Oraqle

A Depth-Aware Secure Computation Compiler

Vos, Jelle; Conti, Mauro; Erkin, Zekeriya

DOI
10.1145/3689945.3694808

Publication date
2024

Document Version
Final published version

Published in
WAHC '24

Citation (APA)
Vos, J., Conti, M., & Erkin, Z. (2024). Oraqle: A Depth-Aware Secure Computation Compiler. In *WAHC '24: Proceedings of the 12th Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (pp. 43-50). Association for Computing Machinery (ACM). https://doi.org/10.1145/3689945.3694808

Oraqle: A Depth-Aware Secure Computation Compiler

Jelle Vos
Delft University of Technology
Delft, The Netherlands
J.V.Vos@tudelft.nl

Mauro Conti*
University of Padua
Padua, Italy
mauro.conti@unipd.it

Zekeriya Erkin
Delft University of Technology
Delft, The Netherlands
Z.Erkin@tudelft.nl

Abstract

In the past decade, tens of homomorphic encryption compilers have been released, and there are good reasons for these compilers to exist. Firstly, homomorphic encryption is a powerful secure computation technique in that it is relatively easy for parties to switch from plaintext computation to secure computations when compared to techniques like secret sharing. However, the technique is mathematically involved and requires expert knowledge to express computations as homomorphic encryption operations. So, these compilers support users who might otherwise not have the time or expertise to optimize the computation manually. Another reason is that homomorphic encryption is still computationally expensive, so compilers allow users to optimize their secure computation tasks. One major shortcoming of these compilers is that they often do not allow users to use high-level primitives, such as equality checks, comparisons, and AND and OR operations between many operands. The compilers that do are either based on TFHE, requiring large bootstrapping keys that must be sent to the evaluator, or they only work in the Boolean domain, excluding many potentially more performant circuits. Moreover, compilers must reduce the multiplicative depth of the circuits they generate to minimize the noise growth inherent to these homomorphic encryption schemes. However, many compilers only consider reducing the depth as an afterthought. We propose the Oraqle compiler, which solves both problems at once by implementing depth-aware arithmetization, a technique for expressing high-level primitives as arithmetic operations that are executable by homomorphic encryption libraries. Instead of generating one possible circuit, the compiler generates multiple circuits that trade off the number of multiplications with the multiplicative depth. If the depth of the resulting circuits is low enough, they may be evaluated using a BFV or BGV library that does not require bootstrapping keys. We demonstrate that our compiler allows for significant performance gains.

CCS Concepts

• **Security and privacy → Cryptography**.

Keywords

homomorphic encryption; arithmetization; compiler

*Also with Delft University of Technology.

WAHC '24, October 14–18, 2024, Salt Lake City, UT, USA.

ACM ISBN 979-8-4007-1241-8/24/10
https://doi.org/10.1145/3689945.3694808

ACM Reference Format:
Jelle Vos, Mauro Conti, and Zekeriya Erkin. 2024. Oraqle: A Depth-Aware Secure Computation Compiler. In *Proceedings of the 12th Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3689945.3694808

1 Introduction

In the past decade, the field of somewhat and fully homomorphic encryption (FHE) has seen significant advancements, leading to the development of tens of homomorphic encryption compilers. These compilers are essential tools for enabling users who might otherwise not have the time or expertise to transition from plaintext computations to secure computation, and they do so with relative ease compared to other techniques like secret sharing.

Despite their utility, existing FHE compilers still have limitations. A critical shortcoming is their limited support for high-level primitives such as equality checks, comparisons, and AND and OR operations involving multiple operands. This is because FHE schemes compute circuits of additions and multiplications over some algebra. For the schemes that we consider in this work, this algebra is typically the commutative ring of integers modulo some q. In general, it is not straightforward to express high-level primitives as arithmetic circuits in this algebra. However, it is a misconception that generating these arithmetic circuits is impossible for every q. When q is a prime, the plaintext algebra is the finite field $\mathbb{F}_p$, in which any function can be expressed as an arithmetic circuit.

Expressing high-level operations as arithmetic circuits is a process called arithmetization. Some compilers do support the arithmetization of high-level primitives, but they only allow doing so for the plaintext algebra $\mathbb{F}_2$, thereby restricting the circuits to the Boolean circuits. This is a significant restriction that potentially ignores many more efficient circuits. For example, if we want to compute an AND operation between 16 operands, we require many more multiplications in Boolean circuits than in arithmetic circuits, where q can be larger. These multiplications are significantly more expensive to compute than additions. Concretely, we require 15 multiplications in $\mathbb{F}_2$, and only 4 in $\mathbb{F}_{17}$ (see Section 4).

There are also compilers that provide arithmetization of high-level primitives by relying on FHE schemes that support programmable bootstrapping. Instead of computing circuits consisting of additions and multiplications, these schemes compute circuits of additions and programmable bootstrapping operations, which are essentially lookup tables. A common example of such a scheme is TFHE [6]. While these schemes are typically computationally efficient, they require every evaluator to have large bootstrapping keys, which are in the order of tens to thousands of megabytes in size. Our work focuses on BFV/BGV-type schemes [3, 4, 9], which do not require the evaluator to have bootstrapping keys.

