ROMEO: Conversion and Evaluation of HDL Designs in the Encrypted Domain

Charles Gouert, Nektarios Georgios Tsoutsos
{cgouert, tsoutsos}@udel.edu
University of Delaware

Abstract

As cloud computing becomes increasingly ubiquitous, protecting the confidentiality of data outsourced to third parties becomes a priority. While encryption is a natural solution to this problem, traditional algorithms may only protect data at rest and in transit, but do not support encrypted processing. In this work we introduce ROMEO, which enables easy-to-use privacy-preserving processing of data in the cloud using homomorphic encryption. ROMEO automatically converts arbitrary programs expressed in Verilog HDL into equivalent homomorphic circuits that are evaluated using encrypted inputs. For our experiments, we employ cryptographic circuits, such as AES, and benchmarks from the ISCAS'85 and ISCAS'89 suites.

I. INTRODUCTION

As corporations and individuals produce an ever-increasing amount of sensitive data, a high demand has arisen for outsourcing these vast data sets to the cloud. Storing large amounts of data locally incurs high monetary and time costs in order to develop and maintain the necessary hardware infrastructure. Even though outsourcing large data sets to the cloud can be expensive, the benefits outweigh the disadvantages in many situations. Nevertheless, there are glaring problems with this approach: the security of the outsourced data is entirely dependent on the cloud service providers (CSPs), and curious CSPs can view the sensitive data stored on their servers.

As more users outsource their data to the cloud, attackers devise new methodologies to compromise the CSP servers hosting sensitive information. In fact, many research efforts have proposed and demonstrated viable attacks in this area [1]–[4]. Several unique solutions have been proposed to combat various attacks on cloud servers [5], [6], but have not seen widespread adoption. As the security of outsourced data lies entirely in the hands of the CSP, users need to take measures to ensure the confidentiality of their data.

A natural solution to these major problems outlined above is encryption, which can protect data *at rest* and *in transit*. Foe example, secure database frameworks such as Arx [7] and CryptDB [8] utilize encryption to protect stored data. Encryption prevents CSPs from viewing plaintext data and ensures privacy even if the cloud servers are compromised by attackers. However, these benefits come with a serious drawback: if the outsourced data is dynamic and should change over time, standard encryption remains limited. Indeed, to update and perform computations with the outsourced encrypted data, the data must first be pulled from the cloud, decrypted, used for computation, re-encrypted, and then re-uploaded to the cloud. This lengthy and computationally intensive process defeats the purpose of outsourcing in the first place.

To allow the cloud to carry out operations on encrypted data, it is necessary to utilize special algorithms that protect *data in use*. Fully homomorphic encryption (FHE), often referred to as the “holy grail” of cryptography [9], enables arbitrary computation on encrypted data and can eliminate the lengthy process previously discussed. FHE allows the cloud to carry out meaningful computations while remaining completely oblivious to details about the plaintext data [10].

While open-source homomorphic encryption libraries are readily available today, they are prohibitively difficult to use for non-crypto savvy programmers. Various parameters must be set properly to ensure sufficient levels of security, complicated objects and variables must be initialized (and later properly deleted), and a deep understanding of the library’s API is required to properly carry out computations on ciphertexts. Also, for many libraries, ciphertext noise must be continuously monitored to determine when special noise-reduction steps are required to ensure successful decryption.

In this work, we present ROMEO: a novel framework that eliminates the steep learning curve of FHE by automatically converting arbitrary Verilog programs to equivalent homomorphic programs compatible with the state-of-the-art TFHE library [11]. Security parameters, key generation and management, ciphertext generation, and freeing memory are handled transparently and abstracted away from the user. Specifically, our contributions can be summarized as follows:

- Automated conversion of algorithms expressed in Verilog into equivalent homomorphic circuits that enable privacy-preserving processing of encrypted data on the cloud.
- A novel compiler that translates combinational and sequential netlists into standard C++ code implementing equivalent fully homomorphic operations.
- A versatile execution engine that enables homomorphic evaluation of state machines and sequential algorithms using encrypted clock signals.

