CoPHEE: Co-processor for Partially Homomorphic Encrypted Execution

Mohammed Nabeel*, Mohammed Ashraf*, Eduardo Chielle*, Nektarios G. Tsoutsos†, and Michail Maniatakos*
*Center for Cyber Security, New York University Abu Dhabi
†Electrical and Computer Engineering Department, University of Delaware

***Abstract*—The recent disclosure of the Spectre and Meltdown side-channel vulnerabilities offers yet another example of modern computer architectures prioritizing performance optimizations over security and privacy. The devastating impact of data leakage, however, emphasizes the need for new processor designs that provide native support for data privacy using cryptography. In this paper, we report on a year-long effort to design, implement, fabricate, and validate CoPHEE: a novel co-processor design that mitigates data leakage risks using partially homomorphic encrypted execution. ASIC designs for encrypted execution impose unique challenges, such as the need for non-traditional arithmetic units (modular inverse, greatest common divisor), very wide datapaths (2048 bits), and the requirement for secure multiplexer units enabling general-purpose execution on encrypted values. Our fully-functional co-processor chip is fabricated in 65nm CMOS technology, and communicates to a main processor via UART. This paper offers an elaborate overview of all steps and design techniques in the ASIC development process, ranging from RTL design to fabrication and validation. We evaluate our co-processor using data-oblivious C++ benchmarks, while our RTL files are available in an open-source repository.**

***Index Terms*—Data Privacy, Encrypted Execution, Partially-Homomorphic Encryption, Hardware Root-of-Trust, ASIC.**

I. INTRODUCTION

Cloud services have become popular as a robust form of outsourcing computation and storage. Third party cloud services nowadays contain user sensitive information regarding health, financial status, etc., in their databases. This process raises concerns about the security and privacy of the outsourced data, exacerbated by noteworthy compromises in recent years [1], despite the investment of cloud service providers in security. At the same time, more advanced threats, such as side channel leakage [2] and hardware Trojans [3], introduce new attack vectors against the privacy of outsourced computation.

Existing commercial cryptography solutions protect data at rest, mitigating privacy issues with data transfer and data storage. However, these solutions are not capable of manipulating encrypted data, i.e., perform operations directly on the encrypted domain. Even solutions in hardware, like Intel SGX, require the data to be processed as plaintext, which renders the entire microprocessor core and cache memories vulnerable to hardware Trojans and side channel attacks [4], [5].

In recent years, homomorphic encryption schemes have improved significantly. With the recent advancements in Fully Homomorphic Encryption (FHE) [6], non-trivial data manipulation directly in the encrypted domain became a reality, albeit non-practical. FHE schemes use homomorphic Boolean circuits and obtain the same result as if the data were manipulated in the open. Thus, no sensitive information is ever decrypted for processing, mitigating privacy risks. Nevertheless, existing FHE cryptosystems are not yet practical [7], since they incur excessive overheads in area and performance: Area overheads are in the order of megabytes per ciphertext [8], which prohibits efficient hardware implementations of FHE schemes. Performance-wise, FHE cryptosystems need to manipulate millions of bits to perform an operation [9], and they further require frequent *bootstrapping* for noise reduction [10]. Recent improvements in FHE reduced the bootstrapping overhead down to milliseconds [11], allowing the execution of dozens of gates in a second. This performance, however, is still orders of magnitude slower than non-FHE cryptosystems.

Partially Homomorphic Encryption (PHE) schemes offer a more practical and efficient alternative to FHE. These asymmetric cryptosystems support a single homomorphic operation (i.e., addition or multiplication), which can be applied for unlimited times without any need for noise reduction or bootstrapping. In modern PHE cryptosystems like Paillier [12] and RSA [13], the modular multiplication of ciphertexts is homomorphic to plaintext addition and multiplication respectively. Hence, since there exist very efficient instantiations of modular multipliers (e.g., [14]), our thesis is that any PHE-based algorithm that manipulates encrypted values natively can benefit from dedicated hardware units.

Since PHE schemes support only one homomorphic operation, they do not immediately offer the computational completeness of FHE schemes, given that universal computation requires two orthogonal operations (i.e., both addition and multiplication) [15]. In PHE-protected algorithms, however, this limitation only affects any runtime decisions controlled by encrypted values, and can be mitigated using a secure multiplexer as part of the root of trust. For example, Cryptoleq is a One Instruction Set Computer (OISC) architecture that supports universal computation of PHE-protected algorithms with practical overheads [16]. Internally, Cryptoleq is based on the Turing-complete Subleq abstract machine [17], and implements the Paillier cryptosystem along with a secure multiplexer that is obfuscated in software using single-instruction self-modifying code. As a protection mechanism, however, Cryptoleq's software obfuscation does not offer the same flexibility and guarantees as secure hardware units, since strong obfuscator programs do not generally exist (i.e., there are functions that cannot be obfuscated) [18].

Contributions: To address the performance and universality limitations while executing PHE-based algorithms, this work develops a novel *Co-processor for Partially Homomorphic Encrypted Execution* (CoPHEE), which targets the Paillier encryption scheme. Our processor is instantiated using 2048-bit encrypted operands and can be readily used to accelerate a broad range of secure applications, such as voting protocols, threshold cryptosystems, watermarking and secret sharing schemes, as well as server-aided polynomial evaluation protocols [12]. Towards that end, we develop special arithmetic units for modular multiplication (ModMul), exponentiation (ModExp), inversion (ModInv) and greatest common divisor (GCD). Likewise, to extend support for ciphertext-based control flow decisions in PHE-protected algorithms, we adopt Cryptoleq's blueprint and instantiate a secure multiplexer in trusted hardware, effectively minimizing the required trust surface to a single operation. Since CoPHEE operates directly on encrypted data, any observable difference in the execution flow would reveal side-channel information about the value of the corresponding control ciphertexts; therefore, the encrypted algorithms accepted by CoPHEE must have constant-time execution paths and predefined termination conditions.

Our CoPHEE chip has been fabricated at Global Foundries with 65nm CMOS technology and, to the best of our knowledge, it is the first of its kind to be fabricated. This paper presents a comprehensive chronicle from the initial design to post-silicon validation, and discusses implementation challenges and lessons learned, serving as a baseline for future work towards the design of homomorphic processors.

II. Preliminaries

A. Homomorphic Encryption and the Paillier Cryptosystem

Homomorphic encryption is a special form of encryption that allows meaningful manipulations directly on encrypted data. This property enables outsourcing to a third party the evaluation of a function that *hides its inputs using encryption*. Formally, if Enc denotes encryption, Dec denotes decryption, and $f()$ is a regular function applied on plaintext values a, b, then homomorphic encryption supports the following property:

$$f(a,b) = Dec\Big(g(Enc(a), Enc(b))\Big), \quad (1)$$

where $g()$ is the homomorphic counterpart of $f()$ that operates on encrypted values $Enc(a), Enc(b)$. For example, the Paillier cryptosystem [12] defines the encryption of value a as $Enc(a,r) = r^n(n+1)^a \bmod M$, where $M = n^2$, $n = pq$ is the product of two primes p, q, and r is a random value. In Paillier, the supported $f(a,b)$ is the modular addition $a + b \bmod \sqrt{M}$, and its homomorphic counterpart $g()$ is the modular multiplication of encrypted values $Enc(a) \cdot Enc(b) \bmod M$.

