CS252R (Fall 2023)

Optimizing Boolean Circuits for Fully Homomorphic Encryption

ANDREW SIMA, Harvard University, USA

This paper studies methods for optimizing boolean circuits in the context of fully homomorphic encryption (FHE). With respect to encrypted computation and securing machine learning, the complexity of FHE and lack of standardized programming paradigms remain challenges in its integration into applications. One solution to this is an FHE compiler—a toolchain converting high-level languages into an intermediate representation respecting a chosen FHE encryption scheme. However, computational overhead remains a hurdle, prompting exploration into optimizing boolean circuits via tools like the ABC toolchain and logic gate rearrangement techniques. Following methods from the literature, we are able to replicate state-of-the-art program synthesis-based circuit optimization systems, with comparable metrics to the best reported benchmarks.

Additional Key Words and Phrases: Fully Homomorphic Encryption, Privacy-Preserving Computation, Boolean Circuits, Logic Synthesis

1 INTRODUCTION

As the usage of Large Language Models (LLMs) increases exponentially in various modern software applications, leakage of training data has become a heated topic of discussion. It has been demonstrated that adversaries can efficiently extract substantial amounts of training data from various language models, including but not limited to open-source, semi-open, and closed models [1]. Ultimately, this hinders the adoption of LLMs in industries handling more sensitive information, such as law and medicine.

In this paper, we address using encrypted computation to alleviate these issues, specifically fully homomorphic encryption (FHE), an encryption scheme that allows computation without needing to perform decryption between individual compute operations. The idea of performing machine learning techniques on encrypted data is not a new one, and FHE has previously been applied towards securing both k-NN algorithms [2] and statistical analysis [3].

However, a long standing issue of using these encryption schemes is the high barrier to entry: it requires an experienced cryptographer to write programs in these schemes that ensure both security and computational efficiency. Furthermore, standardized programming paradigms and uniformity in API design is severely lacking; each encryption tool/scheme is usually a standalone project. Scoping out this unified standard is an active area of work [4][5].

A logical solution towards democratizing access to FHE is a toolchain that can convert a high level language like Python into an intermediate representation (IR) that respects the chosen FHE encryption scheme, hiding the internal encryption specific operations into the IR instruction set. We can then compile the IR into a boolean circuit represented by a hardware description language (HDL) like Verilog, from which it is easy to get an efficient binary or even make use of in SoC design. This type of toolchain/software is generally referred to as an FHE compiler, and many prototypes have been produced over the years [6].

However, even with cutting edge compilers, the headache of computing on encrypted data is compounded due to expensive compute, with it being order of magnitudes slower than plaintext computation [7]. Hence, another active area of research is optimizing the end product of the FHE compiler, namely the boolean circuit; intuitively, this makes for a more computationally binary resulting in performance speedups for both training and inference. One industry standard for manipulating boolean circuits is the ABC toolchain, which bundles tools for equivalence checking and baseline optimizations [8]. Further attempts have been made at identifying specific arrangement of gates and using boolean logic to rewrite these structures in a more computationally efficient arrangement [9]. More recent work incorporates program synthesis as a means to boolean logic rewrite rule generation, improving upon the handwritten rule generation used previously [7].

In this project, we attempt to reproduce state of the art circuit optimization systems using program synthesis methods.

2 RELATED WORK

2.1 Fully Homomorphic Encryption

The literature on encryption schemes is vast, so for brevity, we formalize an interface for FHE, touch briefly on partially homomorphic encryption (PHE), before discussing some widely used implementations. As a formal interface for FHE, we define $\mathbb{Z}_2 = \{0, 1\}$ the plain text place, Ω the encrypted text space, pk for public key, and sk for secret key; then it makes sense to define functions (taken from [7])

$$Encrypt_{pk} : \mathbb{Z}_2 \to \Omega, \tag{1}$$

$$Decrypt_{sk} : \Omega \to \mathbb{Z}_2, \tag{2}$$

$$Add_{pk} : \Omega \to \Omega, \tag{3}$$

$$Multiply_{pk} : \Omega \to \Omega. \tag{4}$$

Then given two messages m_1 and m_2, the following properties must hold for this encryption scheme to be FHE (i.e. multiplication and addition are preserved across encryption/decryption).