The remainder of the paper is organized as follows: Section II provides a brief background on FHE, modern implementations, and the TFHE library employed in this work. Section III presents an overview of our ROMEO framework, while Section IV presents our experimental evaluation. Section V offers comparisons with related works, and Section VI presents our concluding remarks.

II. PRELIMINARIES

A. Basics of Homomorphic Encryption

Homomorphic encryption (HE) allows users to perform operations on encrypted data without ever exposing the plaintext. In particular, for an arbitrary function $\mathcal{F}$ on plaintexts "a" and "b" there is an HE-equivalent function $\mathcal{G}$ on the encryptions of "a" and "b" so that $\mathcal{F}(a, b) = \mathrm{Dec}(\mathcal{G}(\mathrm{Enc}(a), \mathrm{Enc}(b)))$ (i.e., decrypting the value of $\mathcal{G}$ on ciphertexts yields the value of $\mathcal{F}$ on plaintexts). Since HE schemes never expose plaintext data while carrying out computations, this form of encryption enables companies and individuals to outsource sensitive data to untrusted third parties, such as the cloud, and dictate them to perform homomorphic operations on that data.

Various "partially" HE schemes, such as RSA [12], Paillier [13], and ElGamal [14], have existed for several decades and support only certain homomorphic operations (such as addition or multiplication, but not both). In addition, there exist "leveled" HE schemes that allow evaluating Boolean circuits up to a certain depth [15]; the latter lacks a mechanism to deal with noise accumulated in ciphertexts after each operation. In fact, as more operations are performed on ciphertexts, they could eventually become non-decryptable and completely useless. Thus, the circuit depth must be restricted to keep the ciphertext noise within acceptable levels.

In 2009, Gentry proposed a groundbreaking FHE scheme that enables evaluating Boolean circuits of arbitrary depth without noise problems [10]. Specifically, Gentry was able to reset ciphertext noise using a technique called *bootstrapping*, which entails evaluating a ciphertext decryption circuit homomorphically. Surprisingly, this technique reduces ciphertext noise to safe levels and allows unlimited computations. Gentry's method paved the way for the first generation of FHE.

The implementations of first generation FHE schemes were much slower than today's state-of-the-art libraries. For example, one bootstrapping operation took between 30 seconds with weak security parameters and approximately 30 minutes with strong security parameters using one of the first available FHE libraries [16]. In 2012, it was also demonstrated that the AES circuit could be evaluated homomorphically within 36 hours [17]. At that time, homomorphic encryption was infeasible for use outside of the academic sphere due to its slow speeds and low memory efficiency.

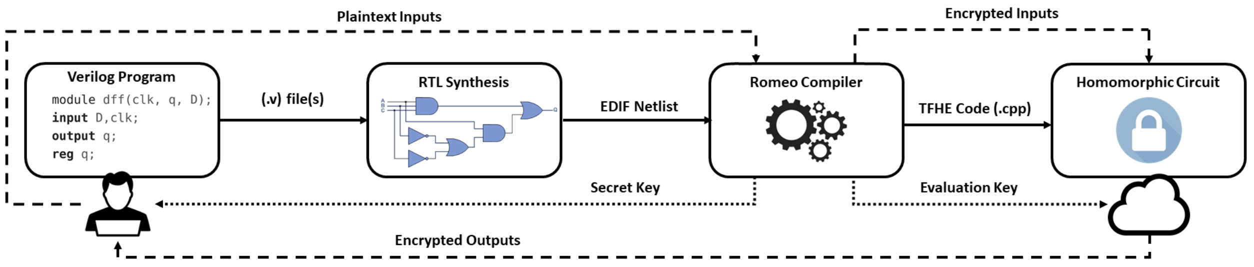
Over time, new FHE schemes have been proposed that drastically improved the speed of bootstrapping and other homomorphic operations. Gentry, Sahai, and Waters started this trend with their seminal 2013 paper proposing a scheme known as the *GSW cryptosystem*, which reduced the execution time of homomorphic addition and multiplication by transforming them into matrix addition and multiplication respectively [18]. Notably, this scheme also does not require the untrusted third party carrying out computations on ciphertexts to have an evaluation key. A novel scheme introduced in 2014 called FHEW [19], built upon GSW to create a library that could execute bootstrapping procedures in less than one second. Initially, FHEW supported only the FHE equivalent of a NAND operation with bootstrapping; this was chosen because it is a functionally complete operation (i.e., it can implement any arbitrary function). Building upon the principles of FHEW, in 2017 a new open source library called "TFHE: Fast Fully Homomorphic Encryption over the Torus" has been proposed [11].

