

Depth-Aware Arithmetization of Common Primitives in Prime Fields

Jelle Vos

*Cyber Security Group
Delft University of Technology
Delft, Netherlands
J.V.Vos@tudelft.nl*

Mauro Conti

*SPRITZ
University of Padua
Padua, Italy
conti@math.unipd.it*

Zekeriya Erkin

*Cyber Security Group
Delft University of Technology
Delft, Netherlands
Z.Erkin@tudelft.nl*

Abstract—A common misconception is that the computational abilities of circuits composed of additions and multiplications are restricted to simple formulas only. Such arithmetic circuits over finite fields are actually capable of computing any function, including equality checks, comparisons, and other highly non-linear operations. While all those functions are computable, the challenge lies in computing them efficiently. We refer to this search problem as arithmetization. Arithmetization is a key problem in secure computation, as techniques like homomorphic encryption and secret sharing compute arithmetic circuits rather than the high-level programs that programmers are used to. 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 to compute than additions. However, the multiplicative depth of a circuit arguably plays an even more important role in deciding the computational cost: For homomorphic encryption, it strongly affects the choice of cryptographic parameters and the number of bootstrapping operations required, which are orders of magnitude more expensive to compute than multiplications. In fact, if we can limit the multiplicative depth of a circuit such that we do not need to perform any bootstrapping, we can omit the large bootstrapping keys required to perform them all together. 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 take into account that squaring can be cheaper than arbitrary multiplications, and we study how to compose them.

1. Introduction

Within cryptography, the area of secure computation has come to know many different techniques, such as lattice-based homomorphic encryption and secret sharing, each with different trade-offs. Secret sharing-based secure computation typically suffers from a high degree of interaction, incurring network latency. On the other hand, somewhat and fully-homomorphic encryption allows for computations to be

performed locally, but the computational cost is significantly higher. Other techniques, such as garbled circuits, provide even more trade-offs. The main problem that we want to solve in secure computation is to find the most efficient protocols that compute a given function. For secret sharing, this typically comes down to minimizing the interactivity, and for homomorphic encryption, this comes down to minimizing the computational cost.

As part of this optimization problem, it is often not straightforward how to compute a given function using secure computation techniques because they require a problem to be expressed in terms of additions and multiplications in some algebraic structure. We refer to this sub-problem as *arithmetization*. So far, most works find an arithmetization of some function that minimizes the number of multiplications, known as the multiplicative size. This objective stands to reason because multiplications are significantly more expensive to compute than additions in all of the techniques mentioned above. However, these techniques do not consider the multiplicative depth of a circuit, which is the largest number of multiplications in any path through the circuit.

In this work, we propose a new type of arithmetization called depth-aware arithmetization, which considers both a circuit’s multiplicative size and multiplicative depth. In doing so, depth-aware arithmetization allows one to significantly reduce the number of interactions in secret sharing and the size of the parameters needed in lattice-based homomorphic encryption schemes, 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. 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 $0.5 \leq \sigma \leq 1.0$. In Table 1, we explain how the multiplicative depth and multiplicative cost indeed capture

important efficiency aspects in different secure computation techniques. We note that in some arithmetic garbling schemes, the multiplicative depth also plays an important role in the efficiency of a circuit [1].

TABLE 1. THE ROLE OF THE MULTIPLICATIVE DEPTH IN SECURE COMPUTATION, AND THE WAY IN WHICH SQUARING IS CHEAPER.

Technique	Multiplicative depth	Squaring
Lattice-based HE	Noise growth	Cheaper tensor products
Secret sharing	Number of interactions	Smaller Beaver tuples
Arithmetic garbling	-	Smaller gates

While our work applies to multiple secure computation techniques, it directly applies to lattice-based homomorphic encryption schemes. The reason is that, for these schemes, there is a highly non-trivial relation between a circuit’s depth and cost in determining the overall efficiency. To accommodate this non-trivial relationship, the output of depth-aware arithmetization is not one circuit but a collection of circuits that optimally trade off depth and cost. Moreover, many use cases rely solely on local computations, which suits our restriction that no intermediate values are revealed.

