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 three modes of encrypted evaluation: a) a gate mode that consists of Boolean gates, b) a small-precision lookup table mode which significantly reduces the size of the circuit by combining multiple gates into lookup tables, and c) a high-precision lookup table mode tuned for multi-bit arithmetic evaluations. Finally, HELM introduces a scheduler that leverages the parallelism inherent in arithmetic and Boolean circuits to efficiently evaluate encrypted programs. We evaluate HELM with the ISCAS'85 and ISCAS'89 benchmark suites, as well as real-world applications such as image filtering and neural network inference. In our experimental results, we report that HELM can outperform prior works by up to 65×.

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

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 or the data could be exposed if the server is compromised by attackers.

An intuitive solution to these issues is encryption, which can protect data both when it is *at rest* and *in transit*. Encryption prevents 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.

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

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

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

fully homomorphic encryption (FHE), often referred to as the "holy grail" of cryptography [1], 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) [2].

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 [3, 4], they are challenging to use without extensive cryptographic knowledge, especially as different libraries offer different APIs and implement different FHE schemes [5–7]. Recent works have attempted standardizing benchmarks and compilers [8–12], however, properly setting cryptographic parameters for security (such as the polynomial ring dimension) 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 decisions based on underlying plaintext values. In many cases, this is not straightforward to address for a given application such as private sorting, which has been the focus of prior works [13, 14]. We remark that prior work proposed a solution to control flow decisions on encrypted data, but requires the client to remain online and participate during the encrypted program evaluation [15]. Additionally, for many libraries, monitoring ciphertext noise 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 [16]. Notably, HELM requires no knowledge of the challenges imposed by encrypted computing from users. 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 privacy-preserving 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 circuits with optimal formats.

2 Preliminaries

2.1 Fully Homomorphic Encryption

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. We omit any discussion of partial HE as it is not functionally complete and therefore unsuitable for general-purpose computation. Leveled HE, on the other hand, is functionally complete but scales poorly for complex applications; the size of ciphertexts as well as the computational overhead of HE operations scales with the computational depth.

The 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 [17]. 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 [18] 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. The CGGI cryptosystem [16] 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. 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 plaintext value of a ciphertext to a valid encryption with a value dependent on the selected coefficient after the blind rotation and extraction. This generalized bootstrapping technique is called *programmable bootstrapping* [19, 20]. Because of this powerful technique and the low latency of bootstrapping, we target the CGGI cryptosystem in this work and leverage the state-of-the-art TFHE-rs [21] library as our chosen cryptographic backend.

2.2 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 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 cannot 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. We note that some earlier bridging techniques, such as those employed in FHE-DiNN [22] and REDsec [23], 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 bridging methodologies can be found in Section 3.

3 Automatic Translation of Circuits to the Encrypted Domain

Many existing frameworks, such as the Google Transpiler [10] and Concrete [24], allow users to program in high-level languages such as C++ and Python. However, many standard language features are incompatible with encrypted evaluation, such as external libraries, system calls, and dictionaries. This can lead to frustration on behalf of the programmers, as they are required to adhere to the restrictions imposed by encrypted computation. Additionally, these frameworks typically do not work with off-the-shelf or pre-existing codebases, and need to be transformed to interface with the target framework. On the other hand, Verilog programs can be transformed directly into Boolean circuits that can be evaluated with HELM, imposing no restrictions on the developer aside from the fact that the program must be synthesizable. To accomplish this conversion, the first step is to use *logic synthesis* to convert Verilog programs to Boolean netlists consisting

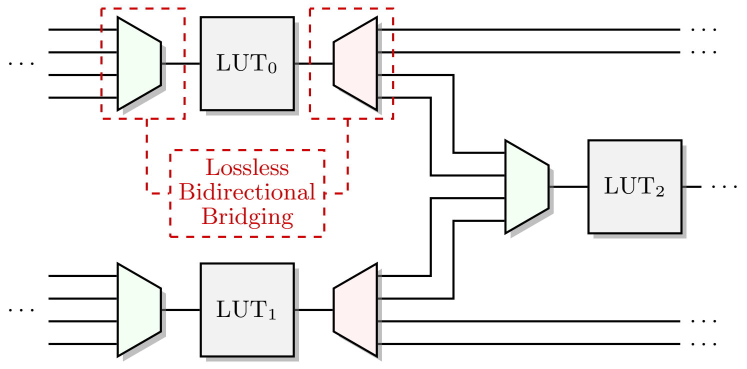
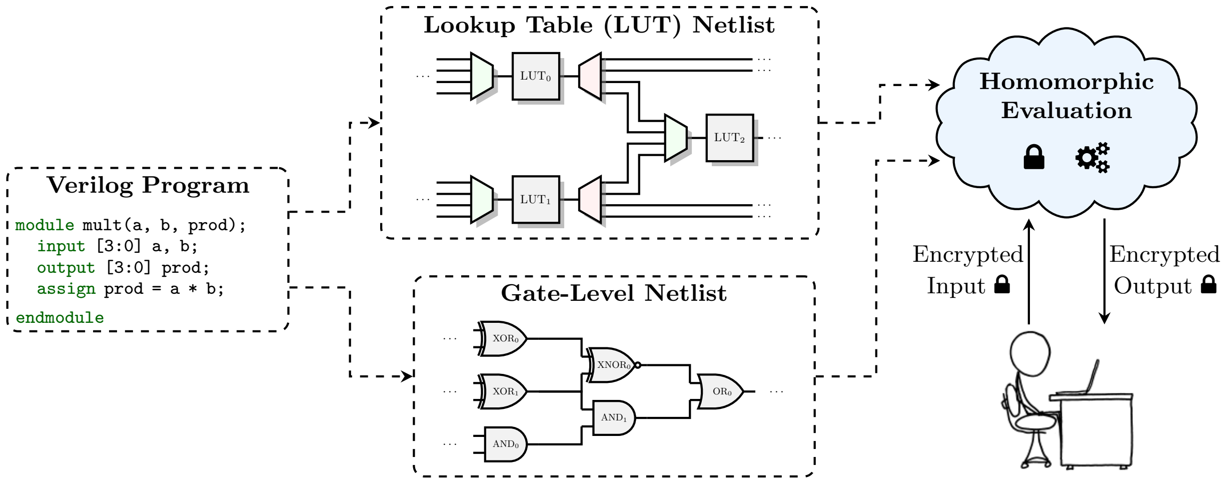


