HELM: Navigating Homomorphic Encryption through Gates and Lookup Tables

Charles Gouert*, Dimitris Mouris*, and Nektarios Georgios Tsoutsos

University of Delaware
{cgouert, jimouris, tsoutsos}@udel.edu

Abstract. As cloud computing continues to gain widespread adoption, safeguarding the confidentiality of data entrusted to third-party cloud service providers becomes a critical concern. While traditional encryption methods offer protection for data at rest and in transit, they fall short when it comes to where it matters the most, i.e., during data processing.

To address this limitation, we present HELM, a framework for privacy-preserving data processing using homomorphic encryption. HELM automatically transforms arbitrary programs expressed in a Hardware Description Language (HDL), such as Verilog, into equivalent homomorphic circuits, which can then be efficiently evaluated using encrypted inputs. HELM features two modes of encrypted evaluation: a) a gate mode that consists of standard Boolean gates, and b) a lookup table mode which significantly reduces the size of the circuit by combining multiple gates into lookup tables. Finally, HELM introduces a scheduler that enables embarrassingly parallel processing in the encrypted domain. We evaluate HELM with the ISCAS'85 and ISCAS'89 benchmark suites as well as real-world applications such as AES and image filtering. Our results outperform prior works by up to 65×.

Keywords: Applied Cryptography · Circuit Evaluation · Hardware security · Homomorphic Encryption · Lookup Tables · Privacy-preserving Computation · Secure Computation · Trustworthy Hardware

1 Introduction

In recent years, the cloud computing paradigm has been widely adopted by a plethora of organizations in order to outsource computationally intensive tasks to powerful, remote servers maintained and operated by third-party service providers. This allows companies to avoid developing and maintaining their own computing infrastructure and provides high degrees of scalability. However, cloud users may be reticent to trust a third party with their data, particularly if it contains proprietary or personal information. The third-party service provider could plausibly view user data residing on their servers if they are incentivized.

Additionally, attackers have increasingly turned their attention to cloud-based infrastructure as each cloud server can potentially host sensitive data from numerous users. In fact, numerous research endeavors have proposed viable attacks in this domain [1–3]. While several solutions to counter these cloud server attacks have been proposed [4, 5], their adoption remains limited. As the security of outsourced data lies solely with the service provider, users must take measures to ensure the confidentiality of their data.

An intuitive solution to the outlined issues is encryption, which can safeguard data both when it is *at rest* and *in transit*. For instance, secure database frameworks like Arx [6] utilize encryption to protect stored data. Encryption prevents both attackers and cloud service providers from accessing plaintext data, ensuring privacy even if the remote servers are compromised. However, traditional encryption techniques have a significant drawback: updating or modifying encrypted data requires the user to download, decrypt, process, and re-encrypt data before uploading it back to the cloud. This is a time-consuming and computationally intensive process, defeating the purpose of outsourcing in the first place.

[*] The first two authors have equal contributions and appear in alphabetical order.

This work has been supported by NSF CAREER award #2239334.

To address this challenge, special privacy-enhancing technologies that protect data while *in use* must be employed to enable the cloud to perform computation directly on encrypted data. One of these techniques is fully homomorphic encryption (FHE), often referred to as the "holy grail" of cryptography [7], which allows arbitrary computation over ciphertexts. This eliminates the previously mentioned lengthy process to update encrypted data (i.e., decrypting, processing, re-encrypting, and uploading back to the cloud). No details regarding the underlying plaintext data are leaked at any point of the computation with FHE aside from the data size and shape (since the algorithm itself is not protected by FHE) [8].

While this technology is very powerful, it poses significant challenges to developers wishing to integrate it into their frameworks. Although there are multiple open-source homomorphic encryption libraries available [9,10], they are challenging to use for programmers without extensive cryptographic knowledge, especially as different libraries offer different APIs and implement different FHE schemes [11–13]. Recent works have attempted standardizing benchmarks and compilers [14–18], however, properly setting cryptographic parameters for security (such as the polynomial ring dimension and ciphertext modulus) and adopting algorithms to the restrictions imposed by encrypted computation still remain challenging tasks. For example, the execution flow of the program needs to be oblivious to the encrypted data; in other words, the algorithm should not make any runtime decisions based on the underlying plaintext value. In many cases, this is not straightforward to address for a given application such as private sorting. Additionally, for many libraries, monitoring ciphertext noise continuously is necessary to determine when special noise-reduction steps are needed for successful decryption.

To solve these problems, we introduce HELM: a framework that automatically converts synthesizable Verilog programs into homomorphic algorithms and seamlessly integrates them with the cutting-edge CGGI scheme (also known as TFHE) that implements operations primarily over encrypted Boolean numbers [19]. HELM is designed to accelerate the encrypted evaluation of Verilog programs while eliminating the challenging learning process associated with FHE. Verilog was chosen as a target front-end because it integrates seamlessly with RTL synthesis and Boolean optimization frameworks, which allows for rigorous optimization of arbitrary FHE programs that utilize CGGI, which exposes encrypted logic gate operations to users. More specifically, HELM features three modes of operation (i.e., gate mode and two look-up table modes that differ in the plaintext encoding strategy) and a scheduler to automatically dispatch the encrypted evaluation of any circuit across multiple CPU threads in parallel. Regarding usability, HELM handles cryptographic parameterization, key generation and management, ciphertext generation, and memory allocation and deallocation in a transparent manner, shielding these complexities from the user. To summarize, our contributions can be summarized as follows:

- We bring to bear decades of hardware design research to optimize homomorphic circuits and allow for efficient private outsourced computation.
- We exploit the inherent circuit parallelism in combinational and sequential netlists to accelerate encrypted evaluation on multi-core systems.
- We investigate multiple ciphertext encodings to evaluate arithmetic and Boolean-intensive circuits with optimal formats.

Roadmap: The rest of the paper is organized as follows: Section 2 provides a brief background on FHE and the CGGI scheme, which is utilized in this work. Section 3 presents our methodology for converting circuits to the encrypted domain, while Section 4 presents an overview of the HELM framework. In Section 5, we present our experimental evaluations with a variety of benchmarks, Section 6 provides comparisons with related works and Section 7 presents our concluding remarks.

2 Preliminaries

2.1 Basics of Homomorphic Encryption

Homomorphic encryption (HE) allows performing operations directly on encrypted data without the need for decryption or exposing the plaintext. These encrypted operations are addition (i.e., `Add`) and multiplication (i.e., `Mult`) and take two ciphertexts as inputs and return a new ciphertext as output. With these operations, one can compute any arbitrary function on the encrypted data. In the encrypted domain, HE allows computing:

$$\texttt{Dec}(\texttt{Add}(\texttt{Enc}(x), \texttt{Enc}(y))) = x + y,$$
$$\texttt{Dec}(\texttt{Mult}(\texttt{Enc}(x), \texttt{Enc}(y))) = x \times y.$$

Since HE does not reveal plaintext data while carrying out computations, this form of encryption enables any party (e.g., companies and individuals) to outsource sensitive data to untrusted third parties, such as a cloud service provider, and wait for the encrypted result. The cloud service provider will have no knowledge about the underlying data as all the operations are homomorphic, i.e., they are carried out on the ciphertexts.

2.2 Homomorphic Encryption Schemes

All homomorphic encryption schemes support computation directly on encrypted data; however, not all HE schemes have the same computational capabilities. In particular, modern HE constructions can be divided into three distinct classes: partial HE, leveled HE, or fully HE.

Partial HE (PHE). PHE is the weakest class and allows for either unbounded addition or multiplication on encrypted data, but not both. As a result, these schemes do not consist of a functionally complete set of operations and are only suitable for specific applications, such as data aggregation [20]. Several schemes exhibit partial HE properties, such as the ubiquitous RSA cryptosystem [21], which allows for multiplication on encrypted data provided that no padding is used, and the Paillier cryptosystem [22] that supports ciphertext addition.

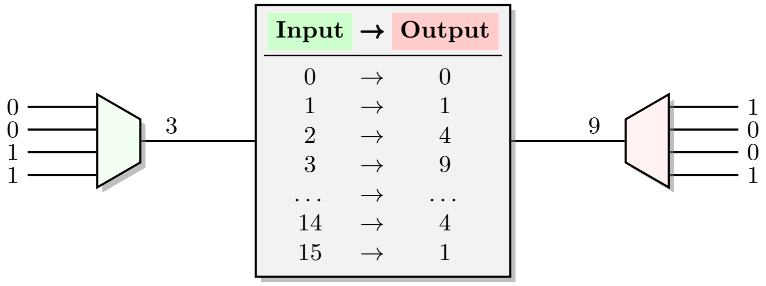
Leveled HE (LHE). LHE, on the other hand, allows for both addition and multiplication in the encrypted domain. However, leveled homomorphic encryption schemes rely on the learning with errors (LWE) [23] or ring-learning with errors (RLWE) problem [24] and random noise is added to encrypted data to guarantee security. Unfortunately, whenever you perform computation with noisy ciphertexts, the target operation affects the noise as well and causes it to grow. In the case of addition, the noise grows slightly, whereas multiplication causes large noise growth. Eventually, if the noise magnitude exceeds a certain threshold, it will corrupt the underlying message and cause an incorrect result upon decryption with a high probability. This restricts the possible number of operations that can be carried out on ciphertexts and the only way to evaluate algorithms with a larger computational depth is to select different parameters that yield larger ciphertexts and slower encrypted operations. For very deep algorithms that require dozens of subsequent multiplications on encrypted data, this approach becomes prohibitively slow. Cryptosystems that are typically used as LHE schemes include BGV [12] and BFV [11] (which encrypt vectors of modular integers), and CKKS [13] (which encrypts vectors of floating point numbers).

