context of secure medical data processing to demonstrate the improvements over previous methods. Specifically, we use the BGV cryptosystem to analyze an imaginary medical survey and to perform machine learning-based inference on the Diagnostic Wisconsin Breast Cancer Database [34]. We first show how to confidentially compute the mean and variance, which can be used to perform simple statistical analyses, e.g. for processing medical surveys. Next, we show how to confidentially perform inference using a previously-trained logistic regression model. For our experiments, we used $p = 6,143$ for which we can choose ring dimension $m = 2^{16}$ to allow for a large enough ciphertext modulus to perform polynomial evaluation. With these parameters, we can perform the computations on 1024 slots in parallel.

[2]We used version 0.11: https://docs.zama.ai/tfhe-rs

Table 5: Average run time of 10 executions in seconds and amortized (across 1,024 slots) in ms for computing the mean, variance, and logistic regression inference. Our most shallow circuits consistently outperform IZ21.

	IZ21 [24]				Ours ($\sigma = 1$)			
	Depth	Size	Time	Amort.	Depth	Size	Time	Amort.
Mean	17	126	25.6	25 ms	14	126	21.5	**21 ms**
Variance	18	633	92.2	90 ms	15	638	81.1	**79 ms**
Inference	18	253	53.2	52 ms	15	253	45.9	**45 ms**

9.1 Mean and variance estimation

The challenging part in computing the mean and variance is computing the (integer) division. A division by some divisor d in $\mathbb{F}_p$ is not as simple as a multiplication by d^{-1}, because this is only correct when d divides the canonical representative of the input. Instead, we express the operation as:

$$\left\lfloor \frac{x}{d} \right\rceil = \left\lfloor \frac{x + \lfloor \frac{d}{2} \rfloor}{d} \right\rfloor = d^{-1}\left(x + \left\lfloor \frac{d}{2} \right\rfloor - \mathrm{Mod}_d\left(x + \left\lfloor \frac{d}{2} \right\rfloor\right)\right). \quad (37)$$

Iliashenko et al. [23] have previously shown that the (univariate) polynomial for the Mod_d function is convenient for certain choices of divisor d. Specifically, when $p = -1 \pmod d$, the polynomial is sparse; it contains only non-zero coefficients for odd monomials (reminiscent of the polynomial used for computing comparisons). We use our polynomial evaluation techniques from Section 6 to perform this computation. A suitable divisor for our choice of p is $d = 512$.[3]

To evaluate the efficiency of our approach, we generate random responses $R_1, \ldots, R_{512}$ to the survey. Note that the actual responses do not influence the run time of the circuit (as these are encrypted). The survey might ask the number of recovery days, which may be some number in the range $[0, 10]$. Given that the mean $\bar{R}$ may be made public, we first compute the mean, and then use it to estimate the variance. We present the results in Table 5, in which we compare our results to the result obtained by evaluating the modular reduction using the ideas from IZ21 applied to the modular reduction polynomial instead of the comparison polynomial.

9.2 Logistic regression inference

To perform logistic regression in $\mathbb{F}_p$, we map the continuous inputs to a subset of discrete elements. We normalize the inputs to $[-1, 1]$ and map them (with rounding) to the discrete range $[-10, 10] \in \mathbb{F}_{6143}$. The weights and intercept must also be discretized. For this reason, we used the Gurobi solver [19] for training, optimizing a mixed integer non-linear program representing logistic regression on 100 training samples, approximating the loss function. We constrain the weights to the range $[-100, 100] \in \mathbb{F}_{6143}$, ensuring that the operations behave well modulo p. This model takes seconds to train and achieves 85% accuracy on the remaining 469 samples. Inference requires determining if the dot product of the features with the weights exceeds the intercept, for which we use a comparison, allowing us to perform a similar experiment as above, comparing against IZ21, as shown in Table 5.

[3] We note that it is also possible to approximate other divisors by compensating the numerator with an additional factor.