B. Homomorphic Encryption Libraries

To date, several open-source homomorphic encryption libraries are available. *HElib* [16], the first publicly available HE library, performs mathematical operations on multi-bit ciphertexts and can compute any polynomial function of arbitrary degree. However, this library has several drawbacks that make it impractical for general purpose computation. First, bootstrapping speeds and evaluation times remain high compared to newer libraries. In addition, HElib exposes a complex API that requires users to tune multiple security parameters, as well as manually keep track of ciphertext noise and determine when bootstrapping should be applied.

In 2018, Microsoft released their own homomorphic library called *SEAL* [20]. While this library provides users with a simpler API that allows conducting additions and multiplications on ciphertexts, it is not capable of FHE in its current state. Instead, SEAL provides *leveled* homomorphic encryption, which does not offer a bootstrapping function and therefore allows for only a finite number of operations on ciphertexts. While this may be suitable for some applications, it is not sufficient for general-purpose computation (as in ROMEO) that requires support of circuits of arbitrary depth.

FHEW, as described previously, initially implemented only NAND evaluations on encrypted bits, while in 2017, increased functionality was added to the library, including NOR, OR, AND, and NOT evaluations. While FHEW is fully homomorphic and provides fast bootstrapping speeds compared to prior schemes, its successor, TFHE, boasts even faster speeds [21]. *TFHE* is a fast FHE library first released in 2017, which is a successor to FHEW and operates exclusively on Boolean circuits. All ciphertexts are encrypted as binary values: plaintext data is converted to binary, encrypted bit by bit, and stored in a ciphertext array that has a size of approximately $2.2kB * N$, where N is the number of bits in the plaintext. The TFHE library offers the ability to carry out any logic gate function on ciphertexts and handles bootstrapping automatically after each gate evaluation (except the NOT gate that does not need bootstrapping). Since TFHE supports evaluation of all types of logic gates (i.e., it offers multiple functionally-complete sets



```
module dff(clk, q, D);
input D,clk;
output q;
reg q;
```

Fig. 1. ROMEO Outline. Verilog designs are converted to netlists and then passed to the ROMEO compiler. The compiler administers keys, receives inputs from the user, and generates an encrypted circuit to the cloud for outsourcing. When the cloud finishes the circuit evaluation, the resulting ciphertext is sent to the user.

of operations), it supports arbitrary computation on encrypted data. This property, as well as the fact that it can evaluate circuits of arbitrary depth, classify it as fully homomorphic. TFHE provides very competitive bootstrapping speeds and gate evaluation times. Thus, TFHE is an ideal candidate for use with ROMEO.

III. The Romeo Framework

ROMEO offers the following functionality: it consumes Verilog programs and outputs a homomorphic circuit operating on encrypted data, which can be evaluated by an untrusted remote party. To accomplish this, the first step is to use *synthesis* to convert Verilog programs to netlists consisting of logic gates and primitive memory structures like flip flops. Next, the generated netlist serves as an input to ROMEO's special compiler that parses the circuit, determines the correct execution order of the gates, and generates an equivalent and efficient homomorphic program. An outline of our framework is illustrated in Figure 1.