Fig. 1. Example of Lookup Tables (LUTs) Circuit.

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. 2.



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

Fig. 2. 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 [25], 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 [26]. 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 a series of algorithms. First, HE-friendly logic optimizations are applied, such as minimization and removing redundant wires. Next, all cells are mapped to standard logic gates and small multiplexers, or many-to-one lookup tables and the resulting netlist is saved 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` and `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 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. We note that operations between secure ciphertexts and constant values are defined in the Homomorphic Encryption Standard [27] and are incorporated in state-of-the-art HE libraries such as OpenFHE [28], Lattigo [4], and TFHE-rs [21]. 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 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 can generate chains of identical wires when synthesizing large and complex programs, resulting in convoluted netlists. If the set of identifiers referring to one wire is not collapsed to a 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 be finally 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 [29].

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.
 1: procedure CIRCUIT-SORT(cells)
 2:     for gate in cells do                                  ▷ Find direct ancestors.
 3:         for wire in gate.inputs do                        ▷ Check origin of inputs.
 4:             if wire is output from another gate then
 5:                 gate.depends_on ← wire.originator
 6:     level ← 0                                             ▷ Counter for levels.
 7:     while unevaluated gates remain do
 8:         if gate.evaluated then
 9:             continue                                      ▷ Gate exists in a level.
10:         for gate in cells do
11:             if gate.depends_on = None then                ▷ Input gates.
12:                 gate.evaluated ← True
13:                 levels[level].append(gate)                ▷ Append gate.
14:             else
15:                 ready ← True
16:                 for prev_gate in gate.depends_on do
17:                     if prev_gate.evaluated = False then
18:                         ready ← False                     ▷ Ancestor not ready.
19:                 if ready then
20:                     gate.evaluated ← True
21:                     levels[level].append(gate)
22:         level ← level + 1
23:     return levels
```

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. Aside from clock-gating, we note that no clock signals are utilized after the unrolling. To simulate FF behavior, the ciphertext corresponding to the data input of the FF is copied to the ciphertext corresponding to the FF data output of the next clock cycle.

Lastly, due to the termination problem, there is no way for the computing party to determine how many cycles are appropriate for correct evaluation. For circuits such as AES encryption, the number of cycles required to generate valid outputs is fixed and independent of the specific input values. AES-128 will always require 10 rounds/cycles for both encryption and decryption. In other cases, the client would be required to specify the desired number of cycles to execute during homomorphic evaluation. The final decrypted result may be inaccurate if too many or too few cycles are evaluated; as such, HELM incorporates an automated mechanism that mitigates the problem of selecting the wrong number of cycles. If a design incorporates a ready or valid signal that is set when the computation is finalized, HELM introduces additional logic to obliviously ensure that the outputs reflect the values generated during the cycle when the ready or valid signal goes high. Otherwise, all output wires are zeroed out upon decryption, indicating that the number of cycles selected by the client was too few to generate valid outputs. We accomplish this by introducing a set of parallelizable multiplexers at the end of each cycle that select between x and the computed value of each output wire, with the ready or valid signal serving as the control input of the multiplexer. In this case, x is initialized to an encryption of zero and replaced on subsequent cycles by the output of the multiplexer. In this way, the valid output is latched when the ready or valid signal is set and all subsequent cycles that may corrupt the output are effectively ignored in terms of modifying the output wires. We note that the computational complexity of the added logic scales linearly with the number of outputs in the circuits and is independent of the overall size of the circuit.

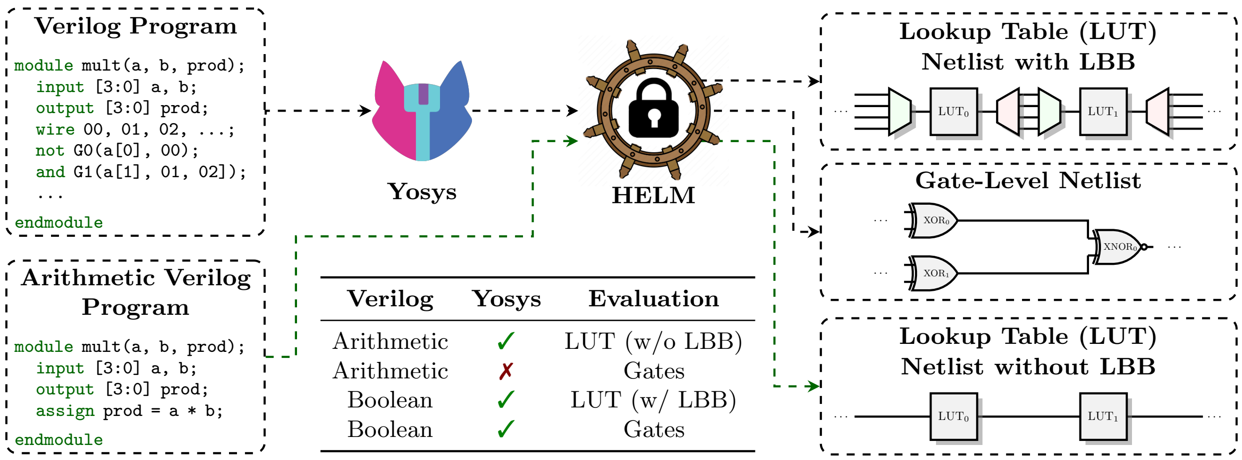
4 Oblivious Circuit Execution with FHE

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. 3. 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 on CPUs

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


Verilog Program
Yosys
HELM
Lookup Table (LUT) Netlist with LBB
LUT$_0$
LUT$_1$
Gate-Level Netlist
XOR$_0$
XOR$_1$
XNOR$_0$
Arithmetic Verilog Program
Lookup Table (LUT) Netlist without LBB