Fully HE (FHE). The last, and most powerful form of HE is called fully HE and allows for unbounded multiplication and addition on encrypted data. LHE schemes can be transformed into FHE schemes through the introduction of a *bootstrapping* mechanism that can reset ciphertext noise to nominal levels to allow for additional computation [25]. However, this operation is far more expensive than other encrypted operations; in the case of the previously identified cryptosystems typically used in LHE (i.e., BGV/BFV and CKKS), a single bootstrap can take up to several minutes on a CPU. The DM cryptosystem [26] addressed this issue by introducing a scheme built to dramatically reduce the latency of bootstrapping. Contrary to prior schemes, DM ciphertexts encrypt a single bit of plaintext and the core operations take the form of Boolean gates. This computational model also allows for more flexibility than standard arithmetic operators, as it is possible to directly compute non-linear functions with circuits (instead of using large polynomial approximations) to compute operations such as comparisons and non-linear activation functions in machine learning applications. The CGGI cryptosystem [19] is similar to DM and incorporates an even faster bootstrapping mechanism that can be evaluated in approximately 10 milliseconds on a CPU. Currently, this cryptosystem has the most efficient bootstrapping technique in terms of latency and was therefore chosen as the target cryptographic backend for this work.

2.3 Programmable Bootstrapping

A crucial feature of the DM and CGGI FHE cryptosystems is the ability to evaluate any non-linear function during bootstrapping. This takes advantage of the inherent programmability of the bootstrapping employed in these schemes and serves as a generalization of the gate bootstrapping case. A polynomial with crafted coefficients that encodes the set of desired output messages is rotated by an encrypted value and the first encrypted coefficient corresponding to the constant term of the polynomial is extracted. These two procedures, called *blind rotation* and *extraction*, form the core bootstrapping steps.

By encoding chosen lookup table (LUT) entries in the coefficients of the polynomial to be rotated, one can create a mapping between the underlying plaintext value of a ciphertext to a valid encryption with a value dependent on the selected coefficient after the blind rotation and extraction. This allows computing arbitrary univariate functions by evaluating a function in the plaintext domain across all possible inputs and encoding them in the polynomial utilized during bootstrapping. This generalized bootstrapping technique is called *programmable bootstrapping* [27,28]. Although the LUT needs to be relatively small to maintain small and efficient cryptographic parameter sets, it can encode any arbitrary function and thus lead to a significant performance boost as it replaces expensive operations that otherwise would require multiple additions and multiplications. An example of a 4-to-4 LUT that encodes the function $f(x) = x^2$ is demonstrated in Fig. 1.



Input	→	Output
0	→	0
1	→	1
2	→	4
3	→	9
...	→	...
14	→	4
15	→	1

Fig. 1. Example of Lookup Table (LUT) Evaluation.

2.4 Lossless Bidirectional Bridging (LBB)

The underlying plaintext data type of a ciphertext dictates which type of operations can be conducted on the encrypted data. For instance, if a ciphertext encodes an integer, adding or multiplying the ciphertext polynomials corresponds to addition or multiplication over the underlying plaintext values. In the case of CGGI, a single ciphertext can encrypt multi-bit inputs (depending on the specific parameters chosen) and the programmable bootstrapping operation can readily map ciphertexts encoding multi-bit values to another multi-bit result. Indeed, this scenario is far more efficient for realizing an N-to-N lookup table than running N programmable bootstraps over N binary ciphertexts, as a single bootstrap is needed with a multi-bit encoding.

The primary challenge of leveraging this approach is that the multi-bit output is not independent, and the individual bits of the underlying plaintext value can not be simply extracted. To access the individual bits of the multi-bit output, one can perform a *bridging* procedure that converts the lookup table output ciphertext to a vector of ciphertexts encoding each bit, to allow for correct routing to downstream lookup tables. A high-level depiction of the bridging technique is shown in Figure 1, where a vector of four ciphertexts encrypting one bit each (that collectively encode the binary value `0011`) is combined to form a single ciphertext representing an encryption of 3 to serve as input to a homomorphic LUT. Using the reverse procedure, the single ciphertext output of the lookup table that encodes 9 can be decomposed into the binary string `1001`. We note that some earlier bridging techniques, such as those employed in FHE-DiNN [29] and REDsec [30], result in a precision loss, conversely, this work focuses on lossless bridging that maintains the same precision between inputs and outputs. Further details about our specific bridging methodologies can be found in Section 3.

2.5 Homomorphic Encryption Libraries

To date, several open-source homomorphic encryption libraries have been released. HElib [31], the first publicly available HE library, performs mathematical operations on multi-bit ciphertexts and can compute any polynomial function of arbitrary degree. However, this library has several drawbacks that make it impractical for general-purpose computation. First, bootstrapping speeds and evaluation times remain high compared to newer libraries. In addition, HElib exposes a complex API that requires users to tune multiple security parameters, as well as manually keep track of ciphertext noise and determine when bootstrapping should be applied.

In 2018, Microsoft released its own homomorphic library called SEAL [9]. While this library provides users with a simpler API that allows conducting additions and multiplications on ciphertexts, it is not capable of FHE in its current state. Instead, SEAL provides *leveled* homomorphic encryption, which does not offer a bootstrapping function and therefore allows for only a finite number of operations on ciphertexts. While this may be suitable for some applications, it is not sufficient for general-purpose computation (as in HELM) that requires the support of circuits of arbitrary depth.

FHEW and its DM scheme, as described previously, initially implemented only `NAND` evaluations on encrypted bits, while in 2017, increased functionality was added to the library, including `NOR`, `OR`, `AND`, and `NOT` evaluations. While DM is fully homomorphic and provides fast bootstrapping speeds compared to prior schemes, its successor, CGGI, boasts even faster speeds [32].

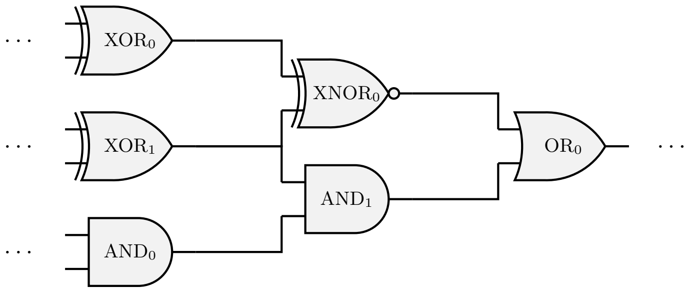


Fig. 2. Example of Gates Circuit.

TFHE is a fast FHE library first released in 2017 that implements the aforementioned CGGI scheme. TFHE is a successor to FHEW and operates exclusively on Boolean circuits. All ciphertexts are encrypted as binary values: plaintext data is converted to binary, encrypted bit by bit, and stored in a ciphertext array that has a size of approximately 2.2 KB×N, where N is the number of bits in the plaintext. The TFHE library offers the ability to carry out any logic gate function on ciphertexts and handles bootstrapping automatically after each gate evaluation (except the `NOT` gate that does not need bootstrapping). Since TFHE supports the evaluation of all types of logic gates (i.e., it offers multiple functionally complete sets of operations), it supports arbitrary computation on encrypted data. This property, as well as the fact that it can evaluate circuits of arbitrary depth, classifies it as fully homomorphic. Although TFHE provides competitive bootstrapping speeds and gate evaluation times, it has been outperformed by TFHE-rs [33], a pure Rust implementation of the TFHE scheme for both Boolean and integer arithmetic. Additionally, TFHE-rs enables programmable bootstrapping, a core feature of the CGGI cryptosystem that is missing from the original TFHE library. In addition to standard Boolean circuits (e.g., Fig. 2), the programmable bootstrapping feature enables the evaluation of LUT-based circuits such as the one depicted in Fig. 3. Thus, TFHE-rs is an ideal candidate for use with HELM.

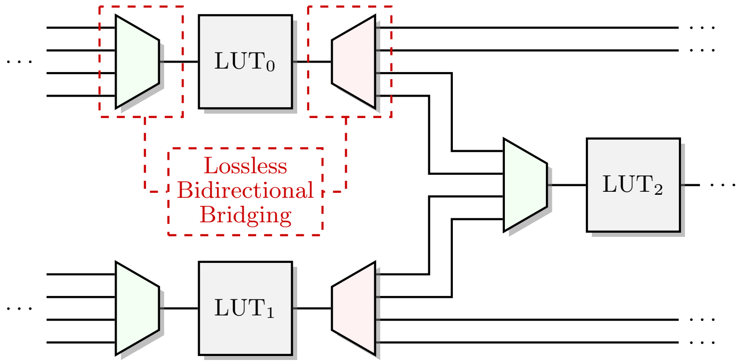
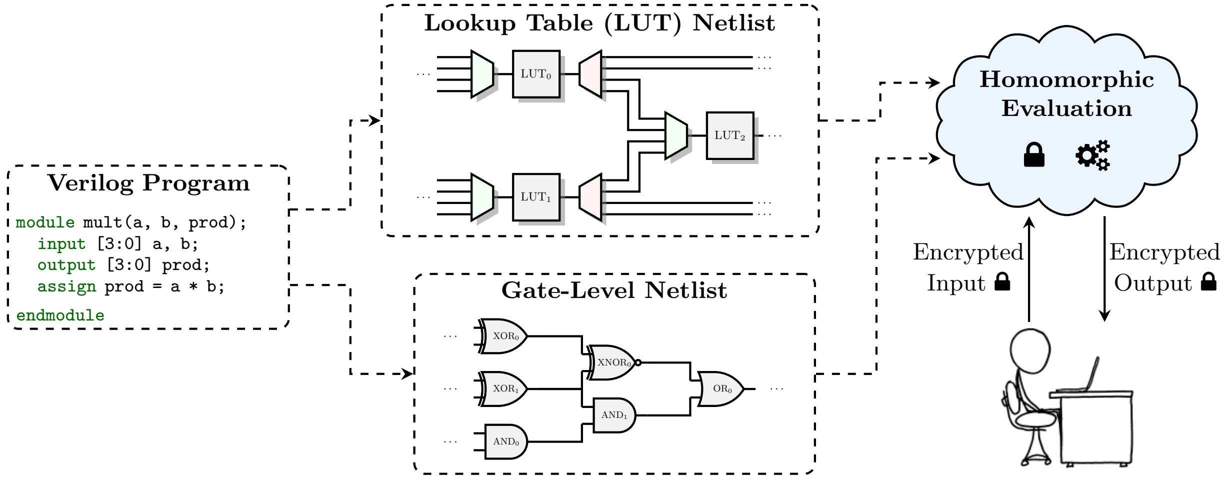


Fig. 3. Example of Lookup Tables (LUTs) Circuit.