Generally, each homomorphic cryptosystem supports specific functions f, and their counterparts g: PHE cryptosystems support only one f (e.g., either addition or multiplication), while FHE cryptosystems support two orthogonal functions f_1, f_2 (e.g., both addition and multiplication) that form a functionally complete set of operations. Nevertheless, the corresponding g functions of PHE are significantly more efficient that the g_1, g_2 functions of FHE, so FHE remains prohibitive for practical applications [7].

B. Cryptoleq's Secure Multiplexer for Universal Computation

The homomorphic operation of Paillier enables the implementation of single-instruction abstract machines that can execute algorithms with runtime decisions that are not controlled by encrypted values [16]. In this case, to also support ciphertext-based runtime decisions, it is possible to extend the homomorphic properties of Paillier using a *secure multiplexer*. The latter operates within a root-of-trust and selects between two ciphertexts Y, Z using a cryptographic predicate $\mathcal{P}$ over a third ciphertext X. To support universal computation on Paillier ciphertexts, the aforementioned blueprint was introduced in Cryptoleq as function G, where the sign of $Dec(X)$ is the predicate for selecting between $Z = Enc(0)$ (i.e., the homomorphic *identity* element) and its second input Y:

$$G(X,Y) = \mathcal{P}(X) \cdot Y + \Big(1 - \mathcal{P}(X)\Big) \cdot Enc(0), \quad (2)$$

where $\mathcal{P}(X) = 0$ if $Dec(X) \leq 0$, otherwise $\mathcal{P}(X) = 1$. In this case, $\mathcal{P}(X)$ can be computed efficiently using modular exponentiation of X to a private parameter FKF [16].

C. Threat Model with Hardware Root-of-Trust

The Paillier cryptosystem offers provable security guarantees against Chosen-Plaintext Attacks, based on the hardness of decisional composite residuosity problem (DCRP) [12]. According to DCRP, for a given integer x and composite $n = pq$, it is hard to decide whether there exists an integer y such that $x \equiv y^n \bmod n^2$, without knowing the factors of n. Inheriting the security guarantees of DCRP, our threat model assumes that it is intractable to recover any value protected using Paillier when the bitsize of n is sufficiently large (i.e., at least 1024 bits). In addition, our threat model assumes the existence of a secure multiplexer in trusted hardware (i.e., CoPHEE is our root-of-trust), which protects exponentiations to FKF and cannot be tampered with by adversaries.

III. CoPHEE Design Flow Overview

The main objective of the CoPHEE co-processor is native support for modular operations over integers of size up to 2048 bits, which is essential for accelerating PHE cryptosystems such as Paillier. Our modular operations comprise multiplication, exponentiation and inversion. Moreover, our chip implements secure multiplexing, computes the GCD of 2048-bit integers, generates truly random numbers, and features a UART interface to communicate with the main processor. CoPHEE is implemented using commercial tools and industry standard Netlist–to–GDSII design flow, and is fabricated at Global Foundries with a 65nm technology node. Specifically, we used the Multi-Project Wafer (MPW) fabrication service from MOSIS, with a die area of $9mm^2$ and a target frequency of 100 Mhz (constrained by the maximum speed of the provided IO pads). To avoid complex integration of on-chip

TABLE I: IO pin description of CoPHEE

IO Pin	Direction	Description
VDDIO	in	3.3V voltage supply for IO pads
VDDC	in	1.2V voltage supply for the core logic
VSS	in	ground supply for IO pads and core logic
nRESET	in	active low reset
Clk	in	clock input (max. frequency: 100 Mhz)
RX	in	UART receive line from host
TX	out	UART transmit line to host
HostIRQ	out	interrupt to the host processor
GPIO	out	for post-silicon debugging

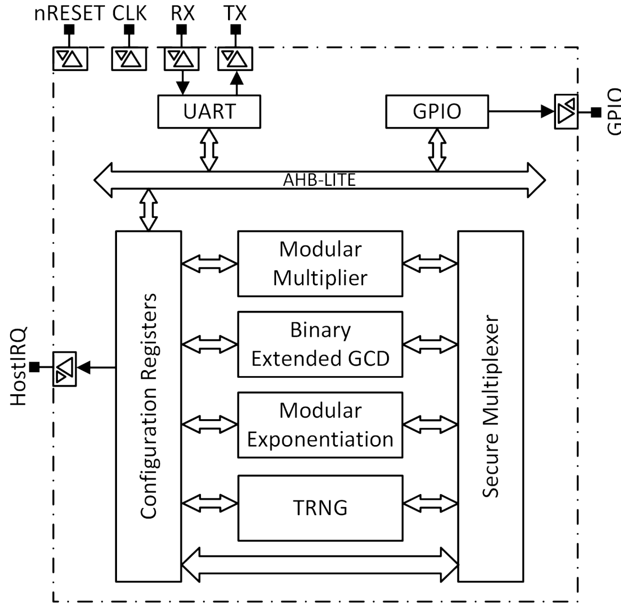

Fig. 1: Bus architecture diagram of CoPHEE. Resets and clock are connected to all blocks (their connections are not shown).

clock sources and minimize the risk of non-working silicon, CoPHEE uses an externally supplied clock. Moreover, our chip has two voltage supplies, namely 3.3 V (IO pads) and 1.2 V (logic core), and uses a Dual In-line Package (DIP).

A. External Interfaces

In Table I we present the main IO pins of CoPHEE and how they interface with the host computer. Using the receiver line RX of the UART interface, the host computer can program the value of each operand and then trigger a desired operation (e.g., modular multiplication). As soon as the trigger bit in the CoPHEE configuration registers is set through UART, the requested operation starts executing on the co-processor. When the operation terminates, CoPHEE sets the interrupt line HostIRQ to signal the host that the requested output is ready. After receiving this interrupt, the host processor reads the computed result via UART using the transmitter line TX and clears the interrupt HostIRQ. Moreover, CoPHEE also features GPIO to assist debugging and post-silicon validation.

B. Internal Data Flow

In Fig. 1 we present the internal bus architecture diagram of CoPHEE that is based on a single-master two-slave system, where the master communicates to the slaves using a 32-bit AHB-Lite bus protocol. Our goal is to make the AHB-Lite design parameterizable to facilitate the addition of masters or slaves to the bus. In CoPHEE, the UART is the only master on the bus, while the configuration registers unit and the GPIO are slaves. The configuration registers unit comprises special registers for the Paillier key, public modulus and encrypted operands, as well as registers for triggering operations, storing computed results and operation status. Likewise, the GPIO can provide status information during testing (e.g., one can blink an LED through GPIO to signal an error).