$$Decrypt_{sk}(Add_{pk}(Encrypt_{pk}(m_1), Encrypt_{pk}(m_2))) = m_1 + m_2, \tag{5}$$

$$Decrypt_{sk}(Multiply_{pk}(Encrypt_{pk}(m_1), Encrypt_{pk}(m_2))) = m_1 \times m_2. \tag{6}$$

Note that partially homomorphic encryption are schemes where one but not both of properties (5) and (6) are satisfied; a good example is the ubiquitous RSA, which respects the multiplication but not addition property. The production-grade implementations of FHE begin with what are called the *second generation schemes*, including thje Brakerski-Gentry-Vaikuntanathan (BGV), Brakerski-Fan-Vaikuntanathan (BFV), Cheon-Kim-Kim-Song (CKKS), Fast Fully Homomorphic Encryption Over the Torus (CGGI) schemes [6] [10]. Select implementations can be found as Microsoft SEAL[11] and TFHE[12].

On a high level, the purpose of these encryption libraries is to provide functionality such as key generation, encoding, encryption, addition, and multiplication in a module-like fashion, abstracting away the low-level details from developers. Whereas previous iterations took on the order of 30 minutes to complete a simple addition operation, the performance improvement in second generation schemes comes from internal SIMD batching of multiple message texts in computing their respective encrypted texts[13].

Furthermore, these encryption schemes all make use of the “Learning with Errors” (LWE) technique in which a small enough amount of noise is added to the encrypted text such that decryption is still possible [14]. This is especially problematic for multiplication operations because the noise becomes compounded, requiring techniques such as bootstrapping, where the message is periodically decrypted after a given number of operations in order to "reset" the noise[10].

Putting the specific details of scheme engineering aside, the encryption scheme most suitable to the project would be TFHE, because it specifically processes a binary encoding compatible with boolean circuits.

2.2 Homomorphic compilers

As touched on briefly above, FHE compilers are higher-order tools that abstract away the low level homomorphic operations from developers, instead translating programs written in higher-level languages into their FHE-based counterparts, and relying on the FHE libraries discussed in the previous section to perform the actual encryption, decryption, and homomorphic computation.

On a more framework specific level, the most popular frameworks include Cingulata[15], E^3[16], and Marble[17], all of which have an input language of C++ and compile down to an arithmetic / boolean circuit representation. While the C++ FHE compiler implementations are quite interoperable with many of the encryption scheme implementations, there is a varying range of support for lower-level circuit depth optimizations (see section 3) and manual configuration required to support system-level optimizations like SIMD batching and noise maintenance[6].

The best production grade general purpose FHE implementation is arguably the XLS transpiler released by Google, supporting both C++ and DSLX, a DSL dataflow-oriented functional language used to specify FHE-friendly routines that can be more conveniently converted to Verilog[10]. There also exist compilers that operate on inputs other than C++, such as Ramparts taking input in Julia[18], Alchemy taking input in Haskell[19], and PyTFHE[20] and EVA[21] taking inputs in Python. However, these frameworks either suffer from long compilation times due to its symbolic execution design (Julia), or poor user experience and interoperability with input encoding schemes (Alchemy).

