Simulation-Based Engineering Lab
University of Wisconsin-Madison

Technical Report TR-2020-10

# CryptoEmu: An Instruction Set Emulator for Computation Over Ciphers

Xiaoyang Gong and Dan Negrut

December 28, 2020

## Abstract

Fully homomorphic encryption (FHE) allows computations over encrypted data. This technique makes privacy-preserving cloud computing a reality. Users can send their encrypted sensitive data to a cloud server, get encrypted results returned and decrypt them, without worrying about data breaches.

This project report presents a homomorphic instruction set emulator, CryptoEmu, that enables fully homomorphic computation over encrypted data. The software-based instruction set emulator is built upon an open-source, state-of-the-art homomorphic encryption library that supports gate-level homomorphic evaluation. The instruction set architecture supports multiple instructions that belong to the subset of ARMv8 instruction set architecture. The instruction set emulator utilizes parallel computing techniques to emulate every functional unit for minimum latency. This project report includes details on design considerations, instruction set emulator architecture, and datapath and control unit implementation. We evaluated and demonstrated the instruction set emulator's performance and scalability on a 48-core workstation. CryptoEmu shown a significant speed up in homomorphic computation performance when compared with HELib, a state-of-the-art homomorphic encryption library.

**Keywords**: Fully Homomorphic Encryption, Parallel Computing, Homomorphic Instruction Set, Homomorphic Processor, Computer Architecture

# Contents

# 1 Introduction

In the conventional cloud service model, users share data with their service provide (cloud) to outsource computations. The cloud receives encrypted data and decrypts it with the cloud's private key or the private key shared between the user and the cloud. Thus, the service provider has access to user data, which might contain sensitive information like health records, bank statements, or trade secrets. Privacy concerns have been raised along with the wide adoption of cloud services. In 2019, over 164.68 million sensitive records were exposed in the United States [1].

In the worst-case scenario, the cloud service provider cannot be trusted. User data is inherently unsafe if it is in plain text. Even if the service provider is honest, cloud service is prone to fail victims of cybercrime. Security loopholes or sophisticated social engineering attacks expose user privacy on the cloud, and a successful attack usually results in a massive user data leak. One way to eliminate this type of risk is to allow the cloud to operate on the encrypted user data without decrypting it. Fully Homomorphic Encryption (FHE) is a special encryption scheme that allows arbitrary computation over encrypted data without knowing the private key. An FHE enabled cloud service model shown in Fig. 1. In this example, the user wants to compute the sum of 1, 3, 5 in the cloud. The user first encrypts data with FHE, then sends the cipher (shown in Fig. 1 as bubbles with blurry text) to the cloud. When the cloud receives encrypted data, it homomorphically adds all encrypted data together to form an encrypted sum and returns the encrypted sum to the user. The user decrypts the encrypted sum with a secret key, and the result in cleartext is 9 – the sum of 1, 3, and 5. In the entire process, the cloud has no knowledge of user data input and output. Therefore, user data is safe from the insecure cloud or any attack targeted at the cloud service provider.

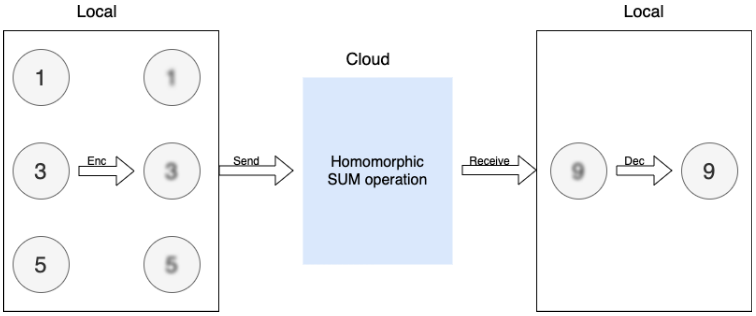


Figure 1: Homomorphic Encryption
Blurry text in the figure denotes encrypted data.

Over the years, the research community has developed various encryption schemes that enable computation over ciphers. TFHE [2] is an open-source FHE library that allows fully homomorphic evaluation on arbitrary Boolean circuits. TFHE library supports FHE operations on unlimited numbers of logic gates. Using FHE logic gates provided by TFHE, users can build an application-specific FHE circuit to perform arbitrary computations over encrypted data. While TFHE library has a good performance in gate-by-gate FHE evaluation speed and memory usage [3], a rigid logic circuit has reusability and scalability issues for general-purpose computing. Also, evaluating a logic circuit in software is slow. Because bitwise FHE operations on ciphers are about one billion times slower than the same operations on plain text, computation time ramps up as the circuit becomes complex.

Herein, we propose a solution that embraces a different approach that draws on a homomorphic instruction set emulator called CryptoEmu. CryptoEmu supports multiple FHE instructions (ADD, SUB, DIV, etc.). When CryptoEmu decodes an instruction, it invokes a pre-built function, referred as functional unit, to perform an FHE operation on input ciphertext. All functional units are built upon FHE gates from TFHE library, and they are accelerated using parallel computing techniques. During execution, the functional units fully utilize a multi-core processor to achieve an optimal speedup. A user would simply reprogram the FHE assembly code for various applications, while relying on the optimized functional units.

