Data Privacy Made Easy: Enhancing Applications with Homomorphic Encryption

Charles Gouert, and Nektarios Georgios Tsoutsos

Abstract

Homomorphic encryption is a powerful privacy-preserving technology that is notoriously difficult to configure and use, even for experts. The key difficulties include restrictive programming models of homomorphic schemes and choosing suitable parameters for an application. In this tutorial, we outline methodologies to solve these issues and allow for conversion of any application to the encrypted domain using both leveled and fully homomorphic encryption.

The first approach, called Walrus, is suitable for arithmetic-intensive applications with limited depth and applications with high throughput requirements. Walrus provides an intuitive programming interface and handles parameterization automatically by analyzing the application and gathering statistics such as homomorphic noise growth to derive a parameter set tuned specifically for the application. We provide an in-depth example of this approach in the form of a neural network inference as well as guidelines for using Walrus effectively.

Conversely, the second approach (HELM) takes existing HDL designs and converts them to the encrypted domain for secure outsourcing on powerful cloud servers. Unlike Walrus, HELM supports FHE backends and is well-suited for complex applications. At a high level, HELM consumes netlists and is capable of performing logic gate operations homomorphically on encryptions of individual bits. HELM incorporates both CPU and GPU acceleration by taking advantage of the inherent parallelism provided by Boolean circuits. As a case study, we walk through the process of taking an off-the-shelf HDL design in the form of AES-128 decryption and running it in the encrypted domain with HELM.

I. Introduction

Fully homomorphic encryption (FHE) has received increasing attention in the research community and industry as new cryptographic schemes and application-specific optimizations have demonstrated that the technology is fast approaching feasibility in realistic scenarios. This powerful class of encryption allows

C. Gouert and N. G. Tsoutsos are with the Department of Electrical and Computer Engineering, University of Delaware, Newark, DE. E-mail: {cgouert, tsoutsos}@udel.edu.

users to execute algorithms on encrypted data and serves as a robust solution to privacy concerns in the context of cloud computing. Users can encrypt sensitive data locally, upload the resulting ciphertexts to the cloud, instruct the cloud to execute an algorithm on the ciphertexts, receive the encrypted output, and decrypt it locally to view the correct plaintext result of the computation. In this way, organizations can outsource computationally expensive calculations to the cloud, while having strong data privacy guarantees.

While standard encryption mechanisms such as AES have been used to enforce data confidentiality for outsourced data in the past, these solutions only work for storage and transmission. In this case, if any meaningful computation needs to be done on the encrypted data, the user would need to pull the data from the cloud, decrypt it, perform the computation locally on the plaintext data, re-encrypt it, and finally re-upload to the cloud. Conversely, HE can help accomplish the same task without engaging the user, since the computation can be performed directly on the encrypted data stored on the cloud. Because the *data in use* remains encrypted with HE, the cloud has no knowledge of the underlying plaintext. In addition, if a cloud server is compromised, the attacker will be unable to decrypt the ciphertexts without the secret key (known only to the user). A diagram depicting this secure cloud computing scenario is shown in Figure 2.

Only a few years ago, the idea that FHE could be adopted in a cloud computing context outside of research circles was a distant dream. When first conceived in 2009 by Gentry [1], it was estimated that the cost of performing a Google search over encrypted data would be a trillion times slower than a regular search [2]. Now, state-of-the-art FHE libraries [3] and novel computational techniques tailored for encrypted computation [4] have accelerated the evaluation time of homomorphic algorithms by several orders of magnitude. Further, leveled homomorphic encryption (LHE) provides an efficient alternative to FHE and can achieve much faster speeds for certain types of applications. Even still, software optimizations and hardware acceleration has led to significant speedups over the years. General trends in terms of performance since the inception of FHE is shown in Figure 1.

Unfortunately, while the efficiency of both FHE and LHE operations continues to increase, usability is not advancing at the same speed. For instance, choosing optimal parameter sets at an acceptable security level for most HE libraries remains a notoriously difficult problem. While one set of parameters may result in efficient computation for one application, the same set of parameters may result in poor performance for a different set of computations on encrypted data. Even for crypto-savvy programmers, the process of optimizing encryption parameters is time consuming and requires a high degree of trial-and-error.

Additionally, the programming models associated with FHE and LHE are somewhat restrictive and users must write their applications in the form of a directed acyclic graph (DAG). Depending on the

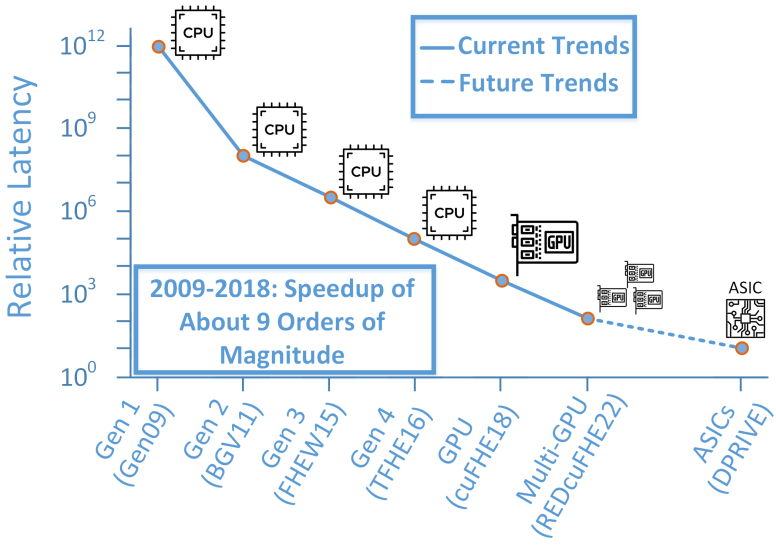

Fig. 1: **FHE Speed Trend:** Early FHE was prohibitively expensive, but was improved over time with software-oriented optimizations. Eventually, these improvements stagnated and further speedups were gained through hardware acceleration in the form of GPUs and ASICs. The DPRIVE initiative started by the US Defense Advanced Research Projects Agency (DARPA) for FHE ASICs aims for a $10\times$ slowdown relative to plaintext computation in the near future.

scheme and configuration, the DAG must take the form of a Boolean circuit composed of standard logic gates or an arithmetic circuit with solely addition and multiplication nodes. This design approach is not scalable for complex applications such as neural network inference, which may require circuits composed of millions of gates or arithmetic operations.

Prior work has identified efficient and secure parameter sets, most notably the homomorphic encryption standard of 2018 [5], and existing HE libraries typically provide a set of default parameters, but these parameter sets are not tailored for any given application. In addition, the level of security may not be appropriate for some users as many parameter sets are preconfigured for 80 bits of security or less, which today is considered relatively low. For HE schemes such as BGV [6], default parameter sets may not be well suited for high degrees of *batching*. The latter is a very powerful technique applicable to a subset of HE schemes that allows multiple plaintext values to be encoded in one ciphertext. When an HE operation is conducted with a batched ciphertext, each plaintext element will be affected, in a similar way to SIMD-style computation. For many types of applications, such as neural network training and inference, this strategy can drastically improve throughput. The size of the plaintext vector that can be encoded is determined by the parameter set and can vary from less than 10 to several thousand. One can

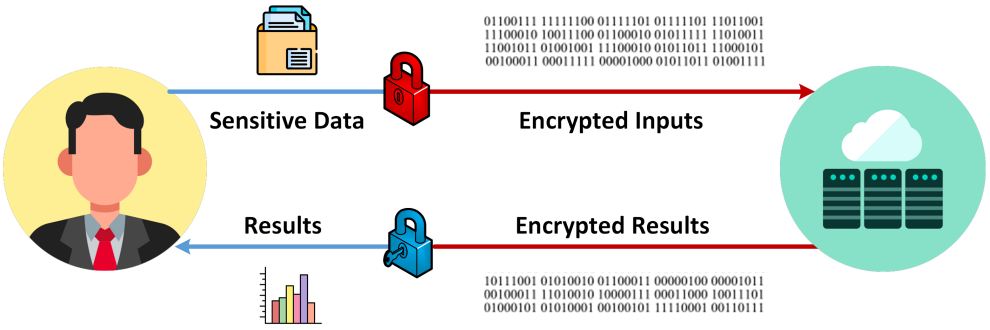

Fig. 2: **Secure Outsourced Computation:** A user has sensitive plaintext, encrypts the data, sends the ciphertext (in red) to a cloud server that computes analytics, and returns the encrypted results to the user, who decrypts them locally.

achieve speedups of orders of magnitude compared to ill-suited configurations when selecting parameters well tuned for the target computation.

In terms of usability, several compilers have aimed to allow programmers to express algorithms in high-level code and automatically map the input programs into a series of HE primitives. For instance, the TenSEAL library [7] provides a high-level Python API for linear algebra operations between encrypted tensors of arbitrary shape. Under the hood, the linear algebra operations (like matrix multiplication) are mapped to a sequence of encrypted arithmetic operations with the Microsoft SEAL library [8], which implements the BFV, BGV, and CKKS schemes that operate over encrypted integers or floats.

This tutorial aims to address the problem of selecting good parameters for any given application and to allow users to more easily wield the full power that current state-of-the-art HE libraries provide. To this end, we introduce the following:

- A methodology for seamlessly evaluating existing HDL designs with FHE,
- Automatic parameterization techniques for efficient LHE evaluation,
- A case-study applying the developed techniques for two realistic use-cases in the form of AES decryption and neural network inference.

The next section gives a high-level overview of homomorphic encryption and its different variants, as well as a brief look at some of the most important HE schemes. Then, we identify methodologies for optimized parameter tuning and converting existing designs for LHE, followed by FHE. For both cases, we demonstrate methodologies to convert non-trivial applications to the encrypted domain and analyze the performance.

II. Homomorphic Encryption Primer

Homomorphic encryption can be subdivided into different categories that define their relative computational powers. The most limited type of HE is known as partial homomorphic encryption, or PHE. These cryptosystems allow for unbounded multiplication [9], [10] *or* addition [11], [12] on encrypted data. Notably, PHE can not be used to implement arbitrary algorithms with only one arithmetic operator, which enables only simple use cases, such as aggregation.

Leveled HE (LHE), on the other hand, supports both addition *and* multiplication over ciphertext data. Therefore, this class of HE exposes a functionally complete set of operations and can be used to implement arbitrary algorithms. The only caveat is that the algorithm must have a *bounded depth*. LHE constructions rely on variants of the *learning with errors (LWE)* problem [13] for security guarantees (unlike PHE which relies on other hard problems like factorization). The key ramification is that as a plaintext is encrypted with LHE, a small amount of noise is injected to make the encryption harder to break without knowledge of the secret key. Consequently, the noise grows as additions and multiplications are conducted on the encrypted data. The noise grows linearly when two ciphertexts are added together, while multiplication between two ciphertexts causes an exponential noise growth. Once the magnitude of the noise exceeds a threshold, with high probability the noisy ciphertext will not yield the expected result when decrypted, which offers limited utility for the client. Nevetheless, if the depth of the application, defined by the number of multiplications performed successively on ciphertext data, is bounded, it is possible to choose parameter sets that will allow the LHE scheme to successfully evaluate the entire circuit without exceeding the noise threshold. At the same time, very deep circuits require large parameter sets, resulting in significant memory and computational overheads.

Three of the most popular LHE schemes include BFV [14], BGV [6], and CKKS [15]. BGV and BFV encrypt integers modulo a configurable plaintext modulus, while CKKS encrypts floating point or complex numbers. In these schemes, a ciphertext is represented as a pair of high-degree polynomials with large coefficients modulo a high-degree irreducible cyclotomic polynomial of order m, which limits the maximum polynomial degree. The size of each coefficient is reduced by a ciphertext modulus q, which is a product of primes. In practice, m defines the number of coefficients in each ciphertext polynomial and q defines the coefficient size. As a result, the overall ciphertext size scales with both q and m. More formally, each ciphertext consists of a pair of polynomials in $\mathbb{Z}_q[X]/(X^m + 1)$.