TABLE II: Subset of CoPHEE Configuration Registers

Register Name	Description	Bit Size
UARTMTX_PAD_CTL	IO Pad control for UART TX	32
UARTMRX_PAD_CTL	IO Pad control for UART RX	32
HOSTIRQ_PAD_CTL	IO pad control for Host Interrupt	32
GPIO0_PAD_CTL	IO pad control for GPIO	32
UARTM_BAUD_CTL	Baud control for UART	32
UARTM_CTL	UART control (parity, polarity, etc.)	32
CLEQCTL2	Log_2 of N	32
SIGNATURE	Stores Chip ID	32
CLCTLP	Trigger bits for modular blocks	32
CLCTL	Control bits	32
CLSTATUS	Flag bits (busy, inverse error, etc.)	32
N	Modulus N	1024
NSQ	Square of N	2048
ARGA	Argument A for modular blocks	2048
ARGB	Argument B for modular blocks	2048
ARGC	Argument C for modular blocks	2048
RAND0	Random number 0 for secure mux	1024
RAND1	Random number 1 for secure mux	1024
MUL_RES	Result register for multiplication	2048
EXP_RES	Result register for exponentiation	2048
INV_RES	Result register for inversion	2048
DBG_REG	Debug register	2048

Table II shows a representative subset of the 39 Configuration Registers in CoPHEE. Our configuration registers map to the 0x4002_0000 – 0x4002_FFFF memory range, while the GPIO maps to the 0x4003_0000 – 0x4003_FFFF range. In our design, the memory base address follows the ARM Cortex M series memory map convention for peripheral addresses.

C. Design Blocks

CoPHEE implements a set of modular arithmetic acceleration blocks, namely: (1) an *interleaved* modular multiplication unit, (2) a modular exponentiation unit based on Montgomery multiplication [19], and (3) a modular inversion unit based on the binary extended GCD algorithm, which outputs the GCD of its inputs along with the modular inverse of its first input using the second input as the modulus. The design details of these blocks, as well as the design of our secure multiplexer and true random number generator (TRNG) blocks are presented in this section.

1) Modular Multiplication: Our chip implements an interleaved modular multiplier for integers up to 2048 bits [19]. This multiplier is more efficient compared to other complicated algorithms (such as Montgomery multiplication), when the two inputs are multiplied *only once*. Conversely, transforming the inputs to a different multiplication domain

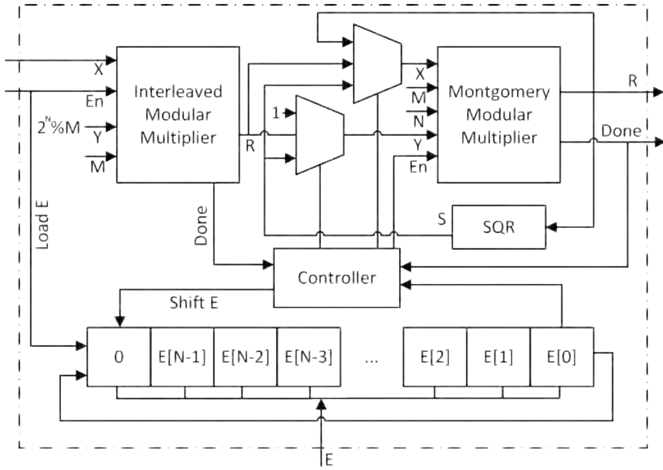

Fig. 2: Modular exponentiation block

(e.g., the Montgomery domain) offers better amortized performance only when multiple multiplications are cascaded (e.g., as in modular exponentiations).

2) Modular Exponentiation: For modular exponentiation, CoPHEE implements a Montgomery multiplier that offers significant advantages in terms of modular reductions [19]. Contrary to the modular reductions in interleaved multipliers that are implemented using subtractions, the Montgomery multiplication requires merely a bitwise right-shift operation, which is more efficient. Nevertheless, the Montgomery algorithm requires that its operands undergo an additional transformation to the Montgomery domain before multiplications can be applied. Specifically, if N is the bit width of an odd modulus M, each operand should be multiplied with 2^N (i.e., bitwise shift-left) and reduced modulo M. Likewise, the final result is transformed to an ordinary integer using multiplication with the inverse of 2^N modulo M. Due to these transformations, the Montgomery algorithm is beneficial only when several multiplications are performed within the Montgomery domain, which is the case of modular exponentiation.

Our modular exponentiation operates in three stages and the corresponding CoPHEE block is illustrated in Fig. 2. The first stage transforms the exponentiation base X to the Montgomery domain (i.e., multiplication with $2^N \bmod M$), while the second stage uses the bits of exponent E to perform *right to left* binary exponentiation by squaring [19]. Our chip implements a controller that determines which operands enter our Montgomery multiplier by scanning the bits of E (from LSB to MSB). Specifically, the controller executes repeated squaring (in the Montgomery domain) of base X for each bit of E, while if the exponent bit equals 1, the corresponding square is also multiplied with X. The third stage transforms the result from the Montgomery domain to an ordinary integer by triggering a final Montgomery multiplication with 1 (i.e., a multiplication with the inverse of 2^N modulo M).

3) Modular Inversion: The modular inverse of an integer X over a modulus M exists when $GCD(X, M)$ equals 1, and it can be efficiently computed using the binary extended GCD algorithm (Algorithm 1) [19]. Specifically, on input X and M, the binary extended GCD computes values A, B and $G = GCD(X, M)$ that satisfy the equation $A \cdot X + B \cdot M = G$ (3) (i.e., Bézout's identity); if G equals 1 then $A = X^{-1} \bmod M$. CoPHEE implements a modular inversion block that receives inputs X, M from the host processor and starts computing $X^{-1} \bmod M$ as soon as its En input is toggled.

Internally, our modular inversion block creates two instances of Equation 3: $X_g = A_x \cdot X + B_x \cdot M$ (4), and $Y_g = A_y \cdot X + B_y \cdot M$ (5), and initializes the variables $\{A_x, B_x, A_y, B_y\}$ to $\{1, 0, 0, 1\}$ so that the initial values of $\{X_g, Y_g\}$ are $\{X, M\}$ (line 3 in Algorithm 1). Then, our modular inversion block iterates the following three steps:

1) While both X_g and Y_g are even, divide them by 2 and multiply the GCD G by 2.
2) If only one of them is divisible by 2, their corresponding equation is divided by 2. In this case, if any of the coefficients in the equation is not divisible by 2, add $X \cdot Y - X \cdot Y$ to the right side of the equation and restructure it in the form $A \cdot X + B \cdot Y$ (lines 14 and 22 of Algorithm 1). Then, both coefficients become divisible by 2.
3) When both X_g and Y_g are not divisible by 2, check if X_g is greater than Y_g. If so, the Equation 4 is updated by subtracting itself from the Equation 5. Otherwise, the Equation 5 is updated by subtracting itself from the Equation 4.

As soon as X_g equals Y_g, the algorithm terminates and the $Done$ output is set to high for one clock cycle. If the output G (i.e., the GCD of $\{X, M\}$) is equal to 1, then the output INV contains the desired value $X^{-1} \bmod M$.