A. RTL Synthesis

To handle synthesis, ROMEO's back-end uses the Yosys Open SYnthesis Suite [22], which is an open source toolchain performing RTL synthesis along with basic circuit optimization functionality. Our framework receives Verilog source code files as input and instructs the Yosys back-end to apply the following:

1) perform optimizations including removing unused wires and replacing process blocks with flip-flops;
2) map cells to standard logic gates and small multiplexers;
3) write resulting netlist as an EDIF (Electronic Design Interchange Format) file.

B. Combinational Circuit Conversion

Once an EDIF netlist is generated by Yosys, ROMEO's compiler transforms it into a standard C/C++ program composed of homomorphic operations exposed by TFHE's API. First, the EDIF netlist is scanned and the compiler identifies all gates and wires in the circuit. On a second pass, connections between gates and wires are made and the circuit detailed in the EDIF file is now fully constructed. Finally, the C/C++ source file is created and all ciphertext structures required for the circuit (i.e., one ciphertext per wire) are initialized.

To begin conversion, our compiler takes plaintext inputs from the user in binary and generates C++ code that calls TFHE functions to encrypt them. The now encrypted inputs are loaded into their corresponding input wires in the HE circuit using TFHE's copy gate functionality, which introduces negligible overhead. Next, ROMEO constructs a Directed Acyclic Graph (DAG) to determine the execution order of all gates in the HE circuit. This is necessary as homomorphic gate evaluations are serialized and, for each gate, all dependent gate evaluations must be completed before the current gate's input wires are assigned the correct ciphertext values. The DAG construction is outlined in Algorithm 1: the graph is traversed until all gate operations have been written consecutively to the generated C++ file. Finally, the ciphertexts corresponding to output wires are saved in a file and all ciphertext structures are destroyed.

C. Sequential Circuit Conversion

Evaluating sequential circuits in the encrypted domain requires a more involved approach than purely combinational circuits. For one, the incorporation of a clock signal poses an important challenge for homomorphic evaluation: before using the clock signal as an input to an encrypted domain function, the current clock state must be encrypted. It is not possible, however, to mix plaintext clock signals with ciphertexts, and there are two approaches to address the requirement of encrypted clock signals: the user could either encrypt a large number of 0's and 1's prior to circuit evaluation and upload these values to the cloud, or instruct the cloud to encrypt these values as needed on-the-fly. In this work, we employ the latter approach in order to minimize the computation on the user side, as well as reduce the communication overhead between the user and remote cloud server.

In addition to the encrypted clock challenge, TFHE does not offer support for sequential circuit components such as flip flops (FFs). Thus, to incorporate FF functionality into homomorphic circuits, ROMEO implements a *gate re-evaluation technique* illustrated in Algorithm 2. First, we begin by instructing the cloud to generate an encrypted clock signal that initializes to '0' (and inverts after every complete pass through the circuit). Then, the cloud proceeds to evaluate the circuit like a combinational circuit; when a FF is reached, the data input to the FF is stored for the next round and the output takes on the FF

Algorithm 1: Determine Order of Gate Evaluations

```
for gate in circuit do
    for wire in gate.inputs do
        if wire is output from other gate then
            gate.dependsOn += wire.originator;
while unevaluated gates remain do
    if gate.evaluated == True then
        continue;
    for gate in circuit do
        if gate.dependsOn == ”” then
            gate.evaluated = True;
            write_gate_to_file(gate);
        else
            ready = True;
            for prevGate in gate.dependsOn do
                if prevGate.evaluated == False then
                    ready = False;
            if ready == True then
                gate.evaluated = True;
                write_gate_to_file(gate);
return;
```

input from the previous round. On subsequent passes through the circuit, only gates that depend on the output of FFs and gates upon which FFs are dependent are re-calculated. Purely combinational logic networks separated from sequential components are only executed on the initial pass as their outputs will not change over time.