10 Conclusion

In this work, we introduced the concept of depth-aware arithmetization, in which we generate arithmetic circuits for high-level operations while considering the trade-off between multiplicative depth and multiplicative cost. We proposed methods for the depth-aware arithmetization of exponentiations, polynomial evaluation, and AND/OR operations. In turn, these primitives allow one to perform equality checks, comparisons, and perform operations such as veto voting. They may also be composed into larger circuits.

Our methods have limitations. For example, they can take minutes to arithmetize circuits with only a handful of comparisons. Moreover, they are not necessarily optimal: we only provide optimal methods for exponentiation circuits.

There is still room for future work. One may look for:

- Faster methods for generating optimal addition chains with depth constraints and/or precomputed values.
- An optimal method for polynomial evaluation, although this may be as hard as solving a system of multivariate polynomials.
- Other polynomial evaluation methods, e.g. mixing or generalizing the methods that we use in this work.
- An optimal method for AND/OR operations, or a proof that our current approach is optimal.
- Efficient ways of composing arithmetized primitives.
- Methods for arithmetizing multiple polynomial evaluations at once, reusing the precomputed powers across evaluations.
- Experimenting with different plaintext moduli p, which allows for a trade-off between the number of slots and the size of the generated circuits.

Our work paves the way to make several secure computation tasks more efficient and more user-friendly. For example, these algorithms can be used to automatically generate efficient arithmetic circuits for high-level circuits in the context of homomorphic encryption, where this is currently inefficient, or left as an exercise for the protocol designer. These techniques can also be used in the context of other secure computation techniques, such as arithmetic garbling.

11 Ethics considerations

Our work considers the protection of private data using established cryptographic primitives, so it extends (rather than reduces) the capabilities of secure computation. In fact, the techniques in this paper are general enough to compute any circuit securely. At first sight, this may seem to imply that there are no ethics considerations since we do not decrease confidentiality. However, we argue that one needs to place our work in a real-world context to analyze its ethical considerations.

The stakeholders of our work may be classified as researchers and developers who create tools for homomorphic encryption, organizations that use our techniques to deploy secure computation tasks, and end users whose personal data may be computed on. We briefly discuss ethical concerns towards each group of stakeholders.

One aspect that affects all stakeholders is the correctness of our work, because if the circuits generated using the techniques in this paper are not correct, they may inadvertently leak auxiliary information about the inputs. This may lead to privacy leakage in the real world, system outages, or even security incidents. For example, if the circuit is used to judge whether a financial transaction is fraudulent, an incorrect result may lead to benign transactions to be halted whereas malicious transactions may be missed. Moreover, the output may reveal personal information about the sender or receiver. To ensure correctness, we described the mathematical foundations of our circuit generation algorithms in full detail and we ran tests to ensure the circuits generated by our implementation are indeed computing what they are meant to.

For researchers and developers who create homomorphic encryption tools, it is important that our techniques are described in detail, that their limitations are clear, and that they are reproducible. We took great care in writing this paper and in listing limitations in future work in full detail in the conclusion. For reproducibility, we also refer the reader to the open science section below.

Most importantly, we ask that organizations take care when deploying these techniques to perform secure computation in the real world, considering whether what is being computed does not lead to loss of privacy or other negative consequences to end users. For example, a machine learning model trained using homomorphic encryption on private data may still reveal private information about the input data, even though all computations were performed ‘securely’. In short: confidentiality does not equal privacy. While our work helps to securely compute high-level circuits, one must take responsibility to analyze the real-world consquences of what is being computed.