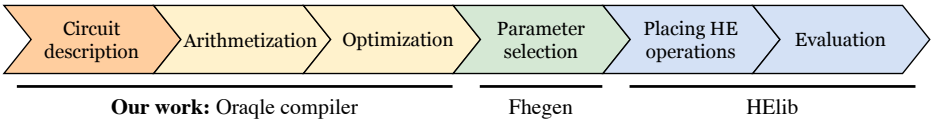


Figure 1: Typical pipeline in a homomorphic encryption compiler and the parts covered by our work.

A second problem in arithmetization for FHE is that a circuit's efficiency is relies strongly on the multiplicative depth of an arithmetic circuit. This metric is defined as the highest number of multiplication on any path through the circuit. The reason is that FHE ciphertexts contain some noise as part of the underlying cryptographic hardness assumption. As FHE schemes perform more homomorphic operations on ciphertexts, this noise grows. At some point, the noise may become so large as to override the plaintext that the ciphertext originally encrypted. Since the noise grows most strongly during ciphertext multiplications, the multiplicative depth is a useful metric for measuring noise growth. Knowing the multiplicative depth of the circuit that will be computed allows one to choose parameters that are large enough to accommodate the expected amount of noise growth. However, larger parameters make multiplications more expensive to compute. As a result, we have two metrics that we want to minimize to increase the efficiency of arithmetic circuits: the number of multiplications, which are expensive to compute, and the multiplicative depth, which impacts the cost of individual multiplications.

In this work, we propose a new compiler called Oraqle that solves both the problem of arithmetizing high-level operations and the problem of multiplicative depth reduction simultaneously. The Oraqle compiler implements depth-aware arithmetization, a concept recently introduced by Vos et al. [21]. By restring the plaintext algebra to $\mathbb{F}_p$, where p is prime, we can arithmetize any high-level function. Unlike other compilers, the Oraqle compiler does not focus on reducing either the number of multiplications or the multiplicative depth, but it reduces both. In doing so, it returns multiple circuits that trade off these two metrics. To be precise, it generates a front of circuits that trade off the multiplicative depth and the multiplicative cost, which is a number that considers that squaring operations are cheaper to compute than arbitrary multiplications.

Our work is not the first to trade off the multiplicative depth with the number of multiplications. However, depth reduction has previously only been considered as an optimization stage that comes after arithmetization [1, 5, 15, 22]. These works input any arithmetic circuit, so they cannot exploit the knowledge of the high-level operations that these circuits perform. Besides, it is possible to arithmetize a high-level operation in multiple ways, resulting in radically different arithmetic circuits. These techniques cannot recover all possible circuits generated using depth-aware arithmetization.

In Figure 1, we describe a typical pipeline for compiling high-level circuits into homomorphic encryption circuits. Notice that the Oraqle compiler considers depth reduction in the *arithmetization* stage, whereas other compilers consider depth reduction in the *optimization* stage. We note that our work only addresses the first parts of the pipeline, and we rely on other work for the later parts. We believe that this decoupling is a positive development. For example, a user (or compiler) could use our compiler to generate arithmetic circuits and another compiler to generate parameters and place homomorphic encryption operations. In the Oraqle compiler, we rely on fhegen [16] to select parameters and HElib [13] for the placement of relinearization and modswitch operations, as well as the final evaluation of the circuit using the BGV cryptosystem [4].

In short, while there are already many homomorphic encryption compilers, the Oraqle compiler is unique in the sense that it:

- Arithmetizes any high-level operation in $\mathbb{F}_p$, supporting equality checks, comparisons, and AND and OR operations between many operands, among others.
- Minimizes both the number of multiplications and the multiplicative depth during arithmetization, generating multiple circuits that trade off these two metrics.
- Considers the fact that squaring is often cheaper to compute homomorphically than arbitrary ciphertext multiplications.

In this paper, we present the practical workings of the Oraqle compiler. We demonstrate that our compiler produces more efficient arithmetizations of comparison operations ($x < y$) than other work for circuits over $\mathbb{F}_p$ where p is prime. Next to that, we use our compiler to demonstrate that arithmetic circuits (i.e. where $p > 2$) can be more performant than Boolean circuits. We show this for doing an equality check ($x = y$) between two 64-bit inputs.

Our paper is structured as follows. We start by reviewing other general-purpose homomorphic encryption compilers for BFV/BGV-type schemes in Section 2. After that, we proceed in the same order as the pipeline diagram in Figure 1. So, in Section 3 we explain how users can describe high-level circuits in Oraqle. In Section 4 we explain how we implement depth-aware arithmetization. We also explain some heuristics & approximations that can be used to speed up circuit generation time. Next, in Section 5, we describe how semantic common subexpression elimination can further optimize the generated arithmetic circuits. After that, we describe in Section 6 how we use *fhegen* and *HElib* to compile the circuits to an executable binary. We present some results in Section 7, and finish with an overview of limitations and a conclusion in Sections 8 & 9.