The most popular lattice-based homomorphic encryption schemes rely on the hardness of (variants of) the learning with errors problem. The hardness is determined in part by a small amount of noise that is added to samples. In homomorphic encryption schemes derived from this problem, ciphertexts contain noise that grows linearly during homomorphic additions and exponentially during multiplications. For successful decryption, this noise must stay under some bound. Parameters 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. In other words, the size of those parameters are largely determined by the multiplicative depth of the evaluated circuit. At the same time, large parameters negatively impact the efficiency of each homomorphic multiplication.

Our work is not the first to reduce the depth of arithmetic circuits. Some works [2], [3], [4], [5] 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 (unlike Boolean circuits, where this is much more common).

Unlike the methods described above, our work considers depth reduction during arithmetization, which allows us to generate circuits that these methods could not. To demonstrate this, we consider the function $x_1 \vee x_2 \vee \dots \vee x_7$ and generate two depth-aware arithmetizations in \mathbb{F}_5 . Figure 1 shows an arithmetization of size 4 and depth 4, whereas Figure 2 shows an arithmetization of size 5 and depth 3. The latter circuit cannot be generated from the first circuit using the methods

described above. Moreover, it is not immediately clear which of these two circuits would be faster to compute using lattice-based homomorphic encryption. We use fhegen [6] to generate parameters and execute the circuits on an 8-core M1 CPU to find that the size-4 circuit takes 41.3 ± 0.5 milliseconds, while the size-5 circuit took only 22.8 ± 1.7 milliseconds. While previous methods typically prioritize the size-4 circuit, the size-5 circuit outperforms the former because it supports a smaller ring dimension.

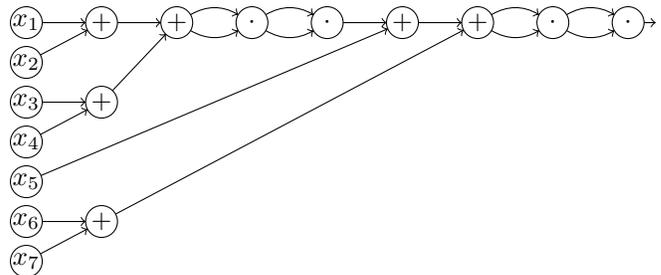


Figure 1. An arithmetization of $x_1 \vee x_2 \vee \dots \vee x_7$ in \mathbb{F}_5 with a multiplicative size of 4 and a multiplicative depth of 4.

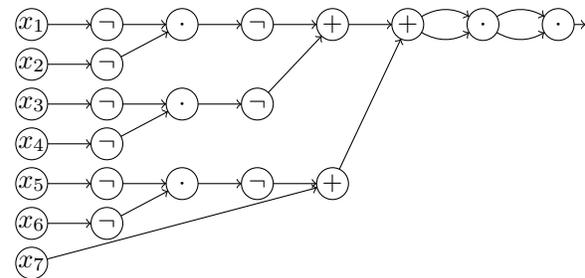


Figure 2. An arithmetization of $x_1 \vee x_2 \vee \dots \vee x_7$ in \mathbb{F}_5 with a multiplicative size of 5 and a multiplicative depth of 3.

In the rest of the work, we will compare circuits rather than run times. Since we generate the entire Pareto front, the user can define any run time estimation script to select the desired circuit. Note that we will be looking at p that are not necessarily NTT-friendly, but our algorithms are generic. One can still run the same circuit many times in parallel on different data.

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.

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 a parameter in two existing techniques to generate circuits for univariate polynomial evaluation. We use these circuits for arithmetizing 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, we compose these primitives into larger circuits. Finally, we conclude in Section 9.

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:

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

Definition 2.2 (Multiplicative cost). The multiplicative cost of a circuit is a weighted sum of the cost of all multiplications in a circuit. One might define the cost of squaring operations to be a factor σ of that of arbitrary multiplications, yielding:

$$\text{cost}(C_1, \dots, C_n) = \sigma|[X \times X \in C_1] \cup \dots \cup [X \times X \in C_n]| + |[X \times Y \in C_1 \mid X \neq Y] \cup \dots \cup [X \times Y \in C_n \mid X \neq Y]| .$$

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 [7], [8]. 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 [2], [3], [4], [5] that show how to reduce the multiplicative depth of a circuit beyond multiplication trees by distributing products.

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) [9], window methods [9], and ones based on heuristics [10] 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 .

Abbas & Gustafsson [11] 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. While the formulation allows one to find optimal arithmetic circuits, it is slow in practice. In Section 5, we translate this MILP to a MaxSAT formulation that is significantly faster to solve. We also provide several optimizations.