12 Open science

We pledge to open source our implementation to ease reproducing our work and improving on it in the future. All our experiments can be reproduced using this open source repository.[4] It will feature an MIT license. We note that we did not use any data sets, so our artifacts only include code. The list of artifacts is as follows:

- Code for generating the circuits: we will open source all our implementations of the arithmetization algorithms described in this paper.
- Code for generating circuits pertaining to previous work: we also open source our implementations of arithmetic circuits for comparisons as generated by the T2 compiler and as proposed by Iliashenko and Zucca.
- Code for running experiments and plotting the graphs.
- Code for generating the content of the tables, but not the latex code itself as this was written by hand.

Our implementation is written entirely in Python and its dependencies are already available open source, so one can run our experiments without the use of proprietary or paid software. One exception is the Rust code we used to execute the cardio circuits using TFHE. This code will also be available under the same license.

Acknowledgments

The authors would like to thank the reviewers and, in particular, the shepherd, for their careful reviews and useful suggestions. This work is partly supported by the European Union’s Horizon Europe research and innovation program under grant agreement No. 101094901, the Septon and 101168490, the Recitals Projects.

References

[1] Muhammad Abbas and Oscar Gustafsson. Integer linear programming modeling of addition sequences with additional constraints for evaluation of power terms. *CoRR*, abs/2306.15002, 2023.

[2] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. How to garble arithmetic circuits. In Rafail Ostrovsky, editor, *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, pages 120–129. IEEE Computer Society, 2011.

[4] You can find this repository at https://doi.org/10.5281/zenodo.15613817

[3] David W. Archer, José Manuel Calderón Trilla, Jason Dagit, Alex J. Malozemoff, Yuriy Polyakov, Kurt Rohloff, and Gerard W. Ryan. RAMPARTS: A programmer-friendly system for building homomorphic encryption applications. In Michael Brenner, Tancrède Lepoint, and Kurt Rohloff, editors, *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography, WAHC@CCS 2019, London, UK, November 11-15, 2019*, pages 57–68. ACM, 2019.

[4] Pascal Aubry, Sergiu Carpov, and Renaud Sirdey. Faster homomorphic encryption is not enough: Improved heuristic for multiplicative depth minimization of boolean circuits. In Stanislaw Jarecki, editor, *Topics in Cryptology - CT-RSA 2020 - The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24-28, 2020, Proceedings*, volume 12006 of *Lecture Notes in Computer Science*, pages 345–363. Springer, 2020.

[5] Gilles Audemard and Laurent Simon. Glucose 4.2.1 SAT Solver. https://www.labri.fr/perso/lsimon/glucose/, 2023. Accessed: 2025-06-12.

[6] François Bergeron, Jean Berstel, Srecko Brlek, and Christine Duboc. Addition chains using continued fractions. *J. Algorithms*, 10(3):403–412, 1989.

[7] Charlotte Bonte and Ilia Iliashenko. Homomorphic string search with constant multiplicative depth. In Yinqian Zhang and Radu Sion, editors, *CCSW'20, Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop, Virtual Event, USA, November 9, 2020*, pages 105–117. ACM, 2020.

[8] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886. Springer, 2012.

[9] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. *Electron. Colloquium Comput. Complex.*, TR11-111, 2011.

[10] Sergiu Carpov, Pascal Aubry, and Renaud Sirdey. A multi-start heuristic for multiplicative depth minimization of boolean circuits. In Ljiljana Brankovic, Joe Ryan, and William F. Smyth, editors, *Combinatorial Algorithms - 28th International Workshop, IWOCA 2017, Newcastle, NSW, Australia, July 17-21, 2017, Revised Selected Papers*, volume 10765 of *Lecture Notes in Computer Science*, pages 275–286. Springer, 2017.

[11] Sergiu Carpov, Thanh-Hai Nguyen, Renaud Sirdey, Gianpiero Costantino, and Fabio Martinelli. Practical privacy-preserving medical diagnosis using homomorphic encryption. In *9th IEEE International Conference on Cloud Computing, CLOUD 2016, San Francisco, CA, USA, June 27 - July 2, 2016*, pages 593–599. IEEE Computer Society, 2016.