Notably, the cloud remains oblivious to the number of clock cycles necessary to finish a circuit evaluation. This stems from the fact that the cloud has no knowledge about the plaintext values assigned to wires and signals in the circuit. Thus, users can define in advance how many clock cycles are necessary for the circuit to complete its evaluation. While ROMEO's compiler is generating the homomorphic circuit for outsourcing, it will prompt the user for the number of timesteps required during evaluation. The compiler will re-evaluate the necessary logic gates for each additional timestep. In ROMEO, combinational circuits are treated as sequential circuits with a single timestep.

Algorithm 2: Optimized Circuit Re-evaluation

```
Function re-eval(gate):
    if gate precedes flip-flop then
        flag gate for re-evaluation;
        for prevGate in gate.dependsOn do
            re-eval(prevGate);
    else if gate follows flip-flop then
        flag gate for re-evaluation;
        for nextGate in gate.next do
            re-eval(nextGate);
    return;
```

D. Circuit Verification using Debug Mode

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

To rapidly verify the accuracy of the circuit in debug mode, ROMEO can encrypt circuit inputs using the evaluation key instead of the private key. Normally, the former key is used to encrypt non-sensitive constant values for computation with sensitive encrypted ciphertexts and TFHE treats such ciphertexts generated with the evaluation key as "trivial", assuming that both the third party and the user know the corresponding plaintext values. The executing overhead for FHE gates processing these "trivial" ciphertexts is very fast at approximately 10 microseconds per gate evaluation. This is three orders of magnitude faster than the typical FHE gate evaluation speed of 13 ms [11]. Using this feature, users can evaluate correctness of FHE circuits very efficiently. We remark that ROMEO's debug mode can only be used locally, as it is insecure to encrypt data with the evaluation key while outsourcing to the cloud.

IV. EXPERIMENTAL EVALUATION

The ROMEO framework was used to convert all combinational and sequential circuits from the ISCAS '85 [23] and ISCAS '89 [24] benchmark suites to the encrypted domain. In addition, we converted

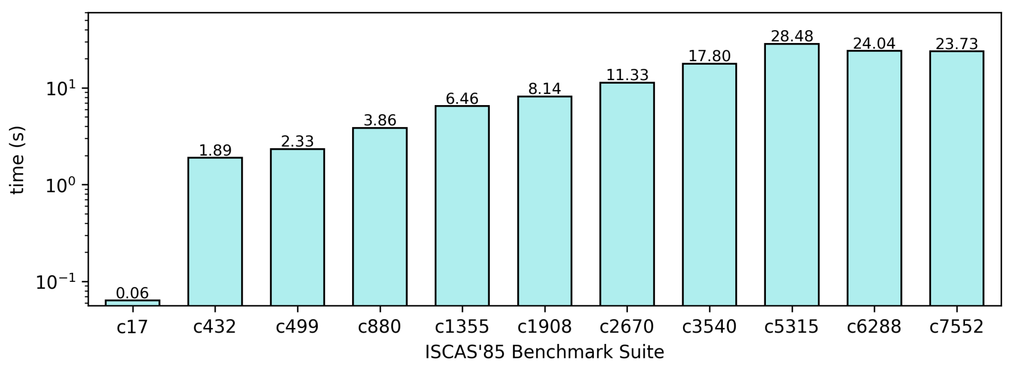


Fig. 2. Encrypted circuit evaluation times for the ISCAS '85 benchmark suite.

five encryption benchmark circuits to demonstrate the robustness of our framework. These benchmarks were chosen due to their widespread use, the broad range of circuit sizes, and the inclusion of both combinational and sequential circuits. All experiments were performed on an Ubuntu 18.10 host with 8 GBs of RAM and an i7-8650U CPU. The TFHE security parameter (λ) was set at the default value for 110 bits of security. Lastly, the reported times were averaged over 10 executions per circuit and each execution was assigned one exclusive processor core.