3 Automatic Translation of Circuits to the Encrypted Domain

HELM offers the ability to transform hardware description language (HDL) designs to an equivalent homomorphic circuit that performs computation on encrypted data, which can be evaluated by an untrusted remote party. To accomplish this, the first step is to use *logic synthesis* to convert Verilog programs to Boolean netlists consisting of logic gates, lookup tables, and primitive memory structures (like flip-flops). Next, the generated netlist serves as an input to HELM's custom compiler that parses the circuit, determines a correct execution order of the gates, and generates an equivalent and efficient homomorphic program. An outline of our framework is illustrated in Fig. 4.



```
module mult(a, b, prod);
  input [3:0] a, b;
  output [3:0] prod;
  assign prod = a * b;
endmodule
```

Fig. 4. HELM Outline. Verilog designs are converted to netlists and then passed to the HELM execution engine. The execution engine administers keys, receives inputs from the user, and generates an encrypted circuit for the cloud to evaluate (which can be either a LUT-based or gate-based netlist). When the cloud finishes the circuit evaluation, the resulting ciphertext is sent to the user.

3.1 RTL Synthesis

To handle synthesis, HELM's back-end uses the Yosys Open SYnthesis Suite [34], which is an open-source toolchain performing RTL synthesis as well as circuit optimizations. Under the hood, many synthesis operations such as technology mapping (the process of converting circuit cells to a specified gate technology) are handled by the ABC framework [35]. Importantly, ABC provides the ability to map circuits of logic gates to multi-bit LUTs as this is a common use-case in other contexts, such as generating FPGA-compatible netlists. Our framework receives Verilog source code files as input and instructs the Yosys back-end to apply the following algorithms:

1. Perform HE-friendly optimizations including removing unused wires, replacing process blocks with flip-flops, and logic minimization;
2. Map cells to standard logic gates and small multiplexers, or many-to-one lookup tables;
3. Write the resulting netlist as a structural Verilog file.

Importantly, due to the differing ciphertext formats required for logic gate evaluation and multi-bit LUTs, we do not support mixed circuits that consist of both LUTs and Boolean gates; however, both types of circuits can contain sequential circuit elements, such as flip-flops.

3.2 Preprocessing Verilog to an FHE-friendly Format

The first step involves running the netlist through a preprocessor that transforms it into a custom format for efficient processing by the HELM execution engine. Specifically, the gates and LUTs are formatted in a consistent manner with a unique cell identifier and all wire inputs appear before outputs. There are three primary considerations when preparing a Verilog file input to a format that can be properly parsed by the HELM execution engine.

Consideration 1: Constant wires. In Yosys, some wires are driven by constant signals (i.e., `1` or `0`) after the synthesis process. To capture this case, the preprocessor inserts two custom cells that correspond to trivial encryptions of `0` or `1`. Generating trivial encryptions involves encoding the bit as a polynomial of the same degree as the secure ciphertext inputs. However, these trivial encryptions are noiseless and are not encrypted with the secret key, so can be readily generated by the computing party. These noiseless ciphertexts are necessary to generate in order to allow the constant/non-secret value to serve as an auxiliary input to homomorphic operations with securely encrypted data. Notably, operations that mix trivial encryptions with secure encryptions result in a secure encryption and the security of the user-supplied input is not impacted in any way. When the execution engine dispatches these two nodes to workers, the constant value can be used in gate evaluations with secure ciphertexts. Since these node types have no inherent dependencies, they are always among the first operations evaluated by HELM during circuit evaluation.

Consideration 2: Direct I/O connections. In some circuit designs, an input wire may directly route to an output. With no intermediate node, HELM can easily miss the association between the two wires as no operation exists between them. We utilize a similar strategy employed for constant wires and introduce a buffer cell that maps the input directly to the output. Internally, HELM evaluates this cell by performing a ciphertext-ciphertext copy operation instead of the trivial encryption needed for dealing with wires driven by constant signals. These specific buffer cells are also evaluated by HELM in the first set of cell evaluations since they only depend on input wires, which are available as soon as evaluation begins.

Consideration 3: Non-unique wire identifiers. Yosys tends to generate chains of identical wires when synthesizing large and complex programs, resulting in convoluted netlists. If the set of identifiers referring to a single wire is not collapsed to a single unique identifier, HELM may miss the association between them and attempt to treat each identifier as a separate ciphertext object. To deal with this problem, HELM iterates through all of the wire mappings provided by Yosys and replaces all identical mappings with a single unique identifier or an output wire identifier, if one is present at the end of the chain of identifiers.

3.3 A Strawman Approach to Encrypted Circuit Evaluation

A straightforward way to evaluate a circuit homomorphically involves a direct translation from the Yosys output netlist to an FHE program. With this method, the netlist can be parsed and sorted topologically to resolve dependencies. Then, each gate can be converted to a few lines of FHE code that invokes the corresponding library functions to evaluate the operation. The resulting FHE code can finally be readily executed by a third party, but the evaluation will be entirely sequential as there is no notion of which gates can be executed concurrently. While this approach can perform relatively well for thin circuits, wide circuits will result in prohibitive latencies as each individual gate requires several milliseconds to execute. This approach represents a baseline incorporated by prior work [36].

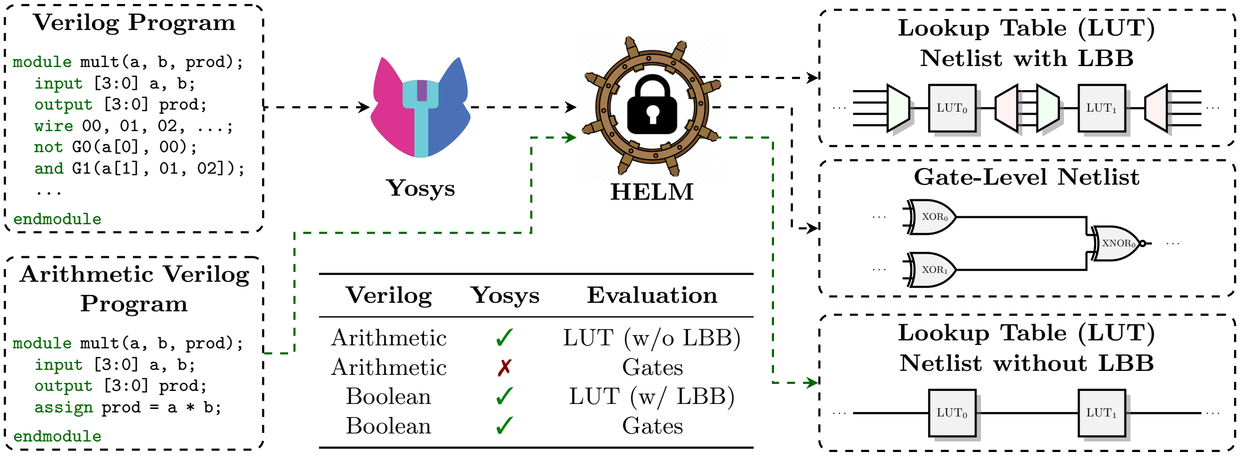
3.4 Combinational Circuit Conversion

First, the HELM execution engine creates associations between all cells and wires. Importantly, the cells present in the graph representation generated by the preprocessor are not sorted and appear in the same order as the raw Yosys output (with the exception of buffer gates, which are prepended to the start of the module). Our execution engine performs a topological sort across the unordered set of cells and divides the cells into sets of *levels* (shown in Algorithm 1). Thus, after sorting, each level in HELM consists of gates that can be evaluated concurrently. Each gate appears in level $N + 1$ if it directly depends on a prior gate from level N.

Before evaluation, the input wires of the circuit are loaded with encrypted data supplied by the client. Additionally, any constant wires that are known at compile time and are not secret values are loaded with trivial encryptions of `0` or `1`. To evaluate the circuit, HELM iterates through all previously identified circuit levels and dispatches the homomorphic operations equally across all available CPU threads. We observe that both LUT-based and gate-based circuits exhibit ample parallelism and most circuit levels are wide enough to effectively utilize all CPU threads for our representative benchmarks.

Algorithm 1 Partition Circuit into Levels

```
Input: cells                                              ▷ An unordered set of gates.
procedure CIRCUIT-SORT(cells)
    for gate in cells do                                  ▷ Find direct ancestors.
        for wire in gate.inputs do                        ▷ Check origin of inputs.
            if wire is output from another gate then
                gate.depends_on ← wire.originator
    level ← 0                                             ▷ Counter for levels.
    while unevaluated gates remain do
        if gate.evaluated = True then
            continue                                      ▷ Gate exists in a level.
        for gate in cells do
            if gate.depends_on = None then                ▷ Input gates.
                gate.evaluated ← True
                levels[level] ← levels[level] + gate      ▷ Append gate.
            else
                ready ← True
                for prev_gate in gate.depends_on do
                    if prev_gate.evaluated = False then
                        ready ← False                     ▷ Ancestor not ready.
                if ready = True then
                    gate.evaluated ← True
                    levels[level] ← levels[level] + gate
        level ← level + 1
    return levels
```



```
module mult(a, b, prod);
  input [3:0] a, b;
  output [3:0] prod;
  wire 00, 01, 02, ...;
  not G0(a[0], 00);
  and G1(a[1], 01, 02]);
  ...
endmodule
```

```
module mult(a, b, prod);
  input [3:0] a, b;
  output [3:0] prod;
  assign prod = a * b;
endmodule
```

Verilog	Yosys	Evaluation
Arithmetic	✓	LUT (w/o LBB)
Arithmetic	✗	Gates
Boolean	✓	LUT (w/ LBB)
Boolean	✓	Gates

Fig. 5. HELM Modes. HELM can process both arithmetic and Boolean circuits. Arithmetic circuits bypass the logical synthesis step provided by Yosys and can be directly evaluated by HELM, which uses non-LBB LUTs to evaluate them. On the other hand, circuits with bitwise operations are synthesized by Yosys and then the resulting netlist is processed by Yosys, which will evaluate them with LBB LUTs or encrypted logic gates depending on the technology mapping requested by clients.