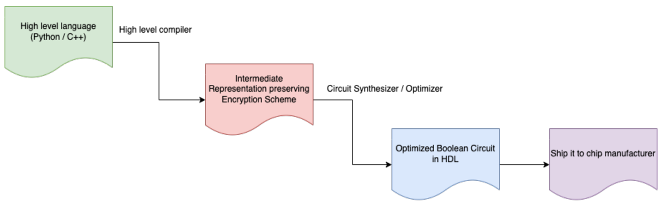
High level language (Python / C++)
High level compiler
Intermediate Representation preserving Encryption Scheme
Circuit Synthesizer / Optimizer
Optimized Boolean Circuit in HDL
Ship it to chip manufacturer

Fig. 1. Stages of an FHE compiler workflow

More recently, machine learning specific FHE compilers have also been an active area of research, such as CHET[22] and SEALion[23]. These compilers operate specifically on tensor manipulation operations such as convolution, matrix multiplication, pooling, and reshaping, and can efficiently compile a program expressed in Tensorflow down to an arithmetic circuit while optimizing for inference performance[6]. However, like the compilers above, there is little consideration for circuit depth optimization once code has been compiled down to the arithmetic circuit.

Further limitations of these compilers, even among the ones that consider circuit optimization, are that most optimizations are implemented using a fixed set of logic rules that were likely handwritten—it is likely that there are better rules that reduce the circuit more efficiently. Therefore, it is worthwhile to study the underlying circuit optimization using program synthesis, specifically using a bottom-up enumerative search (BUS) to ensure that the search space is sufficiently covered.

2.3 Logic Synthesis and Circuit Optimization

Logic synthesis refers to the process by which a hardware description language like verilog is manipulated and transformed into a netlist that can be further manipulated by an electronic design automation (EDA) tool before being processed by a semiconductor manufacturer[24]. The pioneer system was multi-level combinational logic synthesis (MIS)[25] developed in the 1980s at Berkeley. Then with the introduction of the AIGER format representing circuits as And-Inverter Graphs (AIGs), circuits consisting of only AND ($\wedge$) and INV ($\neg$) gates, this system eventually evolved into the modern open-source ABC software system.

ABC provides a standardized suite of useful circuit manipulation features including rewriting, rebalancing, hardware specific mapping, and verification / SAT checking. The software was based on operations over the AIG format, representing both arithmetic and boolean circuits. Because arithmetic circuits are essentially polynomial functions with order equal to the number of inputs, there has countless follow-on work to improve circuit rewriting via algebraic manipulation[26]. Results from graph theory and truth table computation are further applied in Yu, et. al. [27] to improve both rewrites and verification.

Of specific interest in this project are the suite of formal verification tools provided. This suite of tools is also the product of years of refinement, working in conjuction with SMT and Boolean SAT solvers to achieve optimal performance [27]. In our case, we use combinational equivalence checking or CEC (i.e. given two circuits, determine if they describe the same arithmetic/boolean function) to drive our heuristic search and check our rewrite results. Circling back to our original problem of circuit depth optimization, the work closest to state-of-the-art was first done by Carpov, et. al. [9] in which they used two basic rewrite operators, namely AND associativity and XOR distributivity to perform optimizations on top of the circuits previously optimized by ABC, obtaining circuits in many cases almost 50% smaller in depth. It was then in Lee, et. al.[7], that the idea of applying program synthesis was formally introduced to improve upon previous boolean circuit rewrite rules. It was this paper that initially inspired me to undertake this project.

3 PROBLEM DEFINITION

The formal definition of a boolean circuit $c \in \mathbb{C}$, where $\wedge$ and $\oplus$ are the AND and XOR gates respectively, is [7]:

$$c \longrightarrow \wedge(c,c) \mid \oplus(c,c) \mid \text{input} \mid 0 \mid 1. \tag{7}$$

Then given some node $v \in V$, where V is the set of nodes of c, we can define

$$pred : V \longrightarrow 2^V \text{ and } succ : V \longrightarrow 2^V \tag{8}$$

as functions returning the predecessors and successors of the node input. From the literature, the length of successive AND gates on a path, defined as the **multiplicative depth**, determines both the computational complexity and the ciphertext size [9]. Formally, we can define $isAndGate : V \longrightarrow \{0,1\}$, the multiplicative depth function $l : V \longrightarrow \mathbb{N}$, and the reverse multiplicative depth function $r : V \longrightarrow \mathbb{N}$.