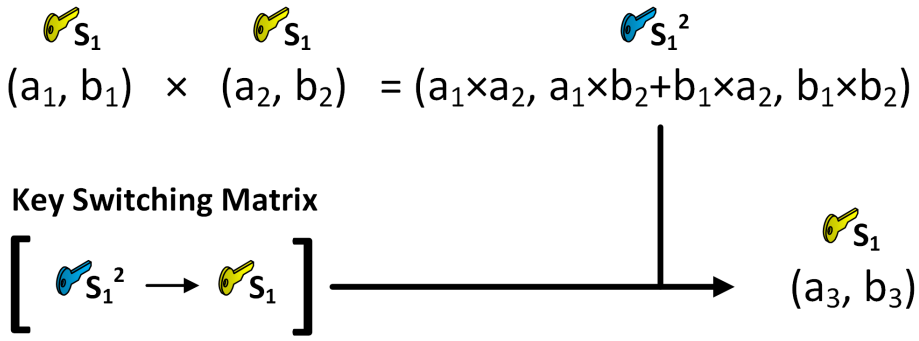

Fig. 4: **Key Switching:** HE multiplication results in a larger product ciphertext that is encrypted under the square of the secret key. Key switching serves to reduce the dimensionality of the product as well as remap the ciphertext back to a secret key.

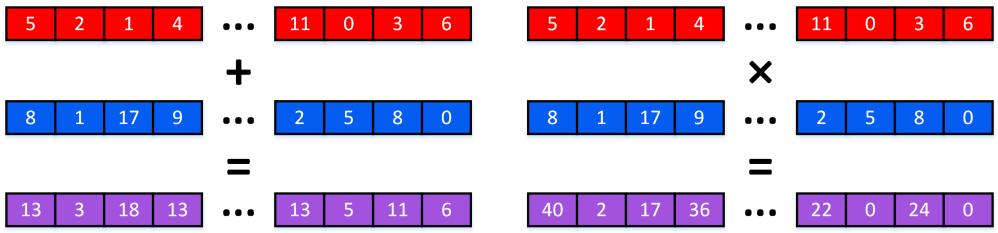

Fig. 3: **Batched Ciphertexts:** LHE enables vectors of plaintext elements to be encrypted into a single ciphertext. When two batched ciphertexts are added, the behavior is identical to vector addition. Multiplication parallels element-wise multiplication between two vectors. The power of batching lies in its ability to enable parallel arithmetic over independent values with no additional cost, thus vastly improving the throughput of HE operations.

These LHE schemes also support batching, which allows for vector-like processing that can vastly improve the throughput of homomorphic operations. Instead of encrypting scalars and single values, one can encrypt vectors of integers or floating point numbers into a single ciphertext. When encrypted arithmetic is performed between ciphertexts, each vector element is affected individually in a similar fashion to SIMD instructions. Notably, additions and multiplications across ciphertexts encrypting vectors incur the same cost as ciphertexts encrypting single values assuming the same q and m. An example of addition with two batched ciphertexts is depicted in Figure 3. Depending on the polynomial degree selected, a single ciphertext can hold potentially thousands of independent plaintext elements.

Addition between two ciphertexts is straightforward as it is simply a set of additions between corresponding tuples of polynomials in the two ciphertexts. On the other hand, multiplication is more expensive

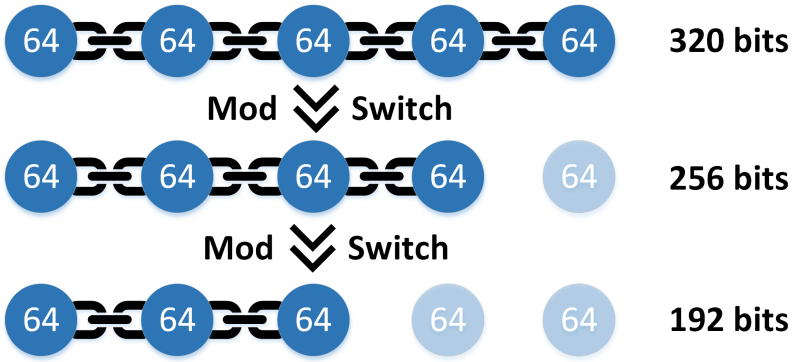

Fig. 5: **Modulus Switching:** This operation manipulates the chain of primes (of size 64 bits in this figure) composing the ciphertext modulus. Each invocation removes a prime from the chain and hence decreases the bit size of all coefficients in the ciphertext.

and results in undesirable side effects. Multiplication involves multiplying the pairs of polynomials together, which effectively introduces a third large polynomial to the product ciphertext. Consequently, each subsequent multiplication will cause the ciphertext to become larger and larger and take up increasing amounts of memory. Luckily, an operation called *key switching* can be used to deal with this issue and effectively constrain the ciphertexts to their original sizes. Key switching also tackles an orthogonal issue related to the secret key: part of the product ciphertext is actually encrypted under the *square of the secret key*. This must be switched back to the original secret key (hence the name key switching), or a different secret key to avoid assuming circular security. Key switching will output a ciphertext with only two polynomials and more noise. This operation following a multiplication is illustrated in Figure 4.

A *modulus switching* operation will typically follow a multiplication (and subsequent key switch). This operation solely affects the ciphertext modulus q, which can be thought of as a chain of prime numbers, as shown in Figure 5. By removing one of the primes from the chain, the ciphertext modulus will correspondingly decrease by the bit size of the removed prime. This effectively scales down the magnitude of the ciphertext coefficients, which also serves to scale down the magnitude of the noise. However, this operation can only be done a limited number of times because there is a finite number of primes in the chain. Even with batching and modulus switching, LHE becomes infeasible for complex algorithms as the ciphertext sizes will scale with the number of multiplications required to evaluate the target circuit. Consequently, arithmetic operations become more expensive with higher polynomial degrees and coefficient sizes in each ciphertext; additionally, the memory requirements increase as well for the same reasons. Therefore, for algorithms exhibiting a large multiplicative depth, or algorithms with an unbounded number of operations, it is more appropriate to use a most powerful form of HE.

Fig. 6: **Bootstrapping:** Starting with a noisy ciphertext under secret key S_1, represented by a red circle, bootstrapping will add another layer of encryption by encrypting the ciphertext under key S_2 with less noise. Next, the inner layer of encryption is stripped away by executing the decryption function homomorphically. The result is a ciphertext under key S_2 encrypting the same message with reduced noise.

Fully homomorphic encryption (FHE) is identical to LHE besides the inclusion of the most effective (and expensive) form of noise reduction, called *bootstrapping*. Consider the following scenario in an LHE context: the cloud computes on a ciphertext until it becomes noisy and then asks the user to refresh the noise on behalf of the cloud. The user can do this by downloading the ciphertext, decrypting it, and then sending the cloud a fresh encryption of the underlying plaintext value in the original ciphertext. Now, the cloud can proceed to do more operations with the ciphertext. However, this is a suboptimal solution as it requires the user to be active during the computation, increases the communication overhead associated with sending a large ciphertext between the cloud server to the user, and it is against the principle of using HE in the first place. Bootstrapping employs the intuition from this scenario and allows it to happen entirely on the cloud server without any interaction from the user.

In more details, bootstrapping performs a decryption procedure *in the encrypted domain*, as illustrated in Figure 6. In order to accomplish this, the user sends the cloud an encryption of their secret key, referred to as the evaluation key. One can envision this as happening inside another layer of encryption (i.e., the green circle), so the plaintext value is never exposed. After evaluating the homomorphic decryption procedure (with the encrypted key), the result is a new ciphertext encoding the same underlying value as the original noisy ciphertext, but its noise magnitude is reduced to a level that allows for more HE operations to be executed. Here, it is important to consider the depth of the decryption circuit, which adds further noise before the total noise can ultimately be reduced, and thus requires some remaining noise budget to work correctly. Additionally, the complexity of the decryption circuit will affect the amount of noise capable of being removed by the bootstrapping procedure itself. Generally, the larger the plaintext modulus, the deeper the decryption circuit becomes.

Unlike other noise reducing techniques like modulus switching, bootstrapping can be invoked an unlimited number of times, allowing FHE schemes to execute algorithms with *unbounded depth*. It

can also be combined with modulus switching, where the latter is used as an initial noise mitigation technique until no more primes can be removed, and then bootstrapping can be invoked. Bootstrapping will regenerate ciphertext primes, so modulus switching can be used in between subsequent bootstraps.

All LHE schemes can become FHE schemes with the inclusion of bootstrapping. Unfortunately, this ability comes with a steep cost: bootstrapping is the major bottleneck in all FHE schemes. For BGV, CKKS, and BFV, this procedure can take dozens of seconds to several minutes depending on parameters. However, while newer schemes have improved bootstrapping latency, these schemes maintain the highest throughput per bootstrap due to batching capabilities.

The first latency-optimized bootstrapping scheme is FHEW [16], which introduced a much faster bootstrapping mechanism that takes less than a second to evaluate on a CPU. This scheme encrypts single bits of plaintext as individual ciphertexts and allows users to evaluate homomorphic Boolean gate operations. Contrary to prior schemes, bootstrapping plays a key role in evaluating the logic gates, so each gate operation requires a bootstrap, with the exception of the `NOT` gate. The bootstrapping operation can be used to perform a programmable mapping operation; for instance, an encryption of `1` can be mapped to an encryption of `0` for no added cost during the bootstrap. The FHEW scheme was improved by CGGI [3], which accelerated bootstrapping to approximately ten milliseconds. In the case of parameterizing these libraries, the key consideration is the desired plaintext space. While both schemes naturally support encoding of bits, the bootstrapping that plays a pivotal role in the computation of logic gates can be extended to evaluate arbitrary univariate functions over multi-bit plaintexts in the form of a programmable lookup table (LUT). The precision of the lookup table (i.e., how many plaintext bits can be mapped to a specified output) is entirely dependent on parameter choice, with higher precision LUTs requiring larger polynomial degrees. In this tutorial, we chose CGGI as a key focus for arbitrary FHE computation due to its fast bootstrapping speeds and ability to perform non-linear operations with ease through Boolean circuits.

III. Converting Applications for LHE Evaluation

LHE is the optimal choice for applications that require high throughput and exhibit a limited multiplicative depth. Additionally, arithmetic-intensive applications are natural candidates for LHE evaluation as all popular LHE schemes encode multi-bit plaintexts and support multiplication and addition as primitive operations. For these reasons, LHE is widely used for shallow neural networks and other applications that incorporate linear algebra operations [7], [17], [18]. This section presents an intuitive methodology for user-driven parameterization and development, as well as an introduction of a framework built on top

of the Microsoft SEAL HE library [8] to rapidly develop LHE applications by incorporating automated techniques for parameter selection.

A. Development with Manual Parameterization

Implementations of BGV [6], BFV [14], and CKKS [15] typically grant users a very high degree of freedom when it comes to customizing a wide variety of encryption parameters. Most importantly, the user can select the size of the prime chain (and hence the ciphertext modulus) and the degree of the irreducible cyclotomic polynomial (which serves to constrain the number of coefficients in ciphertext polynomials).