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

[13] Sangeeta Chowdhary, Wei Dai, Kim Laine, and Olli Saarikivi. EVA improved: Compiler and extension library for CKKS. In *WAHC '21: Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Virtual Event, Korea, 15 November 2021*, pages 43–55. WAHC@ACM, 2021.

[14] Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T. Lee, and Brandon Reagen. Porcupine: a synthesizing compiler for vectorized homomorphic encryption. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 375–389. ACM, 2021.

[15] Jean Paul Degabriele, Jan Gilcher, Jérôme Govinden, and Kenneth Paterson. SoK: Efficient design and implementation of polynomial hash functions over prime fields. In *45th IEEE Symposium on Security and Privacy (SP 2024)*, 2024.

[16] Peter J. Downey, Benton L. Leong, and Ravi Sethi. Computing sequences with addition chains. *SIAM J. Comput.*, 10(3):638–646, 1981.

[17] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, page 144, 2012.

[18] Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. SoK: New Insights into Fully Homomorphic Encryption Libraries via Standardized Benchmarks. *Proceedings on Privacy Enhancing Technologies*, 2023(3):154–172, July 2023.

[19] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*, 2024. https://www.gurobi.com.

[20] Shai Halevi and Victor Shoup. Algorithms in HElib. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 554–571. Springer, 2014.

[21] Darrel Hankerson, Scott Vanstone, and Alfred Menezes. *Guide to Elliptic Curve Cryptography*. Springer Professional Computing. Springer New York, NY, 1 edition, 2004. Springer Science+Business Media New York 2004.

[22] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018.

[23] Ilia Iliashenko, Christophe Nègre, and Vincent Zucca. Integer functions suitable for homomorphic encryption over finite fields. In *WAHC '21: Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Virtual Event, Korea, 15 November 2021*, pages 1–10. WAHC@ACM, 2021.

[24] Ilia Iliashenko and Vincent Zucca. Faster homomorphic comparison operations for BGV and BFV. *Proc. Priv. Enhancing Technol.*, 2021(3):246–264, 2021.

[25] DongKwon Lee, Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi. Optimizing homomorphic evaluation circuits by program synthesis and term rewriting. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 503–518. ACM, 2020.

[26] Michael Ben McLoughlin. addchain: Cryptographic Addition Chain Generation in Go, 10 2021.

[27] Johannes Mono, Chiara Marcolla, Georg Land, Tim Güneysu, and Najwa Aaraj. Finding and evaluating parameters for BGV. In Nadia El Mrabet, Luca De Feo, and Sylvain Duquesne, editors, *Progress in Cryptology - AFRICACRYPT 2023 - 14th International Conference on Cryptology in Africa, Sousse, Tunisia, July 19-21, 2023, Proceedings*, volume 14064 of *Lecture Notes in Computer Science*, pages 370–394. Springer, 2023.

[28] António Morgado, Carmine Dodaro, and João Marques-Silva. Core-guided maxsat with soft cardinality constraints. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 564–573. Springer, 2014.

[29] Mike Paterson and Larry J. Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM J. Comput.*, 2(1):60–66, 1973.

[30] Arnold Schönhage. A lower bound for the length of addition chains. *Theor. Comput. Sci.*, 1(1):1–12, 1975.

[31] Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, volume 3709 of *Lecture Notes in Computer Science*, pages 827–831. Springer, 2005.

[32] Edward G. Thurber and Neill Michael Clift. Addition chains, vector chains, and efficient computation. *Discret. Math.*, 344(2):112200, 2021.

[33] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. SoK: Fully homomorphic encryption compilers. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1092–1108. IEEE, 2021.

[34] William H. Wolberg, Olvi L. Mangasarian, and W. Nick Street. Breast cancer wisconsin (diagnostic) data set. https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic), 1993. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5DW2B.