3.5 Sequential Circuit Conversion

Evaluating sequential circuits in the encrypted domain requires more considerations than purely combinational circuits. For one, clock gating proves a difficult challenge; before the clock signal can serve as a homomorphic gate input, it must be encrypted. This can be done on the client side, where a high number of ciphertexts encoding 0 and 1 are generated prior to circuit evaluation. However, it is much more efficient in terms of both computational and communication overheads to have the computing party that evaluates the FHE operations generate trivial encryptions of the current value of the clock signal since this is not a secret value.

In addition to the clock-gating challenge, the CGGI cryptosystem does not natively offer support for sequential circuit components such as flip-flops (FFs). Thus, to incorporate FF functionality into homomorphic circuits, HELM effectively unrolls the circuit for each requested clock cycle and propagates FF inputs to outputs at the end of each cycle. Specifically, all gates are duplicated C times, where C is the maximum number of cycles required to generate the expected output and is configurable by the client.

4 Oblivious Circuit Execution with FHE

The HELM execution engine consumes a pre-partitioned circuit composed of levels that indicate sets of gates that have no interdependencies and can thus be executed in parallel. HELM takes advantage of this and distributes gates (which are computationally expensive relative to plaintext evaluation) to different CPU threads. Assuming a configuration with T threads and a circuit level with G gates, we can achieve a speedup very close to $T\times$ compared to a single-threaded approach provided $G \gg T$.

In the remainder of this section, we discuss the evaluation strategies and differences between circuits consisting of primitive logic gates and those consisting of lookup tables. An overview of the HELM evaluation modes that can be used to evaluate these two types of circuits is shown in Fig. 5. We note that the methodologies for formulating combinational and sequential circuits, discussed in Section 3, are equally applicable to both LUT-based and gate-based circuits. Lastly, we briefly outline a verification mode to rapidly confirm the functionality of the encrypted circuits.

4.1 Gate-based Circuit Evaluation

The CGGI cryptosystem provides mechanisms for evaluating all basic logic gate operations. All these operations are supplied directly by the TFHE-rs library. When a worker thread spawned by the HELM execution engine is assigned a logic gate evaluation, it first fetches the input ciphertexts, invokes the homomorphic gate primitive, and generates an output ciphertext with the result of the computation.

The computational cost of the available gates falls into one of three categories depending on the number of inputs. Single input gates (i.e., inverters and buffers) are very low cost as they do not require bootstrapping and result in no noise accumulation. Two input gates, which encompass the majority of supported CGGI operations, exhibit roughly the same cost as each involves a linear combination of the inputs followed by a bootstrapping and key-switching operation. Lastly, the encrypted multiplexer is the only natively supported three-input logic gate (taking in an encrypted select signal as well as two data inputs) and requires two bootstraps to evaluate, along with multiple linear combinations between the input ciphertexts. As a result, this gate is approximately twice as costly as a standard two-input gate and is the most expensive Boolean primitive in CGGI.

4.2 LBB LUT-based Circuit Evaluation

The powerful lookup table capabilities provided by the CGGI programmable bootstrapping primitive maps ciphertexts that encode multiple bits to a desired value, which also takes the form of a multi-bit ciphertext. However, it is not directly compatible with complex circuits consisting of many lookup tables as some output bits may be routed to different destinations (as shown in Fig. 3) and there are no straightforward ways to extract a single encrypted bit from a multi-bit ciphertext without evaluating more programmable bootstraps for the extraction.

Our LBB Approach. To overcome this, we leverage *lossless bidirectional bridging* (LBB) that allows us to convert multi-bit ciphertexts to encryptions of individual bits and vice versa. The forward LBB algorithm involves a sequence of constant multiplications with powers of 2 and accumulation to convert a vector of ciphertexts encoding bits to a single integer ciphertext. We note that the noise accumulation is relatively small and exhibits a multiplicative depth of 1, which is easily mitigated by the subsequent lookup table evaluation. To isolate the individual bits of a multi-bit ciphertext after the lookup evaluation, we can perform a series of parallelizable programmable bootstraps that evaluate the function `f(x, y) = x >> y & 1`, where `x` is the encrypted multi-bit value and `y` is a plaintext constant representing the desired bit position to extract.

4.3 LUT-Based Circuit Evaluation Without LBB

Our LBB mechanism provides a way to isolate and combine individual bits by manipulating the underlying encoding of ciphertext objects. We note that this mechanism adds overheads in the form of noise accumulation and additional computation. If a set of input wires are always routed to the same circuit nodes, there is no need to decompose them with LBB to route each wire independently. This pattern is extremely common in *arithmetic circuits*, where each circuit node is an addition, multiplication, subtraction, or division operation on multi-bit values.

As such, HELM supports these circuits in the form of behavioral Verilog, which can be processed without performing the logic synthesis steps with Yosys and evaluated directly with the execution engine. We note that this mode does not support bitwise operations and can only be evaluated if the Verilog program is composed entirely of multi-bit arithmetic of a uniform size (e.g., 8 bits, 16 bits, etc.).

With non-LBB LUTs (also referred to as *arithmetic mode*), all ciphertexts encode N bits of data and can be conceptually viewed as encrypted variants of N-bit unsigned integers. HELM supports integers with word sizes of powers of 2 up to 128 bits of precision. Addition and subtraction between two multi-bit ciphertexts can be done without a homomorphic LUT entirely, as the primitive addition operation is defined in CGGI and entails simply performing element-wise additions between ciphertext polynomial coefficients (the same strategy also applies to subtraction). As a result, these two operations are orders of magnitude faster than adding or subtracting two N-bit encrypted numbers stored in vectors of single-bit ciphertexts. On the other hand, the non-linear multiplication and division operations are more challenging and comprise the majority of the execution time of arithmetic circuits. In HELM, both of these operations are implemented as a series of LUTs with programmable bootstrapping and linear operations between the ciphertexts. We note while other cryptosystems, such as BGV and CKKS, are well-suited for arithmetic-style operations, they exhibit prohibitively long latencies for bootstrapping and are incapable of supporting the division operation without incorporating an expensive and noisy polynomial approximation.

4.4 Circuit Verification using Debug Mode

The HELM framework provides users with a convenient method for testing the correctness of a homomorphic circuit before outsourcing it to a third party. This saves users from the cost and time required to deploy potentially faulty code to the cloud. To enable this debugging functionality, HELM offers additional verification elements: the user's private key is read in by the program to assist with decryption and users are prompted to directly input plaintext values that are immediately encrypted with the private key and loaded into the circuit's input wires. Once the circuit evaluation has been completed, the private key is used to decrypt all output wires and print the corresponding plaintext outputs.

To rapidly verify the accuracy of the circuit in debug mode, HELM can generate trivial encryptions of the circuit inputs. Normally, this technique is used to encrypt non-sensitive constant values for computation with sensitive encrypted ciphertexts. The execution overhead for FHE gates processing these trivial ciphertexts is almost negligible, at approximately 10 microseconds per gate evaluation; this is three orders of magnitude faster than the typical FHE gate evaluation speed of 13 ms [19]. Using this verification capability, users can evaluate the correctness of FHE circuits very efficiently. We remark that HELM's debug mode is intended for local use only, as it is insecure to trivially encrypt sensitive data while outsourcing to the cloud.

5 Experimental Evaluation

5.1 Implementation

HELM was implemented in Rust (v1.72) and uses the Yosys Open SYnthesis Suite framework [34] (v0.9) for synthesizing Verilog programs and converting them to EDIF netlists. Yosys features various parameters that produce different netlists depending on the user's preferences (e.g., optimize for performance, space, etc.). In this work, we synthesized our benchmarks for both LBB LUTs and logic gates with the `proc`, `flatten`, and `synth` flags. To get a netlist with solely gates, we use the Yosys `abc` wrapper to perform a technology mapping to standard cells with `abc -g simple`. Conversely, we use a different technology mapping strategy to get a netlist of LUTs in the form of `abc -lut X` where `X` indicates the width of the LUT inputs. This capability is naturally supported by `abc` as it is commonly used for synthesizing netlists for FPGA targets. HELM also relies on TFHE-rs (v0.3.1) for the encrypted evaluation of gates and lookup tables [33]. The HELM framework is embarrassingly parallelizable as all the gates and lookup tables that are in the same depth of the circuit are executed in parallel. Additionally, as the encrypted evaluation of the LUTs is more expensive than the evaluation of single gates, each LUT evaluation itself uses multiple cores.[1]

5.2 Experimental Setup

For our experiments, we use the combinational and sequential circuits from the ISCAS '85 [37] and ISCAS '89 [38] benchmark suites. In addition, we run three variants of the AES algorithm (i.e., AES-128 with key scheduling, without key scheduling, and just the core AES algorithm which is agnostic to the specific AES instantiation), the CRC-32 benchmark for a stream of 1 KB of data, multiplier circuits of various sizes,

[1] The HELM framework is open source and located at `https://github.com/TrustworthyComputing/helm`, while our Verilog designs and synthesized netlists along with our preprocessor are available at `https://github.com/TrustworthyComputing/hdl-benchmarks`.

matrix multiplication over square matrices, blur filters over encrypted images, the chi-square test, as well as the squared Euclidean distance benchmark. These are widely used real-world applications that can also be used to stress test HELM.