3.3. Arithmetization of Polynomial Evaluation

The arithmetization of polynomial evaluation has been studied in many previous works, but the work by Patterson & Stockmeyer [12] 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 more detail in Section 6, and we show how to tweak them to obtain a depth-size trade-off.

Iliashenko et al. [13], [14] 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 it must hold that p is a Mersenne prime. In our work, we want to allow any 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 [15] 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 [13] 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 comparison circuits.

3.4. ORs & ANDs

ANDs are typically arithmetized using a product $x_1 \wedge x_2 \wedge \dots \wedge x_k = x_1 \times x_2 \times \dots \times x_k$, which leads to a circuit of depth $\lceil \log_2 k \rceil$. The OR operation follows using DeMorgan's law, which does not introduce further non-scalar multiplications. An alternative arithmetization [16] uses a summation and an `IsNonZero` check to compute such operations on many inputs. Figures 1 and 2 show that by combining both arithmetizations in \mathbb{F}_5 , one can find circuits on the depth-size front. Here, `IsNonZero(z)` is efficient to compute because z^4 only requires squaring twice. As a result, these arithmetic circuits require fewer non-scalar multiplications than the equivalent Boolean circuit, which would have size 7 and depth 3.

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 3. The left subfigure shows a depth-3 product, whereas the right subfigure shows a rearranged product



Figure 3. 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.

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 [7], [8].

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 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:

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

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.

We can adapt the equation before to derive a closed-form expression of the depth of the circuit resulting from depth-aware arithmetization of a product. Since we do not modify

Algorithm 1 Depth-aware product of distinct multiplicands

```
1: procedure PRODUCT( $C_1, \dots, C_n$ )
2:   Let  $Q$  be an empty priority queue
3:   for  $n = 1, \dots, n$  do
4:     Insert  $C_i$  into  $Q$  with priority  $\text{depth}(C_i)$ 
5:   while  $|Q| \geq 2$  do
6:     Pop  $X$  and  $Y$  from  $Q$   $\triangleright$  Returns lowest depth
7:      $d \leftarrow 1 + \max(\text{depth}(X), \text{depth}(Y))$ 
8:     Insert  $X \times Y$  into  $Q$  with priority  $d$ 
9:   Pop  $C$  from  $Q$   $\triangleright$  There is only one  $C$  in  $Q$ 
10:  return  $C$ 
```

the subcircuits, we model them as having maximal breadth for their depth, yielding:

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

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 [17], which we use to solve a reformulation of the mixed-integer linear programming (MILP) formulation by Abbas & Gustafsson [11]. Such a solver attempts to find a variable assignment that satisfies a set of hard clauses and as many soft clauses as possible (possibly dropping some). We assign a weight to some of these 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 [11], [18]. Much theoretical work has been done [19] and many heuristics have been proposed [10], [18]. Variants of the problem have also been studied, such as addition sequences [20], 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 [11] 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:

- 1) If $y_{i,j} = 1$, then $x_i = 1$, $x_j = 1$, and $x_{i+j} = 1$.
- 2) Cutting away: $x_{\lceil \frac{k}{2} \rceil} \vee \dots \vee x_{k-1} = 1$.

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 allows taking into account that squaring is cheaper operation. 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 make solving it faster in practice. We add three kinds of cuts:

- Bounds from original [11]
- The bounds derived by Thurber & Clift [21]. Given an upper bound on the cost of the chain, we can use these to find lower bounds for the i th element in the chain. For our MaxSAT formulation, let $T_t(c_{\max})$ return a set of pairs (l, u) such that the i th 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 [22] to work well in practice.¹

1. Our implementation supports the choices offered by PySAT [23].

We can add these cuts using the following hard clauses:

$$\left(\bigvee_{m=\lceil \frac{k}{2} \rceil}^k x_m \right), \quad \forall k \in [2, t]$$

$$\left(\bigvee_{m=l}^u x_m \right), \quad \forall (l, u) \in T_t(c)$$

$$\left(\sum_{i=2}^t x_i \geq s_{\min} \right), \quad \text{encoded with [22]}$$

To determine s_{\min} we combine three lower bounds reported by Schönage [19], where $\nu(t)$ is the Hamming weight of t :

$$s_{\min}(t) \geq \lceil \log_2(t) \rceil, \quad (3)$$

$$s_{\min}(t) \geq \lceil \log_2(t) + \log_2(\nu(t)) - 2.13 \rceil, \quad (4)$$

$$s_{\min}(t) \geq \lceil \log_2(t) + \log_3(\nu(t)) - 1 \rceil. \quad (5)$$

Finally, in a finite field, 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 solve this problem by generating an exponentiation circuit for several $t' = t + i\phi(p)$, with $i = 1, 2, \dots$, and selecting the most efficient.