2 Homomorphic encryption compilers

We briefly discuss existing general-purpose homomorphic encryption compilers for $\mathbb{Z}_q$ plaintext spaces. We exclude works that are solely for TFHE, because these do not execute arithmetic circuits: instead, they are comprised of additions and programmable bootstrapping operations. We only consider works from 2020 and after. We refer the reader to the work by Viand et al. [20] for prior works.

In Table 1, we provide an overview of the works described in this section. We specify several properties in the same order as the pipeline presented in Figure 1. Specifically, we consider each compiler's circuit description interface, by stating their input language

and plaintext algebra. For arithmetization, we discuss whether they support high-level operations, and whether they consider the multiplicative depth during arithmetization. Moreover, we state whether the compilers implement common subexpression elimination (CSE) to reduce the multiplicative size, or depth reduction techniques. Finally, we state whether they automate parameter selection and the placement of relinearization and modswitch operations, as well as the library they use for evaluation.

A takeaway is that few of the previous compilers implement arithmetization for plaintext spaces with $p > 2$, so there is a large space of possible circuits they cannot generate. Moreover, none of the compilers generate circuits in a depth-aware manner for $p > 2$.

T2. The T2 cross-compiler [11] provides a DSL for describing arithmetic circuits, allowing one to generate code for multiple libraries. If p is a prime, the compiler implements arithmetization for equality checks and comparisons, but it does not consider the depth-cost trade-off. The compiler chooses parameters from a predefined list, and places homomorphic encryption operations automatically.

HEIR. The HEIR compiler [10] is based on the multi-level intermediate representation (MLIR) toolchain, which can be reused and extended by other compilers. For this reason, it supports multiple input formats and multiple FHE schemes. At the time of writing, the compiler only translates arithmetic operations on secrets to arithmetic operations on encrypted secrets, so it does not support high-level arithmetization but it can be extended as such. Due to its extensible nature, the compiler inherits common subexpression elimination and tree balancing from MLIR.

Porcupine. The Porcupine compiler [8] focuses on automatic vectorization of arithmetic circuits. It inputs circuits written in a new DSL. While it does not arithmetize high-level operations, it performs depth reduction due to vectorization. Parameters must be selected manually and relinearization operations are placed naively.

HECO. Similar to the HEIR compiler, the HECO compiler [19] relies on the MLIR toolchain. It supports a python front-end and a SEAL backend. It does not implement the arithmetization of high-level primitives, which is currently left to the user. It does perform simple parameter selection and naive placement of relinearization operations. Since it is based on MLIR, it can perform CSE, and it supports a vectorization pass that reduces the multiplicative depth of series of multiplications.

HElium. The HElium compiler is a compiler that focuses on proxy re-encryption, allowing computations on data stored under different keys. The main objective is reducing re-encryption operations. It implements depth reduction in the form of tree rebalancing. As an input language, it uses a new domain-specific language.

Our work: Oraqle. The Oraqle compiler. Strong form of CSE. For the placement of homomorphic encryption operations, we rely on the HElib library, which does so naively (i.e. it may scale down the modulus only to scale it up before the next operation).

3 Programming interface

The programming interface defines the way in which users supply input to the compiler. As shown in Table 1, several works provide a domain-specific language for the user to do so. While this allows one to tailor the language to the use case, it introduces a learning curve. In the Oraqle compiler, we do not use a DSL, and instead allow the user to express circuits in pure Python, supporting a subset of Python functions by overloading operators. This means that we also do not perform introspection or analysis of the abstract syntax tree. While those approaches would allow one to be more expressive, they make it harder for users to change the behavior of the compiler to their needs. A downside is that in some cases, due to language restrictions, the user cannot use a built-in function and instead must resort to a function with a similar name. For example, instead of calling `sum`, the user must call `sum_`. In this section, we first describe how we go from the user's inputs in Python to a high-level circuit description. After that, we provide some examples of Python code and the circuits they describe.

The key way in which we construct high-level circuits that the compiler can arithmetize, is to symbolically execute the Python code by overriding the typical operators. For example, when the user calls `x - y`, this will result in a symbolic `Subtraction(x, y, gf)` node, rather than the interpreter trying to evaluate the expression. We provide several ways in which these symbolic nodes are combined to create different symbolic nodes. For example, additions are automatically flattened into one large `Sum` node. Moreover, if all the inputs to an operation are constants, then the constant is folded. In other words, the output is a constant too.

We capture the semantics of different high-level operations by specifying different types of operations, such as:

- **Fixed nodes**, which have a fixed number of operands.
  - *Commutative binary nodes:* E.g. addition and equality checks.
  - *Non-commutative binary nodes:* E.g. comparisons.
  - *Univariate nodes:* E.g. exponentiation by a constant.
  - *Leaf nodes:* E.g. inputs and constants.
- **Flexible nodes**, which have an arbitrary number of operands.
  - *Commutative & associative reducible nodes with a set of operands:* E.g. AND and OR operations.
  - *Commutative & associative reducible nodes with a multiset of operands:* E.g. sums and products.