We compare HELM with two state-of-the-art works, Romeo [36] and Google Transpiler [16]. All experiments were performed on an `c5.12xlarge` AWS EC2 instance with 48 virtual cores running Ubuntu 22.04. All TFHE-rs security parameter sets utilized correspond to approximately 128 bits of security under the RC.BDGL16 [39] cost model of lattice estimator [40] at the time of writing, which indicates the difficulty of breaking the underlying RLWE instance for specific parameter sets. Lastly, the reported times for the sequential ISCAS benchmarks are averaged over 10 cycles and represent the amortized cost per cycle.

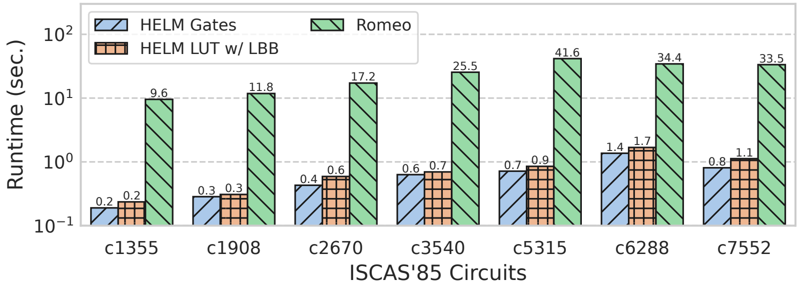


Fig. 6. Encrypted circuit evaluation times for the ISCAS'85 benchmark suite. We remark that the Google Transpiler is only compatible with C/C++ programs, and does not support Verilog circuits as inputs.

5.3 ISCAS'85 and ISCAS'89 Circuits

The homomorphic circuit evaluation times for the ISCAS'85 combinational benchmarks are presented in Fig. 6. Our results show an approximately linear increase in execution time with the number of evaluated gates. Nevertheless, the evaluation time for different gates is not the same. For instance, inverters are evaluated much faster than other logic gates because no bootstrapping is required for this operation. As illustrated in the graph, the `c5315` circuit incurs longer evaluation times than the two largest circuits despite its smaller size. This deviation from expected behavior is attributed to the proportion of inverter gates to the overall number of gates in the circuit. Indeed, the two largest circuits contain approximately 34% inverters while `c5315` contains about 25% inverters. Likewise, the results for the ISCAS'89 sequential circuit benchmarks are presented in Fig. 7. These numbers show the amortized execution cost per cycle (i.e., one complete circuit evaluation), and this cost was amortized over ten clock cycles. In all, we observe that execution times for both benchmark suites scale linearly with the total number of gates and the distribution of the gate types can influence latency to a lesser extent, as we observed with circuits with a higher percentage of `NOT` gates.

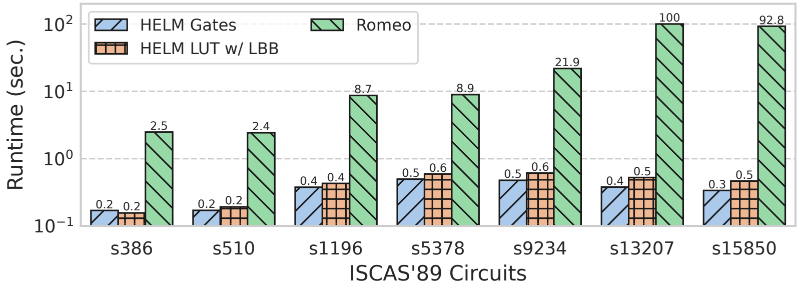


Fig. 7. Amortized evaluation time per cycle (over 10 cycles) for encrypted circuits from the ISCAS '89 benchmark suite. As already mentioned, the Google Transpiler is only compatible with C/C++ programs, and does not support Verilog circuits as inputs.

5.4 Real-world Benchmarks

This class of benchmarks encompasses large workloads that either form key primitives of encrypted applications or constitute useful applications that are representative of what can be accomplished with encrypted computing today.

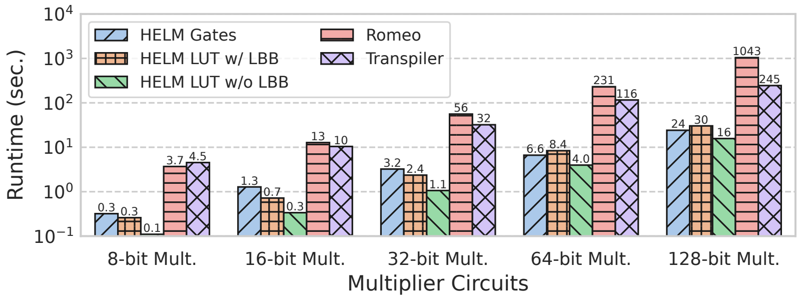


Fig. 8. Encrypted circuit evaluation times for {16, 32, 64, 128}-bit multipliers.

Multi-bit multipliers. Large, multi-bit multipliers form the basis of many applications related to linear algebra and machine learning. As such, we demonstrate the evaluation of these multipliers of sizes up to 128-bit encrypted operands, shown in Fig. 8. We observe that gate mode outperforms the LUTs with LBB and also exhibits better scaling. We chose an LUT width of 2 for the LBB LUTs, as this yields the best performance of the LUT sizes that can be generated by Yosys (up to width 6). However, the LUT without LBB mode performs significantly better than both modes as they can efficiently utilize a small set of large LUTs that are highly parallelizable to perform the multiplication operation. For instance, for a 32-bit multiplication, the LUT without LBB mode is 3× faster than the gate mode. In all modes, HELM outperforms both Romeo [36] by up to 65× (which is entirely sequential by design) and the parallelized Google FHE Transpiler [16] by up to 33×.

Matrix multiplication. Similarly to the previous primitive multiplication benchmark, matrix multiplications form key building blocks of larger, more complex applications and are significantly more computationally expensive than the standard multiplier circuit, which is invoked internally. We study the cost of matrix multiplication across square 5×5 and 10×10 matrices, where each matrix element is a 16-bit unsigned integer. In Table 1, we observe that the LUT encrypted evaluation without LBB is the fastest as this mode directly computes the multiplications as arithmetic operations instead of multiplication circuits. For that reason, the size of the 5×5 netlist in this mode consists of ~ 200 operations, while for the LUTs with LBB and the gates modes the sizes are $\sim 98,000$ LUTs and $\sim 95,000$ gates, respectively. For the 10×10 square matrix multiplication benchmark we observe an approximately $10\times$ blow-up in the netlist sizes and $8 - 9\times$ increase in the runtime. Both the gates and LUTs with LBB perform very similarly, with the LUTs with LBB being marginally faster for the 5×5 matrix product and the gates slightly outperforming the LUTs with LBB for the 10×10 product.

Cyclic Redundancy Check (CRC). In the context of encrypted computing, CRC can be used as both an integrity mechanism as well as an encrypted hash function to obliviously check if two sets of encrypted data encode the same underlying plaintext. The CRC-32 algorithm is primarily composed of bitwise operations, which are well-suited for the CGGI cryptosystem and can be readily evaluated with HELM's LBB-LUT and gate modes. We report the timings for running the CRC-32 algorithm homomorphically over an input set encrypting one kilobyte of plaintext data in Table 1. We observe that HELM in LUT with LBB mode offers the best performance for the CRC-32 evaluation, which aligns with previous results for circuits with primarily bit-wise operations. This particular benchmark results in a fairly narrow, but very deep overall circuit as each clock cycle is unrolled and a block of data is processed sequentially. For this benchmark, we employ a word size of 8 bits as all of the CRC operations occur at the byte level. In this case, HELM in LUTs with LBB mode outperforms the gate mode by approximately 17%.

Chi-squared (χ^2). The Chi-squared test is an important statistical computation used to determine the association between two variables. The test consists of a series of strictly arithmetic operations, making it well-suited for arithmetic mode, which outperforms the other two HELM variants for this application (Table 1). The LUT with LBB mode yielded the worst performance, but still completed the evaluation in approximately 10 seconds. For this benchmark, we used a word size of 32 bits as it corresponds to the bit size of a standard integer and is commonly used for χ^2.

Squared Euclidean distance. The squared Euclidean distance is a mathematical expression for calculating the distance between two points in an n-dimensional space and has various applications, including facial recognition [41]. In our particular scenario, we examine two n-dimensional points denoted as $V = (v_1, v_2, \ldots, v_n)$ and $U = (u_1, u_2, \ldots, u_n)$ in Euclidean space and run experiments for $n = 32$ and $n = 64$. We observe in Table 1 that the arithmetic mode performed best and is approximately 30% faster than the gates and LUTs with LBB because the algorithm is composed of numerous multiplications, additions, and subtractions. To avoid overflow due to the multiplications, we chose a 32-bit word size for this particular benchmark.

Image filters. Lastly, image processing is an exciting application for FHE as it enables cloud services that perform transformations such as sharpening, blurring, and color correction directly on encrypted images. With this approach, no information can be gleaned or leaked regarding the client images except for the dimensions of the image itself. As a representative example, we include two types of blurring filters that operate over arbitrary-size grayscale images.