$$isAndGate(V) = \begin{cases} 1 & \text{if } v \text{ is an AND gate,} \\ 0 & \text{otherwise.} \end{cases} \tag{9}$$

$$l(v) = \begin{cases} 0 & \text{if } |pred(v)| = 0, \\ \max_{u \in pred(v)} l(u) + isAndGate(v) & \text{otherwise.} \end{cases} \tag{10}$$

$$r(v) = \begin{cases} 0 & \text{if } |succ(v)| = 0, \\ \max_{u \in succ(v)} (r(u) + isAndGate(u)) & \text{otherwise.} \end{cases} \tag{11}$$

Then we can define the max multiplicative depth (i.e. depth determining performance of the circuit) as

$$l_{max} = \max_{v \in V} l(v) = \max_{v \in V} r(v) \tag{12}$$

and a **critical node** as any node $v \in V$ satisfying

$$l_v + r_v = l_{max}. \tag{13}$$

A **critical** path is then any path in the boolean circuit c composed of only critical nodes. We can define the formal optimization problem statement as the following[7]: given some boolean circuit c

with inputs $x_1, x_2, \cdots x_n$, we aim to find an equivalent circuit c' such that

$$c(x_1, x_2, \cdots x_n) \equiv c'(x_1, x_2, \cdots x_n) \text{ and } l_{max}(c) > l_{max}(c'). \tag{14}$$

4 METHODS

4.1 Dataset Preparation

There are many different ways to represent boolean circuits, most notably Verilog, VHDL (Very High-Speed Integrated Circuit Hardware Description Language), BLIF (Berkeley Logic Interchange Format) and AIGER (And-Inverter Graph Format) formats. Here we choose Verilog due to its ubiquity and portability to wide variety of other applications.

We begin the dataset preparation process using purposely unoptimized circuits from the EPFL Combinational Benchmark Suite [28].

In order to get the standardized dataset used in Carpov, et. al. [9], we run an initial resynthesis command with the ABC CLI tool as seen in Figure 2. For the rest of the circuit manipulation,

```
[...] ~/abc> ./abc
abc 01> rv <test_verilog_file.v>
abc 02> resyn2
abc 03> wv <verilog_input_to_python.v>
```

Fig. 2. Initial Circuit Resynthesis with ABC

we chose to work in Python and make use of the `circuitgraph` library due to ease of graph manipulation and compatability with verilog. Because circuits in the EPFL dataset contain gates OR ($\vee$) and INV ($\neg$), we first rewrite these gates in terms of AND ($\wedge$) and XOR ($\oplus$) using the following rules:

$$a \vee b = (a \oplus b) \oplus (a \wedge b), \tag{15}$$
$$\neg a = 1 \oplus a. \tag{16}$$

We do this for two reasons: (1) from the literature, the set of operators {"AND", "XOR", "1"} is functionally complete [29] and (2) this greatly simplifies the rewrite rule DSL and BUS search space as we cut the number of enumerations. We implement this initial rewriting of gates in Python as the

```
[...] ~/abc> ./abc
abc 01> cec <test_verilog_file.v> <python_output_file.v>
Networks are equivalent.  Time =     0.00 sec
```

Fig. 3. Circuit Equivalence Verification with ABC and example output

`do_and_xor_transform` function, and verify using CEC that our AND-XOR circuit is semantically equivalent as shown in Figure 3.

4.2 Rewrite Optimizations

Following Carpov, et. al. [9], we begin by adding two basic rewrite rules describing AND ($\wedge$) associativity and XOR ($\oplus$) distributivity to our rewrite rule set.

$$a \wedge b = b \wedge a, \tag{17}$$
$$(a \oplus b) \wedge c = (a \wedge c) \oplus (b \wedge c). \tag{18}$$