In the compiler, we ensure that, for common operations, there is only one way to represent them. For example, we do not allow an AND operation with one operand, or an addition between two constants (this should simply be a constant). As a result, the only time that a `Constant` node exists, is when the entire circuit evaluates to a constant. Otherwise, the constant is part of an operation such as a `ConstantAddition`.

Next, we showcase several examples of the conversions from Python expressions to high-level circuits. These figures are generated by the compiler, which outputs DOT files. Note that these high-level circuits are not yet arithmetized; they describe the function that the user wants to perform, split into common primitives.

Describing high-level circuits. We start with a simple example of a program that a user might run. A user might wish to compute $[x < y]\texttt{AND}[y == z]$. The Oraqle compiler requires the user to first specify the plaintext algebra, prior to the defining the input node x, y, and z. In Listing 1, we use $\mathbb{F}_{31}$ as the plaintext algebra. As one can see, after defining the inputs, the operations are expressed in the same way as in regular Python functions. Finally, the user creates a `Circuit`, which contains an arbitrary number of outputs.
In Figure 3, we show the high-level circuit as generated by the compiler. For non-commutative nodes, the edges enter the node at

Table 1: An overview of homomorphic encryption compilers since 2020 and their compilation stages (see Figure 1).

Compiler		Circuit description		Arithmetization		Optimization		Parameter sel.	Placing HE ops.	Evaluation
Name	Year	Input	Algebra	High-level	Depth-aware	CSE	Depth red.	Automatic	Automatic	Library
Porcupine [8]	2021	Quill	$\mathbb{Z}_q$	-	○	○	●	○	◐	SEAL
HEIR [10]	2023	Multiple	$\mathbb{Z}_q$	-	○	●	◐	●	●	Multiple
HECO [19]	2023	Python	$\mathbb{Z}_q$	-	○	●	●	◐	◐	SEAL
T2 [11]	2023	C++	$\mathbb{Z}_q$	Eq. & comp.	○	○	○	◐	●	Multiple
HElium [12]	2023	HEDSL	-	○	○	○	◐	◐	◐	Multiple
Oraqle	**2024**	**Python**	$\mathbb{F}_p$	**Multiple**	●	●	◐	**fhegen [16]**	◐	**HElib**

the correct side, indicating the direction of the operation (from left to right). In commutative nodes, the edges enter the node anywhere.

Describing arithmetization in the compiler. While the Python interface is useful for users to express the functions they want to compute, it is also used within the compiler to implement transformations such as arithmetization. For example, if the scheme does not support subtractions, the compiler implements a way to arithmetize subtractions into an addition and constant multiplication as `x + -1 * y`. Subtractions can in turn be used to arithmetize if-else operations using the same interface.

Extending arithmetization in the compiler. We also use the Python interface to implement arithmetization external to the compiler, making it easy to compare with other works, such as the comparison circuits as proposed by Gouert et al. [11] for the T2 compiler. The code for this is almost as simple as the equation used to describe the arithmetization. We present this code in Listing 2.

The high-level circuit that the compiler generates from this code can be seen in Figure 4. While it may seem that the compiler implements loop unrolling, this is not the case. The for-loop is executed as is. Since the compiler flattens sums, there is only one addition node at the end of the circuit. Operations like exponentiation by 6 will be arithmetized later, instead of turning them into multiplications at this stage. The reason is that exponentiation can be arithmetized in different ways, trading off the multiplicative cost and the multiplicative depth, as described by Vos et al. [21].

4 Depth-aware arithmetization

The Oraqle compiler implements the depth-aware arithmetization techniques described by Vos et al. [21]. Specifically, they propose how to arithmetize distinct products, exponentiations, polynomial evaluations, and AND and OR operations between multiple operands in a way that trades off the multiplicative cost and the multiplicative depth. In this section, we do not discuss the theory behind the techniques, but we discuss our practical implementation. We begin by explaining our implementation and providing some examples, after which we describe several ways in which the time it takes to generate circuits can be reduced.

4.1 Arithmetization for $p \geq 2$

In the introduction, we gave an example of how AND operations between multiple operands can be performed with fewer multiplications in an arithmetic circuit over $\mathbb{F}_p$ with $p > 2$, than in a Boolean circuit where $p = 2$. The reason is that a Boolean circuit requires the operation $x_1 \wedge \ldots x_{16}$ to be arithmetized as a product $x_1 \times \cdots \times x_{16}$, whereas an arithmetic circuit over $\mathbb{F}_{17}$ allows for many different kinds of circuits. The Oraqle compiler arithmetizes this operation as $(x_1 + \cdots + x_{16})^{16}$, in which the exponentiation only requires four multiplications.

While the Oraqle compiler implements depth-aware arithmetization, it also implements 'regular' arithmetization, in which the compiler only outputs a single circuits. In this mode the compiler seeks to minimize the multiplicative cost, and the multiplicative depth secondarily. The compiler is significantly faster at performing arithmetization in this way, because composition is straightforward: the output of arithmetization of high-level operations is a single arithmetic circuit rather than a Pareto front.