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:

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

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 $c_{\text{max}}(t')$, making the formulation faster to solve and allowing us to skip targets t' for which $\sigma s_{\min}(t) \geq c_{\text{max}}(t')$.

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 4. 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:

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

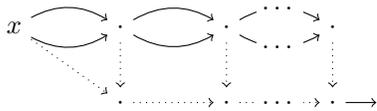


Figure 4. Square & multiply method for computing x^t .

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

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:

$$(d_{k,m+1} \vee \neg d_{i,m} \vee \neg y_{i,j}), \quad \forall (i,j) \in P, \forall m \in [0, d_{\text{max}}]$$

$$(d_{k,m+1} \vee \neg d_{j,m} \vee \neg y_{i,j}), \quad \forall (i,j) \in P, \forall m \in [0, d_{\text{max}}]$$

$$(\neg d_{k,d_{\text{max}}+1}), \quad \forall k \in [2, t]$$

$$(d_{1,0}).$$

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 $d_{\text{max}} + 1$ 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 [11], which uses integers to denote the depth (as they use a MILP solver).

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 $\text{ADDCHAIN}(t, d_{\text{max}}, c_{\text{max}}, \sigma, s_{\min})$, 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  $\text{cost}(C) = \text{cost}^*(C)$ 
3:    $d \leftarrow \lceil \log_2 t \rceil$ 
4:    $c \leftarrow \sigma \lceil \log_2 t \rceil + \nu(t) - 1$ 
5:   while  $c < \text{cost}^*(C)$  and  $d < \text{depth}(C)$  do
6:     Compute  $s_{\min}$  using (3), (4), and (5)
7:      $C' \leftarrow \text{ADDCHAIN}(t, d, c, \sigma, s_{\min})$ 
8:     if  $C' \neq \perp$ 
9:       yield  $C'$ 
10:     $c \leftarrow \text{cost}(C')$ 
11:   $d \leftarrow d + 1$ 

```

5.4. Case Study: Equality Checks

As explained by Iliashenko & Zucca [13], 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 5, 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 5, 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 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 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 [12] provide an asymptotic bound:

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

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:

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

This is an equality because we cannot evaluate x^d with fewer multiplications. In Paterson & Stockmeyer’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 algorithms proposed by Paterson & Stockmeyer [12], 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 [24].

The algorithm, parameterized by an integer $1 \leq k \leq d$, starts by precomputing the monomials X^2, X^3, \dots, 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 $X \times X^4 = X^5, X^2 \times X^4 = X^6, \dots, X^4 \times X^4 = X^8$, etc. Given these precomputed powers, the key idea behind this algorithm is the following identity:

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

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 [12] 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:

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

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

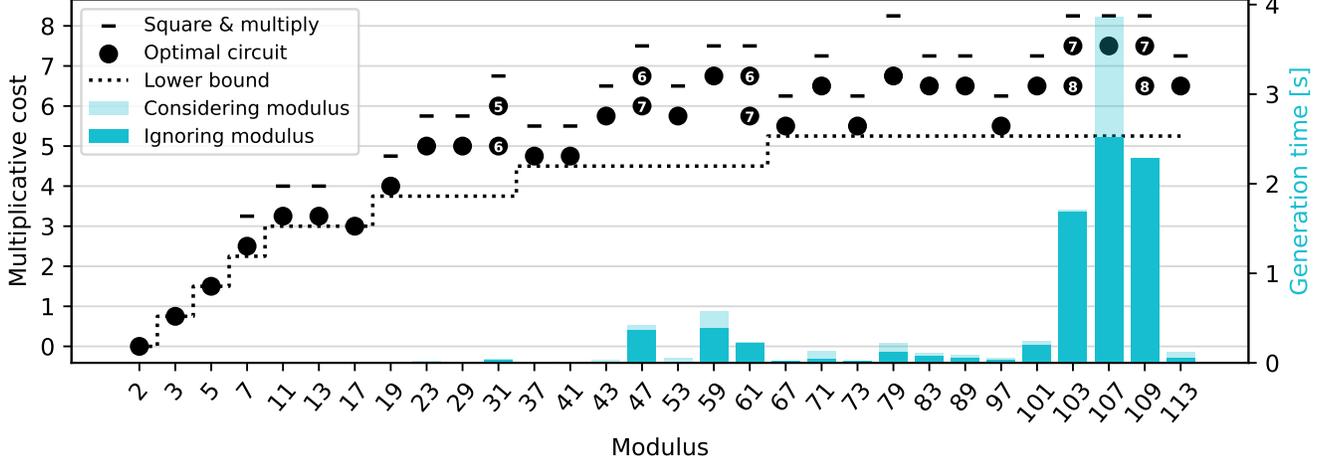


Figure 5. Equality circuits generated using square & multiply and our MaxSAT formulation, where $\sigma = 0.75$. Square & multiply is only optimal when p is of the form $2^k + 1$. When we find a depth-cost trade-off, we denote the depth in the markers. The run time of our algorithm is hard to predict, 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.

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, \dots, 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 $i = (2^n - 1)k \bmod \phi(p) \leq d$, where $\phi()$ is the totient function, 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, \dots, t_n with depths d_1, \dots, 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:

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

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:

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

As a result:

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

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

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

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

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

If it takes $k - 1$ multiplications to compute X^2, \dots, X^k and $n - 1$ squarings to compute $X^{2^k}, X^{4^k}, \dots, X^{2^{n-1}k}$, 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 $i = 2, \dots, k$ are $\lceil \log_2 i \rceil$, and the depths of $X^{2^i k}$ for $i = 1, \dots, n - 1$ 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.

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 . Our depth-aware arithmetization method for polynomial evaluation is to simply try all three methods on all values $1 \leq k \leq d$. 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 6 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 $k = 9$.

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 [13] 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 [15], in which the comparison is implemented as a number of equality checks:

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

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 [13], 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

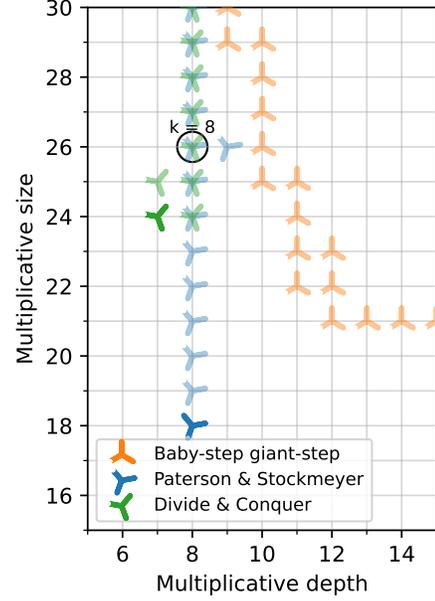


Figure 6. Polynomial evaluation circuits for computing $x \pmod{7}$ with $p = 127$. This is a degree 126 polynomial, so the parameter for minimizing the multiplicative size determined by Paterson & Stockmeyer is $k = \sqrt{\frac{126}{2}} \approx 8$. The optimum occurs when $k = 8$, and the actual optimum occurs when using the BSGS method. By varying k we can generate three circuits that trade off multiplicative size and depth.

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. In some cases, this allows the method to find circuits on the front.

TABLE 2. COMPARISON CIRCUITS FOR DIFFERENT MODULI p ; $\sigma = 1.0$.

Method	$p = 29$		$p = 43$		$p = 61$		$p = 101$		$p = 131$	
	Depth	Size	Depth	Size	Depth	Size	Depth	Size	Depth	Size