4) True Random Number Generation: The TRNG design in CoPHEE is based on a bi-stable circuit [20], and as illustrated in Fig. 3c, we employ 16 individual TRNG blocks spread across our chip. This TRNG design improves randomness by exploiting the inherent process variations of the fabricated chip, and our random number stream is generated by XORing the outputs of all 16 TRNG blocks [20]. Moreover, we remove any potential 0/1 bias ("deskewing") by post-processing the random number stream using a *von Neumann extractor*. Since we desire our random numbers to be co-prime with the (odd) public modulus, we fix the LSB of each random number to 1; if the GCD of the random number and the modulus is not 1, the random number is incremented by 2 until the GCD becomes 1. To calculate a new random number, an explicit request to the TRNG block is required.

5) Secure Multiplexer: The secure multiplexer of CoPHEE adopts Cryptoleq's blueprint and implements a state machine that employs all the aforementioned design blocks, as illustrated in Algorithm 2. Specifically, our secure multiplexer receives: (a) two encrypted inputs X, Y, (b) a function of the private (decryption) key FKF, and (c) two random numbers $RAND0$ and $RAND1$. Our state machine computes the modular exponentiation X^{FKF} (line 6) and checks the sign of the result (line 8); if this sign is less than or equal to zero, our secure multiplexer outputs an encryption of the value 0, otherwise it outputs a random re-encryption of encrypted

Algorithm 1: Binary Extended GCD

```
1  INPUT: X[N-1 :0], M[N-1 :0], En, Clk;
2  OUTPUT: G[N-1 :0], INV[N-1 :0] Done;
3  G = Not(X[0] | Y[0]); X_g = X; Y_g = M; A_x = 1; B_x = 0; A_y = 0; B_y = 1;
4  if En == 1 @ Positive edge of Clk then
5      while X_g[0] == Y_g[0] == 0 @ Positive edge of Clk do
6          X_g = X >> 1; Y_g = Y >> 1; G = 2 * G;
7      end
8      while X_g != Y_g do
9          while X_g[0] == 0 @ Positive edge of Clk do
10             X_g = X_g >> 1;
11             if A_x[0] == B_x[0] == 0 then
12                 A_x = A_x >> 1; B_x = B_x >> 1;
13             else
14                 A_x = (A_x + Y) >> 1;
                   B_x = (B_x - X) >> 1;
15             end
16         end
17         while Y_g[0] == 0 @ Positive edge of Clk do
18             Y_g = Y_g >> 1;
19             if A_y[0] == B_y[0] == 0 then
20                 A_y = A_y >> 1; B_y = B_y >> 1;
21             else
22                 A_y = (A_y + Y) >> 1;
                   B_y = (B_y - X) >> 1;
23             end
24         end
25         if X_g > Y_g then
26             X_g = X_g - Y_g; A_x = A_x - A_y;
               B_x = B_x - B_y;
27         else
28             Y_g = Y_g - X_g; A_y = A_y - A_x; B_y = B_y - B_x;
29         end
30         G = X_g << G; INV = A_x; Done = 1;
31     end
32 end
```

Algorithm 2: Secure Multiplexer

```
1  INPUT: X[N-1 :0], Y[N-1 :0], FKF[N-1 :0], RAND0[N-1 :0], RAND1[N-1 :0], M[N-1 :0] En, Clk;
2  OUTPUT: R[N-1 :0];
3  if En == 1 @ Positive edge of Clk then
4      RAND0 = Modular Exponentiation of RAND0 with exponent √M;
5      RAND1 = Modular Exponentiation of RAND1 with exponent √M;
6      D = Modular Exponentiation of X with exponent FKF;
7  end
8  if D <= 0 then
9      R = Modular multiplication of RAND0 and RAND1 ;
10 else
11     R = Modular multiplication of RAND0 and Y;
12 end
```

input Y. We remark that these outputs are computationally indistinguishable.

In each invocation of our secure multiplexer, our TRNG generates two fresh random numbers to ensure that consecutive invocations cannot repeat the same encrypted outputs. The latter prevents side-channel attacks that infer ciphertext information by comparing the inputs and outputs. Nevertheless, since generating new random numbers using a TRNG can incur high overheads, it is also possible to generate one truly random (seed) number and then compute a sequence of additional pseudorandom values using modular squaring. We remark that modular squaring of a (random) number is essentially Rabin's one-way function, since 'factoring' and 'computing square roots' have equivalent computational complexity when the prime decomposition of the modulus is unknown [21].

6) Auxiliary Communication & Control Blocks: To support communication and control, CoPHEE incorporates the following auxiliary blocks: (1) UART master (used to interface with the external host computer), (2) configuration registers unit (used to store the operands, modulus and results), (3) GPIO (used to assist debugging during post-silicon validation), and (4) AHB bus interconnect (used to transfer data inside the chip). In our chip, the UART is the master, while the configuration registers unit and GPIOs are the slaves. Moreover, our configuration registers unit consists of 39 registers, with their size varying from 32-bit to 2048-bit. All registers are byte-addressable. Some registers are readable and writable (e.g., operands, modulus), some are readable-only (e.g., result), while others are writable-only (e.g., operation trigger).

D. Pre-Silicon Verification

We verified the functionality of our RTL design using both simulation and FPGA-based validation. The simulation was performed using Synopsys VCS at the top-level and block-level using random inputs (since the 2048-bit operand range cannot be exhaustively tested). Moreover, for our FPGA design, we implemented a scaled-down version of CoPHEE as the 2048-bit data width of the original design was incompatible with the available resources of our FPGAs. Specifically, the maximum data width that could be loaded on a Digilent Nexys 4 was 256 bits (running at 25 MHz), while a 512-bit version of the co-processor exceeds the capacity of the significantly larger Kintex-7 and Virtex-5 FPGA boards.

E. Synthesis

The CoPHEE RTL code is synthesized using a 65nm standard cell library from Global Foundries and a clock constraint of 100 Mhz. As the UART and GPIOs are the only interfaces of CoPHEE (i.e., both are asynchronous), there is no specific IO timing constraint. Following common practices, the standard cell library used for synthesis was the one characterized for the worst voltage (1.08V), temperature (125C), resistance, and capacitance. Synthesizing with such a library ensures that we can achieve the target frequency.

For synthesis, we used the Synopsys Design Compiler (DC), while for post-synthesis and formal verification we used the Cadence Conformal, and we were able to ensure that the RTL code and the synthesized netlist are functionally equivalent. In Table III we presents the area and timing estimations of the major CoPHEE blocks after synthesis. The largest design is the binary extended GCD, followed by the configuration registers that store a total of 2.73 KB. As expected, the modular multiplier and the modular exponentiation unit are

TABLE III: Post-synthesis area and timing estimations

Blocks	Area (um^2)	Worst path delay (ns)
Configuration registers	512,724	4.770
Binary extended GCD	548,454	9.433
Interleaved mod. multiplier	315,487	9.374
Modular exponentiation	304,064	9.389
AHB bus	1,261	4.850
UART master	3,466	4.930
TRNG	28,658	NA

TABLE IV: Layout physical parameters

Parameter	Value
IU (Initial Utilization)	36 %
FU (Final Utilization)	47 %
MA (Macro Area)	0 um^2
HIO (IO PAD Height)	120 u
CIO (Core to IO spacing)	110 u
A (Aspect ratio)	1
RCA (Required std cell Area)	1956692.52 um^2
CW (Core Width)	2340 u
CH (Core Height)	2340 u
DW (Die Width)	2800 u
DH (Die Height)	2800 u

roughly the same size, and the rest of the modules occupy significantly less area.