4.2 Depth-aware arithmetization for $p \geq 2$

When it comes to depth-aware arithmetization, the compiler outputs multiple arithmetic circuits for each high-level circuit if it can find a trade-off between the multiplicative cost and the multiplicative depth. We provide an example for performing equality checks in $\mathbb{F}_{31}$, which is the smallest plaintext modulus for which a trade-off occurs. Listing 3 shows the Python input for this function. The Oraqle compiler allows the user to specify the cost of a squaring operation relative to a ciphertext multiplication, which we denote by σ. Calling `arithmetize_depth_aware()` defaults to $\sigma = 1.0$.

The compiler internally makes several calls to the MaxSAT compiler to generate multiple arithmetic circuits with different multiplicative depth. The results are in Figure 5. Here, red-colored multiplications denote non-constant multiplications, which are expensive to compute. The compiler here generates one circuit with a multiplicative depth of 5, and a multiplicative cost of 7, and another with depth 6 and cost 6. It is not clear which circuit is more efficient, especially under composition, until we evaluate them. One limitation is that, in the current version of the compiler, we only implement depth-aware arithmetization for circuits with a single output.

4.3 Practical optimizations

We discuss three optimizations that reduce the times it takes to generate these circuits without changing them.

The current bottleneck in our implementation is the MaxSAT solver that we use to arithmetize exponentiation circuits. Our implementation uses the RC2 solver [17] implemented in PySAT [14], and defaults to the Glucose 4.2.1 SAT solver [2]. While it is not always possible to reduce the number of exponentiations that we must arithmetize, we employ caching to significantly reduce the number of calls made to the MaxSAT solver.

Another optimization is that constant folding allows us to sometimes skip arithmetization altogether. For example, if the a subcircuit in a `Product` node evaluates to 0, we can output a constant. The same applies to other nodes with multiple operands.

Finally, *commutative & associative reducible nodes with a set of operands* (see Section 3), the inputs are actually modeled as a set. This means that if during arithmetization, an AND operation receives the same operand twice (or one that is equivalent), it ignores the second. We discuss equivalence in the context of semantic subexpression elimination in Section 5.

4.4 Heuristics & approximations

We propose several ways in which the user may speed up circuit generation time at the cost of a potentially worse circuit. This is necessary for larger circuits and larger plaintext moduli, which increase the duration of arithmetization. The reason is that the depth-aware arithmetization techniques proposed by Vos et al. [21] in some cases perform exhaustive searches.

One context in which we can avoid exhaustive search, is in polynomial evaluation. All the polynomial evaluation methods described in [21] require the compiler to choose a parameter k, which affects both the multiplicative depth and cost of the circuits that are generated. Vos et al. propose to try all values $1 \leq k < p$ but this requires a significant amount of computation and many calls to the MaxSAT solver. Paterson & Stockmeyer [18] instead analytically derive a single value for k, but Vos et al. show that this is not always optimal in practice. We propose a heuristic, where we only evaluate several values for k around the value derived by Paterson & Stockmeyer. In our experiments, we find that the optimal k is typically only 1 value away. The compiler only searches for values up to twice the size of the analytically-optimal k, which makes circuit generation approximately twice as fast in practice.

5 Optimization of arithmetic circuits

After generating arithmetic circuits, different compilers perform different forms of post-processing in an attempt to reduce the multiplicative cost or the multiplicative depth. These are transformations from arithmetic circuits to other arithmetic circuits. The Oraqle compiler currently implements one optimization in the form of semantic common subexpression elimination, which is similar to that implemented by the EVA compiler [7] for a different kind of homomorphic cryptosystem.

Common subexpression elimination is a technique that has been applied to many homomorphic encryption compilers and many regular compilers alike. The simple idea is to never compute the same thing twice. While this seems obvious, it is not uncommon for arithmetization (or compilation) to introduce common subexpressions. In our work, we implement semantic common subexpression elimination, meaning that the compiler can also recognize two subexpressions to be equivalent but not identical. We give an example in Figures 6a & 6b of two circuits that the compiler can tell to be equivalent.

The way we implement these equivalence checks efficiently, is to ensure that equivalent expressions have the same hash. This is not always possible, but it is easy to do for properties such as commutativity. The Oraqle compiler computes the hash for commutative & associative nodes by sorting the hashes of all the operands before computing the hash, making the order of operands irrelevant. It can also be done for high-level operations that are non-commutative but each others inverses. For example, $x < y$ is equivalent to $y > x$.

6 Code generation

Since the Oraqle compiler currently focuses on arithmetization, it does not perform parameter selection, placement of homomorphic encryption operations, or evaluation. Instead, it relies on fhegen [16] and HElib [13]. In this section, we describe the steps from an arithmetic circuit to an executable binary chronologically. We note that one might also use other tools to finish compiling the arithmetic circuits generated by the Oraqle compiler.

Register allocation. To reduce the memory footprint of the final binary, we implement a register allocation step, which determines at any time throughout the computation how many ciphertexts must be stored. We note that these are logical registers containing FHE ciphertexts, and not actual hardware registers. We do so using a topological graph traversal of the arithmetic circuits. After this step, each node in the arithmetic circuit knows to which register it can assign the result of its computation.