This report is organized as follows. Section 2 provides a primer on homomorphic encryption and summarizes related work. Section 3 introduces TFHE, an open-source library for fully homomorphic encryption. TFHE provides the building blocks for CryptoEmu. Section 4 describes CryptoEmu's general architecture. Section 5 and 6 provide detailed instruction set emulator implementations and gives benchmark results on Euler, a CPU/GPU supercomputer. Section 7 analyzes CryptoEmu's scalability and vulnerability, and compared CryptoEmu with a popular FHE software library, HELib [4]. Conclusions and future directions of investigation/development are provided in Section 8.

# 2 Background

**Homomorphic Encryption.** Homomorphic encryption (HE) is an encryption scheme that supports computation on encrypted data and generates an encrypted output. When the encrypted output is decrypted, its value is equal to the result when applying equivalent computation on unencrypted data. HE is formally defined as follows: let $Enc()$ be an HE encryption function, $Dec()$ be an HE decryption function, $f()$ be a function, $g()$ be a homomorphic equivalent of $f()$, and $a$ and $b$ be input data in plaintext. The following equation holds:

$$f(a, b) = Dec(g(Enc(a), Enc(b))) \ .$$

An HE scheme is a *partially homomorphic encryption* (PHE) scheme if $g()$ supports only either addition or multiplication. An HE scheme is a *somewhat homomorphic encryption* (SWHE) scheme if a limited number of $g()$ is allowed to be applied to encrypted data. An HE scheme is a *fully homomorphic encryption* (FHE) scheme if any $g()$ can be applied for an unlimited number of times over encrypted data [5].

The first FHE scheme was proposed by Gentry [6]. In HE schemes, the plaintext is encrypted with Gaussian noise. The noise grows after every homomorphic evaluation until the noise becomes too large for the encryption scheme to work. This is the reason that SWHE only allows a limited number of homomorphic evaluations. Gentry introduced a novel technique called “bootstrapping” such that a ciphertext can be homomorphically decrypted and homomorphically encrypted with the secret key to reduce Gaussian noise [6,7]. Building off [6], [8] improved bootstrapping to speedup homomorphic evaluations. The TFHE library based on [3] and [9] is one of the FHE schemes with a fast bootstrapping procedure.

**Related work.** This project proposed a software-based, multiple-instruction ISA emulator that supports fully homomorphic, general-purpose computation. Several general-purpose HE computer architecture implementations exist in both software and hardware. HELib [10] is an FHE software library the implements the Brakerski-Gentry-Vaikuntanathan (BGV) homomorphic encryption scheme [11]. HELib supports HE arithmetic such as addition, subtraction, multiplication, and data movement operations. HELib can be treated as an assembly language for general-purpose HE computing. Cryptoleq [12] is a software-based one-instruction set computer (OISC) emulator for general-purpose HE computing. Cryptoleq uses Paillier partially homomorphic scheme [13] and supports Turing-complete SUBLEQ instruction. HEROIC [14] is another OISC architecture implemented on FPGA, based on Paillier partially homomorphic scheme. Cryptoblaze [15] is a multiple-instruction computer based on non-deterministic Paillier encryption that supports partially homomorphic computation. Cryptoblaze is implemented on the FPGA.

# 3 TFHE Library

TFHE [2] is an FHE C/C++ software library used to implement fast gate-by-gate bootstrapping. The idea of TFHE is straightforward: if one can homomorphically evaluate a universal logic gate and homomorphically evaluate the next universal logic gate that uses the previous logic gate’s output as its input, one can homomorphically evaluate arbitrary Boolean functions, essentially allowing arbitrary FHE computations on encrypted binary data. Figure 2 demonstrates a minimum FHE gate-level library: NAND gate. Bootstrapped NAND gates are used to construct an FHE XOR gate. Similarly, any FHE logic circuit can be constructed with a combination of bootstrapped NAND gates.

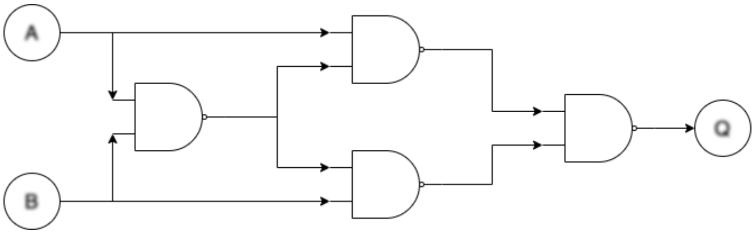


Figure 2: Use of bootstrapped NAND gate to form arbitrary FHE logic circuit. Blurry text in the figure denotes encrypted data.