IV. Physical Design

The CoPHEE chip was fabricated using the Multi Project Wafer (MPW) program of MOSIS, and the 5_02_00_00_LB metal layer stack; specifically, the metal and via layers are M1, V1, M2, V2, M3, V3, M4, V4, M5, WT, BA, WA, BB, VV, and LB. These layers have preferred routing direction (horizontal or vertical), and this stack offers enough signal routing resources in first 5 metal layers, so the top two are used mostly for power/ground and clock structure. The layer LB was only used for the IO pads, and the external bonding to the chip during packaging was done using these LB cups on the pads. The GF-PDK for the 65LPE process provided us all the technology-related files, and we used the physical and timing libraries for standard cells from ARM, while ARAGIO provided the IO Pad libraries. Our chip was implemented flat, without any physical hierarchy.

A. Place and Route

1) Die size estimation: In Table IV we show the physical parameters with respect to our layout after multiple iterations. The minimum chip size supported by Global Foundries for 65LPE through MOSIS is $9mm^2$. Since this is more than the estimated die size of our chip, we utilize the entire area.

2) Floor Planning: Our layout outline along with IO pad placement adheres to the IO pad placement guidelines from Global Foundries. We have a total of 27 IO pads, where 8 of them are for VDD/VSS core power/ground supply, 8 DVDD/DVSS are for IO power/ground, while the remaining 11 are signal pads. One supply and one ground pad would be sufficient, but we utilize the empty spaces in the IO pad ring to improve the power structure robustness. The empty spaces between the pads are filled with filler pads to maintain continuity of the internal power/ground and other special signals, while ENDCAPs and well tie cells were also distributed in the core region as per the foundry requirements. Finally, the quality of the input netlist we checked using Zero Interconnect Delay analysis.

3) Power Planning: We created a core power ring illustrated with the red and white lines around the core in Fig. 3a, which are located in metal layers BA (red vertical) and BB (white horizontal). The thick white and red lines connect the ring to the pads, and we have power straps created in layers BA, BB, M5 and M4. The mesh structure inside the core region in Fig. 3a shows the power straps distribution: our PG rails are in M1 and are connected to the power straps in M4 through power via stack from M4. We remark that the M4 strap runs vertically (preferred routing direction is vertical) and vias can be dropped on every intersection with horizontal M1 rails.

4) Placement And Optimization: Prior to the standard cell placement, we grouped and distributed our TRNG modules using bounds/regions in the chip to leverage on-chip variation (bound is a feature used to restrict placement of specific cells according to the user specification). We note that the standard cells in the TRNG modules were not allowed to be optimized. In Fig. 3c we illustrate the TRNG module distribution, where the red highlighted standard cell groups compose our TRNG modules. After fixing these groups in position, the rest of the standard cells went through placement and optimization. Our design was then analyzed for timing, congestion, area and power, passing all requirements. We also enabled high threshold voltage (HVT) and regular threshold voltage cells (RVT) during optimization: HVT are low-leakage high-delay cells, while RVT are medium-leakage medium-delay cells. We remark that LVT (low threshold) cells, which are high-leakage low-delay cells, were used only in the final timing closure. This approach allows us to limit the power leakage in accordance to standard practices.

In Fig. 3b we illustrate the distribution of the main modules in our design after placement. Notably, the GCD module consumes the largest portion of the design and is confined to the right side of the chip, while the modular multiplication and exponentiation units are the second and third largest. We also performed a trial placement run by creating specific regions in the chip for these modules, where each region restricted the placement of a module to a specified location of the chip. Nevertheless, since the timing results did not show any improvement compared to the placement without regions, this trial was discarded.

5) Clock Tree Synthesis: For the implementation of our chip, we performed Clock-Tree Synthesis (CTS) using the BUFH_X4M_A9TR, BUFH_X5M_A9TR, BUFH_X6M_A9TR, and BUFH_X9M_A9TR buffers, as well as the INV_X4M_A9TR, INV_X5M_A9TR, INV_X6M_A9TR, and INV_X9M_A9TR inverters from the ARM standard cell library. The middle part of the name indicates the driving strength of the cell (e.g., X9M). This list of buffers comprises

TABLE V: Post-CTS statistics

Parameter	Value
clock name	HCLK
CTS synthesis corner	slow
Number of levels	45
Number of Sinks	67628
Number of clock tree buffers	9921
Global Skew	162 ps
Longest Insertion delay	2.410 ns
Shortest Insertion delay	2.248 ns
Standard cell utilization	43.63 %

TABLE VI: Design statistics through PnR

Parameter	Initial	Place	CTS	Route
# of Standard cells	693333	775548	784795	786526
# of Sequential cells	67628	67628	67628	67628
# of Combinational cells	625708	707923	717170	718901
# of Buffer/Inverter cells	91139	707923	175163	176894
Standard Cell Utilization	35.45 %	43.44 %	46.35 %	46.56 %
# of Signal nets	695425	777637	771789	773520
HVT cells	100 %	55.8 %	67.7 %	67.8 %
RVT cells	0 %	44.2 %	32.3 %	32.2 %
Total wire length (μm)	NA	NA	1242079	46668040

RVT cells of medium driving strength that allow reduced On Chip Variation (OCV), a robust clock network, and less power consumption.

A Non-Default Rule (NDR) of double width and double spacing was also created and assigned to the clock trunk nets (all the clock nets are trunk nets, except for those connected to sinks directly). Our clock nets were routed using metal layers M4, M5, BA, and BB, and assigned as *soft fixed*, which ensures that the clock network remains intact during signal routing (i.e., changes are restricted). In addition, we applied a multi-corner optimization to fix design rule violations, as well as setup and hold timing closure, and achieved a skew of 162 ps with nominal count of clock buffers/inverters. Our post-CTS statistics are presented in Table V.

6) Signal Routing and Optimization: In Table VII we present the percentage of redundant vias for various via layers. We were able to achieve more than 75% conversion of single to multi-cuts for the lower via layers V1, V2, V3, V4, yet a lower percentage was achieved for higher layers. Likewise, in Table VI we present design statistics over various stages in the Place & Route (PnR). We remark that the standard cell count increases as the design moves from initial to final routing stages, primarily due to the buffers/inverters inserted in the design to fix design rule violations, clock tree synthesis and timing issues. Our design started with 100% HVT cells and ended up with 67.8%, as HVT cells were swapped with RVT cells to address timing and DRV fixes. The total wire length under 'CTS' corresponds to the clock net only, while under 'Route' corresponds to all nets (Table VI).