```
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. 3. 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.

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 Gate-based Circuit Evaluation on GPUs

We observe that due to the small parameter sizes needed to evaluate gates with the CGGI cryptosystem, we can fit a multitude of ciphertexts in the memory of a GPU for concurrent evaluation. Specifically, we extend the multi-threaded CPU approach and deploy the entire circuit level in one large batch for GPU-based encrypted gate evaluations. First, we revamped the `concrete-core` library [24] by extending the set of supported operations with new GPU-based Boolean gate evaluations. Additionally, we added new support for multiple CUDA streams to allow for multiple simultaneous kernel launches, achieving more concurrency and thus higher GPU utilization. Next, we extended HELM with a GPU mode that uses our modified `concrete-core` instead of TFHE-rs, as the latter does not incorporate support for GPUs, while our revamped library focuses on CUDA-accelerated Boolean gates. Unlike FHE Boolean gates, the generalized multi-bit LUTs implemented in `concrete-core` are better suited for *approximate computation* as they have a non-negligible probability of error, so in the case of precise Boolean evaluation these LUTs can cause incorrect results; therefore `concrete-core` Boolean gates are preferred over LUTs for our GPU acceleration.

Each gate type differs in the number and types of operations involved, albeit all gates with the exception of the inverter consist of linear operations between ciphertexts followed by a bootstrapping procedure and keyswitch. As such, a different set of kernels is launched for each gate type, where each kernel implements a key primitive such as ciphertext addition, scalar multiplication, and bootstrapping. Before the evaluation of each level, all ciphertexts are grouped into different vectors depending on gate type and then uploaded to the device. Next, the CPU launches numerous kernels to evaluate each gate type at the level on the GPU, and then the resulting ciphertext vectors are transmitted back to the CPU. Evaluation proceeds until all circuit levels are evaluated, similar to the CPU case.

4.3 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. 1) 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.

To overcome this, we introduce *lossless bidirectional bridging* (LBB) that allows us to convert multi-bit ciphertexts to encryptions of individual bits and vice versa. The implementation of LBB is a cornerstone of HELM's circuit evaluation when evaluating LUT circuits in a Boolean context. As such, our LBB technique has a significant impact on the overall latency of circuit evaluation. The goal of the forward LBB algorithm is to convert X ciphertexts encrypting `1` or `0` and compute an output ciphertext that combines them into an encryption of a single X-bit number. A straightforward method involves choosing parameter sets that support X bit data encodings and employing a sequence of constant multiplications with powers of 2 and accumulation to convert a vector of ciphertexts encoding bits to a single integer ciphertext. Unfortunately, this approach results in computational overhead that scales linearly with the number of ciphertexts to be combined as the polynomial degree of the ciphertexts must be increased. This renders the FHE operations more expensive. Instead, by taking advantage of the encoding strategy employed by the CGGI backends, we can devise a significantly better approach that allows for lower ciphertext polynomial degrees.

Both libraries split the capacity of a ciphertext into segments for actual data, which will be recovered upon decryption, and intermediate results are referred to as a carry region. Some parameter sets sacrifice data precision by creating capacity for intermediate computation, and we note that sets incorporating a carry capacity are usually smaller than those that do not. For instance, in TFHE-rs, the default parameter set that supports 5 bits of data precision and 1 bit of carry capacity uses a smaller LWE polynomial with 40 fewer coefficients (and thus a smaller ciphertext size) compared to the set that supports 6 bits of data precision with no carry capacity. Effectively, using the 5+1 parameter set over the 6-bit one would result in better performance. The carry region is typically ignored upon decryption; the 5+1 parameter set is meant to allow the user to encrypt and decrypt 5 bits, with the carry bit only used for temporary data utilized during the computation.

Therefore, to compute a 2:1 LUT forward bridging with a 1+1 parameter set, we concatenate two ciphertexts encrypting bits by saving the data bit of one ciphertext into the carry bit of another ciphertext. In this way, the concatenation of the two ciphertexts behaves in an identical manner to the ciphertext formed through an approach that achieves bridging without carry capacity, but allows the use of smaller encryption parameters and is thus more efficient.