To speed up the algorithm, we also cannot rewrite arbitrary critical paths. We specifically examine the set of critical paths P that begin and end in an AND ($\wedge$) gate, and contain only XOR ($\oplus$) gates in between; furthermore, in order to see a decrease in the overall multiplicative depth, we must rewrite all parallel critical paths. To ensure that we see results with every rewrite, given nodes $x, y, z \in V$, we only execute the rule if

$$l(y) < l(x) \text{ and } l(z) < l(x) \tag{19}$$

i.e. only x has greater multiplicative depth than both y and z [9]. Other speedups come from the repeated XOR distributivity trick, in which we continually apply (18) within a single iteration to get the following identity

$$(x \oplus (y1 \oplus \cdots \oplus yk)) \wedge z = (x \wedge z) \oplus ((y1 \oplus \cdots \oplus yk) \wedge z). \tag{20}$$

Moving into the part of the algorithm requiring program synthesis, in order to run the BUS algorithm for generating rewrite rules, we need to define the appropriate DSL a general boolean circuit. We can then generate the set of boolean circuits of depth 0, 1, and 2 according to our DSL,

$$
\begin{aligned}
S &\longrightarrow d_n && (21)\\
d_n &\longrightarrow d_{n-1} \wedge d_{n-1} \mid d_n \oplus d_n \mid d_{n-1} && (22)\\
&\vdots\\
d_1 &\longrightarrow d_0 \wedge d_0 \mid d_1 \oplus d_1 \mid d_0 && (23)\\
d_0 &\longrightarrow 0 \mid 1 \mid x_0 \mid \cdots \mid x_k && (24)
\end{aligned}
$$

Fig. 4. Boolean Circuit DSL. S represents the starting symbol, x_i denotes the inputs, d_i refers to a circuit of depth i

removing any enumerated circuits with greater depth. The reason we can do this is similar to why we choose to work only with the set P: because the rewrite rule of multiplicative depth 2 can just be reapplied to the circuit to reduce the overall multiplicative depth.

4.3 Heuristics and Training Loop

The training loop algorithm we devised is a modified version of the minimization heuristic presented in Carpov, et. al. [9]. The termination conditions are similar in that there are only 2 possible: (1) max iteration count is reached or (2) no critical path targets can be found for all rules in the rewrite ruleset. Furthermore, we also use a heuristic to choose which critical path will be rewritten first, namely

$$
\begin{aligned}
&\texttt{priority\_c} : \text{the total number of critical node predecessors of all} && (25)\\
&\qquad\text{path nodes in decreasing order.} && (26)
\end{aligned}
$$

There are two main differences with our algorithm: (1) the rewrite ruleset is generated previous to running the heuristic using BUS and (2) for each iteration of the training loop, the algorithm repeatedly applies the same rewrite rule until their are no more viable targets for the iteration, before moving onto the next rewrite rule. After the training loop exits, the verification procedure in Figure 3 is repeated to ensure that circuit equivalence is preserved.

```
def heuristic_based_training_loop(c, rw_ruleset, prior_func, max_iter):
    iter = 0
    cout = c.copy()
    while iter < max_iter:
        iter += 1
        continue_cnt = 0
        # 1. create copy
        coutp = cout.copy()
        # 2. loop through ruleset
        for rule in rw_ruleset:
            # 3. filter viable critical paths
            P = filter_paths(get_critical_paths(coutp))
            if len(P) == 0:
                continue_cnt += 1
                continue
            # 4. repeatedly apply rule
            while len(P) != 0:
                # 5. sort and select the path with highest priority
                selected_path = prior_func(P)
                # 6. apply rewrite
                do_rewrite(selected_path)
                P = filter_paths(get_critical_paths(coutp))
        # 7. save after all rewrite rules tried
        if lmax(coutp) > lmax(cout):
            cout = coutp
    return cout
```

Fig. 5. Python pseudocode for heuristic-based training loop

5 EVALUATION

The results of the initial dataset preparation are found in Table 1. We specifically chose 5 circuits all with multiplicative depth less than 20, namely `ctrl`, `dec`, `int2float`, `router`, and `cavlc`. We

Circuit Name	Multiplicative Depth	# of AND Gates	Size
ctrl	8 / 8	107 / 105	180 / 226
dec	3 / 3	304 / 304	312 / 328
router	19 / 19	170 / 176	277 / 404
int2float	15 / 14	213 / 208	386 / 474
cavlc	16 / 16	655 / 659	1219 / 1531

Table 1. Circuit Statistics after Initial Dataset Preparation. For each cell, the metric on the left is the one reported in the literature, and the one on the right is the metric obtained by this work.

can see that our preparation is successful, as we have the same metrics reported by both Carpov, et. al. [9] and Lee, et. al. [7]. Furthermore, Table 2 displays our results after running the algorithm described in section 4.3.