A. *ISCAS Combinational Circuits*

The homomorphic circuit evaluation times for the ISCAS '85 combinational benchmarks are presented in Figure 2. Our results show an approximately linear increase in execution time with the number of evaluated gates. Nevertheless, the evaluation time for different gates are not the same: for instance, inverters are evaluated much faster than other logic gates because no bootstrapping is required for this operation. As illustrated in the graph, the `c5315` circuit incurs longer evaluation times than the two largest circuits despite its smaller size. This deviation from expected behavior is attributed to the proportion of inverter gates to the overall number of gates in the circuit. Indeed, the two largest circuits contain approximately 34% inverters while `c5315` contains about 25% inverters.

B. *ISCAS Sequential Circuits*

The results for the ISCAS '89 sequential circuit benchmarks are presented in Figure 3. These numbers show the amortized execution cost per cycle (i.e., one complete circuit evaluation). This cost was amortized over ten clock cycles. As with the combinational results, a roughly linear increase in execution time is observed with increasing numbers of gates as anticipated. However, more variance is observed due to the varying number of gates that need to be re-evaluated for each cycle. This is entirely dependent on the circuit configuration.

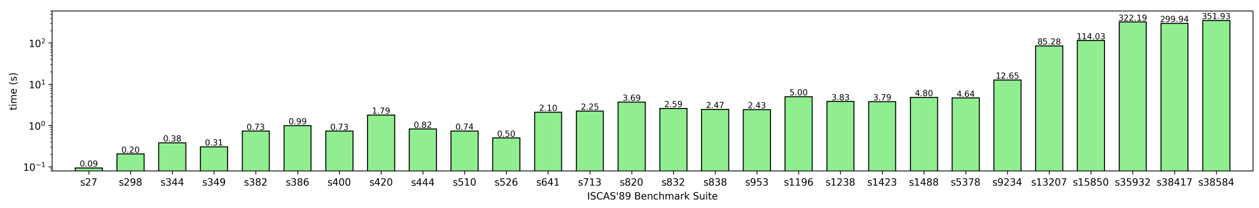


Fig. 3. Amortized evaluation time per cycle (over 10 cycles) for encrypted circuits from the ISCAS '89 benchmark suite.

C. Encryption Circuits

To further illustrate the robustness of the ROMEO framework, we tested its performance using five circuits implementing the following well-known encryption algorithms: DES [25], AES [26], PRESENT [27], SIMON and SPECK [28]. The last three algorithms are *lightweight* block ciphers and their circuits are suited for homomorphic evaluation. In more details, PRESENT has an 80-bit key and a 64-bit block size, while SIMON and SPECK ciphers [28] support a variety of block and key sizes (in this work, we implemented the 128/128 variants with 128-bit block size and 128-bit key size). Moreover, DES uses 56 bit keys (with 8 parity bits added for a total of 64 bits) and a 64 bit block size. Finally, AES, the most widely used encryption cipher today, uses a 128 bit key and a 128 bit block size [26]. Our experimental results in Figure I show that the homomorphic evaluation of PRESENT was the fastest, with SIMON and SPECK being slightly slower. Conversely, the homomorphic evaluation of DES took approximately 24 minutes and AES required 13.5 minutes due to the complexity and larger size of these circuits. In the case of DES, we attribute the slow speed due to the substitution step, which is implemented with look-up tables; since it is not possible to branch on encrypted data, all possible outputs must be computed for each look-up table evaluation.

TABLE I

EVALUATION TIMES FOR STANDARD ENCRYPTION ALGORITHMS

Cipher	**Evaluation Time (s)**	**Cycles**	**Gate Evaluations**	**Input Wires**	**Output Wires**
PRESENT	107.35	31	12256	144	64
SIMON	129.28	68	13698	256	128
SPECK	152.70	32	17821	256	128
DES	1461.29	16	167058	120	64
AES	810.65	10	61113	256	128