[35] Mingfei Yu and Giovanni De Micheli. Expediting homomorphic computation via multiplicative complexity-aware multiplicative depth minimization. Cryptology ePrint Archive, Paper 2024/1015, 2024. https://eprint.iacr.org/2024/1015.

A Depth-aware arithmetization of ANDs

Algorithm 3 Finds `AND` circuit with fullness $\leq f$ & cost $< c$.

1: **procedure** $\text{AND}(X_1, \ldots, X_k, f, c, E, p)$
2: Ensure that $\text{fuln}(X_1) \geq \cdots \geq \text{fuln}(X_k)$
3: **if** $k = 1$ ▷ *Base cases*
4: **if** $\text{fuln}(X_1) \leq f$ and $c > 0$
5: **return** X_1
6: **return** $\perp$
7: **if** $f < 1$ or $c \leq 0$
8: **return** $\perp$
9: **if** $\text{cost}^*(X_1 \wedge \cdots \wedge X_k) \geq c$
10: **return** $\perp$
11: $C_{\text{out}} = \perp$
12: **for** $C_{\text{exp}} \in E$ **do**
13: $C = X_1 \times \cdots \times X_k$ ▷ *C_1 circuit*
14: **if** $\sum_{i=1}^{k} \text{fuln}(X_i) \leq f$ and $\text{cost}(C) < c$
15: $C_{\text{out}} \leftarrow C$, $c \leftarrow \text{cost}(C)$
16: $f_{\text{exp}} \leftarrow 2^{\lceil \log_2 f \rceil - \text{depth}(C_{\text{exp}})}$ ▷ *Max fuln for C_2*
17: **if** $k < p$
18: $C \leftarrow \overline{C_{\text{exp}}(\overline{X_1} + \cdots + \overline{X_k})}$ ▷ *C_2 circuit*
19: **if** $\bigwedge_{i=1}^{k} \text{fuln}(X_i) \leq f_{\text{exp}}$ and $\text{cost}(C) < c$
20: $C_{\text{out}} \leftarrow C, c \leftarrow \text{cost}(C)$
21: **continue**
22: **if** $\bigwedge_{i=1}^{k} \text{fuln}(X_i) \leq f_{\text{exp}}$ ▷ *C_2 works for all X_i*
23: **if** $\text{cost}(C_{\text{exp}}) \geq c$
24: **continue**
25: cache $\leftarrow \{\}$
26: **for** $i = 1, \ldots, k-1$ **do**
27: $C' \leftarrow \overline{C_{\text{exp}}(\overline{X_i} + \cdots + \overline{X_{i+p-2}})}$
28: $X \leftarrow C', X_1, \ldots, X_{i-1}, X_{i+p-1}, \ldots, X_k$
29: **if** $\{\text{fuln}(x) \mid x \in X\} \in$ cache
30: **continue**
31: Add $\{\text{fuln}(x) \mid x \in X\}$ to cache
32: $C \leftarrow \text{AND}(X, f, c - \text{cost}(C_{\text{exp}}), E, p)$
33: **if** $C \neq \perp$
34: $C_{\text{out}} \leftarrow C, c \leftarrow \text{cost}(C)$
35: **else** ▷ *We can isolate X_i that must use C_1*
36: Find t s.t. $\text{fuln}(X_t) > f_{\text{exp}}$, $\text{fuln}(X_{t+1}) \leq f_{\text{exp}}$
37: **if** $t = 0$ or $t \geq c$
38: **continue**
39: $f_{\text{new}} \leftarrow f - \sum_{i=1}^{t} \text{fuln}(X_i)$
40: $C' \leftarrow \text{AND}(X_{t+1}, \ldots, X_k, f_{\text{new}}, c - t, E, p)$
41: **if** $C' \neq \perp$
42: $C \leftarrow C' \times X_1 \times \cdots \times X_t$
43: $C_{\text{out}} \leftarrow C, c \leftarrow \text{cost}(C)$
44: **return** C_{out}