Translation into instructions. At this point, we can compile the arithmetic circuit with register allocation to what is essentially assembly for FHE operations. That is, input nodes are translated to instructions that place a named input into a register, arithmetic nodes perform operations on several registers, placing the result in a (possibly overlapping) register, and output nodes instruct are translated to instructions that output a given register. Importantly, the compiler traverses the graph in the same way that it did before.

Translation to a program. Finally, we generate C++ code that can be compiled into an actual binary. This takes two steps, as the code includes both the parameters with which the FHE schemes will be instantiated, as well as the actual circuit evaluation. We use the methods provided by fhegen [16] to generate parameters for the default settings in HElib using the OpenFHE cost model. To do so, we derive several metrics from the arithmetic circuit. We make one modification to support $p = 2$, which is to decrement the polynomial degree m by 1, as p may not divide m. After this step, code generation is a direct translation from the FHE instructions to the functions implemented in HElib for performing homomorphic operations. The resulting code can be compiled using any C++ compiler. HElib here performs two stages of the pipeline as presented in Figure 1: it places and performs relinearization and modswitch operations, and it evaluates the homomorphic operations.

7 Results

In this section, we provide two results using the Oraqle compiler.[1] We first show that choosing $p > 2$ does lead to circuits with better practical performance than fixing $p = 2$. Next, we show that an optimistic implementation of the arithmetization technique used in the T2 compiler produces circuits with worse practical performance than the Oraqle compiler does. We execute our experiments on a strong computer with a Threadripper 7970X CPU. The CPU has 64 threads, but we only use a single thread to compile and evaluate the circuits. When it comes to memory, it has 4x64GB DDR5 RAM.

[1] Our compiler is open source and can be found at: https://github.com/jellevos/oraqle

Arithmetic versus Boolean circuits. Since the Oraqle compiler allows compiling any function into an arithmetic circuit for any plaintext modulus p that is prime, we can evaluate the performance of different p for the same operation. We consider here the function that checks whether two 64-bit inputs are equal and provide the results in Table 2 and we set $\sigma = 0.75$. Choosing $p > 2$ allows circuits with lower multiplicative cost at the expense of a higher depth. Moreover, $p = 2$ does not allow the ring dimension m to be a power of two. This choice of m means that homomorphic operations are slower, even though m can be smaller. We see that choosing $p = 5$ leads to an arithmetic circuit that is almost twice as fast to compute as the Boolean circuit.

Table 2: Run time for a circuit checking whether two 64-bit integers are equal. We consider the front of solutions across all $2 \leq p \leq 257$ that are prime.

Circuits			Parameters				Results
Modulus p	Depth	Cost	Ring dimension m	r	Bits	c	Run time (s)
2	6	63	16385	1	142	1	3.28
5	7	58	32768	1	178	1	1.67
17	8	51	32768	1	217	1	1.96

Arithmetization in other compilers. While there are many homomorphic encryption compilers, they typically target the earlier or later stages of the compilation pipeline. To still facilitate a comparison, we compare the less-than circuits generated by our compiler with those generated by the T2 compiler as described in the paper by Gouert et al. [11]. While the techniques described by Vos et al. [21] and Gouert et al. only perform comparisons between half of the elements in $\mathbb{F}_p$, we propose a new arithmetization that performs three calls to the half-comparisons. We denote these half-comparisons by $\prec$. We precompute $x_{\text{small}} = [x \prec \frac{p-1}{2}]$ and $y_{\text{small}} = [y \prec \frac{p-1}{2}]$. Our arithmetization for both works is as follows, where $\overline{x}$ represents negation of a Boolean variable:

$$[x < y] = \overline{x_{\text{small}} \oplus y_{\text{small}}}[x \prec y] + (x_{\text{small}} \wedge y_{\text{small}}) \,. \tag{1}$$

In Figure 2 we compare the actual run time of the T2 circuits with the circuits generated by the Oraqle compiler for a growing plaintext modulus. Our circuits consistently outperform the T2 circuits, often by an order of magnitude, even though these sometimes have a lower multiplicative depth. The reason is that these circuits have a significantly higher multiplicative cost.

8 Limitations

Packing and rotations would allow for the number of multiplications to be reduced. Since we do not consider packing, we consider plaintext moduli that are not necessarily NTT-friendly, which would allow constant additions and multiplications with arbitrary vectors. This is also the reason why we currently only support code generation for HElib: other libraries do not support plaintext moduli that are not NTT-friendly.

The compiler does not yet take common inputs into account. E.g. when performing multiple polynomial evaluations, we could reuse the precomputations. The same applies to computing multiple products with the same operands.