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. We note that this reverse bridging technique can be combined with the table lookup in a single PBS per bit by computing `f(x, y) = lut[x] >> y & 1`. Additionally, in the case of many-to-one LUTs, the reverse bridging technique can be omitted as the output of any given LUT is only a single bit in length. The forward and reverse bridging procedures result in no net noise growth as the forward pass involves relatively low-noise additions to achieve concatenation (which is immediately followed by a bootstrap to evaluate the LUT itself). The reverse bridge, meanwhile, consists of programmable bootstraps, which reduce noise while decomposing the bits of the output.

We note that bi-directional bridging can be done with scheme switching, which involves converting from one FHE scheme to another through a computationally intensive process that involves bootstraps. For instance, the OpenFHE library [28] supports this capability between FHEW and CKKS. However, we note that the cost of converting between the schemes is approximately 1.5 seconds for ciphertexts encoding 1280 bits of data on our experimental server at 128 bits of security. This leads to an amortized cost of 1.2 milliseconds per bit. Conversely, HELM can achieve a cost of 250 microseconds per bit on the same server with 2:1 LUTs with LBB.

4.4 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 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.

5 Experimental Evaluation

5.1 Implementation and Experimental Setup

HELM was implemented in Rust (v1.72) and uses the Yosys Open SYnthesis Suite framework [25] (v0.9) for synthesizing Verilog programs and converting them to netlists. We compare HELM with three state-of-the-art works, Romeo [29], Concrete [24] and Google Transpiler [10]. All CPU experiments were performed on an `c5.12xlarge` AWS EC2 instance with 48 virtual cores. Conversely, GPU gate experiments were run on a single A100 SXM GPU with 80 GB of GPU memory. All security parameter sets utilized correspond to approximately 128 bits of security under the RC.BDGL16 [30] cost model of lattice estimator [31] 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.

5.2 ISCAS'85 and ISCAS'89 Circuits

The homomorphic circuit evaluation times for the ISCAS'85 combinational benchmarks are presented in Fig. 4. 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

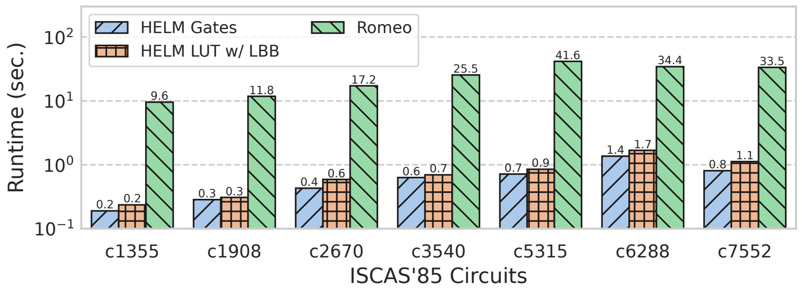


Fig. 4. Encrypted circuit evaluation times for the ISCAS'85 benchmark suite. We remark that the Google Transpiler is only compatible with C/C++ programs and does not support Verilog circuits as inputs.

in the graph, the `c5315` circuit incurs longer evaluation times than the two largest circuits despite its smaller size. This deviation is due 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. 5. 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 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. Table 1 shows the runtimes for `c7552` and `s15850`; we observe that the GPU gates mode is slower than the CPU gates mode because the circuits are narrow and therefore low device utilization is achieved. The CPU-GPU memory transfers, and the kernel launch overheads dominate the execution time in this case. We utilize `s344`, which consists of 11 output wires, to evaluate the additional logic added to circuits with a ready or valid signal as this is the only ISCAS circuit that incorporates this functionality. When processed with Yosys, we observe that this circuit consists of 118 combinational gates per cycle; with the automatically inserted logic to enforce the circuit behavior with a ready signal, 11 multiplexers are appended after the combinational logic of each cycle (one for each output wire). On average, we observe that the latency of each cycle is increased by approximately 20.4 milliseconds relative to the baseline encrypted evaluation due to the added multiplexer logic. We emphasize that the number of gates added is independent of the overall circuit size and only scales with the number of output wires, so large circuits with a limited number of outputs will have a significantly smaller proportional cycle latency increase.

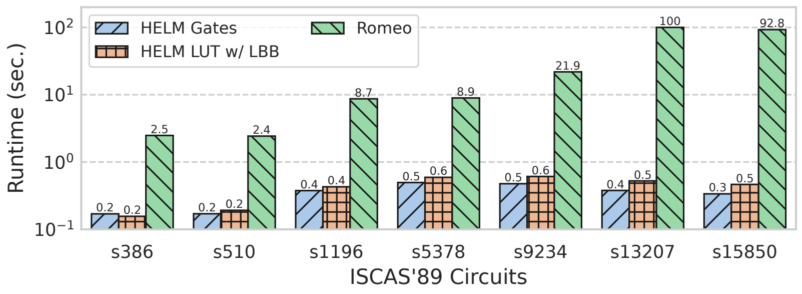


Fig. 5. Amortized evaluation time per cycle (over 10 cycles) for encrypted circuits from the ISCAS '89 benchmark suite. The Google Transpiler is only compatible with C++ programs and does not support Verilog circuits as inputs.

5.3 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.