These two parameters must be balanced to achieve a suitable level of security. On one hand, having a longer prime chain allows one to increase the number of possible modulus switches, allowing for a greater number of leveled operations. However, the size of the prime chain has a negative impact on security and must be minimized to achieve just enough levels to evaluate any given application. On the other hand, increasing the polynomial degree has a positive impact on security. This parameter also serves to govern the speed of homomorphic operations; operations involving ciphertexts with larger polynomials will be slower.

In our experience, the most straightforward methodology for finding the optimal parameters for a given application involves first focusing on tuning the ciphertext modulus size and then altering the polynomial degree to achieve a desired security level and number of slots. The first step involves analyzing the application and identifying the *multiplicative depth*, i.e., the maximum number of multiplications that happen on a single ciphertext. For complex applications, this can be non-trivial, in which case a rough approximation can be used instead. Once this is determined, a general rule of thumb is that you should add $d + 1$ primes to the chain, where d is the multiplicative depth. The size of the primes vary per implementation, but usually the primes in the chain are chosen to be approximately the same size. For CKKS, it's essential that the intermediate primes are approximately the same bitwidth in order to keep the scale factor consist after a rescaling operation. However, it is most common for the first and last prime to be larger than the intermediate primes to maintain a high precision for encryption and decryption. Consequently, since the composite number of the prime chain is the modulus of the polynomial coefficients of ciphertext polynomials, the larger the prime chain, the more memory is required to hold ciphertext data.

Once the bit size of the initial ciphertext modulus is decided, one can then run multiple iterations of the HE program to fine-tune this value. It is recommended to use small polynomial degrees for these tests (i.e., up to degree 4096) which yield low security in practice but can allow one to rapidly verify

TABLE I: **Walrus API**: Walrus provides a set of high-level functions that allow users to encode plaintexts, encrypt and decrypt ciphertexts, and conduct operations between ciphertexts as well as operations between ciphertexts and plaintexts. For simplicity, we omit the public key material that must be passed to the functions for encrypted evaluation. Here, ct_i represent ciphertexts, pt_i represents an encoded plaintext, while A represents a public scalar that is neither encoded nor encrypted.

Type	Function	Operands			Description
Enc/Dec	`WalrusEncryptSingle`	ct	A		$ct = Enc(A)$
	`WalrusEncryptVector`	ct	$A_0...A_N$		$ct = Enc(A_0...A_N)$
	`WalrusDecryptSingle`	A	ct		$A = Dec(ct)$
	`WalrusDecryptVector`	$A_0...A_N$	ct		$A_0...A_N = Dec(ct)$
Arithmetic	`WalrusAdd`	ct_0	ct_1	ct_2	$ct_0 = ct_1 + ct_2$
	`WalrusAddPlain`	ct_0	ct_1	pt	$ct_0 = ct_1 + pt$
	`WalrusSub`	ct_0	ct_1	ct_2	$ct_0 = ct_1 - ct_2$
	`WalrusSubPlain`	ct_0	ct_1	pt	$ct_0 = ct_1 - pt$
	`WalrusMult`	ct_0	ct_1	ct_2	$ct_0 = ct_1 \times ct_2$
	`WalrusMultPlain`	ct_0	ct_1	pt	$ct_0 = ct_1 \times pt$

correctness. If the final decrypted answer matches the expected value for given input pairs across multiple runs, the modulus is sufficiently large. Lastly, the modulus should be decreased until the answer becomes non-deterministic (meaning that the result is too noisy) and the last size that resulted in correct evaluation should be selected as the final modulus size. This entire process is tedious even for experts and can be extremely time-consuming for large applications like encrypted neural network inference.

B. Rapid Development with Walrus

In this tutorial, we discuss an FHE parameter optimization and evaluation library called *Walrus*, which utilizes SEAL as a cryptographic backend, provides a friendly C++ frontend, and requires no manual parameterization on behalf of users. An overview of the user interface for HE operations with Walrus is shown in Table I. Additionally, the `WalrusParameterize` function is invoked to parameterize the program and is invoked with the output ciphertexts; the returned SEAL parameter set can be used to facilitate the actual homomorphic computation. To facilitate this, Walrus applications are executed in two passes; a first pass will be executed on the cloud side with dummy data that will serve to gather various statistics about the arithmetic circuit representing the application. Conversely, the second pass will be run on the cloud using the parameters derived in the initial pass and processes the actual

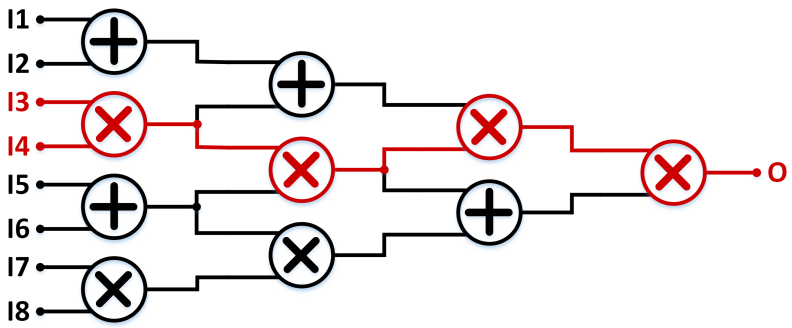

Fig. 7: **Critical Noise Path:** The red path in this arithmetic circuit indicates the path with maximal noise growth; the first pass in Walrus identifies this path and chooses parameters accordingly.

encrypted inputs supplied by the user. Specifically, Walrus keeps track of the number of additions and multiplications conducted with a ciphertext as well as the noise budget remaining before a ciphertext is rendered undecryptable with high probability at each point in the circuit.

After the first pass, Walrus analyzes the gathered circuit statistics and deduces the critical path of the circuit in terms of noise growth. Figure 7 visualizes the critical path with respect to noise growth (highlighted in red) for a simple arithmetic circuit. With this information, Walrus chooses an appropriate parameter set that will ensure correct decryption for the target application. The user can optionally specify additional constraints, such as the plaintext precision or a minimum number of ciphertext slots for applications where high throughput is necessary. Walrus takes these constraints and the maximum required noise budget and chooses the smallest parameter set from a collection of efficient, pre-built parameter sets supplied by the underlying SEAL library that satisfies all provided constraints.

In some cases, Walrus will be unable to find a parameter set that satisfies either the noise budget or the user-provided constraints. For the former case, Microsoft SEAL only supports polynomial degrees up to 2^{15} and does not support FHE evaluation for arbitrarily deep applications. As a result, SEAL could only be used to evaluate arithmetic circuits with a maximum multiplicative depth of approximately fourteen; if the critical noise path contains more than fourteen multiplications from an input to an output, then it is impossible to correctly evaluate the circuit with the SEAL backend. Instead, these applications can be executed with the methodology presented in the next section about converting applications for FHE evaluation. Likewise, for the latter case, the user may request a plaintext precision that can't be achieved with the SEAL parameter sets or the desired number of slots is unachievable.

After Walrus performs the first pass, it chooses appropriate cryptographic parameters for secure and efficient outsourced evaluation. On the second pass through the circuit, Walrus consumes encrypted inputs

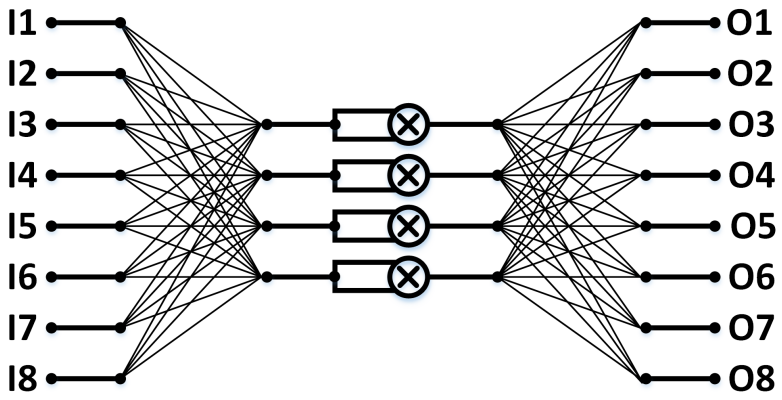

Fig. 8: **Neural Network Inference:** We present an architecture with eight input and output neurons and four hidden neurons. The network consists of two fully-connected layers with a square activation function in between.

(using the parameters derived from the first pass), executes each HE operation on the cloud side, and then returns the encrypted results back to the client for decryption.

C. A Case Study: Neural Network Inference

Privacy-preserving neural network inference is a popular use-case for HE and is well-suited for MLaaS (machine-learning as a service), where a cloud service provider that owns an ML model can offer classification services for client inputs. MLaaS can be utilized for a wide variety of applications, such as categorizing images and aiding in medical diagnoses. However, regulations such as HIPAA prevent health practitioners from sharing patient health data with third party cloud servers. In this case, HE can be utilized to encrypt the medical data and allow the cloud to run the ML model directly on the encrypted data.

As a case study, we will construct a neural network with an architecture resembling Figure 8, yet with a larger number of neurons. The first and third layer are fully-connected layers, which are akin to matrix-vector products between the vectorized inputs and a matrix of trained weights. In between the fully-connected layers is an activation function, which in this case is a square function (i.e., $y = x^2$). The square function was chosen because it can be implemented with one HE multiplication. Other popular non-linear activation functions, such as the rectified linear unit (ReLU) function or the sigmoid function, are non-polynomial and can't be implemented directly with HE. The only recourse for utilizing these functions is to use a polynomial approximation, which can add unacceptable multiplicative depth to a network if close approximations of the non-polynomial functions are required.

Notably, the network parameters, such as the weights, are not encrypted as we assume that the cloud owns the ML model and therefore their plaintext values are known to the computing party (i.e., the cloud server). In the case where the client also owns the model, the weights can be intuitively encrypted in the same fashion as the inputs and uploaded to the cloud. However, the latency will increase significantly as mixed operations between plaintext and ciphertext are less expensive than operations between two ciphertexts. In fact, prior work has found that using encrypted weights versus plaintext weights can result in a slowdown of over $2\times$ for a similar network for MNIST classification [19].

```
vector<vector<Plaintext>> cloud_weights;
void cloud_generate_weights() {
  // Create random generator (Mersenne twister with a state size of 19937
     bits)
  random_device rd;
  mt19937 gen(rd());
  // Set the range of the generator (non-zero 4 bit numbers)
  uniform_int_distribution<uint64_t> distribution(1, 15);
  // Generate random numbers and convert to SEAL Plaintexts
  for (int i = 0; i < 2; i++) { // 2 fully-connected layers
    vector<Plaintext> layer_weights;
    // Weight matrix shape: # input neurons x # output neurons
    for (int j = 0; j < IO_LEN * NUM_HIDDEN_NEURONS; j++) {
      // Get random value from distribution
      uint64_t sample = distribution(gen);
      // Generate SEAL plaintext encoding sample
      auto tmp = Plaintext(util::uint_to_hex_string(&sample, size_t(1)));
      layer_weights.push_back(tmp);
    }
    cloud_weights.push_back(layer_weights);
  }
}
```

Listing 1: **Cloud Network Parameter Setup**: Since we assume the cloud owns the model, the network weights do not need to be encrypted. However, they must be encoded as SEAL plaintexts in order to serve as operands along with encrypted data for arithmetic operations. In this code snippet, we generate two weight matrices (one for each fully-connected layer) and initialize them with plaintexts representing 1. In a realistic setting, this value can be replaced by trained weights.