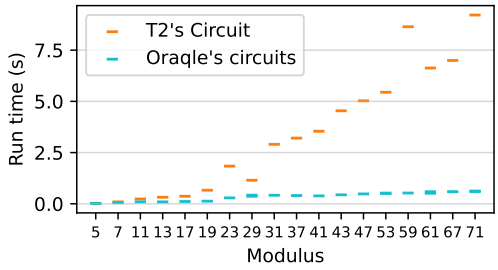


Figure 2: Arithmetization of a less-than operation in T2 and in Oraqle for $\sigma = 1.0$. In some cases the Oraqle compiler outputs multiple circuits but they perform similarly.

The compiler also does not provide a layer of abstraction for integers (or e.g. real numbers) that exceed the plaintext space. In this stage, the user would have to implement this logic by hand.

We currently only perform depth-aware arithmetization on circuits with a single output. We argue that this is mostly a practical limitation and not a theoretical limitation.

As seen from our experiments, the cost of FHE cannot be completely described by the multiplicative depth and cost. These merely serve as metrics. There are other factors, such as the polynomial degree m being a power or two of not. Moreover, it matters how and when homomorphic encryption 'maintenance' operations are placed such as relinearizations and modswitches. Some circuits are more amenable to reducing the number of maintenance operations than others. Another example is that multiplications at the end of the circuit become slightly cheaper to compute as the ciphertext modulus shrinks.

9 Conclusion

In conclusion, while previous homomorphic encryption compilers play a crucial role in enabling users to transition from plaintext to secure computations, they typically only implement automatic arithmetization for Boolean circuits, or they require large bootstrapping keys. The Oraqle compiler addresses these issues by implementing depth-aware arithmetization for BFV/BGV-type cryptosystems with prime plaintext moduli, allowing it to express high-level primitives as arithmetic operations suitable for homomorphic encryption libraries. This allows one to find more efficient circuits than when considering depth reduction only as an afterthought.

Future work might focus on the following enhancements:

- Incorporating SIMD, which will require handling larger plaintext modulus for arbitrary plaintext vectors.
- Implementing multi-threading, accelerating compilation.
- Optimizing addition chain generation.
- Integrating sorting networks and other complex structures.
- Implementing early stopping, e.g. using a maximum depth.
- Extending the compiler's capability to handle nodes with an arbitrary number of outputs.

A Circuit visualizations

We present several circuits generated by the Oraqle compiler in Figures 3, 4, 5, and 6.

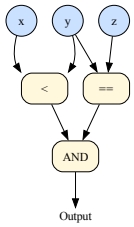


Figure 3: An example circuit with high-level operations.

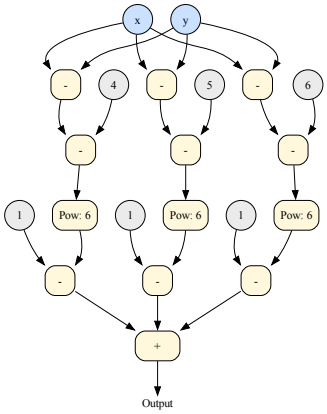
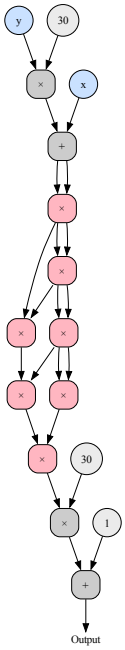
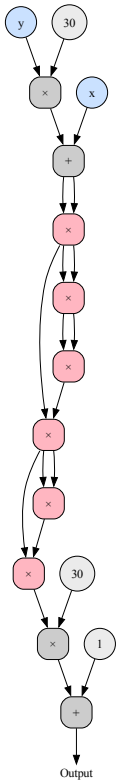


Figure 4: High-level circuit of a comparison operation as proposed by Gouert et al. [11] when $p = 7$.

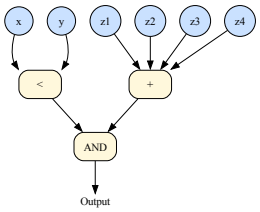


(a) Depth-5 circuit

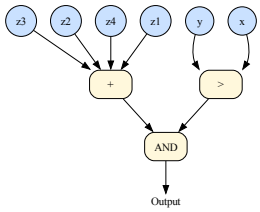


(b) Depth-6 circuit

Figure 5: Depth-aware arithmetization of $x = y$ in $\mathbb{F}_{31}$.



(a) Original circuit



(b) Equivalent circuit

Figure 6: Two circuits that are not identical but equivalent.

B Code samples

We present several code samples used to generate these circuits.

```
gf = GF(31)
x = Input("x", gf)
y = Input("y", gf)
z = Input("z", gf)

comparison = x < y
equality = y == z
both = comparison & equality

circuit = Circuit([both])
```

Listing 1: A simple example of a three-input function using high-level operations.

```
gf = GF(p)
x = Input("x", gf)
y = Input("y", gf)

comparison = 0

for a in range((p + 1) // 2, p):
    comparison += 1 - (x - y - a) ** (p - 1)

circuit = Circuit([comparison])
```

Listing 2: Implementation of a comparison operation as proposed by Gouert et al. [11] in the Oraqle compiler.

```
gf = GF(31)
x = Input("x", gf)
y = Input("y", gf)

equality = x == y

circuit = Circuit([equality])
arithmetic_circuits = circuit.arithmetize_depth_aware()
```

Listing 3: Implementation of an equality operation over $\mathbb{F}_{31}$.