B. Sign-Off Analysis

1) Static Timing Analysis: For our timing analysis, we used an uncertainty of 200ps for setup, and 50ps for hold, as recommended by Global Foundries. Our design was further

TABLE VII: Redundant via statistics

Layer	# of multi-cut vias	# of total vias	% of multi-cut vias
V1	2003289	2579845	77.65
V2	2129266	246275	86.46
V3	688693	869281	79.23
V4	417410	544818	76.61
WT	76473	134395	56.90
WA	59085	100430	58.83

analyzed for design rule violations, such as the maximum transition, capacitance, and fanout. The few violations we identified after the initial Static timing analysis (STA) were fixed using Engineering Change Order (ECO).

2) Physical Verification: In this step, we verify the final layout against foundry manufacturing rules and insert dummy metal/poly fills using the Calibre software to meet the foundry's density requirements. The fill GDS obtained from Calibre is merged with the design GDS in our layout tool (Virtuoso), generating a final design ready for Design Rule Check (DRC) and Layout Versus Schematic (LVS) analysis. We used the Cadence PVS tool to run DRC, LVS, and Antenna checks, and fixed any violations before re-running the checks to ensure a clean GDS for our tapeout. Fig. 4a shows the design layout from Virtuoso after the "streamIn" process (i.e., merging of the primitive GDS with the design GDS) and the addition of a "seal ring" as per foundry guidelines (shown as thin lines running around the chip boundary with diagonal routing around corners).

3) Rail Analysis: For our rail analysis, we used Synopsys primeRail and imported the Milkyway database and signal parasitics. The analysis did not raise any static or dynamic violations, which is attributed to the robust power structure of CoPHEE; the worst static and dynamic drop was 10.6 mV and 24 mV, respectively. In Fig. 4b we present the static effective rail voltage drop, where most of our power pads are located at the left of the chip, so the highest static drop appears on the right side (red color). Likewise, in Fig. 4c we show the dynamic rail voltage drop, which appears specific to a limited number of gates (red spots).

C. Post-Silicon Validation

CoPHEE was packaged in a 28 pin DIP, and was connected to a breadboard for silicon bring up and testing. For interfacing with a host computer, we used a UMFT230XA development board that features an FTDI chip for USB-to-UART conversion. The UMFT230XA board can provide a 3.3V supply for the IO pad of CoPHEE, as well as a clock output (used as the clock source of the chip). Moreover, the required 1.2V supply was generated using a DC-DC adjustable step-down module that converts the 5V source of the UMFT230XA board. In addition, an Arduino was responsible for receiving interrupt signals from CoPHEE and for transmitting these events to the host computer. Our post-silicon validation setup is shown in Fig. 5.

Our post-silicon validation confirmed that the fabricated chip is fully functional, with a discovered bug in the "debug

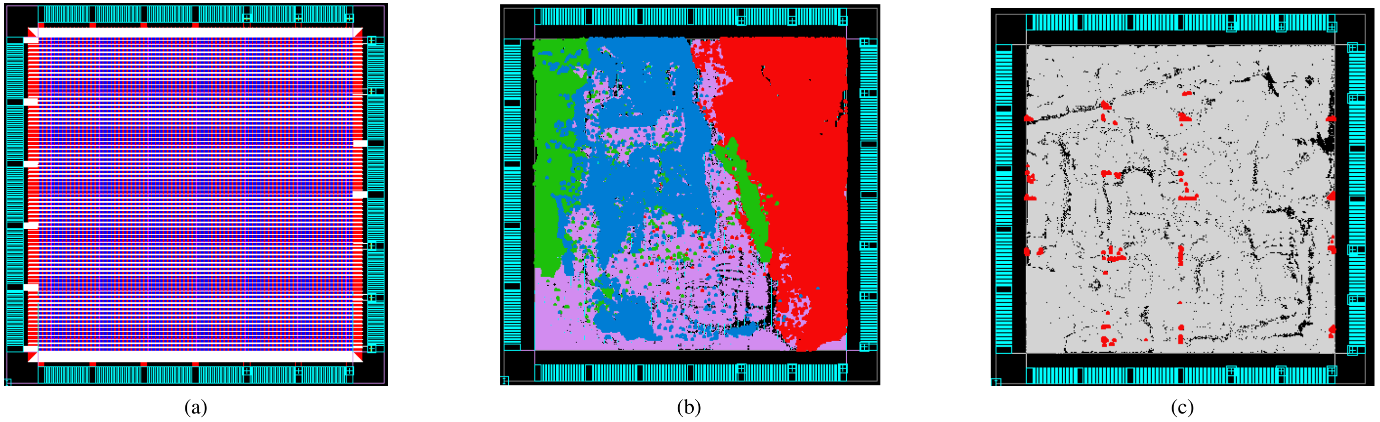

Fig. 3: Place & Route output: (a) Power network; (b) Placement distribution (binary extended GCD: red; interleaved modular multiplier: green; modular exponentiation: blue; other modules: purple); (c) TRNG distribution (red).

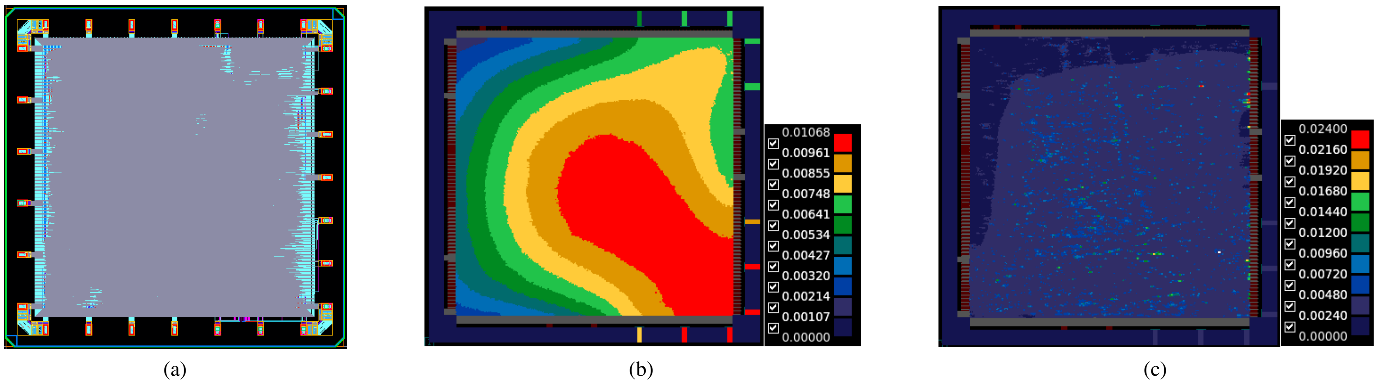

Fig. 4: SignOff results: (a) Chip GDS view from Virtuoso; (b) Chip static rail drop; (c) Chip dynamic rail drop.