Listing 1 showcases a sample network parameter setup phase on the cloud side that entails encoding the raw weights (in this case 4-bit integers) into plaintext objects compatible with the SEAL cryptographic backend. For demonstration, we use randomly generated weights instead of trained parameters. `IO_LEN` corresponds to the size of the network inputs and outputs (which are assumed to have the same shape for the purposes of demonstration as shown in the diagram of Figure 8) and `NUM_HIDDEN_NEURONS` indicates the number of inputs and outputs for the activation function in the middle of the network. We note that the weights are represented in this case as a 2D vector, with the outer dimension corresponding to the layer (i.e., the first or second fully-connected layer) and the inner dimension corresponds to the weight matrix of size `IO_LEN` $\times$ `NUM_HIDDEN_NEURONS` unrolled row-wise.

```
EncryptionParameters cloud_first_pass(vector<WalrusCtxt>& input_ct) {
  vector<WalrusCtxt> hidden_neurons(NUM_HIDDEN_NEURONS);
  vector<WalrusCtxt> out_neurons(IO_LEN);
  // First Fully-Connected Layer
  for (int i = 0; i < NUM_HIDDEN_NEURONS; i++) {
    hidden_neurons[i] = WalrusCtxt(scheme_type::bgv);
    for (int j = 0; j < IO_LEN; j++) {
      WalrusCtxt tmp = WalrusCtxt(scheme_type::bgv);
      // Multiply input ciphertexts by plaintext weight and store in tmp
      WalrusMultPlain(tmp, input_ct[j], cloud_weights[0][i*IO_LEN+j]);
      // Accumulate results
      WalrusAdd(hidden_neurons[i], hidden_neurons[i], tmp);
    }
  }
  // Activation Function (x^2)
  for (int i = 0; i < NUM_HIDDEN_NEURONS; i++) {
    WalrusMult(hidden_neurons[i], hidden_neurons[i], hidden_neurons[i]);
  }
  // Second Fully-Connected Layer
  for (int i = 0; i < IO_LEN; i++) {
    out_neurons[i] = WalrusCtxt(scheme_type::bgv);
    for (int j = 0; j < NUM_HIDDEN_NEURONS; j++) {
      WalrusCtxt tmp = WalrusCtxt(scheme_type::bgv);
      // Multiply previous layer outputs by plaintext weight and store in
          tmp
      WalrusMultPlain(tmp, hidden_neurons[j],
          cloud_weights[1][i*NUM_HIDDEN_NEURONS+j]);
      // Accumulate results
      WalrusAdd(out_neurons[i], out_neurons[i], tmp);
    }
  }
  return WalrusParameterize(out_neurons);
}
```

Listing 2: **Initial Pass for Parameterization**: This pass doesn't utilize any key material and the input ciphertexts are not loaded with actual values. Instead, Walrus uses this pass to build a model of the application and compute statistics about noise growth. After all of the operations have been invoked, the `WalrusParameterize` function will select the most appropriate parameter set for the application that SEAL provides.

After the cloud initializes the network parameters, either the client or the cloud can execute the initial pass of the circuit that will serve to parameterize the actual encrypted inference (shown in Listing 2). As discussed, this first pass is executed on the cloud side (with dummy input data); notably, Walrus will not actually evaluate the HE operations at this stage, and this step only serves to aggregate noise statistics during each invocation of an HE operation like `WalrusMult` or `WalrusAdd`. Walrus effectively models the growth of noise in the circuit and uses this pass to determine the maximum multiplicative depth of the circuit.

Once the circuit simulation has concluded, the output neurons of the network are passed to the `WalrusParameterize` function that will choose an appropriate parameter set based on the gathered

noise statistics. The algorithm first considers the maximum multiplicative depth of all provided outputs to determine the minimum ciphertext modulus needed. Recall that each multiplication is followed by a mod switching operation that deletes one of the primes in the ciphertext modulus. As a consequence, if the largest multiplicative depth is 7, the ciphertext modulus must have at least 8 primes (one prime needs to remain in the modulus after the 7 mod switches). Next, the addition depth is considered; the authors have experimentally verified the number of additions that can be done with the provided SEAL parameter sets through manual experiments. If the smallest parameter set that satisfies the multiplicative depth can't support the number of additions in the input algorithm, Walrus increases the modulus. After the target modulus is selected, the appropriate polynomial degree is selected to yield at least 128 bits of security according to the recommendations put forward by the HE standard [5]. At this point, a SEAL class object encompassing the derived parameters is created. This final parameter set will be utilized on the client-side for key generation and encryption of the initial inputs.

```
vector<WalrusCtxt> cloud_second_pass(Encryptor& encryptor, Evaluator&
    evaluator,
     RelinKeys& relin_keys, vector<WalrusCtxt>& input_ct) {
  // Create variables for intermediate and output values
  vector<WalrusCtxt> neurons(NUM_HIDDEN_NEURONS);
  vector<WalrusCtxt> out_neurons(IO_LEN);
  // First Fully-Connected Layer
  for (int i = 0; i < NUM_HIDDEN_NEURONS; i++) {
    // Instantiate each output neuron and load with encryptions of 0
    neurons[i] = WalrusCtxt(scheme_type::bgv);
    encryptor.encrypt_zero(neurons[i].ctxt);
    // Multiply input neurons with weights and accumulate
    for (int j = 0; j < IO_LEN; j++) {
      WalrusCtxt tmp = WalrusCtxt(scheme_type::bgv);
      tmp.WalrusEncryptSingle(encryptor, 0);
      WalrusMultPlain(tmp, input_ct[j], cloud_weights[0][i*IO_LEN+j],
          evaluator);
      WalrusAdd(neurons[i], neurons[i], tmp, evaluator);
    }
  }
  // Activation Function (x^2)
  for (int i = 0; i < NUM_HIDDEN_NEURONS; i++) {
    WalrusMult(neurons[i], neurons[i], neurons[i], evaluator, relin_keys);
  }
  // Second Fully-Connected Layer
  for (int i = 0; i < IO_LEN; i++) {
    out_neurons[i] = WalrusCtxt(scheme_type::bgv);
    out_neurons[i].WalrusEncryptSingle(encryptor, 0);
    // Mod switch twice to ensure out_neurons ctxts have the same modulus
        for accumulation
    evaluator.mod_switch_to_next_inplace(out_neurons[i].ctxt);
    evaluator.mod_switch_to_next_inplace(out_neurons[i].ctxt);
    for (int j = 0; j < NUM_HIDDEN_NEURONS; j++) {
      WalrusCtxt tmp = WalrusCtxt(scheme_type::bgv);
      tmp.WalrusEncryptSingle(encryptor, 0);
      WalrusMultPlain(tmp, neurons[j],
```

```
          cloud_weights[1][i*NUM_HIDDEN_NEURONS+j], evaluator);
      evaluator.mod_switch_to_next_inplace(tmp.ctxt);
      WalrusAdd(out_neurons[i], out_neurons[i], tmp, evaluator);
    }
  }
  return out_neurons;
}
```

Listing 3: **Encrypted Evaluation**: The second pass consists of the actual HE evaluation with secure input ciphertexts provided by the user. Additionally, public key material is passed to the function to allow for HE operations with SEAL and mod switching invocations are added to ensure that ciphertexts have identical moduli before arithmetic occurs between them.

After the client has generated encryption keys and sent the public key material and input ciphertexts to the cloud (shown in Listing 4), the cloud evaluates the circuit homomorphically. We can see that the encrypted evaluation in Listing 3 is very similar to Listing 2 with a few differences. First, the `encrypt_zero` function calls are needed to initialize the output neurons of each layer to zero. Additionally, extra input parameters are needed for the Walrus arithmetic functions to facilitate HE evaluation, such as the `relin_keys` for multiplications. Lastly, the `mod_switch_to_next_inplace` function of the SEAL `evaluator` engine is needed to ensure that ciphertexts are compatible with each other for a given operation. After a multiplication operation, a modulus switching operation is invoked to reduce the noise, which also decreases the size of the ciphertext modulus associated with that ciphertext. In order to perform computations with two ciphertexts, both inputs should have the same ciphertext modulus; the `mod_switch_to_next_inplace` function facilitates this and removes the next prime from the polynomials of the ciphertext with a larger modulus. Listing 3 returns the ciphertext vector that contains the raw classification scores for each input instead of the parameter set returned in Listing 2.

```
void client() {
  // Create ciphertext objects for parameterization
  vector<WalrusCtxt> inputs(IO_LEN);
  for (size_t i = 0; i < IO_LEN; i++) {
    inputs[i] = WalrusCtxt(scheme_type::bgv);
  }
  // Get optimal parameters from Walrus (running on the cloud)
  EncryptionParameters parms = cloud_first_pass(inputs);
  SEALContext context(parms);
  // Generate secret key for decryption and public key for encryption
  KeyGenerator keygen(context);
  SecretKey secret_key = keygen.secret_key();
  PublicKey public_key;
  keygen.create_public_key(public_key);
  // Generate keyswitching keys needed for multiplications
  RelinKeys relin_keys;
  keygen.create_relin_keys(relin_keys);
  // Create engines for HE operations
  Encryptor encryptor(context, public_key);
  Evaluator evaluator(context);
  Decryptor decryptor(context, secret_key);
```

```
22   BatchEncoder batch_encoder(context);
23   // Create generator for 8-bit numbers (based on Mersenne twister)
24   random_device rd;
25   mt19937 gen(rd());
26   uniform_int_distribution<uint64_t> distribution(1, 255);
27   // Encrypt batched inputs
28   size_t slot_count = batch_encoder.slot_count();
29   for (size_t i = 0; i < IO_LEN; i++) {
30     vector<uint64_t> in_vec(slot_count);
31     for (size_t j = 0; j < slot_count; j++) {
32       in_vec[j] = distribution(gen);
33     }
34     inputs[i].WalrusEncryptVector(batch_encoder, encryptor, in_vec);
35   }
36   // Run encrypted inference on the cloud
37   vector<WalrusCtxt> enc_res = cloud_second_pass(encryptor, evaluator,
        relin_keys, inputs);
38   // Decrypt and postprocess the classification results
39   vector<uint64_t> top_scores(slot_count);
40   vector<int> top_class(slot_count);
41   // Find top score for each input
42   for (size_t i = 0; i < enc_res.size(); i++) {
43     auto res_vec = enc_res[i].WalrusDecryptVector(batch_encoder,
         decryptor);
44     for (int j = 0; j < res_vec.size(); j++) {
45       if (res_vec[j] > top_scores[j]) {
46         top_scores[j] = res_vec[j];
47         // Chosen class is the index of the max element
48         top_class[j] = i;
49       }
50     }
51   }
52   for (size_t i = 0; i < slot_count; i++) {
53     cout << "Classification #" << i << ": " << top_class[i] << endl;
54   }
55   return;
56 }
```

Listing 4: **Client-side Operations**: The user is responsible for generating encryption keys and creating various objects needed to facilitate the HE operations supported by the SEAL backend. Additionally, encryption of secure inputs and decryption of the final results are handled by the user. In this specific case, the decrypted classification scores are post-processed to identify the highest scoring class for each input.

The script shown in Listing 4 demonstrates all operations executed by the client for the inference application. First, empty ciphertext objects are created for the first pass executed by the cloud to determine the correct encryption parameters (i.e., lines 3-8). These `WalrusCtxt` input objects hold no actual input data, but will be instantiated with noise metadata automatically that will serve as input to the first pass, which will manipulate this metadata as the circuit simulation proceeds. After the proper encryption parameters are derived from the first pass, the client creates the necessary encryption keys, all of which are public aside from the secret key used for decryption. Additionally, all necessary engines for HE evaluation are created (lines 19-22). Engines constitute classes in the underlying SEAL library that expose

functionality for different types of HE operations. Specifically, the `Encryptor` engine encapsulates the public key and is used solely for encrypting data into ciphertexts. Similarly, the `Decryptor` engine encapsulates the secret key and is only utilized by the client to decrypt ciphertexts back into plaintext elements. The `BatchEncoder` engine is needed to turn vectors of plaintext elements into a plaintext object for encryption. Lastly, the `Evaluator` engine contains the functionality for performing HE arithmetic operations. From the client perspective, they must generate the engines, which are passed to the high-level Walrus functions.