References

[1] Pascal Aubry, Sergiu Carpov, and Renaud Sirdey. 2020. 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 (Lecture Notes in Computer Science, Vol. 12006)*, Stanislaw Jarecki (Ed.). Springer, 345–363. https://doi.org/10.1007/978-3-030-40186-3_15

[2] Gilles Audemard and Laurent Simon. 2024. Glucose SAT Solver. https://github.com/audemard/glucose. Accessed: 2024-07-27.

[3] Zvika Brakerski. 2012. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7417)*, Reihaneh Safavi-Naini and Ran Canetti (Eds.). Springer, 868–886. https://doi.org/10.1007/978-3-642-32009-5_50

[4] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2012. (Leveled) fully homomorphic encryption without bootstrapping. In *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, Shafi Goldwasser (Ed.). ACM, 309–325. https://doi.org/10.1145/2090236.2090262

[5] Sergiu Carpov, Pascal Aubry, and Renaud Sirdey. 2017. 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 (Lecture Notes in Computer Science, Vol. 10765)*, Ljiljana Brankovic, Joe Ryan, and William F. Smyth (Eds.). Springer, 275–286. https://doi.org/10.1007/978-3-319-78825-8_23

[6] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: Fast Fully Homomorphic Encryption Over the Torus. *J. Cryptol.* 33, 1 (2020), 34–91. https://doi.org/10.1007/S00145-019-09319-X

[7] Sangeeta Chowdhary, Wei Dai, Kim Laine, and Olli Saarikivi. 2021. 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, 43–55. https://doi.org/10.1145/3474366.3486929

[8] Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T. Lee, and Brandon Reagen. 2021. Porcupine: a synthesizing compiler for vectorized homomorphic encryption. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 375–389. https://doi.org/10.1145/3453483.3454050

[9] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptol. ePrint Arch.* (2012), 144. http://eprint.iacr.org/2012/144

[10] Google. 2024. HEIR: A compiler for homomorphic encryption. https://github.com/google/heir. Accessed: 2024-07-25.

[11] Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. 2023. SoK: New Insights into Fully Homomorphic Encryption Libraries via Standardized Benchmarks. *Proc. Priv. Enhancing Technol.* 2023, 3 (2023), 154–172. https://doi.org/10.56553/POPETS-2023-0075

[12] Mirko Günther, Lars Schütze, Kilian Becher, Thorsten Strufe, and Jerónimo Castrillón. 2023. HElium: A Language and Compiler for Fully Homomorphic Encryption with Support for Proxy Re-Encryption. *CoRR* abs/2312.14250 (2023). https://doi.org/10.48550/ARXIV.2312.14250 arXiv:2312.14250

[13] Shai Halevi and Victor Shoup. 2020. Design and implementation of HElib: a homomorphic encryption library. *IACR Cryptol. ePrint Arch.* (2020), 1481. https://eprint.iacr.org/2020/1481

[14] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. 2018. PySAT: A Python Toolkit for Prototyping with SAT Oracles. In *SAT*. 428–437. https://doi.org/10.1007/978-3-319-94144-8_26

[15] DongKwon Lee, Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi. 2020. 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*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 503–518. https://doi.org/10.1145/3385412.3385996

[16] Johannes Mono, Chiara Marcolla, Georg Land, Tim Güneysu, and Najwa Aaraj. 2022. Finding and Evaluating Parameters for BGV. Cryptology ePrint Archive, Paper 2022/706. https://eprint.iacr.org/2022/706 https://eprint.iacr.org/2022/706.

[17] António Morgado, Carmine Dodaro, and João Marques-Silva. 2014. 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 (Lecture Notes in Computer Science, Vol. 8656)*, Barry O'Sullivan (Ed.). Springer, 564–573. https://doi.org/10.1007/978-3-319-10428-7_41

[18] Mike Paterson and Larry J. Stockmeyer. 1973. On the Number of Nonscalar Multiplications Necessary to Evaluate Polynomials. *SIAM J. Comput.* 2, 1 (1973), 60–66. https://doi.org/10.1137/0202007

[19] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. 2023. HECO: Fully Homomorphic Encryption Compiler. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 4715–4732. https://www.usenix.org/conference/usenixsecurity23/presentation/viand

[20] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. 2021. SoK: Fully Homomorphic Encryption Compilers. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1092–1108. https://doi.org/10.1109/SP40001.2021.00068

[21] Jelle Vos, Mauro Conti, and Zekeriya Erkin. 2024. Depth-Aware Arithmetization of Common Primitives in Prime Fields. Cryptology ePrint Archive. https://eprint.iacr.org/2024/1200

[22] Mingfei Yu and Giovanni De Micheli. 2024. Expediting Homomorphic Computation via Multiplicative Complexity-aware Multiplicative Depth Minimization. Cryptology ePrint Archive, Paper 2024/1015. https://eprint.iacr.org/2024/1015 https://eprint.iacr.org/2024/1015.