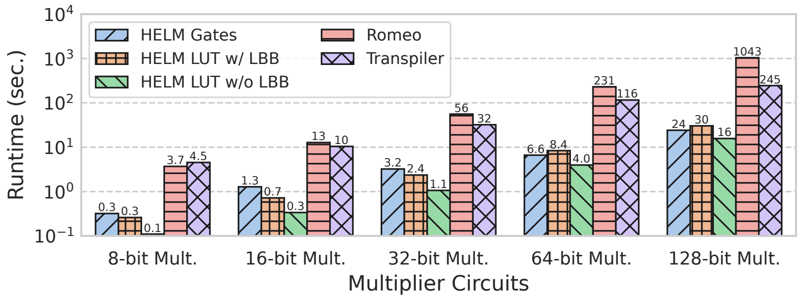


Fig. 6. Encrypted circuit evaluation times for {16, 32, 64, 128}-bit multipliers.

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. 6. 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 [29] by up to 65× (which is entirely sequential by design) and the parallelized Google FHE Transpiler [10] 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 a 10× 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. Further, we note that the GPU gate evaluations outperform the CPU gates by approximately 2.6× for both matrix sizes and outperforms the other CPU-based modes as well. Relative to the Concrete library, which also utilizes CGGI, HELM in LUT w/o LBB mode is approximately 2.2× faster for 10×10 matrix multiplication.

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 CPU gate mode by approximately 17%. The GPU outperforms the CPU gates by approximately 3.3$\times$ for this particular benchmark.

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 [32]. 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 on the CPU and is approximately 30% faster than the gates and LUTs with LBB because the algorithm is composed of numerous multiplications, additions, and subtractions. Even so, the GPU gates outperform the CPU arithmetic mode by nearly 2$\times$. To avoid overflow due to the multiplications, we chose a 32-bit word size for this particular benchmark.

Image filters. 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 [10]. We take advantage of the form of the elements of the filter to compute the slot-wise multiplications as bit shifts, which are efficient in CGGI. Then, each entry is summed and the sum is normalized using a right shift by 4.

Each pixel is represented as an encryption of a byte in HELM's arithmetic mode that corresponds to the value of the grayscale color channel. We use a parameter set corresponding to 16 bits of precision for the default filter, as computing the average of nine 8-bit numbers generates an intermediate sum that requires 12 bits of precision to avoid overflows. For further optimization, we also introduce an optional compression step on the client side to reduce the size of pixels to 4 bits (by mapping 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 AES and CRC, which require bitwise operations, the two algorithms are composed exclusively of arithmetic operations and bit shifts which enables HELM to evaluate them with arithmetic mode as well as gates and LUTs with LBB. 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 of compression. Additionally, the Gaussian blur outperforms the box blur because the normalization can be achieved by a low-cost right shift, which is very efficient with TFHE-rs. In fact, we observe that the Gaussian blur is roughly 1.3$\times$ faster than the box blur; this is primarily due to the division step required to compute the average of the summed pixels.

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.)
		CPU Eval. (sec.)	GPU Eval. (sec.)	Num Gates	Eval. (sec.)	Num LUTs	Eval. (sec.)	Num LUTs	
c7552 (ISCAS'85)	1	0.8[†]	0.93	1.32 K	1.12	0.96 K	N/A[‡]	N/A	33.45
s15850 (ISCAS'89)	1	0.34[†]	0.32	0.48 K[¶]	0.46	0.44 K[¶]	N/A[‡]	N/A	92.76
AES Core	8	8.96	3.52	15 K	8.3[†]	14.7 K	N/A[‡]	N/A	187.3
AES-128 w/ Key Sched.	8	116.8	45.8	196 K	104.1[†]	152.7 K	N/A[‡]	N/A	2490
AES-128 w/o Key Sched.	8	94.3[†]	36.1	159 K	85.5[†]	186.1 K	N/A[‡]	N/A	2179.2
CRC-32 (1024 cycles)	8	1009	300.9	728.1 K	859.7[†*]	700.4 K	N/A[‡]	N/A	DNF[‡]
Box Blur (64×43 opt)[§]	8	449.6[†*]	169.6	806 K	405.6[†*]	732 K	968.9	24768	DNF[‡]
Gauss. Blur (64×43 opt)[§]	8	337.9	125.7	568 K	307[†]	553.7 K	586.4	38528	DNF[‡]
128-bit Multiplier	128	24	13.9	49.4 K	31.8	33.3 K	16[†]	1	1043
Matrix Mult. ($5x5 \times 5x5$)	16	55.7	21.4	95.9 K	55.6	98.8 K	41.8[†]	0.2 K	1287.3
Matrix Mult. ($10x10 \times 10x10$)	16	447.3	171.9	774 K	447.6	796.1 K	351.9[†]	1.9 K	DNF[‡]
Chi-squared	32	9.9	4.8	17.1 K	10.1	17.1 K	6.8[†]	14	205.2
Square Euclid. dist. ($n = 32$)	32	54.0	21.8	106.2 K	54.4	98 K	41.5[†]	95	1115.1
Square Euclid. dist. ($n = 64$)	32	109.1	42.4	207.4 K	108.4	196 K	83.1[†]	191	2486.8
LR Inference	16	10.5	5.1	18 K	9.9	18.2 K	7.1[†]	48	337.9
NN Inference (Sign)	16	2974	1121	5.2 M	3123	5.3 M	2051[†]	14.8 K	DNF[‡]
NN Inference (ReLU)	16	3372	1265	5.8 M	3284	5.8 M	2258[†]	14.9 K	DNF[‡]

[†] The light blue background indicates the fastest mode for a given benchmark (CPU only).