Next, the client encrypts the desired inputs, and batches N inputs together into the same ciphertext (where N is the number of ciphertext slots that can be exploited with the determined parameter set). In this way, N inferences can be evaluated simultaneously with no additional cost. For the purposes of our case study, each input is a vector of 8-bit values. At this point the client sends the input ciphertexts along with the public key material to the cloud for evaluation (line 37). We note that this is emulated in the tutorial, but one can utilize a TCP library or other network protocol to transfer data between the client and remote server.

The client can decrypt the final classification results after the cloud finishes the computation. We remark that the homomorphic evaluation in Listing 3 does not compute an encryption of the selected class that the input was assigned; instead it computes an integer score for each class, where the magnitude of the score is correlated to the probability that the input belongs to that particular close. This is intentional design choice as computing the maximum of a set of encrypted values is a costly procedure; therefore, it is up to the client to postprocess the raw classification scores corresponding to each class and select the highest score.

Figure 9 showcases the inference latencies and throughput for a variety of network sizes. The reported times measure Listing 3, which corresponds to the actual encrypted operations on the cloud side. The throughput is computed as the latency divided by the number of inferences that are batched together. We can see that the performance gets exponentially slower when doubling the size of each layer, which is due to both the greatly increased number of arithmetic operations and the increase in noise growth, which requires Walrus to choose larger parameters. For instance, the smallest network configuration can be evaluated with a polynomial degree of 2^{13}, while the $100 - 50 - 100$ and $200 - 100 - 200$ networks require the polynomial degrees to be doubled. Doubling the polynomial degree results in significantly slower polynomial arithmetic operations due to both the increased coefficient size and the number of coefficients.

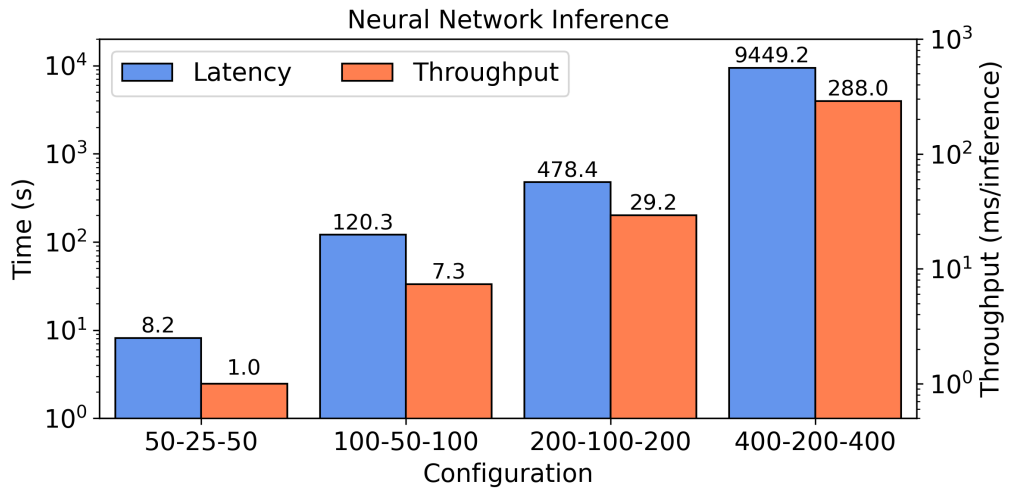

Fig. 9: **Neural Network Performance:** Each network is identified by the number of neurons in the layer (i.e., $50-25-50$ corresponds to a fully connected layer with 50 inputs and 25 outputs followed by an activation function over the 25 output neurons of the first layer, etc.). Walrus selected a polynomial degree of 8192 for the $50-25-50$ network, a degree of 16384 for both the $100-50-100$ and $200-100-200$ networks, and a poly degree of 32768 for the largest $400-200-400$ network.

D. Debugging Walrus Applications

From an HE perspective, Walrus will not allow users to run an application where no parameter set supported by SEAL can satisfy the constraints. For instance, a very deep program with a large multiplicative depth will trigger an exception in the `WalrusParameterize` function at the end of the first pass.

A common issue, as seen in the neural network inference, is ensuring that all input ciphertexts to an arithmetic operation have the same ciphertext modulus. Walrus will automatically decrease the modulus of the product ciphertext of a multiplication, but will not automatically match the moduli of the input ciphertexts of any given computation. If there is a mismatch, an exception will be thrown by the SEAL library. There are generally two techniques for resolving this issue: manually inserting SEAL modulus switching invocations akin to the neural network example or comparing the moduli of both input ciphertexts directly using SEAL. The former approach is suitable for simple applications, while the latter is somewhat cumbersome but necessary for complex programs. The latter approach involves extracting the identifier for the current parameters of a ciphertext by first extracting a unique identifier called `parms_id` from the ciphertext object. This identifier can be used to fetch a pointer to the parameters corresponding to the identifier by using the SEAL Context object (i.e. `context.get_context_data(parms_id)`. The actual parameters can be extracted with `auto &parms = parms_ptr->parms();` and then

the number of primes in the modulus can be found with `parms.coeff_modulus().size()`. Using the number of primes of both input ciphertexts, one can deduce which ciphertext has the highest level and also how many mod switching operations are required to match the parameters. For instance, if ct_a has a modulus length of 5 and ct_b has a modulus length of 3, ct_a must be mod switched twice in order to arrive at the same modulus as ct_b. Note that it is impossible to add primes to the modulus of a ciphertext.

IV. Converting Applications for FHE Evaluation

The HELM library [20] is a compiler written in Rust that can take existing HDL designs written in synthesizable Verilog and automatically convert and execute the resulting netlist with Boolean FHE. Compared to the Walrus framework, HELM is better suited for arbitrary computation and can outperform Walrus for applications with many non-linear, bitwise, or comparison operations. Additionally, HELM is tuned specifically for evaluation on high performance cloud servers with many CPU cores or a powerful GPU.

A. RTL Synthesis and Preprocessing

In order to generate a netlist in the form that HELM expects for encrypted evaluation, we utilize the open-source Yosys synthesis suite [21]. Aside from generating a gate circuit from a Verilog input, Yosys performs an important role in optimizing the underlying circuit, which will be evaluated virtually with FHE. Many optimizations proposed for electronic design automation and hardware development can also benefit Boolean FHE applications, such as removing redundant gates. Additionally, Yosys can utilize the ABC verification and synthesis tool [22] to rigorously optimize the combinational logic in the circuit, which ultimately results in fewer FHE operations. Prior work has explored encoding the computational costs of different Boolean FHE gates as a function of area in a custom technology library [23]. With this technique, Yosys can try to prioritize less expensive logic gate operations in the encrypted domain as it tries to reduce the overall area of the circuit. When opting for such a custom tech library, one should only define the logic gates natively supported by the cryptographic backend, so that all available gates have an FHE counterpart. HELM supports two cryptographic backends in the form of TFHE-rs [24] and a modified version of `concrete-core` [25] capable of executing standard two input logic gates, inverters, 2:1 multiplexers, as well as many-to-one lookup tables.

After Yosys has optimized the circuit, HELM requires it to be formatted in Verilog. The output netlist then needs to be passed to the HELM preprocessor, which will remove auxiliary information from the netlist, such as the clock signals and connections to ground and VCC, as this is not needed for FHE evaluation. We note that HELM does not support clock gating for this reason, but it is capable of emulating

sequential circuit elements. This unique ability of HELM and practical considerations are covered later in this section.

Next, HELM will find duplicate wires in the netlist and collapse them into a single wire; Yosys often refers to a single wire by multiple identifiers, which can cause incorrect connections and redundant ciphertexts if these wires are treated distinctly by HELM. As such, HELM detects these duplicate wire assignments and changes all identifiers to a single value. For wire assignments that directly propagate an input wire to an output, HELM adds a low-cost buffer gate that initiates a ciphertext copy operation. Lastly, HELM adds an encryption gate for constant wires where the value is public; this step is necessary in order to allow the constant wire values to be mixed with other secure ciphertexts.

```
module chi_squared (N0, N1, N2, alpha, beta_1, beta_2, beta_3);

// Define input ports
input [15:0] N0;
input [15:0] N1;
input [15:0] N2;
// Define output parts
output [15:0] alpha;
output [15:0] beta_1;
output [15:0] beta_2;
output [15:0] beta_3;
// Declare tmp variables (single assignment)
wire [15:0] tmp_0, tmp_1, tmp_2, tmp_3, tmp_4;
wire [15:0] tmp_5, tmp_6, tmp_7, tmp_8, tmp_9;

// Compute alpha = (4*N0*N2 - N1^2)^2
assign tmp_0 = 4 * N0;
assign tmp_2 = tmp_0 * N2;
assign tmp_1 = N1 * N1;
assign tmp_9 = tmp_2 - tmp_1;
assign alpha = tmp_9 * tmp_9;
// Compute beta_1 = 2 * (2*N0 + N1)^2
assign tmp_3 = 2 * N0;
assign tmp_4 = tmp_3 + N1;
assign tmp_5 = tmp_4 * tmp_4;
assign beta_1 = tmp_5 * 2;
// Compute beta_2 = (2*N0 + N1) * (2*N2 + N1)
assign tmp_6 = 2 * N2;
assign tmp_7 = tmp_6 + N1;
assign beta_2 = tmp_4 * tmp_7;
// Compute beta_3 = 2 * (2*N2 + N1)^2
assign tmp_8 = tmp_7 * tmp_7;
assign beta_3 = tmp_8 * 2;

endmodule
```

Listing 5: **HELM Chi-Squared Verilog Design**: This behavioral verilog code takes three 16 bit inputs and computes four 16 bit output variables through a series of multi-bit arithmetic operations. This code can be directly ran with HELM's arithmetic mode, or converted to a netlist with Yosys and executed with HELM's LUT or gate modes depending on the specific technology mapping command used.

B. Preparing Inputs and Decrypting Outputs

HELM is ready to evaluate the circuit after the preprocessor generates a modified, but functionally equivalent, version of the netlist made by Yosys. However, without providing values for the input wires, HELM will evaluate the circuit with the input wires loaded with encryptions of zero. There are two methods to supply HELM with actual input wires: using a CSV file or passing the values via command line in a series of key/value pairs. The latter approach is well suited for debugging and for circuits with few input ports, as it becomes tedious otherwise.

```
Wire, Value, Width
N0, 18, 16
N1, 19, 16
N2, 20, 16
```

Listing 6: **CSV Input File for Chi-Squared**: This listing outlines the structure of the corresponding CSV input files for HELM running the Chi-squared example circuit. Notably, HELM can save the final decrypted outputs to a CSV with an identical format. For the Chi-squared example, it consists of four output ports (`alpha`, `beta_1`, `beta_2`, `beta_3`).

As an example, Listing 5 shows a Verilog design provided in HELM that can be used to evaluate the Chi squared statistical test on encrypted data. This design takes in three 2-byte inputs (named N0, N1, and N2), performs a series of arithmetic operations, and generates four 2-byte outputs. Via command-line, this example can be run using `cargo run --bin helm --release -- --verilog chi-squared.v -w N0 12 16 -w N1 13 16 -w N2 14 16`, which will initialize N0 with `0x12`, N1 with `0x13`, and N2 with `0x14`. The `16` parameter that follows the data value for each wire specifies the width of the port. Conversely, one can also create a CSV file with the following structure: the first line of the file reads `wire, value, width` and then every subsequent line specifies the port identifier, followed by the data to be encrypted into that port, and then the length of the data in bits. The CSV version of the previous command is shown in Listing 6.