The first filter consists of a box blur that computes a succession of 3 × 3 average blur kernels across the encrypted image. Each kernel computes the average of the 9 input pixels to generate each output pixel of the resulting, blurred image. The second filter is a Gaussian blur that applies a 3 × 3 filter where each entry is a power of 2 [16]:

$$F = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 2^0 & 2^1 & 2^0 \\ 2^1 & 2^2 & 2^1 \\ 2^0 & 2^1 & 2^0 \end{bmatrix}.$$

We take advantage of the form of the elements of F to compute the Hadamard product (i.e., slot-wise multiplications) as a series of bit shifts, which are efficient in the CGGI cryptosystem. Then, each entry is summed and the resulting value is normalized by computing a right shift by 4 (i.e., division by 16).

Each pixel is represented as an encryption of an 8-bit value in HELM's arithmetic mode that corresponds to the value of the grayscale color channel which can range from 0-255. However, it is not sufficient to use FHE parameter sets that yield 8 bits of precision, as computing the average of nine 8-bit numbers generates an intermediate sum that requires 12 bits of precision to avoid overflows. As such, we use a parameter set corresponding to 16 bits of precision for the default blurring filter. For further optimization, we also introduce an optional compression step on the client side to reduce the bit size of each pixel to 4 bits (achieved using a linear transformation to map the 0-255 grayscale color channel to the range 0-15). With this technique, we can utilize a smaller FHE parameter set that yields 8 bits of precision without potentially overflowing the intermediate sum. Upon decryption, the client can perform another linear transformation to scale the grayscale channel values back to 0-255.

The performance results for both filters with compression for a 64 × 43 grayscale image are depicted in Table 1. Unlike the AES and CRC benchmarks, which require bitwise operations, the two algorithms are composed exclusively of arithmetic operations and bit shifts which enables HELM to evaluate them using the

Table 1. Evaluation times for realistic benchmarks.

Benchmark	Word Size (Bits)	HELM Gates		HELM 2:1 LUT w/ LBB		HELM LUT w/o LBB		Romeo Eval. (sec.)
		Eval. (sec.)	Num Gates	Eval. (sec.)	Num LUTs	Eval. (sec.)	Num LUTs	
c7552 (ISCAS'85)	1	0.8†	1.32 K	1.12	0.96 K	N/A‡	N/A	33.45
s15850 (ISCAS'89)	1	0.34†	0.48 K¶	0.46	0.44 K¶	N/A‡	N/A	92.76
AES Core	8	8.96	15 K	8.3†	14.7 K	N/A‡	N/A	187.3
AES-128 w/ Key Sched.	8	116.8	196 K	104.1†	152.7 K	N/A‡	N/A	2490
AES-128 w/o Key Sched.	8	94.3†	159 K	85.5†	186.1 K	N/A‡	N/A	2179.2
CRC-32 (1024 cycles)	8	1009	728.1 K	859.7$^{\dagger *}$	700.4 K	N/A‡	N/A	DNF‡
Box Blur (64 × 43 opt)§	8	449.6$^{\dagger *}$	806 K	405.6$^{\dagger *}$	732 K	968.9	24768	DNF‡
Gauss. Blur (64 × 43 opt)§	8	337.9	568 K	307†	553.7 K	586.4	38528	DNF‡
128-bit Multiplier	128	24	49.4 K	31.8	33.3 K	16†	1	1043
Matrix Mult. (5x5 × 5x5)	16	55.7	95.9 K	55.6	98.8 K	41.8†	0.2 K	1287.3
Matrix Mult. (10x10 × 10x10)	16	447.3	774 K	447.6	796.1 K	351.9†	1.9 K	DNF‡
Chi-squared	32	9.9	17.1 K	10.1	17.1 K	6.8†	14	205.2
Square Euclid. dist. ($n = 32$)	32	54.0	106.2 K	54.4	98 K	41.5†	95	1115.1
Square Euclid. dist. ($n = 64$)	32	109.1	207.4 K	108.4	196 K	83.1†	191	2486.8

† The light blue background indicates the fastest mode for a given benchmark.

‡ N/A (Not Applicable): Arithmetic mode does not support bitwise operations. DNF (Did Not Finish): Does not complete in less than 2 hours.

§ The image blurring benchmarks use grayscale images and a 3 × 3 filter with the indicated pixel dimensions. "opt" indicates our compression technique.

* Composed as a series of smaller images due to logic synthesis constraints.

¶ Number of gates in each cycle.

arithmetic mode in addition to the gates and the LUTs with LBB. We note that the compression technique results in a $2 - 3\times$ runtime improvement, at the cost of a somewhat degraded output image due to the precision loss arising from the compression step. Additionally, the Gaussian blur outperforms the box blur because the normalization can be achieved by a very low-cost right shift, which is very efficient with TFHE-rs. In fact, we observe that the Gaussian blur is roughly 1.3× faster than the box blur; this is primarily due to the division step required to compute the average of the summed pixels.

Overall, the LUT with LBB mode is the most performant for the image processing benchmarks, outperforming the LUT without LBB mode by approximately 2× and the gates mode by $\sim 1.1\times$. This is due to the rigorous logic optimizations that Yosys can perform and the fact that shifts, while cheap in arithmetic mode, are free for binary ciphertexts, as they just involve simple copy operations as each encrypted bit can be trivially isolated. We note that LUT with LBB and gate modes incur significant setup costs due to the logic synthesis process, while the arithmetic mode can operate with behavioral Verilog directly. In fact, on the experimental server, Yosys was not able to complete the logic synthesis for the 64 × 43 box blur, which is implemented as a very large combinational circuit. As a result, we divided the image into eight slices and performed eight separate circuit evaluations to get a close approximation of the workload for the full image. We remark that Romeo was unable to evaluate both filters in less than two hours. Lastly, we note that the latency of both filters scales linearly with the input image size, with a 4× slowdown resulting from doubling the height and width of the input image (i.e., processing a 128 × 85 image).

Scheme Hopping on Cloud Servers. One of the challenges of using HE for secure outsourced computation in scenarios when there are numerous encrypted data inputs and outputs is the communication overhead between the client and server. For instance, in gates or LBB-LUT mode, each bit of plaintext is encrypted independently and results in a data expansion factor of three to four orders of magnitude depending on parameter choices. This problem can be resolved by implementing a decryption circuit of a traditional cryptographic algorithm such as AES *in the encrypted domain*. With this approach, the client can send small AES ciphertext data representing secret data to be processed (which results in negligible size expansion relative to plaintext data) to the cloud along with a homomorphic encryption of the AES secret key. Then, the cloud server can evaluate the AES decryption circuit homomorphically with the encrypted key and the result will be a valid homomorphic encryption of the underlying plaintext. At this point, the cloud can proceed to perform encrypted computation and return the resulting homomorphic ciphertexts to the client for decryption with the homomorphic secret key.

Table 1 details the computational overhead of evaluating the AES core algorithm along with specific implementations of AES-128 decryption with and without key scheduling. Incorporating the key scheduling into the homomorphic algorithm decreases the computational overhead on the cloud side at the cost of adding additional computation for the client and increases communication overhead as the client must homomorphically encrypt and send all eleven round keys to the server. For AES decryption with key scheduling, HELM is approximately 28× faster than Romeo [36] in both gates and LUT with LBB modes. For reference, the implementation of AES with the BGV cryptosystem by Gentry *et al.* [42] provides two AES variants that differ in the encryption parameters chosen, both of which pre-compute the round keys on the client side. The first variant operates in an LHE context and requires four minutes to evaluate, but exhausts the majority of the noise budget, severely limiting the number of subsequent homomorphic operations on the encrypted data. Conversely, the second variant incorporates bootstrapping and can continue computing on the encrypted data after the AES decryption procedure, but is over 4× slower than the LHE variant. Compared to Romeo and the bootstrapped implementation of Gentry *et al.* for AES-128 decryption without key-scheduling, HELM is 23× and 11.5× faster respectively.

5.5 User Overhead

From the user's perspective, there is a one-time cost to generate a keypair (which can be used for multiple circuits) along with the cost of encrypting inputs with the secret key and finally decrypting outputs. These costs vary based on the encrypted evaluation mode as each mode requires separate parameter sets and entails different encrypted datatypes. Notably, the decryption operation time is negligible with an amortized cost of less than 1 microsecond per bit.

Gates Mode. On average, key generation takes approximately 770 ms with 128 bits of security for gates mode and an encryption cost of 22 μs per bit of plaintext. For reference, the total encryption time for the 128-bit multiplier benchmark (total of 256 bits) is 17.7 ms, while the decryption time is 4.33 ms (i.e., for a total of 128 bits, since the word size is fixed). This benchmark represents the average use case in terms of user overhead. Conversely, the 10×10 square matrix multiplication benchmark, which has a high number of inputs, takes 360 ms as the user has to encrypt $10 \times 10 \times 16$ (bits) $\times 2$ (matrices) $= 3200$ bits. Finally, its decryption for a total of $10 \times 10 \times 16 = 1600$ bits takes 54.5 ms.

LUT with LBB Mode. For the 2:1 LUT with LBB mode used for the experimental evaluation, the key generation requires approximately 3.8 seconds on average with an encryption cost of 1.9 ms per bit of plaintext. Overall, we observe that the LUT with LBB mode exhibits slower client-side operations as the parameters required are larger than the default parameters employed by TFHE-rs for gate evaluations. The encryption and decryption for the 128-bit multiplier benchmark require 377.7 ms and 4.53 ms, respectively. For the 10×10 matrix multiplication, the encryption takes approximately 6 seconds while decryption takes 104 ms.

LUT without LBB Mode. Compared to the other modes, the LUT without LBB mode uses the largest parameters as it demands a greater storage capacity for each ciphertext. However, instead of generating an encryption or computing a decryption for each bit of plaintext, multiple bits can be encrypted or decrypted at the same time. Key generation consumes approximately 3 seconds on average and the amortized encryption cost is 43 microseconds per bit. The 10×10 square matrix product takes 138 ms for encryption and only 1.7 ms for decryption while the 128-bit multiplication takes 11.1 ms for encryption and 0.1 ms for decryption.