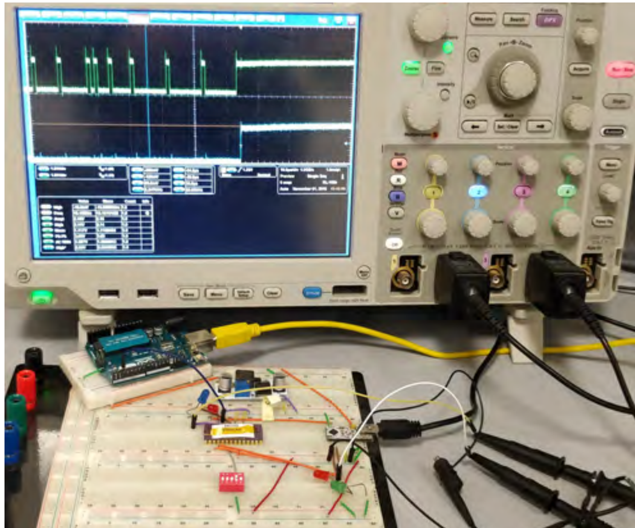

Fig. 5: Photo of CoPHEE experimental setup.

read" path that reads random numbers and tests their randomness. Specifically, there was a hard-coded bit-width value in the read path of the configuration registers, which prevents us from reading the debug register (last register in the path). Interestingly, this bug escaped our FPGA-based validation, as the latter was performed on the scaled-down version where the 256 bit-width was incorrectly hard-coded.

V. Experimental Results

In this section, we quantify the performance of CoPHEE by executing C++ benchmarks and compare it against emulation of PHE operations on a general-purpose x86 architecture. We selected six data-oblivious benchmarks from the TERMinator suite [22] that are classified into two categories: (1) kernel benchmarks, which stress arithmetic and logical operations, such as Bubble Sort (BS), Matrix Multiplication (MM), Insertion Sort (IS) and Sieve of Eratosthenes (SoE), and (2) microbenchmarks, such as the multiplicative-intensive Factorial (FAC) and the addition-intensive Fibonacci (FIB). In our experiments, we used a 2.7 GHz Intel i7-7500U with 8GB RAM, running Ubuntu 16.04.5 LTS with GCC 8.1.0 and GMP 6.1.1, while our CoPHEE chip was running at 100 MHz.

Fig. 6 summarizes the performance benefits of CoPHEE using the aforementioned benchmarks, normalized to the execution time of the benchmarks running on an x86 processor using GMP for emulating the homomorphic operations. In particular, we employ CoPHEE on two different scenarios: (1) as a hardware root-of-trust, where only the secure multiplexer is utilized, and (2) as a cryptographic accelerator, where

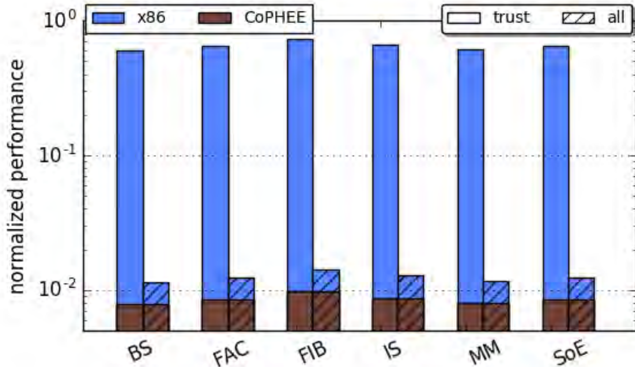

Fig. 6: CoPHEE performance acceleration when it is used: (a) as a hardware root-of-trust only ('trust'), and (b) both as an accelerator and root-of-trust ('all'). Performance is normalized to the x86-only execution.

all PHE operations are offloaded to the co-processor. Using CoPHEE as a root-of-trust only, the load on the x86 host is reduced by the speed-up offered by the hardware encrypted multiplexer. Since the secure multiplexer requires computationally intensive modular exponentiations, offloading it to an ASIC accelerator significantly improves its performance. In this case, the execution time on the x86 host is attributed to the large number of modular multiplications and inverses executed in software using the GMP library.

Conversely, when CoPHEE acts as accelerator for all homomorphic operations (label 'all' in Fig. 6), we observe an average of two orders of magnitude improvement in execution time compared to x86 using GMP. In this scenario, the main CPU only executes setting-up operations, such as creating objects, moving data, processing integers, etc. The two orders of magnitude improvement is obtained by the first prototype of CoPHEE running at 100MHz. Extrapolating from this result, we expect one more order of magnitude improvement, assuming that the next iteration of the co-processor operates in the GHz range.

Fast communication between the main CPU and CoPHEE is essential in order to unlock the power of CoPHEE. In the first fabricated prototype, UART requires 1.68E-03 seconds to complete a transaction between the main CPU and CoPHEE, assuming the transfer of three 2048-bit operands (two source operands, one result) and the transfer of the 32 trigger bits. A faster off-chip communication protocol, such as InterChip USB, would accelerate communication by approximately two orders of magnitude (≈1.29E-05 seconds per operation, assuming 480 Mbits/s transfer rates). Apparently, the fastest way to move data between the main CPU/memory and CoPHEE would be a system-on-chip approach where CoPHEE is also located on the same bus. Assuming a 32-bit ARM architecture, on-chip communication on AHB-Lite would further accelerate communication by 3-4 orders of magnitude (≈9.65E-08 seconds per operation). Thus, the selection of the communication protocol and potential integration of CoPHEE depends on the requirements of the deployed application.

VI. Related Work

Following Gentry's discovery of FHE [6] and its time-intensive CPU-based implementation [23], the research community focused on hardware approaches to speed up the homomorphic computation. Wang et al. [24] proposed accelerating the million-bit modular multiplications required in Gentry's cryptosystem, using Strassen's Fast Fourier Transform (FFT) followed by a Barrett modular reduction, both implemented on an NVIDIA C2050 GPU. Experimental results show a performance improvement of about $7\times$ compared to existing CPU-based implementations.

Recently, several FPGA-based accelerators for FHE operations have also been proposed. Cousins et al. [25] use a Xilinx Virtex-7 to implement a co-processor targeting lattice-based homomorphic encryption schemes. Their co-processor accelerates bottleneck operations such as the forward and inverse Chinese Remainder Transform (CRT), as well as more efficient operations such as ring addition, subtraction and multiplication. This approach achieves 2 orders of magnitude speedup compared to a reference CPU-based implementation of the López-Alt, Tromer and Vaikuntanathan (LTV) cryptosystem [26], where either PCI-E or Ethernet was used to communicate with a host computer. Likewise, Ozturk et al. [27] propose an LVT-based accelerator for homomorphic evaluations by computing high degree polynomial multiplications (2^{14} and 2^{15} degree polynomials) using a Virtex-7 FPGA. In their approach, the host CPU uses PCI-E to communicate with the accelerator, and their multiplier is $102\times$ faster compared to a CPU-based implementation, which corresponds to a $28.5\times$ homomorphic evaluation speedup for the AES block cipher. Moreover, recent work was focused on FFT acceleration for cryptosystems based on Ring Learning With Errors [28]–[30], while Migliore et al. [31] investigate the benefits of the Karatsuba algorithm compared to FFT for FHE operations in the Fan-Vercauteren scheme, and report 23% performance increase for one half logic utilization in the FPGA.