[‡] 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.

Overall, the LUT with LBB mode is the most performant for the image processing benchmarks of the CPU modes, outperforming the LUT without LBB mode by approximately $2\times$ 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 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 synthesis, 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. The GPU gates outperform the CPU-based LUT without LBB mode by approximately $2.4\times$ for both blur filters. Lastly, we note that the latency of both filters scales linearly with the input image size, with a $4\times$ 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 is the communication overhead between the client and server. For instance, in gates or LBB-LUT mode, a data expansion factor of three to four orders of magnitude results upon encryption 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 (with no 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 an FHE 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 and AES-128 decryption variants. Incorporating key scheduling decreases the latency on the cloud side at the cost of adding client-side latency and increasing communication overhead as the client must homomorphically encrypt and send all round keys to the server. For AES decryption with key scheduling, HELM is approximately 28$\times$ faster than Romeo [29] in both gates and LUT with LBB modes. For reference, the AES implementation with the BGV cryptosystem by Gentry *et al.* [33] provides two variants that differ in FHE parameters, both of which pre-compute the round keys on the client side. The first operates in an LHE context and requires four minutes to evaluate, but exhausts the noise budget, limiting the subsequent homomorphic operations on the encrypted data. The second variant incorporates bootstrapping, but is over 4$\times$ 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$\times$ and 11.5$\times$ faster respectively. Finally, relative to the Concrete library, HELM evaluates the decryption circuit (without key-scheduling) 46$\times$ faster. Notably, Concrete was unable to evaluate AES-128 decryption with key scheduling as it could not derive a correct and secure parameter set.

Privacy-Preserving Machine Learning. Machine learning as a service (MLaaS) is a powerful paradigm that allows users to outsource classification to a cloud server. Oblivious classification is possible with FHE, keeping both inputs and outputs confidential. We consider the scenario where a client has trained a machine learning model and provides the cloud server with the encryption of both network parameters (to uphold the confidentiality of proprietary models) as well as encrypted inputs. Logistic regression (LR) inference is commonly used for both binary and multi-class classification problems and can be viewed as a one-layer neural network with a sigmoid activation function. Following the T2 framework [8], we adopt a degree-3 polynomial approximation for the non-linear sigmoid function. We configure the LR model to work with inputs with 4 attributes that can be classified into 3 possible classes (which is in line with popular datasets such as Iris [34]). Due to the prevalence of multi-bit arithmetic operations, the LUT w/o LBB mode is the most performant for LR inference; however, we note that the gate mode with the GPU backend reduces the latency by a further 28%.

Additionally, we perform neural network inference using the same architecture as FHE-DiNN [22] for MNIST classification, which consists of two fully-connected layers (which are akin to matrix-vector products) and a sign activation (i.e., extract the sign bit of each input neuron). We scale the input monochrome image to 8×8, resulting in an input vector of length 64, where each entry (i.e., one pixel) is a one byte value. Additionally, there are 100 hidden neurons and 10 classes (1 for each possible handwritten digit). We also generate a variant of this network that employs the ReLU activation function instead of sign, utilizing the degree-2 approximation proposed by Ali et al. [35] As a Boolean circuit, we remark that both neural network variants are quite large with over 5 million gate or LBB LUT evaluations. On the other hand, this can be accomplished in roughly 15,000 additions and multiplications with the LUT w/o LBB mode. As such, similarly to the LR inference, LUT w/o LBB mode is the most performant on the CPU. With the gate mode running on a GPU, the latency is further reduced by approximately 45%. Lastly, Concrete evaluates the neural network inference with the sign activation in approximately 2465 seconds, which is 1.2$\times$ slower than HELM in LUT w/o LBB mode. Similarly to AES-128 decryption with key scheduling, Concrete was unable to evaluate the network variant with the ReLU network due to auto parameterization failures.

5.4 Impact of Security Considerations

While the standard 128 bits of security is well-suited for most use-cases today, particular applications warrant employing stronger security guarantees, such as those in the military sector. With the LWE-based HE cryptosystems, it is straightforward to modify the parameters to provide stronger security guarantees; the key way to accomplish this is to increase the polynomial degree while keeping both the coefficient size constant and noise injection rate constant (note that security is proportional to the ciphertext polynomial degree and inversely proportional to the coefficient size). To achieve 256 bits of security, one can double the polynomial degree relative to parameter sets that yield 128 bits of security. We remark that the tradeoff for increasing security is larger memory overheads (due to the ciphertext size growing by $2\times$) and increased latency as the polynomial operations become more expensive and the computational complexity of bootstrapping increases. For all three modes of HELM, we find that the latency of the individual encrypted circuit elements (e.g., gates and lookup tables) exhibit a latency increase of approximately $4\times$ for 256 bits of security relative to the 128 bits of security baseline used for our experiments. Therefore, it is expected that the application runtimes will be similarly impacted as all core encrypted operations exhibit near-identical slowdown.

5.5 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 remark that all modes result in negligible overheads for the client; encryption and decryption for all benchmarks take hundreds of milliseconds (at most) and key generation requires between one and three seconds on our experimental server for all modes. 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. 4 and 5 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. 6, 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. The recent Systematization of the Knowledge (SoK) work of the T2 compiler [8] 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. Furthermore, similarly to T2, the E^3 [36] 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 [9] surveyed state-of-the-art FHE compilers like CHET [37], Cingulata [38], and EVA [11].