T2 compiler	5	84	6	147	7	210	7	400	8	520
IZ21	7	12	7	13	9	14	8	16	9	21
IZ21 + CSE	7	12	7	12	9	13	8	16	9	19
Our work	6	11	7	12	7	15	8	16	8	20
	7	10			8	14				

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:

$$X_1 \wedge \cdots \wedge X_k = X_1 \times \cdots \times X_k . \quad (22)$$

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

$$\text{cost}(C_1) = k - 1 + \text{cost}(X_1, \dots, X_k) , \quad (23)$$

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

OR operations are sometimes arithmetized as follows:

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

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:

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

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

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:

$$X_1 \wedge \cdots \wedge X_k = \overline{\overline{X_1} \vee \cdots \vee \overline{X_k}} . \quad (28)$$

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 $\text{cost}(C_2(C_{\text{exp}})) < \text{cost}(C_1) \iff \text{cost}(C_{\text{exp}}) < k - 1$. 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:

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

When $k \geq p$, 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 $\text{cost}(C_{\text{exp}}) \leq p - 1$, 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 $\text{cost}^*(C_{\text{exp}}) \leq p - 1$ always holds.

Using the strategy described above, we get that:

$$N(k) = c + N(k - (p - 1) + 1) , \quad (30)$$

$$= 2c + N(k - p - (p - 1) + 1) , \quad (31)$$

$$\vdots \quad (32)$$

$$= rc + N(k - r(p - 1) + r) , \quad (33)$$

$$= rc + N(k + r(2 - p)) . \quad (34)$$

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

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

Notice that increasing c always increases the total multiplicative cost, apart from the case where $k < p$ and $c \geq k - 1$, 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:

$$\text{fuln}(X + Y) = 2^{\max(\text{depth}(X), \text{depth}(Y))}$$

$$\text{fuln}(X \times Y) = \text{fuln}(X) + \text{fuln}(Y)$$

$$\text{fuln}(v) = 1$$

Notice that:

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

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, in which $\text{cost}(\cdot)(C)$ 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 $d = \lceil \log_2 \text{fuln}(X_1) \rceil$, 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 .

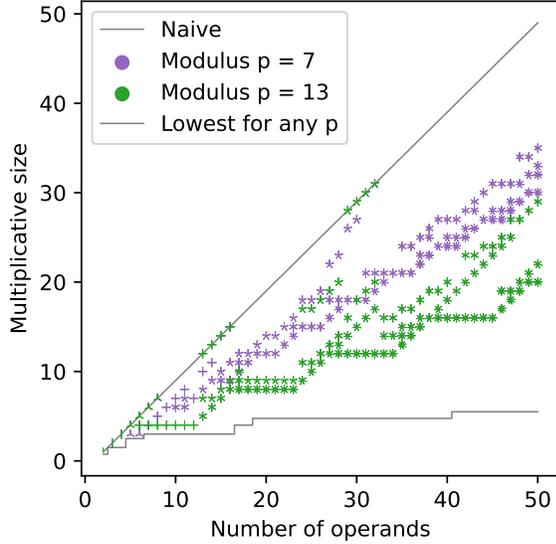


Figure 7. Circuits computing an OR operation with $\sigma = 1.0$, for a growing number of operands. The number of ticks on a marker indicates the depth of the circuit. Depending on the depth that one wants to achieve and the number of operands, it is better to choose $p = 7$ or $p = 13$.

We can keep going in the fashion described above, incrementing d , to generate the entire depth-cost front. Since it is easy to compute $\text{cost}^*(C_3(C_{\text{exp}}))$, 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 7, 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.

Algorithm 3 Finds an AND circuit with fullness $\leq f$ and minimal cost $< c$, returning \perp if it cannot be found.