Circuit Name	Multiplicative Depth	# of AND Gates	Size
ctrl	5 / 6	109 / 108	NA / 341
dec	3 / 3	304 / 304	NA / 344
router	11 / 16	204 / 338	NA / 859
int2float	8 / 9	216 / 459	NA / 1159
cavlc	9 / 12	669 / 1020	NA / 3051

Table 2. Circuit Statistics after Heuristic-Based Training Loop. For each cell, the metric on the left is the one reported in the literature, and the one on the right is the metric obtained by this work. NA indicates that the metric was not reported.

Initially, our metrics indicate that the addition of program synthesis did not provide much improvement over the handwritten rules presented in Carpov, et. al. [9]. One potential explanation of this could be that the test circuits included in our evaluation were too small for meaningful structures to be identified (i.e. the generated rewrite rules did not have an opportunity to be used because of the specific circuit topologies). However, the results indeed demonstrate that for small circuits with size ≤ 500, we are able to achieve slightly worse than the reported state-of-the-art optimization results.

6 FUTURE WORK / CONCLUSION

Overall, I felt that the project was quite successful considering the amount of boilerplate work that needed to be done with respect to understanding the existing tooling and implementing the right circuit manipulation algorithms even to get started. In hindsight, I think I would've picked a different topic, or at least one closer to the software level, as it was challenging to dive right into circuit optimization without previous experience working with computer aided hardware design. That being said, I think it was a great learning experience reading about innovations at all levels of the hardware-software stack, and it was good that I was able to produce metrics that were roughly equivalent to the best reported in the literature.

6.1 Improving the Existing Rewrite Algorithm

6.1.1 Better Heuristics. As stated in Carpov, et. al.[9], multiple heuristics were employed based on critical path length, critical predecessor and successor count, as well as randomized choice. Given more time, I would like to try a few more of these heuristics to ensure that the algorithm is fully covering the search space. Furthermore, according to Aubry, et. al. [30], there exists a specific arrangement of critical nodes called a critical cone such that applying a cone rewrite operator can lead to a smaller depth. But again, this is another case of a handwritten rule, so I'd be curious to see if the BUS implementation on the Boolean Circuit DSL is able to generate the rule.

6.1.2 E-graph Implementation and More Robust Program Synthesis. If more time were allowed, I would implement e-graphs to maintain and generate the rewrite rules. This would be significantly better, both design and performance wise, because it would automatically allow us to identify the rewrite rule which results in the smallest depth immediately [31]. Furthermore, in our current implementation, we simply pool all rewrite rules into the same training iteration and hope that something good comes out of it. To improve this, I would reexamine the theory behind optimal rewrites, substitutions, and ring theory presented in Lee, et. al [7].

6.1.3 Complexity Optimization. As it stands currently, the algorithm is very slow, on the order of $O(n^3 \cdot \log n)$. The main bottleneck is a result of the search algorithm that modifies the Floyd-Warshall algorithm (i.e. for solving the "All-Pairs Shortest Path" problem). In addition, the reapplication of rewrite rules in our heuristic algorithm until no paths exists for a single training iteration is not only not optimal, as it can cause the search to converge too early, but is also a process that can be improved computationally because current it is not guaranteed to terminate. A final alternative path for improving performance could be rewriting the algorithm to use a divide-and-conquer algorithm reported in Lee, et. al. [7].