Decryption is automatically handled by HELM after circuit evaluation and the output is printed directly to the terminal by default. Optionally, one can use the `--output-wires-file` flag to specify the name of a CSV file that will be generated after decryption. The structure of this CSV file is identical to the input CSV files.

C. Choosing the Right Computational Domain with HELM

Unlike Walrus, HELM supports different sets of cryptographic primitives for program evaluation that can have a significant impact on the performance of applications. An overview of the different modes are depicted in Table II.

TABLE II: **HELM Modes**: Each mode of HELM is distinguished by which primitive operations are supported.

Mode	Input Verilog Format	Supported Operations	Word Sizes
Gate	Structural	Standard Logic Gates	Any
LUT	Structural	2:1 LUTs	Any
Arithmetic	Behavioral	Addition & Multiplication	Up to 128 bits

1) FHE Parameterization: Parameterizing FHE involves different considerations than LHE as all FHE parameters by definition can be used to evaluate arbitrarily deep circuits. Instead, one must consider the desired precision of the ciphertexts with CGGI. In Boolean FHE, only one bit of data needs to be encoded, so only one parameter set at 128 bits of security is needed and constitutes the optimal parameter choice. For arithmetic mode, which requires a larger plaintext precision, the polynomial degree must necessarily be larger. In HELM, a polynomial degree of 2^{11} is utilized, while the Boolean ciphertexts employ a polynomial degree of 2^{10}. Conveniently, each mode of HELM has one hardcoded parameter set that works for any program and does not need to be tuned by users.

2) Arithmetic Mode: The Chi squared example in Listing 5 is composed of strictly integer arithmetic operations; this is a less-desirable use-case for Boolean FHE as each multiplication, addition, and subtraction will become a large, computationally expensive subcircuit. For programs of this nature, HELM supports an arithmetic mode, which encodes multiple bits of data into a single ciphertext. In this form, the primitive operations include addition and multiplication, operating similarly to an FHE equivalent of Walrus. In this mode, no logic synthesis is required and HELM can directly parse the behavioral Verilog as long as there is only one arithmetic operator per line and there are no bitwise operations.

3) Boolean Modes: For Boolean ciphertexts, HELM supports both a logic gate mode and a mode that employs exclusively 2:1 LUTs. The difference between the two may seem subtle, as the LUT mode is essentially a generalization of the gates mode, allowing for non-standard cells. However, the bootstrapping technique used for each mode is different, and different cryptographic parameters are required for each. Prior work [20] has shown that the logic gates outperform the LUTs for all ISCAS-85 [26] and ISCAS-89 circuits [27]. However, the LUTs appear superior for larger circuits with hundreds of thousands of gates. From a usability perspective, the only change required for using LUTs versus logic gates mode is a small tweak to the Yosys synthesis steps; Yosys can replace standard gate cells with 2:1 LUTs using the technology mapping capability of ABC with `abc -luts 2`. Nevertheless, this step can sometimes result in a netlist that also employs larger LUT widths; in this case, the `nlutmap` command can be used

to directly specify how many LUTs of different widths can be used. For example, `nlutmap -luts 0,10000000,0` will instruct Yosys that there is a large abundance of 2:1 LUTs, but no availability for other widths. As a result, this command will ensure that Yosys will generate a circuit with solely 2:1 LUTs.

D. Considerations for Sequential Circuits

Sequential circuits can be challenging to evaluate in the encrypted domain, as many rely on a `READY` signal to indicate when the result of a computation is ready. However, because all of the wires in the circuit are encrypted, it's impossible for the cloud to determine when this signal is asserted and halt the computation. This observation is known as the *termination problem* of HE, which stipulates that it is impossible to make runtime decisions based on encrypted data [28]. An example design that showcases this issue is depicted in Listing 7. Here, the target application consists of a state machine with three possible states: an idle state that loads the input data, a division state that divides the data by 2 until a threshold is reached, and a final state asserts the `READY` signal and loads the data into the output.

```
module Divider (clk, rst, input_data, threshold, res, ready);

// Declare input ports
input clk;
input rst;
input [7:0] input_data, threshold;
// Declare output parts
output reg [7:0] res,
output ready;
// Declare temp variables
reg [7:0] data;
reg [2:0] state;

// Trigger logic on rising edge of clk or rst signals
always @(posedge clk or posedge rst) begin
  if (rst) begin
    // Return to initial state
    state <= 3'b000;
    ready <= 0;
    res <= 8'b0;
    data <= 8'b0;
  end else begin
    case (state)
      3'b000: begin
        // Idle
        done <= 0;
        if (input_data > threshold) begin
          // Load input data
          data <= input_data;
          state <= 3'b001;
        end
      end
```

```
33            3'b001: begin
34              // Divide by 2
35              if (data > threshold) begin
36                data <= data >> 1;
37              end else begin
38                state <= 3'b010;
39              end
40            end
41            3'b010: begin
42              // Finish
43              res <= data;
44              ready <= 1;
45              state <= 3'b000;
46            end
47            default: state <= 3'b000;
48          endcase
49        end
50      end
51
52      endmodule
```

Listing 7: Simple state machine that divides an input by 2 until it reaches a user-defined threshold. This is an example of a program with a ready signal that can't be checked by the computing party (e.g., cloud server).

One possible solution to evaluate this circuit over FHE data is for the cloud to send to the client the encrypted `READY` signal after each clock cycle, have the client decrypt it, and then receive a response telling the cloud to continue or halt. Indeed, this approach has been adopted by the VSP homomorphic processor [29]. However, this technique can be abused and lead to sensitive data leakage, including both the input values and the secret encryption key. Instead of sending the encryption of the `READY` signal, the cloud can simply swap it for one of the input ciphertexts and wait for the client's response. If the input ciphertext is an encryption of `1`, the client will tell the cloud to halt execution, effectively acting as a decryption oracle for the cloud server. Additionally, since the evaluation key required for bootstrapping contains encryptions of the user's secret key, the cloud could send these key encryptions to the client as well, effectively leaking bits of the secret key. With knowledge of the secret key, any data encrypted under it can be easily decrypted by the cloud.

Unfortunately, the only way to securely deal with this termination challenge is to have the client inform the server in advance of the maximum number of clock cycles required to derive the expected result. With this information, the cloud can evaluate the sequential circuit using the worst-case number of cycles C. With respect to the example in Listing 7, the largest input value is a single byte. Assuming the worst-case setting of `0x00` as the threshold input, then the number of clock cycles required to evaluate the divider circuit for any input is 8. If the client is willing to potentially leak information about the range of the threshold or the input values, she can also declare a smaller maximum number of clock cycles to evaluate.

For instance, if the threshold is 2^7, then the maximum number of clock cycles for the division state is 1. For some circuits, such as a sequential implementation of AES-128, the number of cycles is always fixed and therefore this approach does not result in any negative performance impacts. To simulate clock cycles, HELM unrolls the circuit, effectively duplicating the combinational logic C times. Flip-flops are interpreted by HELM as delayed buffers that propagate their input values to the outputs at the start of the next clock cycle.

E. Choosing the Correct Hardware Platform with HELM

HELM automatically parallelizes circuit evaluation to make the best use of powerful cloud servers. Specifically, for nodes (i.e. gates or LUTs) at the same circuit depth, HELM dispatches each one to different processing units. For wide circuits, this parallelization technique can drastically decrease latency. As a case in point, HELM was demonstrated to be up to $65\times$ faster than the Romeo framework [30] for large benchmarks. Much of this speedup can be attributed to the fact that the approach employed by Romeo is only suited for sequential processing by a single CPU thread.

HELM can execute HE operations on either multi-core CPUs or a CUDA-enabled GPU. Notably, the GPU backend only supports logic gate circuits. However, for wide circuits with hundreds of thousands of gate evaluations in total, HELM running on an NVIDIA A100 GPU outperforms a 48-threaded CPU implementation (running on a `c5.12xlarge` AWS instance) by approximately $3\times$ [20]. Circuits with consistently wide levels can exploit the ample parallelism supplied by a powerful datacenter GPU like the A100 and achieve considerable speedup versus multi-core CPUs. Conversely, if the target circuit exhibits thin levels with only a few gates that can be evaluated in parallel and a long critical path, a GPU can be detrimental as the overheads associated with data movement can result in a net slowdown relative to a multi-core CPU.

F. Case Study: AES Decryption

At first glance, it seems non-intuitive why AES is a good application for HE; it's a traditional block cipher that doesn't exhibit homomorphic properties and applying another layer of encryption doesn't necessarily result in stronger security guarantees. However, one notable feature of AES and other block ciphers is that there is no significant data expansion when generating ciphertexts from plaintext data. Conversely, HE can result in size expansions of a few orders of magnitude when encrypting data, as each ciphertext becomes a set of high-degree polynomials with large coefficients.

Luckily, there is a technique called *transciphering* that allows HE to take advantage of this lack of data expansion in traditional cryptosystems like AES. If there were a way to transition from AES ciphertexts

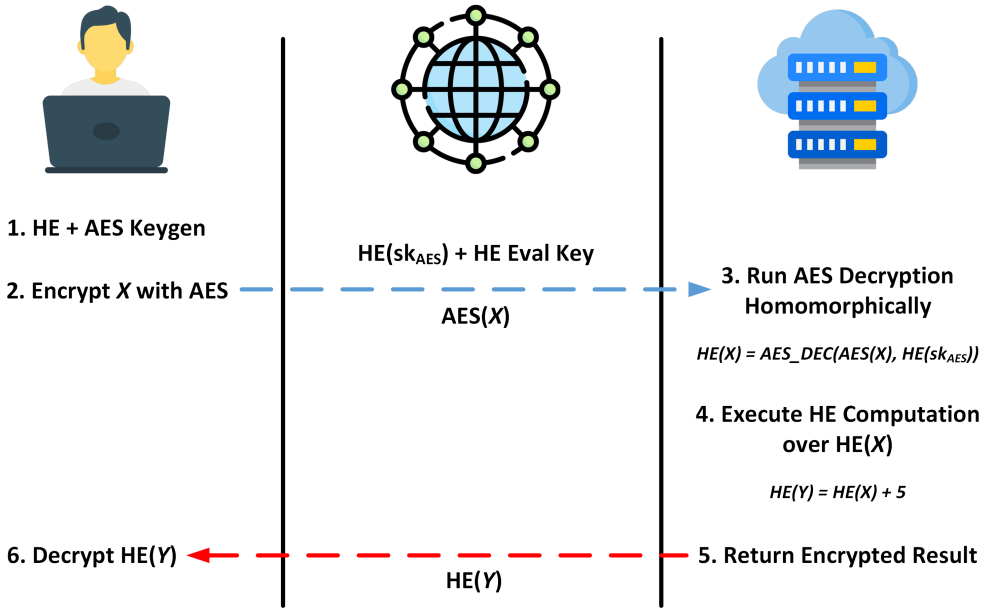

Fig. 10: **Transciphering with AES:** Traditional cryptographic algorithms can be used in tandem with HE to drastically reduce the communication overhead when sending sensitive data from the client to the cloud server.

(used for data transfers) to HE ciphertexts (used for computation), one could drastically reduce the ciphertext size when transmitting data from the client to the computing server. Since FHE allows for arbitrary computation, it stands that it is possible to homomorphically evaluate the *decryption circuit of AES*.