```

1: procedure AND( $X_1, \dots, X_k, f, c, E, p$ )
2:   Ensure that  $\text{fuln}(X_1) \geq \dots \geq \text{fuln}(X_k)$ 

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

```

8. Depth-Aware Composition

In the previous sections, we put forward methods for the depth-aware arithmetization of several common primitives, but many interesting circuits emerge as the composition of these primitives. In this section, we show how to perform depth-aware arithmetization for 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.

While the method described above offers a generic solution of dealing with composition, it can be highly 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 still satisfy the depth limit.

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, and leave it to future work.

To highlight the effectiveness of our methods, we apply them to a practical example that composes all of the primitives described in this work. Specifically, we evaluate them on the *cardio* circuit as proposed by Carpov et al. [25] and used as a benchmark in other works [26]. The circuit computes a number of 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 , and how long these fronts took to generate. 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 427 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.

TABLE 3. FRONTS GENERATED BY OUR METHODS FOR THE *cardio* AND *cardio-elevated* CIRCUITS, DISPLAYING THE COST-DEPTH TRADE-OFF.

Gen. (s)	Cardio risk assessment			Depth	Cardio elevated risk		
	$\sigma = 1.0$	$\sigma = 0.75$	$\sigma = 0.5$		$\sigma = 1.0$	$\sigma = 0.75$	$\sigma = 0.5$
	265	268	264		248	272	272
12	427.0	394.0	361.0	15	436.0	402.0	368.0
13		386.0	345.0	16	.	395.0	354.0
14		382.0	337.0	17	.	391.0	346.0
			
				20	435.0	.	.
			
				22		388.0	341.0

9. 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.

References

- [1] B. Applebaum, Y. Ishai, and E. Kushilevitz, “How to garble arithmetic circuits,” in *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, R. Ostrovsky, Ed. IEEE Computer Society, 2011, pp. 120–129. [Online]. Available: <https://doi.org/10.1109/FOCS.2011.40>
- [2] S. Carpov, P. Aubry, and R. Sirdey, “A multi-start heuristic for multiplicative depth minimization of boolean circuits,” in *Combinatorial Algorithms - 28th International Workshop, IWOCA 2017, Newcastle, NSW, Australia, July 17-21, 2017, Revised Selected Papers*, ser. Lecture Notes in Computer Science, L. Brankovic, J. Ryan, and W. F. Smyth, Eds., vol. 10765. Springer, 2017, pp. 275–286. [Online]. Available: https://doi.org/10.1007/978-3-319-78825-8_23

- [3] P. Aubry, S. Carpov, and R. Sirdey, "Faster homomorphic encryption is not enough: Improved heuristic for multiplicative depth minimization of boolean circuits," in *Topics in Cryptology - CT-RSA 2020 - The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24-28, 2020, Proceedings*, ser. Lecture Notes in Computer Science, S. Jarecki, Ed., vol. 12006. Springer, 2020, pp. 345–363. [Online]. Available: https://doi.org/10.1007/978-3-030-40186-3_15
- [4] D. Lee, W. Lee, H. Oh, and K. Yi, "Optimizing homomorphic evaluation circuits by program synthesis and term rewriting," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, A. F. Donaldson and E. Torlak, Eds. ACM, 2020, pp. 503–518. [Online]. Available: <https://doi.org/10.1145/3385412.3385996>
- [5] M. Yu and G. D. Micheli, "Expediting homomorphic computation via multiplicative complexity-aware multiplicative depth minimization," Cryptology ePrint Archive, Paper 2024/1015, 2024, <https://eprint.iacr.org/2024/1015>. [Online]. Available: <https://eprint.iacr.org/2024/1015>
- [6] J. Mono, C. Marcolla, G. Land, T. Güneysu, and N. Aaraj, "Finding and evaluating parameters for BGV," in *Progress in Cryptology - AFRICACRYPT 2023 - 14th International Conference on Cryptology in Africa, Sousse, Tunisia, July 19-21, 2023, Proceedings*, ser. Lecture Notes in Computer Science, N. E. Mrabet, L. D. Feo, and S. Duquesne, Eds., vol. 14064. Springer, 2023, pp. 370–394. [Online]. Available: https://doi.org/10.1007/978-3-031-37679-5_16
- [7] S. Chowdhary, W. Dai, K. Laine, and O. 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*. WAHC@ACM, 2021, pp. 43–55. [Online]. Available: <https://doi.org/10.1145/3474366.3486929>