5.6 Discussion

Our experiments investigate the three evaluation modes of HELM, each tailored to different circuit characteristics, and shed light on the ideal mode selection based on the circuit. First, we observe that the gates mode demonstrates its superiority in scenarios characterized by a prevalence of `NOT` gates. This is exemplified in our experimental results presented in Table 1 and visualized in Figs. 6 and 7 from the ISCAS'85 and ISCAS'89 benchmark suites, which both contain circuits with higher proportions of `NOT` gates relative to other gate types. In such cases, the gates mode is the most efficient choice, emphasizing the importance of matching the optimization mode to the circuit's underlying gate composition.

Conversely, our experiments reveal that LUTs with LBB exhibit superior performance in general-purpose scenarios. This versatility is exemplified through examples drawn from our benchmark results, notably the AES and Blur filters in Table 1. Furthermore, we underscore that the arithmetic mode (i.e., LUTs without LBB) excels when circuits predominantly involve arithmetic operations, including additions, subtractions, multiplications, and divisions. This assertion is substantiated by the multi-bit multiplier benchmarks in Fig. 8, as well as the arithmetic-heavy experiments (e.g., chi-squared, squared Euclidean distance, matrix multiplication, etc.) in Table 1. Notably, if a circuit operates efficiently in the arithmetic mode, it can also perform well in the binary mode (i.e., gates and LUTs with LBB), but the reverse is not always true, as indicated by cases labeled as "N/A" in Table 1. These findings provide valuable insights into how HELM operates and which mode to use for optimal performance for a given circuit.

6 Related Works

We can categorize related works into two broad classes, the ones that target CPUs (and usually more general-purpose computation) and the ones that target GPUs. Additionally, we observe that prior research efforts prioritize different aspects: some focus on enhancing user-friendliness, others aim at facilitating general-purpose computation, while there are those that focus on fine-tuning optimizations tailored to specific applications – sometimes even encompassing a blend of these three objectives. HELM emerges as a notable example of achieving a balanced synergy among these three goals.

6.1 FHE Compilers & CPU Execution Engines

There is a plethora of works that have focused on making FHE more accessible through high-level compilers that map to the most popular FHE libraries for execution on CPUs. The recent Systematization of the Knowledge (SoK) work of the T2 compiler [14] proposed standardizing benchmarks along with a compiler that interfaces with five different back-ends (HElib, Lattigo, Palisade, SEAL, and TFHE) to facilitate usability. This way, users can implement their programs in one high-level language, then use the compiler to translate it into all the aforementioned libraries, and finally run the FHE programs to find which library is best for their application needs. Furthermore, similarly to T2, the E^3 [32] compiler targets SEAL, HElib, FHEW, PALISADE, and TFHE, however, has limited batching support, a lack of relational operations over integers, and limitation on allowing the users to tweak parameters like the ciphertext modulus. Another recent SoK paper [15] surveyed state-of-the-art FHE compilers like CHET [43], Cingulata [44], and EVA [17].

CHET is designed for CKKS and is geared towards private neural network inference for HEAAN and SEAL, while EVA [17] employs a Domain-Specific Language (DSL) for vector arithmetic focuses on SEAL and CKKS. The Cingulata compiler toolchain [44] allows for the conversion of C++ programs to homomorphic circuits for TFHE and a BFV variant and provides similar functionality to E^3 with some caveats. It requires users to modify their programs to work with the toolchain and, while providing a simpler API than many homomorphic encryption libraries, it requires significant effort on behalf of the user to understand the nuances of the library and its associated structures and data types. The SHEEP library [45] provides the capability for users to employ an assembly-like language to target multiple FHE libraries like TFHE and HElib; however, this approach entails the manual design of FHE circuits, which not only limits usability but also mirrors the methodology required for direct program implementation using the FHE library itself. Marble [46], on the other hand, focuses on usability by providing C++ extensions that allow users to write code similar to a plaintext implementation, utilizing encrypted binary arithmetic with HElib as its FHE backend. Another work that focuses on usability is the Concrete compiler [47]; Concrete allows the users to write Python functions and automatically transforms them to their encrypted equivalents for the CGGI cryptosystem. Finally, Google's Transpiler [16] and ROMEO [36] read C++ and Verilog programs, respectively, as inputs, compile them into optimized circuits through the use of synthesis tools, and finally execute them using TFHE.

It is easy to observe from all the aforementioned works that some focus on usability, others aim at achieving general-purpose computation, while others focus on optimizing the performance for specific applications (e.g., tensor operations). In HELM, we strike a good balance between all three worlds:

- **General-Purpose Computation:** CGGI is the only cryptosystem that allows for efficient general-purpose computation as its fast bootstrapping operation allows for computing any function over encrypted data. While certain computations are possible, and might even be more efficient, with other cryptosystems such as BFV or CKKS, these schemes very quickly reach their peak as bootstrapping is impractical. CGGI on the other hand allows for expressing any operation in the encrypted domain. For that reason, in HELM we utilize CGGI as our underlying cryptosystem.
- **Performance:** Notably, developing encrypted programs for the CGGI cryptosystem is a daunting task as users need to express their algorithms as Boolean circuits. Many works such as [14,16,32,44,47] have created compilers from high-level languages that target CGGI. However, this often results in inefficient solutions since transforming imperative programs (e.g., C, Python) into Boolean circuits is a complicated procedure. Even with utilizing high-level synthesis tools as adopted by [16] (i.e., the input is C++ programs and target is CGGI), simple FHE computation like the Chi-squared test takes around 40 seconds with [44] and [14], over 2 minutes with [16], and over 3.6 minutes with [32]. On the other hand, Chi-squared in HELM takes 10 seconds. This is because our starting point is Verilog and we can leverage existing Boolean optimization toolchains. Finally, HELM utilizes aggressive multi-threading optimizations to evaluate multiple gates that do not have dependencies between them at the same time, achieving significantly faster encrypted evaluation compared to the related works.
- **Usability:** In terms of usability, HELM allows users to develop programs in Verilog instead of a native FHE library. As Verilog is a hardware description language (HDL), it can lead to more efficient and optimized hardware implementations of FHE operations and higher levels of parallelization. Although expressing computations in Verilog might be more challenging than in other high-level languages, Verilog writing HDL programs is far more high-level than the API that most FHE libraries provide, which requires users to explicitly construct netlists directly. This renders HELM both user-friendly and fast at the same time.

6.2 FHE GPU-Accelerated Execution Engines

The works in this class have focused on FHE acceleration using both software and hardware techniques. However, these endeavors have tended to prioritize the acceleration of basic FHE operations, rather than emphasizing user-friendliness and usability. The works of the cuFHE [48] and nuFHE [49] libraries introduce GPU acceleration to the CGGI cryptosystem for single gates and vectors, respectively. The former approach faces challenges due to the costly transfers of ciphertexts between the CPU and GPU for each gate evaluation. In contrast, the latter approach is limited in its applicability to circuit evaluations, as circuits often comprise a variety of gate types and fully utilizing vectorization and the GPU capabilities is not possible. The REDcuFHE [30] project overhauls cuFHE to introduce support for multi-bit plaintext and multi-GPU setups. However, a possible drawback of REDcuFHE is that it places the responsibility of scheduling and managing communication between multiple GPUs squarely on the programmer's shoulders, resulting in usability challenges. Lastly, the work of ArctyrEX [50] also provides multi-GPU compatibility and automates scheduling and communication procedures. Unlike REDcuFHE, ArctyrEX's approach eliminates the need for manual intervention from programmers, enhancing the overall user experience.

Although GPU-accelerated FHE comes with a great performance promise, in reality, it has multiple caveats with regard to usability and even latency. First of all, the programming model in cuFHE, REDcuFHE, and nuFHE expose a gates API which is very similar to the API of the TFHE library; i.e., they require the user to express their program in a Boolean circuit format. ArctyrEX is significantly more usable as it allows developers to write their code in C/C++ and use high-level synthesis to transform the C/C++ code to Boolean circuits. Nevertheless, techniques based on high-level synthesis (HLS) come with potential limitations. First, converting a high-level program (e.g., C/C++) to a circuit results in a suboptimal circuit representation, and second, approaches like ArctyrEX incur high preprocessing costs from HLS, restricting how large the programs can be in practice. Finally, none of the aforementioned libraries considers LUTs. Conversely, HELM offers great performance, while still offering support for programming in Verilog, rather than Boolean circuits.

7 Conclusion

In this work, we introduce the HELM framework for automated conversion from arbitrary synthesizable Verilog HDL designs to encrypted circuits for privacy outsourcing applications. In HELM, Verilog designs are converted to netlists through the process of synthesis, while our bespoke compiler creates an internal construction of the circuit outlined in the netlist and determines the correct execution order for the homomorphic gate evaluations. The resulting homomorphic circuit is converted to Rust and employs the TFHE-rs library and is uploaded to a remote service for encrypted evaluation along with encrypted inputs.

We evaluate HELM with representative circuits from the ISCAS '85 and '89 benchmark suites, as well as several real-life workloads, such as CRC-32, AES, and encrypted image filtering. In all cases, we observed a roughly linear increase in encrypted circuit evaluation time with a growing number of gate evaluations. Leveraging HELM's three powerful modes of homomorphic evaluation, namely gates, LUTs with LBB, and LUTs without LBB, we report performance improvements of 1-2 orders of magnitude over related works.