6.1.4 Miscellaneous. One final interesting result is the `ValueError: Exceeds the limit (4300) for integer string conversion` exception that was thrown by the `circuitgraph` package when processing large boolean circuits of depth > 20. The exception is thrown due to the number of nodes in the boolean circuit exceeding a set maximum; given more time, I would submit a PR that fixes this maximum naming convention issue and shortens the node names.

ACKNOWLEDGMENTS

I would like to acknowledge Professor Amin for the helpful insights and feedback while I was formulating the scope and implementation of the course project. I would also like to give a huge thanks to the CS252R course staff, Will Byrd and Tyler Holloway, for facilitating the course, along with my peers for providing an enjoyable experience this semester.

REFERENCES

[1] M. Nasr, N. Carlini, J. Hayase, M. Jagielski, A. F. Cooper, D. Ippolito, C. A. Choquette-Choo, E. Wallace, F. Tramèr, and K. Lee, "Scalable extraction of training data from (production) language models," 2023.

[2] K. Cong, R. Geelen, J. Kang, and J. Park, "Efficient and secure k-nn classification from improved data-oblivious programs and homomorphic encryption," Cryptology ePrint Archive, Paper 2023/852, 2023, https://eprint.iacr.org/2023/852. [Online]. Available: https://eprint.iacr.org/2023/852

[3] W.-j. Lu, S. Kawasaki, and J. Sakuma, "Using fully homomorphic encryption for statistical analysis of categorical, ordinal and numerical data," 01 2017.

[4] A. Bain, J. C. Mitchell, R. Sharma, D. Stefan, and J. Zimmerman, "A domain-specific language for computing on encrypted data," *IACR Cryptol. ePrint Arch.*, p. 561, 2011. [Online]. Available: http://eprint.iacr.org/2011/561

[5] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan, "Homomorphic encryption standard," Cryptology ePrint Archive, Paper 2019/939, 2019, https://eprint.iacr.org/2019/939. [Online]. Available: https://eprint.iacr.org/2019/939

[6] A. Viand, P. Jattke, and A. Hithnawi, "Sok: Fully homomorphic encryption compilers," *CoRR*, vol. abs/2101.07078, 2021. [Online]. Available: https://arxiv.org/abs/2101.07078

[7] 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 Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 503–518. [Online]. Available: https://doi.org/10.1145/3385412.3385996

[8] R. K. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *International Conference on Computer Aided Verification*, 2010. [Online]. Available: https://api.semanticscholar.org/CorpusID:1251520

[9] S. Carpov, P. Aubry, and R. Sirdey, *A Multi-start Heuristic for Multiplicative Depth Minimization of Boolean Circuits*, 01 2018, pp. 275–286.

[10] S. Gorantala, R. Springer, S. Purser-Haskell, W. Lam, R. J. Wilson, A. Ali, E. P. Astor, I. Zukerman, S. Ruth, C. Dibak, P. Schoppmann, S. Kulankhina, A. Forget, D. Marn, C. Tew, R. Misoczki, B. Guillen, X. Ye, D. Kraft, D. Desfontaines, A. Krishnamurthy, M. Guevara, I. M. Perera, Y. Sushko, and B. Gipson, "A general purpose transpiler for fully homomorphic encryption," *CoRR*, vol. abs/2106.07893, 2021. [Online]. Available: https://arxiv.org/abs/2106.07893

[11] "Microsoft SEAL (release 4.1)," https://github.com/Microsoft/SEAL, Jan. 2023, microsoft Research, Redmond, WA.

[12] "Tfhe," https://github.com/tfhe/tfhe, Jan. 2023.

[13] N. P. Smart and F. Vercauteren, "Fully homomorphic simd operations," *Des. Codes Cryptography*, vol. 71, no. 1, p. 57–81, apr 2014. [Online]. Available: https://doi.org/10.1007/s10623-012-9720-4

[14] P. Pisa, M. Abdalla, and O. C. M. B. Duarte, "Somewhat homomorphic encryption scheme for arithmetic operations on large integers," 12 2012, pp. 1–8.