In the transciphering process (depicted in Figure 10), a user can homomorphically encrypt the symmetric key of AES (which must remain secret to protect the confidentiality of the input data) and send to the cloud a standard AES encryption of the secret data to be computed upon. Note that the AES ciphertext data is orders of magnitude smaller in size than its equivalent ciphertext form in HE. Next, the cloud can compute the AES decryption circuit using HE operations with an HE encryption of the symmetric AES key and the AES ciphertext data supplied by the user. Surprisingly, the output ciphertext will be a valid HE encryption of the original data and can be used in meaningful computation (i.e., the AES protection layer is stripped away and replaced by an HE encryption layer, protected by a secret HE key known only to the user). We remark that only the forward operation is possible (i.e., replacing AES with HE); the cloud cannot send the final result ciphertexts back to the user in AES form. This stems from the fact that doing so would require an HE decryption, which is impossible for the cloud to perform without the HE secret key. For applications with a large amount of input data, this technique can result in significantly

lower network traffic, at the expense of additional computation required to evaluate the AES decryption circuit homomorphically.

```
    def hex_to_bits(hex_string):
        # Interpret hex string in base 2 and remove the leading '0x'
           characters
        binary_string = bin(int(hex_string, 16))[2:]

        # Pad binary to nearest multiple of 4 (each hex character is 4
           bits)
        padding_length = 4 * len(hex_string) - len(binary_string)
        binary_string = '0' * padding_length + binary_string
        return [bit == '1' for bit in binary_string]

    def gen_csv():
        args = parse_args()
        with open(args.csv[0], 'w', newline='') as csvfile:
            writer = csv.writer(csvfile)

            # Add preamble (same for each file)
            first_line = ["wire", " value"]
            writer.writerow(first_line)

            # Iterate through each provided port
            for i in range(len(args.id)):
                bit_values = hex_to_bits(args.val[i][0])
                # Write one line for each bit of the port
                for j in range(len(bit_values)):
                    value = bit_values[-(j+1)]
                    str_value = ""
                    if value:
                      str_value = " true"
                    else:
                      str_value = " false"
                    writer.writerow([f"{args.id[i][0]}[{j}]", str_value])
```

Listing 8: **Generate Input CSV Files**: Input ports with their corresponding input values can be generated with this code snippet. In some cases, automatically generating the CSV input files that are used to load the input wires is less cumbersome than specifying them via command line with HELM.

For the purposes of demonstrating the usability of HELM, we employ an existing open-source AES-128 decryption codebase in Verilog [31]. Listing 8 shows the core of a Python script used to easily generate the CSV file that HELM reads for inputs to the chosen AES circuit. The Python code can be utilized along with a standard AES module to generate keys (and expanded keys if necessary) along with the AES encryptions of the input data on the client side.

Furthermore, we can generate an additional variant that omits the key scheduling procedure from the AES algorithm. Omitting the key scheduling step results in less overall latency, but increases the communication overhead as the client needs to derive each round key locally, encrypt it homomorphically, and send it to the cloud. As ten round keys are required for the AES-128 evaluation, a total of 1280

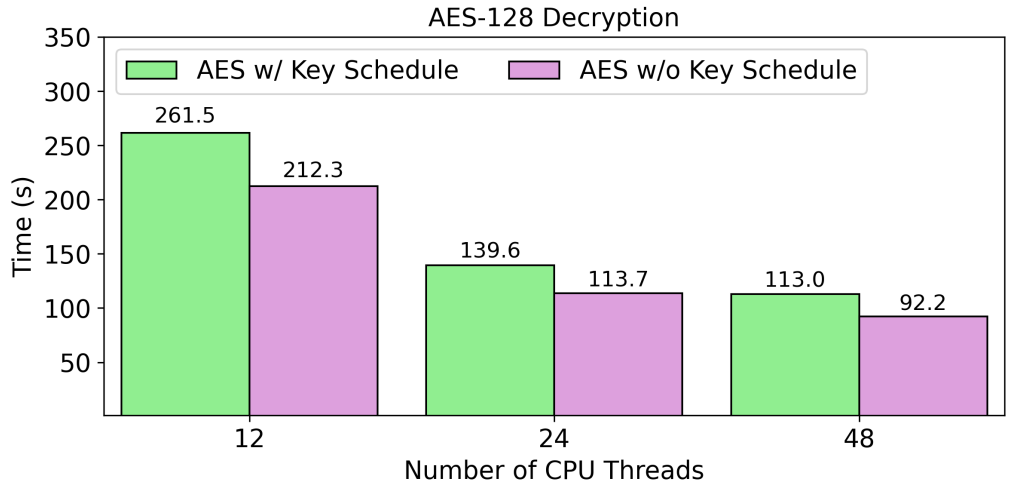

Fig. 11: **AES-128 Decryption Performance:** The AES with key scheduling HE circuit consumes an 128-bit HE-encrypted AES key as well as an AES ciphertext block, while the AES variant without key scheduling consumes an AES ciphertext block and a set of ten 128-bit HE-encrypted AES round keys.

encrypted bits must be transmitted, which adds to the overall network traffic from the client to the server.

Figure 11 shows the results of both AES-128 decryption circuit variants running on a `c5.12xlarge` AWS server. To demonstrate scalability as the CPU threads increase, we restrict the evaluation to 12, 24, and 48 threads. Both variants exhibit near ideal speedup for doubling the number of threads from 12 to 24, but only exhibit modest latency reductions when moving to 48 threads. This is primarily due to the width of the AES circuit, as many circuit levels do not exhibit enough width to exploit all of the CPU threads effectively.

```
    import csv

    # Read the input data from the CSV file
    input_data = []
    with open('aes_verif.output.csv', 'r') as csvfile:
        reader = csv.reader(csvfile)
        next(reader)  # Skip the header
        for row in reader:
            input_data.append(row)

    # Convert 'true' to 1 and 'false' to 0
    def convert_to_bit(value):
        return 1 if value.strip() == '1' else 0

    # Create a dictionary to store bit values indexed by their positions
    bit_dict = {}
    for row in input_data:
        index = int(row[0].split('[')[1].split(']')[0])
        bit = convert_to_bit(row[1])
        bit_dict[index] = bit
```

```
21
22      # Convert the bit values to an integer
23      integer_value = 0
24      for index, bit in sorted(bit_dict.items()):
25          integer_value |= (bit  index)
26
27      # Print the sorted bit_dict
28      print('Sorted bit_dict:', sorted(bit_dict.items()))
29
30      # Print the integer value
31      print('Integer Value:', integer_value)
32
33      # Print the integer value in hexadecimal format
34      hex_value = format(integer_value, 'x')
35      print('Hex Value:', hex_value)
```

Listing 9: This listing converts HELM decryptions to ordered bit strings, hex strings, and integers to make it easier to interpret the results.

The last step is decrypting the final result; as mentioned earlier, HELM can output a CSV detailing the recovered 128 bit output of the AES circuit. However, interpreting the CSV can be tedious as HELM will only include one decrypted bit per line in the output file. The Python script in Listing 9 parses the CSV file line by line, orders the output bits, and constructs a hex string for easy interpretation. When applied to this particular use-case, the code will output a 128 bit result corresponding to the original plaintext prior to AES encryption.

G. Debugging HELM Applications

Unlike Walrus, HELM has no programming API and will simply evaluate a provided netlist that has been generated by the Yosys synthesis tool. If HELM detects an unsupported primitive, such as a logic gate with a fan-in greater than 2, it will throw an exception. Notably, HELM does not support circuits that mix lookup tables and gates as these two sets of primitives require different parameter sets to evaluate. Yosys will always generate a compliant circuit for gates with the following `abc` command: `abc -g simple,-MUX`. Similarly `abc -luts 2` will typically yield a circuit with 2:1 LUTs, though in some cases an additional `nlutmap` command must be employed to force Yosys to squash any larger LUTs in the circuit. If the final answer does not yield the expected result, the netlist can be easily verified with Yosys as well. After Yosys has processed the input behavioral Verilog and completed the logic synthesis process, it is capable of simulating the circuit with provided input pairs using the `eval` command. This command has additional arguments to set each input wire and can additionally generate a truth table with the `table` flag. For sequential circuits, the number of cycles for the simulation can be specified; this feature can also be used to determine the number of cycles to provide to HELM during encrypted evaluation.

V. Unified Case Study: Chi Squared

The Chi squared test, as mentioned briefly in the previous section, is an important statistical measure that has seen utility in previous works focusing on privacy-preserving computation [32], [33], [34]. This benchmark, composed of strictly multi-bit arithmetic operations, can be evaluated with both approaches outlined in the previous two sections. In more detail, the target program takes three 16-bit inputs (N_0, N_1, and N_2) and computes a total of four output variables (α, β_1, β_2, and β_3) through the use of both mixed operations between fixed plaintext constants and encrypted data, as well as addition, multiplication, and subtraction between ciphertexts.

```
vector<WalrusCtxt> cloud_second_pass(Encryptor& encryptor, Evaluator&
    evaluator, RelinKeys& relin_keys, vector<WalrusCtxt>& input_ct) {
  // Chi Squared has four outputs: alpha, beta_1, beta_2, beta_3
  vector<WalrusCtxt> output_variables(4);
  for (int i = 0; i < 4; i++) {
    output_variables[i] = WalrusCtxt(scheme_type::bgv);
  }

  // Create temp variables to hold specific intermediate values
  WalrusCtxt tmp = WalrusCtxt(scheme_type::bgv);
  WalrusCtxt tmp_2 = WalrusCtxt(scheme_type::bgv);

  // Create constants
  uint64_t four = 4;
  uint64_t two = 2;
  Plaintext pt_four = Plaintext(util::uint_to_hex_string(&four,
      size_t(1)));
  Plaintext pt_two = Plaintext(util::uint_to_hex_string(&two, size_t(1)));

  // Compute alpha = (4*N0*N2 - N1^2)^2
  WalrusMultPlain(output_variables[0], input_ct[0], pt_four, evaluator);
  WalrusMult(output_variables[0], output_variables[0], input_ct[2],
      evaluator, relin_keys);
  WalrusMult(tmp, input_ct[1], input_ct[1], evaluator, relin_keys);
  WalrusSub(output_variables[0], output_variables[0], tmp, evaluator);
  WalrusMult(output_variables[0], output_variables[0],
      output_variables[0], evaluator, relin_keys);

  // Compute beta_1 = 2 * (2*N0 + N1)^2
  WalrusMultPlain(output_variables[1], input_ct[0], pt_two, evaluator);
  WalrusAdd(tmp, output_variables[1], input_ct[1], evaluator);
  WalrusMult(output_variables[1], tmp, tmp, evaluator, relin_keys);
  WalrusMultPlain(output_variables[1], output_variables[1], pt_two,
      evaluator);

  // Compute beta_2 = (2*N0 + N1) * (2*N2 + N1)
  WalrusMultPlain(output_variables[2], input_ct[2], pt_two, evaluator);
  WalrusAdd(tmp_2, output_variables[2], input_ct[1], evaluator);
  WalrusMult(output_variables[2], tmp_2, tmp, evaluator, relin_keys);

  // Compute beta_3 = 2 * (2*N2 + N1)^2
  WalrusMult(output_variables[3], tmp_2, tmp_2, evaluator, relin_keys);
```

```
38    WalrusMultPlain(output_variables[3], output_variables[3], pt_two,
          evaluator);
39
40    return output_variables;
41  }
```

Listing 10: This code shows the Walrus evaluation pass for the Chi Squared benchmark. Each output is computed as a series of additions and multiplications. For efficiency, some intermediate values are saved in temporary variables and re-used later for help with computing subsequent output variables.