References

1. Z. Tari, X. Yi, U. S. Premarathne, P. Bertok, and I. Khalil, "Security and privacy in cloud computing: Vision, trends, and challenges," *IEEE Cloud Computing*, vol. 2, no. 2, pp. 30–38, Mar 2015.
2. G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *ASIACCS 16: 11th ACM Symposium on Information, Computer and Communications Security*, X. Chen, X. Wang, and X. Huang, Eds. Xi'an, China: ACM Press, May 30 – Jun. 3, 2016, pp. 353–364.
3. Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One bit flips, one cloud flops: Cross-VM row hammer attacks and privilege escalation," in *USENIX Security 2016: 25th USENIX Security Symposium*, T. Holz and S. Savage, Eds. Austin, TX, USA: USENIX Association, Aug. 10–12, 2016, pp. 19–35.
4. F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *2016 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 2016, pp. 406–418.
5. Y. Han, J. Chan, T. Alpcan, and C. Leckie, "Using virtual machine allocation policies to defend against co-resident attacks in cloud computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 14, no. 1, pp. 95–108, 2015.
6. R. Poddar, T. Boelter, and R. A. Popa, "Arx: An Encrypted Database Using Semantically Secure Encryption," *Proc. VLDB Endow.*, vol. 12, no. 11, p. 1664–1678, jul 2019. [Online]. Available: https://doi.org/10.14778/3342263.3342641
7. D. Micciancio, "A first glimpse of cryptography's holy grail," *Communications of the ACM*, vol. 53, no. 3, p. 96, 2010.
8. C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, 2009.

9. M. Research, “Microsoft SEAL (release 4.0),” https://github.com/Microsoft/SEAL, Mar. 2022, microsoft Research, Redmond, WA.
10. Tune Insight SA, “Lattigo v3,” Online: https://github.com/tuneinsight/lattigo, Feb. 2022.
11. J. Fan and F. Vercauteren, “Somewhat practical fully homomorphic encryption,” Cryptology ePrint Archive, Report 2012/144, 2012, https://eprint.iacr.org/2012/144.
12. Z. Brakerski, C. Gentry, and V. Vaikuntanathan, “(Leveled) fully homomorphic encryption without bootstrapping,” in *ITCS 2012: 3rd Innovations in Theoretical Computer Science*, S. Goldwasser, Ed. Cambridge, MA, USA: Association for Computing Machinery, Jan. 8–10, 2012, pp. 309–325.
13. J. H. Cheon, A. Kim, M. Kim, and Y. S. Song, “Homomorphic encryption for arithmetic of approximate numbers,” in *Advances in Cryptology – ASIACRYPT 2017, Part I*, ser. Lecture Notes in Computer Science, T. Takagi and T. Peyrin, Eds., vol. 10624. Hong Kong, China: Springer, Heidelberg, Germany, Dec. 3–7, 2017, pp. 409–437.
14. 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, pp. 154–172, Jul. 2023.
15. A. Viand, P. Jattke, and A. Hithnawi, “SoK: Fully homomorphic encryption compilers,” in *2021 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 24–27, 2021, pp. 1092–1108.
16. S. Gorantala, R. Springer, S. Purser-Haskell, W. Lam, R. 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,” Cryptology ePrint Archive, Report 2021/811, 2021, https://eprint.iacr.org/2021/811.
17. 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
18. D. Mouris, N. G. Tsoutsos, and M. Maniatakos, “TERMinator Suite: Benchmarking Privacy-Preserving Architectures,” *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 122–125, 2018.
19. I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, “TFHE: Fast fully homomorphic encryption over the torus,” *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, Jan. 2020.
20. D. Mouris and N. G. Tsoutsos, “Masquerade: Verifiable multi-party aggregation with secure multiplicative commitments,” Cryptology ePrint Archive, Report 2021/1370, 2021, https://eprint.iacr.org/2021/1370.
21. R. L. Rivest, A. Shamir, and L. M. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Communications of the Association for Computing Machinery*, vol. 21, no. 2, pp. 120–126, Feb. 1978.
22. P. Paillier and D. Pointcheval, “Efficient public-key cryptosystems provably secure against active adversaries,” in *Advances in Cryptology – ASIACRYPT’99*, ser. Lecture Notes in Computer Science, K.-Y. Lam, E. Okamoto, and C. Xing, Eds., vol. 1716. Singapore: Springer, Heidelberg, Germany, Nov. 14–18, 1999, pp. 165–179.
23. O. Regev, “On lattices, learning with errors, random linear codes, and cryptography,” in *37th Annual ACM Symposium on Theory of Computing*, H. N. Gabow and R. Fagin, Eds. Baltimore, MA, USA: ACM Press, May 22–24, 2005, pp. 84–93.
24. V. Lyubashevsky, C. Peikert, and O. Regev, “On ideal lattices and learning with errors over rings,” in *Advances in Cryptology – EUROCRYPT 2010*, ser. Lecture Notes in Computer Science, H. Gilbert, Ed., vol. 6110. French Riviera: Springer, Heidelberg, Germany, May 30 – Jun. 3, 2010, pp. 1–23.
25. C. Gentry, “Computing on encrypted data (invited talk),” in *CANS 09: 8th International Conference on Cryptology and Network Security*, ser. Lecture Notes in Computer Science, J. A. Garay, A. Miyaji, and A. Otsuka, Eds., vol. 5888. Kanazawa, Japan: Springer, Heidelberg, Germany, Dec. 12–14, 2009, p. 477.
26. L. Ducas and D. Micciancio, “FHEW: Bootstrapping homomorphic encryption in less than a second,” in *Advances in Cryptology – EUROCRYPT 2015, Part I*, ser. Lecture Notes in Computer Science, E. Oswald and M. Fischlin, Eds., vol. 9056. Sofia, Bulgaria: Springer, Heidelberg, Germany, Apr. 26–30, 2015, pp. 617–640.
27. I. Chillotti, M. Joye, and P. Paillier, “Programmable bootstrapping enables efficient homomorphic inference of deep neural networks,” in *Cyber Security Cryptography and Machine Learning: 5th International Symposium, CSCML 2021, Be’er Sheva, Israel, July 8–9, 2021, Proceedings 5*. Springer, 2021, pp. 1–19.
28. Z. Liu, D. Micciancio, and Y. Polyakov, “Large-precision homomorphic sign evaluation using FHEW/TFHE bootstrapping,” in *Advances in Cryptology – ASIACRYPT 2022, Part II*, ser. Lecture Notes in Computer Science,

S. Agrawal and D. Lin, Eds., vol. 13792. Taipei, Taiwan: Springer, Heidelberg, Germany, Dec. 5–9, 2022, pp. 130–160.

29. F. Bourse, M. Minelli, M. Minihold, and P. Paillier, "Fast homomorphic evaluation of deep discretized neural networks," in *Advances in Cryptology – CRYPTO 2018, Part III*, ser. Lecture Notes in Computer Science, H. Shacham and A. Boldyreva, Eds., vol. 10993. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 19–23, 2018, pp. 483–512.
30. L. Folkerts, C. Gouert, and N. G. Tsoutsos, "REDsec: Running encrypted DNNs in seconds," in *ISOC Network and Distributed System Security Symposium – NDSS 2023.* San Diego, CA, USA: The Internet Society, Feb. 2023.
31. S. Halevi and V. Shoup, "Algorithms in HElib," in *Advances in Cryptology – CRYPTO 2014, Part I*, ser. Lecture Notes in Computer Science, J. A. Garay and R. Gennaro, Eds., vol. 8616. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 17–21, 2014, pp. 554–571.
32. E. Chielle, O. Mazonka, N. G. Tsoutsos, and M. Maniatakos, "E^3: A framework for compiling C++ programs with encrypted operands," Cryptology ePrint Archive, Report 2018/1013, 2018, https://eprint.iacr.org/2018/1013.
33. Zama, "TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data," 2022, https://github.com/zama-ai/tfhe-rs.
34. C. Wolf, "Yosys open synthesis suite," http://www.clifford.at/yosys/.
35. R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22.* Springer, 2010, pp. 24–40.
36. C. Gouert and N. G. Tsoutsos, "Romeo: Conversion and Evaluation of HDL Designs in the Encrypted Domain," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.
37. F. Brglez, "A neutral netlist of 10 combinatorial benchmark circuits and a target translator in FORTRAN," in *IEEE ISCAS*, 1985, pp. 663–698.
38. F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *IEEE ISCAS*, 1989, pp. 1929–1934.
39. A. Becker, L. Ducas, N. Gama, and T. Laarhoven, "New directions in nearest neighbor searching with applications to lattice sieving," in *27th Annual ACM-SIAM Symposium on Discrete Algorithms*, R. Krauthgamer, Ed. Arlington, VA, USA: ACM-SIAM, Jan. 10–12, 2016, pp. 10–24.
40. M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," Cryptology ePrint Archive, Report 2015/046, 2015, https://eprint.iacr.org/2015/046.
41. M. D. Malkauthekar, "Analysis of euclidean distance and manhattan distance measure in face recognition," in *Third International Conference on Computational Intelligence and Information Technology (CIIT 2013).* Mumbai: IET, 2013, pp. 503–507.
42. C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the aes circuit," in *Annual Cryptology Conference.* Springer, 2012, pp. 850–867.
43. 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
44. 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
45. N. Barlow, T. Lazauskas, O. Strickson, and A. Gascon, "SHEEP: A homomorphic encryption evaluation platform," Online, 2019, https://github.com/alan-turing-institute/SHEEP.
46. 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.* New York, NY, USA: Association for Computing Machinery, 2018, pp. 49–60.
47. Zama, "Concrete: TFHE Compiler that converts python programs into FHE equivalent," 2022, https://github.com/zama-ai/concrete.
48. W. Dai and B. Sunar, "cuFHE: CUDA-accelerated Fully Homomorphic Encryption Library," https://github.com/vernamlab/cuFHE, 2018.
49. NuCypher, "NuFHE, a GPU-powered Torus FHE implementation," https://github.com/nucypher/nufhe, 2019.
50. C. Gouert, V. Joseph, S. Dalton, C. Augonnet, M. Garland, and N. G. Tsoutsos, "Accelerated encrypted execution of general-purpose applications," Cryptology ePrint Archive, Report 2023/641, 2023, https://eprint.iacr.org/2023/641.