[15] S. Carpov, P. Dubrulle, and R. Sirdey, "Armadillo: A compilation chain for privacy preserving applications," in *Proceedings of the 3rd International Workshop on Security in Cloud Computing*, ser. SCC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 13–19. [Online]. Available: https://doi.org/10.1145/2732516.2732520

[16] E. Chielle, O. Mazonka, H. Gamil, N. G. Tsoutsos, and M. Maniatakos, "E3: A framework for compiling c++ programs with encrypted operands," Cryptology ePrint Archive, Paper 2018/1013, 2018, https://eprint.iacr.org/2018/1013. [Online]. Available: https://eprint.iacr.org/2018/1013

[17] A. Viand and H. Shafagh, "Marble: Making fully homomorphic encryption accessible to all," in *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, ser. WAHC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 49–60. [Online]. Available: https://doi.org/10.1145/3267973.3267978

[18] D. W. Archer, J. M. Calderón Trilla, J. Dagit, A. Malozemoff, Y. Polyakov, K. Rohloff, and G. Ryan, "Ramparts: A programmer-friendly system for building homomorphic encryption applications," in *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, ser. WAHC'19. New York, NY, USA: Association for Computing Machinery, 2019, p. 57–68. [Online]. Available: https://doi.org/10.1145/3338469.3358945

[19] E. Crockett, C. Peikert, and C. Sharp, "Alchemy: A language and compiler for homomorphic encryption made easy," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1020–1037. [Online]. Available: https://doi.org/10.1145/3243734.3243828

[20] J. Ma, C. Xu, and L. W. Wills, "Pytfhe: An end-to-end compilation and execution framework for fully homomorphic encryption applications," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023, pp. 24–34.

[21] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "Eva: An encrypted vector arithmetic language and compiler for efficient homomorphic computation," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 546–561. [Online]. Available: https://doi.org/10.1145/3385412.3386023

[22] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "Chet: An optimizing compiler for fully-homomorphic neural-network inferencing," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 142–156. [Online]. Available: https://doi.org/10.1145/3314221.3314628

[23] T. van Elsloo, G. Patrini, and H. Ivey-Law, "Sealion: a framework for neural network inference on encrypted data," *ArXiv*, vol. abs/1904.12840, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:139104359

[24] E. Seligman, T. Schubert, and M. V. A. K. Kumar, "Chapter 8 - formal equivalence verification," in *Formal Verification (Second Edition)*, second edition ed., E. Seligman, T. Schubert, and M. V. A. K. Kumar, Eds. Academic Press, 2023, pp. 239–277. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780323956123000082

[25] R. K. Brayton, R. L. Rudell, A. L. Sangiovanni-Vincentelli, and A. R. Wang, "Mis: A multiple-level logic optimization system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, pp. 1062–1081, 1987. [Online]. Available: https://api.semanticscholar.org/CorpusID:18340064

[26] C. Yu and M. J. Ciesielski, "Formal verification using don't-care and vanishing polynomials," *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 284–289, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:4672386

[27] C. Yu, M. Ciesielski, and A. Mishchenko, "Fast algebraic rewriting based on and-inverter graphs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 9, pp. 1907–1911, 2018.

[28] L. Amarù, P.-E. Gaillardon, and G. De Micheli, "The epfl combinational benchmark suite," 2015. [Online]. Available: http://infoscience.epfl.ch/record/207551

[29] W. Wernick, "Complete sets of logical functions," *Transactions of the American Mathematical Society*, vol. 51, pp. 117–132, 1942. [Online]. Available: https://api.semanticscholar.org/CorpusID:122726706

[30] 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.* Berlin, Heidelberg: Springer-Verlag, 2020, p. 345–363. [Online]. Available: https://doi.org/10.1007/978-3-030-40186-3_15

[31] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "Egg: Fast and extensible equality saturation," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, jan 2021. [Online]. Available: https://doi.org/10.1145/3434304