The Verilog program corresponding to the Chi squared test is shown in Listing 5. This particular code can be evaluated using all three of HELM's supported backends: 2:1 LUTs, gates, and arithmetic mode. Conversely, Listing 10 showcases the evaluation pass for Walrus for the identical program. Unlike the neural network inference, the Chi squared application does not require any manual modulus switching to match ciphertext parameters on behalf of the user. This is primarily due to the fact that the majority of ciphertext multiplications are squaring operations. Additionally, the code utilizes two additional intermediate ciphertexts to hold the results of specific computations in order to avoid recomputing them later; this further reduces the total depth and computational overhead of the circuit. A more in-depth discussion about modulus switching to match ciphertext parameters can be found in Section III. We remark that a similar technique is also employed by the Verilog code in Listing 5.

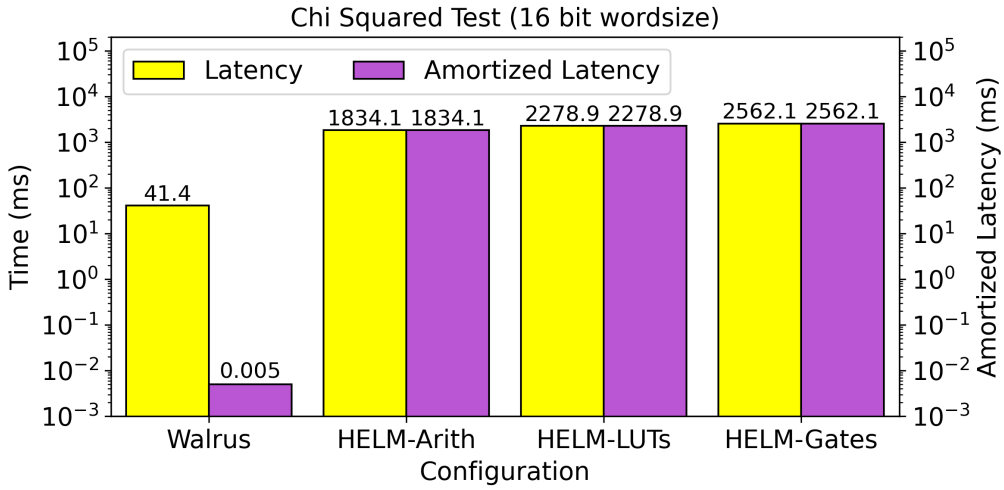

Fig. 12: **Chi Squared Comparative Performance:** The amortized latency axis represents the throughput measurement; as the CGGI cryptosystem does not allow batching, the latency and throughput are identical. From the results, it is evident that the LHE is the proper choice for this particular benchmark due to its shallow depth and prevalence of arithmetic operators. However, for deeper circuits with predominantly bitwise operations like AES, LHE evaluation would be a poor choice.

Figure 12 shows the results of executing the Chi squared benchmark, where Walrus configured with the BGV cryptosystem chose a parameter set with a polynomial degree of 2^{13} at 128 bits of security and HELM was configured with the default 128 bit parameter sets for all three modes (with polynomial degrees varying from 2^{10} to 2^{11}). Of these modes, the arithmetic mode performs the best as it natively supports multi-bit addition and multiplication as primitive operations. Conversely, the corresponding LUT circuit consists of nearly 3500 2:1 LUT operations (where each one requires a bootstrapping), while the gate circuit consists of over 3700 encrypted logic gates. Notably, all HELM variants are parallelized while Walrus is single-threaded; even so, Walrus exhibits significantly lower latency for this benchmark as the depth is shallow (allowing small parameters) and arithmetic operations are more efficient in BGV (native support) compared to CGGI (implemented as multi-bit adder/multiplier circuits).

VI. Lessons Learned

This tutorial has demonstrated how to employ two distinct frameworks for enhancing applications with both leveled and fully homomorphic encryption. Any program that can be implemented as a directed acyclic graph with nodes that involve arithmetic, bitwise, and relational operators can be readily converted with at least one of the outlined frameworks.

The Walrus LHE methodology enables high throughput applications with low latency at the cost of imposing limitations on the applications (i.e. the maximum multiplicative depth is bounded). Additionally, the ciphertexts that Walrus generates are significantly larger in size than those employed by HELM, but the capacity of each ciphertext is far larger as Walrus can fit thousands of multi-bit numbers into a single ciphertext. However, using this ability requires some modifications on behalf of the user and requires re-interpreting the application as a series of vector operations. In the same vein, modifications related to matching the moduli of each ciphertext may be required for certain applications, as illustrated by the mod switching calls in Listing 3. Additionally, it is impossible to branch on encrypted data, so changes in control flow based on a `WalrusCtxt` can't be resolved. As a consequence, applications such as sorting must always incur the worst-case performance. In this case, a data-independent sorting network such as the Batcher network [35] are more optimal as they exhibit a better worst-case complexity than algorithms like Quicksort. We note that it is possible to resolve multiple branches in specific cases, albeit at a steep computational cost. For instance, a ternary operation such as `x = y ? a : b`, where `x` is set to `a` if `y` is true, or `b` otherwise, can be implemented as a series of arithmetic operations. This can be done using the following equation (all variables are encrypted): $x = y \times a + (1 - y) \times b$. The key constraint of this approach is that `y` encrypts `0` or `1`.

Conversely, the HELM framework is optimized for reduced latency of applications that can be interpreted as large Boolean circuits. Unlike Walrus, the only constraint that HELM imposes on the input programs is that it must be synthesizable. Assuming an HDL design satisfies this constraint, it is automatically compatible with HELM and no manual changes to the source code are required on behalf of the user. However, Boolean FHE is not a good choice for applications with a large number of multi-bit arithmetic operations, as seen in Section V where Walrus exhibited both lower latency and significantly higher throughput than all modes of HELM. One of the largest influences on circuits that have multi-bit arithmetic operations is the chosen word size, as larger word sizes drastically increase the number of logic gates required to evaluate the operation. Therefore, it is extremely important to use the smallest word size possible in the input HDL design in order to maximize the efficiency of the FHE operations. Lastly, we note that HLS techniques can be utilized in tandem with Yosys to allow programmers to use a high-level language like C to design applications for HELM. Indeed, prior work has utilized this approach with HELM in the context of performing private set similarity [36]. The primary downside of this approach is the cost of the HLS conversion process from C to Verilog, but we note that this is a one-time cost. For some complex applications, the usability benefits of incorporating HLS into the toolchain may outweigh this downside.

VII. Conclusion

Usability is one of the largest hurdles that HE needs to overcome to see widespread adoption. In this tutorial, we demonstrate two approaches that can be used to efficiently and easily develop applications in the encrypted domain. The first approach discusses a new demonstration framework called Walrus, which serves as a front-end for the SEAL LHE library. Walrus is well-suited to applications that exhibit a limited multiplicative depth, are composed of arithmetic operations, and demand high throughput requirements. For all other applications, we discuss how the HELM framework can enable arbitrary computation with FHE; HELM can readily evaluate existing HDL designs written in Verilog with minimal changes and evaluate the design with FHE efficiently on high performance cloud servers. We demonstrate both methodologies with realistic benchmarks in the form of a shallow neural network inference for Walrus and an AES decryption circuit for HELM.

References

[1] C. Gentry, “Fully homomorphic encryption using ideal lattices,” in *ACM Symposium on Theory of Computing*, 2009, pp. 169–178.

[2] A. Greenberg, “IBM’s Blindfolded Calculator.” [Online]. Available: https://www.forbes.com/forbes/2009/0713/breakthroughs-privacy-super-secret-encryption.html?sh=5a71e2a37124

[3] 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, 2020.

[4] L. Folkerts, C. Gouert, and N. G. Tsoutsos, "REDsec: Running Encrypted Discretized Neural Networks in Seconds," Cryptology ePrint Archive, Report 2021/1100, 2021.

[5] 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.

[6] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.

[7] A. Benaissa, B. Retiat, B. Cebere, and A. E. Belfedhal, "Tenseal: A library for encrypted tensor operations using homomorphic encryption," 2021.

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

[9] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[10] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE transactions on information theory*, vol. 31, no. 4, pp. 469–472, 1985.

[11] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *International conference on the theory and applications of cryptographic techniques*. Springer, 1999, pp. 223–238.

[12] J. D. Cohen and M. J. Fischer, *A robust and verifiable cryptographically secure election scheme*. Yale University. Department of Computer Science, 1985.

[13] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *Journal of the ACM (JACM)*, vol. 56, no. 6, pp. 1–40, 2009.

[14] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *Cryptology ePrint Archive*, 2012.

[15] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 409–437.

[16] L. Ducas and D. Micciancio, "FHEW: bootstrapping homomorphic encryption in less than a second," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 617–640.

[17] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *International conference on machine learning*. PMLR, 2016, pp. 201–210.

[18] E. Aharoni, A. Adir, M. Baruch, N. Drucker, G. Ezov, A. Farkash, L. Greenberg, R. Masalha, G. Moshkowich, D. Murik *et al.*, "Helayers: A tile tensors framework for large neural networks on encrypted data," *arXiv preprint arXiv:2011.01805*, 2020.

[19] C. Gouert, V. Joseph, S. Dalton, C. Augonnet, M. Garland, and N. G. Tsoutsos, "Accelerated encrypted execution of general-purpose applications," *Cryptology ePrint Archive*, 2023.

[20] C. Gouert, D. Mouris, and N. G. Tsoutsos, "HELM: Navigating Homomorphic Encryption through Gates and Lookup Tables," Cryptology ePrint Archive, Paper 2023/1382, 2023, https://eprint.iacr.org/2023/1382.

[21] C. Wolf, "Yosys open synthesis suite," http://www.clifford.at/yosys/.

[22] 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.

[23] S. Gorantala, R. Springer, S. Purser-Haskell, W. Lam, R. Wilson, A. Ali, E. P. Astor, I. Zukerman, S. Ruth, C. Dibak *et al.*, "A general purpose transpiler for fully homomorphic encryption," *arXiv preprint arXiv:2106.07893*, 2021.

[24] 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.

[25] I. Chillotti, M. Joye, D. Ligier, J.-B. Orfila, and S. Tap, "Concrete: Concrete operates on ciphertexts rapidly by extending tfhe," in *WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, vol. 15, 2020.

[26] F. Brglez, "A neutral netlist of 10 combinatorial benchmark circuits and a target translator in FORTRAN," in *IEEE ISCAS*, 1985, pp. 663–698.

[27] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *IEEE ISCAS*, 1989, pp. 1929–1934.

[28] 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.

[29] K. Matsuoka, R. Banno, N. Matsumoto, T. Sato, and S. Bian, "Virtual secure platform: A {Five-Stage} pipeline processor over {TFHE}," in *30th USENIX security symposium (USENIX Security 21)*, 2021, pp. 4007–4024.

[30] 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)*. IEEE, 2020, pp. 1–6.

[31] akashjha25, "AES Decryption," 2021, https://github.com/akashjha25/aes-decryption/tree/main.

[32] C. Gouert, D. Mouris, and N. G. Tsoutsos, "SoK: New Insights into Fully Homomorphic Encryption Libraries via Standardized Benchmarks," *Proceedings on Privacy Enhancing Technologies*, vol. 2023, no. 3, p. 154–172, Jul. 2023.

[33] X. Sun, P. Zhang, M. Sookhak, J. Yu, and W. Xie, "Utilizing fully homomorphic encryption to implement secure medical computation in smart cities," *Personal and Ubiquitous Computing*, vol. 21, pp. 831–839, 2017.

[34] A. Viand, P. Jattke, and A. Hithnawi, "Sok: Fully homomorphic encryption compilers," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1092–1108.

[35] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. New York, NY, USA: Association for Computing Machinery, 1968, pp. 307–314.

[36] R. Shokri, C. Gouert, and N. G. Tsoutsos, "MatcHEd: Privacy-Preserving Set Similarity based on MinHash," 2023, https://github.com/TrustworthyComputing/minhash-HE/blob/main/MatcHEd.pdf.