D. Scheme Hopping on Cloud Servers

The lightweight ciphers in Section IV-C enable practical applications of encrypted computation, such as *scheme-hopping*. With scheme-hopping, users first encrypt their sensitive data with a symmetric encryption algorithm (e.g., compute SIMON ciphertexts that are much smaller than TFHE ciphertexts) and then upload these encryptions to a cloud server; in turn, the cloud server encrypts for a second time each bit of these ciphertext with TFHE. The users also encrypt each bit of their symmetric key (i.e., the SIMON key) with TFHE and upload these encryptions to the cloud server as well. Using ROMEO, the cloud server can generate and evaluate the FHE circuit corresponding to *symmetric decryption* (e.g., SIMON decryption) using the TFHE ciphertexts of the symmetric key and the user data. This process "peels off' the symmetric encryption and result in a TFHE ciphertext on the cloud server. Depending on the size of the initial plaintext, this can drastically reduce the communication overhead between the user and cloud, as uploaded user data are symmetrically encrypted (only the key bits are encrypted with FHE).

To demonstrate this method, we utilized an Amazon EC2 instance with 48 vCPUs and 384 GiB of memory to perform scheme-hopping using SIMON. The local host computed a Simon ciphertext for a 128-bit plaintext, as well as the TFHE encryption of SIMON's key (this resulted in a 128 * 2.2 KB ciphertexts). The TFHE-encrypted Simon key and the 128-bit Simon ciphertext were uploaded to the EC2 instance (this step took 2.1 seconds) and the Amazon server was able to "peel-off" the symmetric encryption and compute a TFHE ciphertext corresponding to the original 128-bit plaintext. This evaluation took 19.63 seconds on the EC2 server, and minimized upload overhead of the local host.

E. User Overhead for TFHE Encryption and Decryption

From the user's perspective, there is a one time cost to generate a keypair (which can be used for multiple circuits) and encrypt inputs with the secret key. On average, key generation takes approximately 770 milliseconds with 110 bits of security and the cost of encryption is 22 microseconds per bit of plaintext. The decryption operation time is negligible at less than 1 microsecond per bit.

V. RELATED WORKS

While fully homomorphic encryption has garnered a great deal of attention in the years since its inception, the majority of research efforts in this field focus on acceleration, improving existing schemes, and specific applications of homomorphic encryption. For instance, recent works have explored the potential of neural network training and inferencing in the encrypted domain [29] [30]. To the best of the authors' knowledge, there is no framework that supports complete conversion of arbitrary HDL designs to encrypted circuits. However, past research efforts have been made to make homomorphic encryption more usable for the average programmer.

The E^3 framework [21] provides users with an API that allows them to flag sensitive variables as "secure" in C/C++ programs. These variables are homomorphically encrypted and each program statement involving these variables will generate a corresponding homomorphic circuit. In addition, E^3 offers users the choice of HElib, FHEW, or TFHE 1.0. However, this approach requires users to modify their source code and does not support arbitrary functionality (e.g., can't process conditionals on encrypted data).

The Cingulata compiler toolchain [31] allows for conversion of C/C++ programs to homomorphic circuits and provides similar functionality to E^3 with some caveats. It requires users to modify their programs to work with the toolchain and, while providing a simpler API than many homomorphic encryption libraries, it requires significant effort on behalf of the user to understand the nuances of the Cingulata library and its associated structures and data types. Conversely, ROMEO abstracts this complexity and enables automated conversion of HDL code into C++ executables.

VI. CONCLUSION

In this work, we have proposed a novel framework for automated conversion from arbitrary synthesizable Verilog HDL designs to encrypted circuits for privacy outsourcing applications. First, Verilog designs are converted to netlists through the process of synthesis. Next, the ROMEO custom compiler creates an internal construction of the circuit outlined in the netlist and determines the correct execution order for the homomorphic gate evaluations. The resulting homomorphic circuit is written to a C++ source code file that employs the TFHE library and can be sent to the cloud for evaluation along with encrypted inputs. For the user's peace of mind, ROMEO provides a debug mode capable of fully simulating the homomorphic circuit locally to verify correct operation.

We tested ROMEO with circuits from the ISCAS '85 and '89 benchmark suites as well as five well-known cryptographic circuits. In all cases, we observed a roughly linear increase in encrypted circuit evaluation time with a growing number of gate evaluations. On a final note, it is possible for users to enhance the usability of this framework further by incorporating high level synthesis (HLS) tools into the toolchain. This would allow for assisted conversion from high level languages such as C/C++ to homomorphic circuits. The ROMEO framework is open source and is available at the following repository: https://github.com/TrustworthyComputing/Romeo.