CHET is designed for CKKS and is geared towards private neural network inference for HEAAN and SEAL, while EVA [11] employs a Domain-Specific Language (DSL) for vector arithmetic focuses on SEAL and CKKS. The Cingulata compiler toolchain [38] 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 [39] 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 [40], 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. The HECO compiler [41] translates high-level programs into Microsoft SEAL programs and automatically determines appropriate batching strategies for efficient SIMD-style computation. On the other hand, HELM focuses on the CGGI cryptosystem and reducing latency instead of primarily throughput. Lastly, HECO only supports leveled HE contexts in its present state, which makes it unsuitable for evaluating programs with a large depth. Google's Transpiler [10] and ROMEO [29] 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. Lastly, Zama's Concrete compiler [24] provides high-level frontends to a Rust implementation of the CGGI cryptosystem. For example, users can write functions in Python with standard operators and a subset of NumPy operations and inform the compiler explicitly which inputs are encrypted and which are plaintext. Unlike HELM, the compiler also requires a set of representative input values to generate the resulting FHE circuit. While this can serve as a way to derive suitable datatypes for the different input variables, if the actual sensitive inputs deviate from the provided input set by more than a marginal amount, it can result in incorrect results during runtime. Additionally, the Concrete compiler requires users to adapt to the quirks of encrypted computing, such as the inability to make control-flow decisions on encrypted data. On the other hand, control-flow decisions in HELM (and other synthesis-based frameworks like Romeo and the Google Transpiler) are resolved implicitly by the synthesis process, which always results in a circuit composed strictly of combinational and sequential elements that can be evaluated through the FHE backend. Lastly, users must balance correctness guarantees and latency as programs with negligible probabilities of error require larger parameters. Compared to Concrete, HELM does not require users to adapt to the HE programming model as any synthesizable Verilog code is compatible. As shown in Section 5, HELM outperforms Concrete by a small amount for arithmetic-intensive benchmarks including neural network inference and matrix multiplication. However, for benchmarks consisting of predominantly Boolean operations like AES, we observe that HELM achieves significant speedups over Concrete.

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 allows for efficient arbitrary 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. Developing encrypted programs with CGGI is a daunting task as users need to express their algorithms as Boolean circuits. Many works such as [8, 10, 24, 36, 38] 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 complicated. Even with high-level synthesis tools as adopted by [10] (i.e., the input is C++ programs and target is CGGI), simple FHE computation like the Chi-squared test takes around 40 seconds with [38] and [8], over 2 minutes with [10], and over 3.6 minutes with [36]. On the other hand, Chi-squared in HELM takes 10 seconds because our starting point is Verilog and we leverage Boolean optimization toolchains. Finally, HELM uses aggressive multi-threading, achieving significantly faster encrypted evaluation compared to the related works.

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, writing HDL programs is far more high-level than the API that most FHE libraries provide, which requires users to 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 [42] and nuFHE [43] 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 is not possible. The REDcuFHE [23] project overhauls the cuFHE CGGI library to introduce support for multi-bit plaintext and multi-GPU setups. However, a drawback of REDcuFHE is that it places the burden of scheduling and managing device communication squarely on the programmer's shoulders, resulting in usability challenges. Additionally, the programming model requires developers to express their programs in a Boolean circuit format directly. Lastly, the work of ArctyrEX [44] also provides multi-GPU compatibility and automates scheduling and communication procedures. Unlike REDcuFHE, ArctyrEX's approach eliminates the need for manual intervention from programmers. However, the reliance on high-level synthesis (HLS) results in high preprocessing costs (which imposes restrictions on the program size) and potentially suboptimal circuit generation. None of the aforementioned libraries consider LUTs and HELM offers great performance while offering support for programming in Verilog as opposed to Boolean circuits.

7 Concluding Remarks

In this work, we introduce the HELM framework for automated conversion from synthesizable Verilog HDL designs to encrypted circuits for private outsourcing. In HELM, HDL designs are converted to netlists through synthesis, while our compiler creates an internal view of the circuit described by the netlist and determines the correct execution order for the homomorphic gate evaluations. The resulting encrypted circuit is uploaded to a remote service for encrypted evaluation with HELM along with encrypted inputs.

We evaluate HELM with circuits from the ISCAS '85 and '89 benchmark suites, and real-life workloads, such as AES and encrypted image filtering. We observed a 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. D. Micciancio, "A first glimpse of cryptography's holy grail," *Communications of the ACM*, vol. 53, no. 3, p. 96, 2010.