- [8] D. W. Archer, J. M. C. Trilla, J. Dagit, A. J. Malozemoff, Y. Polyakov, K. Rohloff, and G. W. Ryan, "RAMPARTS: A programmer-friendly system for building homomorphic encryption applications," in *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography, WAHC@CCS 2019, London, UK, November 11-15, 2019*, M. Brenner, T. Lepoint, and K. Rohloff, Eds. ACM, 2019, pp. 57–68. [Online]. Available: <https://doi.org/10.1145/3338469.3358945>
- [9] D. Hankerson, S. Vanstone, and A. Menezes, *Guide to Elliptic Curve Cryptography*, 1st ed., ser. Springer Professional Computing. Springer New York, NY, 2004, springer Science+Business Media New York 2004.
- [10] F. Bergeron, J. Berstel, S. Brlek, and C. Duboc, "Addition chains using continued fractions," *J. Algorithms*, vol. 10, no. 3, pp. 403–412, 1989. [Online]. Available: [https://doi.org/10.1016/0196-6774\(89\)90036-9](https://doi.org/10.1016/0196-6774(89)90036-9)
- [11] M. Abbas and O. Gustafsson, "Integer linear programming modeling of addition sequences with additional constraints for evaluation of power terms," *CoRR*, vol. abs/2306.15002, 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2306.15002>
- [12] M. Paterson and L. J. Stockmeyer, "On the number of nonscalar multiplications necessary to evaluate polynomials," *SIAM J. Comput.*, vol. 2, no. 1, pp. 60–66, 1973. [Online]. Available: <https://doi.org/10.1137/0202007>
- [13] I. Iliashenko and V. Zucca, "Faster homomorphic comparison operations for BGV and BFV," *Proc. Priv. Enhancing Technol.*, vol. 2021, no. 3, pp. 246–264, 2021. [Online]. Available: <https://doi.org/10.2478/popets-2021-0046>
- [14] I. Iliashenko, C. Nègre, and V. 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*. WAHC@ACM, 2021, pp. 1–10. [Online]. Available: <https://doi.org/10.1145/3474366.3486925>
- [15] C. Gouert, D. Mouris, and N. G. Tsoutsos, "SoK: New Insights into Fully Homomorphic Encryption Libraries via Standardized Benchmarks," *Proceedings on Privacy Enhancing Technologies*, vol. 2023, no. 3, p. 154–172, Jul. 2023.
- [16] C. Bonte and I. Iliashenko, "Homomorphic string search with constant multiplicative depth," in *CCSW'20, Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop, Virtual Event, USA, November 9, 2020*, Y. Zhang and R. Sion, Eds. ACM, 2020, pp. 105–117. [Online]. Available: <https://doi.org/10.1145/3411495.3421361>
- [17] A. Morgado, C. Dodaro, and J. Marques-Silva, "Core-guided maxsat with soft cardinality constraints," in *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014, Proceedings*, ser. Lecture Notes in Computer Science, B. O'Sullivan, Ed., vol. 8656. Springer, 2014, pp. 564–573. [Online]. Available: https://doi.org/10.1007/978-3-319-10428-7_41
- [18] M. B. McLoughlin, "addchain: Cryptographic Addition Chain Generation in Go," Oct. 2021. [Online]. Available: <https://github.com/mmloughlin/addchain>
- [19] A. Schönhage, "A lower bound for the length of addition chains," *Theor. Comput. Sci.*, vol. 1, no. 1, pp. 1–12, 1975. [Online]. Available: [https://doi.org/10.1016/0304-3975\(75\)90008-0](https://doi.org/10.1016/0304-3975(75)90008-0)
- [20] P. J. Downey, B. L. Leong, and R. Sethi, "Computing sequences with addition chains," *SIAM J. Comput.*, vol. 10, no. 3, pp. 638–646, 1981. [Online]. Available: <https://doi.org/10.1137/0210047>
- [21] E. G. Thurber and N. M. Clift, "Addition chains, vector chains, and efficient computation," *Discret. Math.*, vol. 344, no. 2, p. 112200, 2021. [Online]. Available: <https://doi.org/10.1016/j.disc.2020.112200>
- [22] C. Sinz, "Towards an optimal CNF encoding of boolean cardinality constraints," in *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, ser. Lecture Notes in Computer Science, P. van Beek, Ed., vol. 3709. Springer, 2005, pp. 827–831. [Online]. Available: https://doi.org/10.1007/11564751_73
- [23] A. Ignatiev, A. Morgado, and J. Marques-Silva, "PySAT: A Python toolkit for prototyping with SAT oracles," in *SAT*, 2018, pp. 428–437. [Online]. Available: https://doi.org/10.1007/978-3-319-94144-8_26
- [24] J. P. Degabriele, J. Gilcher, J. Govinden, and K. Paterson, "Sok: Efficient design and implementation of polynomial hash functions over prime fields," in *45th IEEE Symposium on Security and Privacy (SP 2024)*, 2024.
- [25] S. Carpov, T. Nguyen, R. Sirdey, G. Costantino, and F. 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*. IEEE Computer Society, 2016, pp. 593–599. [Online]. Available: <https://doi.org/10.1109/CLOUD.2016.0084>
- [26] A. Viand, P. Jattke, and A. Hithnawi, "Sok: Fully homomorphic encryption compilers," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 1092–1108. [Online]. Available: <https://doi.org/10.1109/SP40001.2021.00068>