Encrypted processors using PHE schemes have also been proposed: HEROIC [32] implements a single instruction computer with support for both subtraction-oriented and addition-oriented encrypted computation, using deterministic encryption and without any shared keys. In HEROIC, addition uses the homomorphic properties of the Paillier, subtraction uses the modular inverse of the operand, while branching employs a lookup table as a sign oracle; the processor has been implemented in a VM and on a Kintex-7 FPGA. Similarly, CryptoBlaze [33] is a multi-instruction CPU that is based on MicroBlaze and implements the Paillier cryptosystem to operate on encrypted data. This processor implements eight instructions operating on encrypted data, including *eadd* and *esub* for homomorphic addition and subtraction, as well as *ebrzpos* and *ebrneg* for branching on a non-negative number or a negative number, respectively. Branching is not executed by CryptoBlaze, however, and a request is sent to a host computer to interactively compute the outcome. The CryptoBlaze CPU is implemented on a Virtex-6 FPGA communicating with a

host processor through a 32-bit AXI bus, and offers a $10\times$ speedup compared to HEROIC. Nevertheless, since branch decisions are offloaded to the host, the expensive modular exponentiations used to decrypt control ciphertexts do not benefit from CryptoBlaze's acceleration, and since all decision outcomes are received in the clear through the AXI bus, side-channel information about ciphertexts can be leaked outside the root-of-trust.

VII. Conclusions and Future Work

In this work we present a year-long effort to design, implement, fabricate, and validate `CoPHEE`: a general-purpose co-processor that computes on partially homomorphic encrypted data. The arithmetic units for modular multiplication, exponentiation, inversion, and GCD that have been implemented in this work accelerate the computation of very wide datapaths, while our secure multiplexer and true random number generator enable universal computation in the encrypted domain. To the best of our knowledge, `CoPHEE` is the first academic effort towards constructing a fast and reliable processor capable of processing encrypted data. This paper presents all required steps for a fully functional silicon, from the RTL design to fabrication and validation. Given the silicon, future work will explore side-channel analysis and information extraction through power, timing, and electromagnetic emissions.

Resources

RTL files and a form to request `CoPHEE` (in a 80-pin package) are available at https://github.com/momalab/CoPHEE.

References

[1] C. Barron, H. Yu, and J. Zhan, "Cloud computing security case studies and research," in *World Congress on Engineering*, 2013, pp. 1287–1291.

[2] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *Computer and Communications Security (CCS)*, 2012, pp. 305–316.

[3] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor, "Hardware trojans: Lessons learned after one decade of research," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 22, no. 1, p. 6, 2016.

[4] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR," in *Computer and Communications Security (CCS)*. ACM, 2016, pp. 368–379.

[5] N. G. Tsoutsos and M. Maniatakos, "Fabrication attacks: Zero-overhead malicious modifications enabling modern microprocessor privilege escalation," *IEEE Transactions on Emerging Topics in Computing*, vol. 2, no. 1, pp. 81–93, 2014.

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

[7] M. Naehrig, K. Lauter, and V. Vaikuntanathan, "Can homomorphic encryption be practical?" in *Cloud Computing Security Workshop*. ACM, 2011, pp. 113–124.

[8] M. Varia, S. Yakoubov, and Y. Yang, "HEtest: A Homomorphic Encryption Testing Framework," in *Financial Cryptography and Data Security*. Springer, 2015, pp. 213–230.

[9] Y. Doröz, E. Öztürk, and B. Sunar, "A million-bit multiplier architecture for fully homomorphic encryption," *Microprocessors and Microsystems*, vol. 38, no. 8, pp. 766–775, 2014.

[10] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *Innovations in Theoretical Computer Science Conference*, 2012, pp. 309–325.

[11] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *Asiacrypt*, 2016, cryptology ePrint Archive, Report 2016/870 https://eprint.iacr.org/2016/870.

[12] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in cryptology–EUROCRYPT'99*. Springer, 1999, pp. 223–238.

[13] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Foundations of secure computation*, vol. 4, no. 11, pp. 169–180, 1978.

[14] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[15] R. Rojas, "Conditional branching is not necessary for universal computation in von Neumann computers," *Journal of Universal Computer Science*, vol. 2, no. 11, pp. 756–768, 1996.

[16] O. Mazonka, N. G. Tsoutsos, and M. Maniatakos, "Cryptoleq: A heterogeneous abstract machine for encrypted and unencrypted computation," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 9, pp. 2123–2138, 2016.

[17] O. Mazonka and A. Kolodin, "A simple multi-processor computer based on subleq," *arXiv preprint arXiv:1106.2593*, 2011.

[18] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im) possibility of obfuscating programs," in *Advances in Cryptology–CRYPTO 2001*, 2001, pp. 1–18.

[19] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC press, 1996.

[20] M. Epstein, L. Hars, R. Krasinski, M. Rosner, and H. Zheng, "Design and implementation of a true random number generator based on digital circuit artifacts," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2003, pp. 152–165.

[21] M. O. Rabin, "Digitalized signatures and public-key functions as intractable as factorization," MIT Laboratory of Computer Science, Tech. Rep. MIT/LCS/TR-212, 1979.

[22] 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.

[23] C. Gentry and S. Halevi, "Implementing gentry's fully-homomorphic encryption scheme," in *EUROCRYPT*, 2010, pp. 129–148.

[24] W. Wang, Y. Hu, and L. Chen, "Accelerating fully homomorphic encryption using GPU," in *High Performance Extreme Computing (HPEC)*, 2012, pp. 1–5.

[25] D. B. Cousins, K. Rohloff, and D. Sumorok, "Designing an FPGA-accelerated homomorphic encryption co-processor," *IEEE Transactions on Emerging Topics in Computing*, vol. 5, no. 2, pp. 193–206, 2016.

[26] A. Lopez-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," Cryptology ePrint Archive, Report 2013/094, 2013, https://eprint.iacr.org/2013/094.

[27] E. Öztürk, Y. Doröz, E. Savaş, and B. Sunar, "A custom accelerator for homomorphic encryption applications," *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 3–16, 2016.

[28] S. S. Roy, K. Järvinen, F. Vercauteren, V. Dimitrov, and I. Verbauwhede, "Modular hardware architecture for somewhat homomorphic function evaluation," in *Cryptographic Hardware and Embedded Systems CHES*, vol. 9293, 2015, pp. 164–184.

[29] A. Cilardo and D. Argenziano, "Securing the cloud with reconfigurable computing: An fpga accelerator for homomorphic encryption," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016.

[30] A. Mkhinini, P. Maistri, R. Leveugle, and R. Tourki, "Hls design of a hardware accelerator for homomorphic encryption," in *Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, vol. 9293. IEEE, 2017.

[31] V. Migliore, M. M. Real, V. Lapotre, A. Tisserand, C. Fontaine, and G. Gogniat, "Hardware/software co-design of an accelerator for fv homomorphic encryption scheme using karatsuba algorithm," *IEEE Transactions on Computers*, vol. 67, no. 3, pp. 335–347, 2016.

[32] N. G. Tsoutsos and M. Maniatakos, "The HEROIC framework: Encrypted computation without shared keys," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 6, pp. 875–888, 2015.

[33] F. Irena, D. Murphy, and S. Parameswaran, "CryptoBlaze: A partially homomorphic processor with multiple instructions and non-deterministic encryption support," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2018.