2. C. Gentry, “A fully homomorphic encryption scheme,” Ph.D. dissertation, Stanford University, 2009.
3. “Microsoft SEAL (release 4.1),” https://github.com/Microsoft/SEAL, Jan. 2023, microsoft Research, Redmond, WA.
4. “Lattigo v5,” Online: https://github.com/tuneinsight/lattigo, Nov. 2023, ePFL-LDS, Tune Insight SA.
5. J. Fan and F. Vercauteren, “Somewhat practical fully homomorphic encryption,” Cryptology ePrint Archive, Report 2012/144, 2012. [Online]. Available: https://eprint.iacr.org/2012/144
6. Z. Brakerski, C. Gentry, and V. Vaikuntanathan, “(Leveled) fully homomorphic encryption without bootstrapping,” in *ITCS 2012*, S. Goldwasser, Ed. ACM, Jan. 2012, pp. 309–325.
7. J. H. Cheon, A. Kim, M. Kim, and Y. S. Song, “Homomorphic encryption for arithmetic of approximate numbers,” in *ASIACRYPT 2017, Part I*, ser. LNCS, T. Takagi and T. Peyrin, Eds., vol. 10624, Dec. 2017, pp. 409–437.
8. C. Gouert, D. Mouris, and N. G. Tsoutsos, “SoK: New Insights into Fully Homomorphic Encryption Libraries via Standardized Benchmarks,” *PoPETs*, vol. 2023, no. 3, pp. 154–172, Jul. 2023.
9. A. Viand, P. Jattke, and A. Hithnawi, “SoK: Fully homomorphic encryption compilers,” in *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2021, pp. 1092–1108.
10. 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. [Online]. Available: https://eprint.iacr.org/2021/811
11. R. Dathathri *et al.*, “EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation,” in *Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 546–561.
12. 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.
13. G. S. Cetin, E. Savaş, and B. Sunar, “Homomorphic sorting with better scalability,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 4, pp. 760–771, 2020.
14. A. Chatterjee and I. Sengupta, “Sorting of fully homomorphic encrypted cloud data: Can partitioning be effective?” *IEEE Transactions on Services Computing*, vol. 13, no. 3, pp. 545–558, 2017.
15. ——, “Translating algorithms to handle fully homomorphic encrypted data on the cloud,” *IEEE Transactions on Cloud Computing*, vol. 6, no. 1, pp. 287–300, 2015.
16. 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.
17. C. Gentry, “Computing on encrypted data (invited talk),” in *CANS 09*, ser. LNCS, J. A. Garay, A. Miyaji, and A. Otsuka, Eds., vol. 5888, Dec. 2009, p. 477.
18. L. Ducas and D. Micciancio, “FHEW: Bootstrapping homomorphic encryption in less than a second,” in *EUROCRYPT 2015, Part I*, ser. LNCS, E. Oswald and M. Fischlin, Eds., vol. 9056, Apr. 2015, pp. 617–640.
19. 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)*. Springer, 2021, pp. 1–19.
20. Z. Liu, D. Micciancio, and Y. Polyakov, “Large-precision homomorphic sign evaluation using FHEW/TFHE bootstrapping,” in *ASIACRYPT 2022, Part II*, ser. LNCS, S. Agrawal and D. Lin, Eds., vol. 13792, Dec. 2022, pp. 130–160.
21. 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.
22. F. Bourse, M. Minelli, M. Minihold, and P. Paillier, “Fast homomorphic evaluation of deep discretized neural networks,” in *CRYPTO 2018, Part III*, ser. LNCS, H. Shacham and A. Boldyreva, Eds., vol. 10993, Aug. 2018, pp. 483–512.
23. L. Folkerts, C. Gouert, and N. G. Tsoutsos, “REDsec: Running encrypted DNNs in seconds,” in *NDSS 2023*. The Internet Society, Feb. 2023.
24. Zama, “Concrete: TFHE Compiler that converts python programs into FHE equivalent,” 2022, https://github.com/zama-ai/concrete.
25. C. Wolf, “Yosys open synthesis suite,” http://www.clifford.at/yosys/.
26. 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.
27. M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter *et al.*, “Homomorphic encryption standard,” *Protecting privacy through homomorphic encryption*, pp. 31–62, 2021.

28. A. Al Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee *et al.*, "Openfhe: Open-source fully homomorphic encryption library," in *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2022, pp. 53–63.
29. 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.
30. A. Becker, L. Ducas, N. Gama, and T. Laarhoven, "New directions in nearest neighbor searching with applications to lattice sieving," in *27th SODA*, R. Krauthgamer, Ed. ACM-SIAM, Jan. 2016, pp. 10–24.
31. M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," Cryptology ePrint Archive, Report 2015/046, 2015. [Online]. Available: https://eprint.iacr.org/2015/046
32. 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.
33. C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit," in *CRYPTO 2012*, ser. LNCS, R. Safavi-Naini and R. Canetti, Eds., vol. 7417, Aug. 2012, pp. 850–867.
34. R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of eugenics*, vol. 7, no. 2, pp. 179–188, 1936.
35. R. E. Ali, J. So, and A. S. Avestimehr, "On polynomial approximations for privacy-preserving and verifiable relu networks," *arXiv preprint arXiv:2011.05530*, 2020.
36. 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. [Online]. Available: https://eprint.iacr.org/2018/1013
37. R. Dathathri *et al.*, "CHET: An Optimizing Compiler for Fully-Homomorphic Neural-Network Inferencing," in *Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 142–156.
38. 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.
39. N. Barlow, T. Lazauskas, O. Strickson, and A. Gascon, "SHEEP: A homomorphic encryption evaluation platform," Online, 2019, https://github.com/alan-turing-institute/SHEEP.
40. A. Viand and H. Shafagh, "Marble: Making fully homomorphic encryption accessible to all," in *6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 49–60.
41. A. Viand, P. Jattke, M. Haller, and A. Hithnawi, "{HECO}: Fully homomorphic encryption compiler," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4715–4732.
42. W. Dai and B. Sunar, "cuFHE: CUDA-accelerated Fully Homomorphic Encryption Library," https://github.com/vernamlab/cuFHE, 2018.
43. NuCypher, "NuFHE, a GPU-powered Torus FHE implementation," https://github.com/nucypher/nufhe, 2019.
44. 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. [Online]. Available: https://eprint.iacr.org/2023/641