References

[1] A. J. Duncan, S. Creese, and M. Goldsmith, "Insider attacks in cloud computing," in *IEEE TrustCom*, June 2012, pp. 857–862.

[2] Z. Tari *et al.*, "Security and privacy in cloud computing: Vision, trends, and challenges," *IEEE Cloud Computing*, vol. 2, pp. 30–38, Mar 2015.

[3] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *ASIACCS '16*. New York, NY, USA: ACM, pp. 353–364.

[4] Y. Xiao *et al.*, "One bit flips, one cloud flops: Cross-VM row hammer attacks and privilege escalation," in *USENIX Security*, 2016, pp. 19–35.

[5] F. Liu *et al.*, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *IEEE HPCA*, 2016, pp. 406–418.

[6] Y. Han *et al.*, "Using virtual machine allocation policies to defend against co-resident attacks in cloud computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 14, no. 1, pp. 95–108, 2015.

[7] R. Poddar, T. Boelter, and R. A. Popa, "ARX: A strongly encrypted database system," Cryptology ePrint Archive, Report 2016/591.

[8] R. A. Popa *et al.*, "CryptDB: Protecting Confidentiality with Encrypted Query Processing," in *SOSP*. ACM, 2011, pp. 85–100.

[9] D. Micciancio, "A first glimpse of cryptography's holy grail," *Communications of the ACM*, vol. 53, no. 3, p. 96, 2010.

[10] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, 2009.

[11] I. Chillotti *et al.*, "TFHE: Fast fully homomorphic encryption over the torus," *Journal of Cryptology*, pp. 1–58, 2018.

[12] 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.

[13] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *EUROCRYPT*. Springer, 1999, pp. 223–238.

[14] 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.

[15] D. Boneh, E.-J. Goh, and K. Nissim, "Evaluating 2-DNF formulas on ciphertexts," in *IACR TCC*. Springer, 2005, pp. 325–341.

[16] S. Halevi and V. Shoup, "Algorithms in HElib," in *CRYPTO*. Springer, 2014, pp. 554–571.

[17] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit," in *CRYPTO*. Springer, 2012, pp. 850–867.

[18] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *CRYPTO*. Springer, 2013, pp. 75–92.

[19] L. Ducas and D. Micciancio, "FHEW: bootstrapping homomorphic encryption in less than a second," in *EUROCRYPT*. Springer, 2015, pp. 617–640.

[20] "Microsoft SEAL (release 3.4)," https://github.com/Microsoft/SEAL, Oct. 2019, Microsoft Research, Redmond, WA.

[21] E. Chielle *et al.*, "E3: A framework for compiling C++ programs with encrypted operands," Cryptology ePrint Report 2018/1013.

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

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

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

[25] “Data Encryption Standard,” in *FIPS PUB 46, Federal Information Processing Standards Publication*, 1977.

[26] “Advanced Encryption Standard (AES),” in *FIPS PUB 197, Federal Information Processing Standards Publication*, 2001.

[27] A. Bogdanov *et al.*, “PRESENT: An ultra-lightweight block cipher,” in *CHES*. Springer, 2007, pp. 450–466.

[28] R. Beaulieu *et al.*, “The SIMON and SPECK lightweight block ciphers,” in *DAC*. IEEE/ACM, 2015, pp. 1–6.

[29] K. Nandakumar *et al.*, “Towards deep neural network training on encrypted data,” in *IEEE CVPRW*, 2019.

[30] R. Dathathri *et al.*, “CHET: an optimizing compiler for fully-homomorphic neural-network inferencing,” in *PLDI*. ACM, 2019, pp. 142–156.

[31] CEA-LIST, “Cingulata compiler toolchain,” https://github.com/CEA-LIST/Cingulata.