**TFHE API.** TFHE library contains a comprehensive gate bootstrapping API for the

FHE scheme [2], including secret-keyset and cloud-keyset generation; Encryption/decryption with secret-keyset; and FHE evaluation on a binary gate netlist with cloud-keyset. TFHE API's performance is evaluated on a single core of Intel Xeon CPU E5-2650 v3 @ 2.30GHz CPU, running CentOS Linux release 8.2.2004 with 128 GB memory. Table 1 shows the benchmark result of TFHE APIs that are critical to CryptoEmu's performance. TFHE gate bootstrapping parameter setup, Secret-keyset, and cloud-keyset generation are not included in the table.

| API | Category | Bootstrapped? | Latency (ms) |
|---|---|---|---|
| Encrypt | Encrypt decrypt | N/A | 0.0343745 |
| Decrypt | Encrypt decrypt | N/A | 0.000319556 |
| CONSTANT | Homomorphic operations | No | 0.00433995 |
| NOT | Homomorphic operations | No | 0.000679717 |
| COPY | Homomorphic operations | No | 0.000624117 |
| NAND | Homomorphic operations | Yes | 25.5738 |
| OR | Homomorphic operations | Yes | 25.618 |
| AND | Homomorphic operations | Yes | 25.6176 |
| XOR | Homomorphic operations | Yes | 25.6526 |
| XNOR | Homomorphic operations | Yes | 25.795 |
| NOR | Homomorphic operations | Yes | 25.6265 |
| ANDNY | Homomorphic operations | Yes | 25.6982 |
| ANDYN | Homomorphic operations | Yes | 25.684 |
| ORNY | Homomorphic operations | Yes | 25.7787 |
| ORYN | Homomorphic operations | Yes | 25.6957 |
| MUX | Homomorphic operations | Yes | 49.2645 |
| CreateBitCipher | Ciphertexts | N/A | 0.001725 |
| DeleteBitCipher | Ciphertexts | N/A | 0.002228 |
| ReadBitFromFile | Ciphertexts | N/A | 0.0175304 |
| WriteBitToFile | Ciphertexts | N/A | 0.00960664 |

Table 1: TFHE API Benchmark

In Table 1, outside the "Homomorphic operations" category, all other operations are relatively fast. In general, the latency is around 25ms, with exceptions of MUX that takes around 50ms, and CONSTANT, NOT, COPY that are relatively fast. The difference in speed is from gate bootstrapping. Unary gates like CONSTANT, NOT and COPY do not need to be bootstrapped. Binary gates need to be bootstrapped once. MUX needs to be bootstrapped twice. The bootstrapping procedure is manifestly the most computationally expensive operation in TFHE. This overhead is alleviated in CryptoEmu via parallel computing as detailed below.

# 4 CryptoEmu Architecture Overview

CryptoEmu is a C/C++ utility that emulates the behavior of Fully Homomorphic Encryption (FHE) instructions. The instruction set that CryptoEmu supports is a subset of ARMv8 A32 instructions for fully homomorphic computation over encrypted data. Figure 3 shows the abstract layer for an FHE application. For an FHE application that performs computation over encrypted data, the application will be compiled into FHE assembly that the instruction emulator supports. The instruction set emulator coordinates control units and functional units to decode and execute FHE assembly and returns final results. The design and implementation of CryptoEmu are anchored by two assumptions:

**Assumption 1.** *The instruction set emulator runs on a high-performance multi-core machine.*

**Assumption 2.** *The cloud service provider is honest. However, the cloud is subject to cyber-attacks on the user's data.*

In §7.3 we will discuss modification on CryptoEmu's implementation when Assumption 2 does not hold.

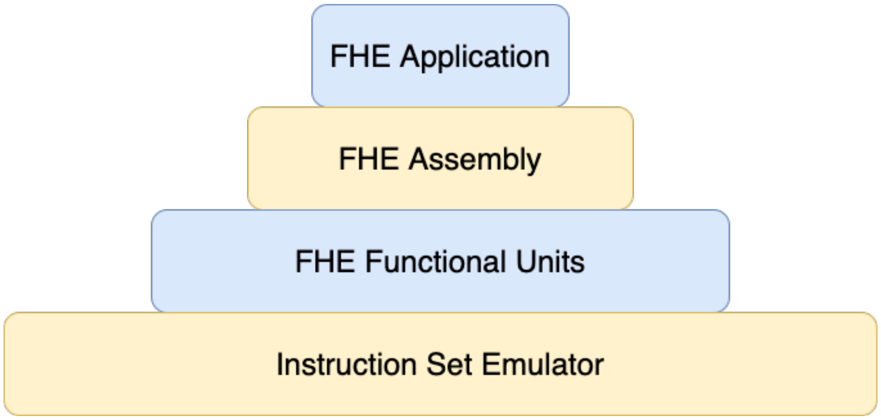


Figure 3: Abstract Layers

**Cloud service model.** Figure 4 shows the cloud service model. The instruction set emulator does what an actual hardware asset for encrypted execution would do: it reads from an unencrypted memory space an `HE instruction`; i.e., it fetches instruction that needs to be executed. The instruction set emulator also reads and writes `HE data` from an encrypted memory space, to process the user's data and return encrypted results to the encrypted memory space. The user, or any device that owns the user's secret key, will communicate with the cloud through an encrypted channel. The user provides all encrypted data to cloud. The user can send unencrypted HE instructions to the cloud through a secure channel. The user is also responsible for resolving branch directions for the cloud, based on the encrypted branch taken/non-taken result provided by the cloud.

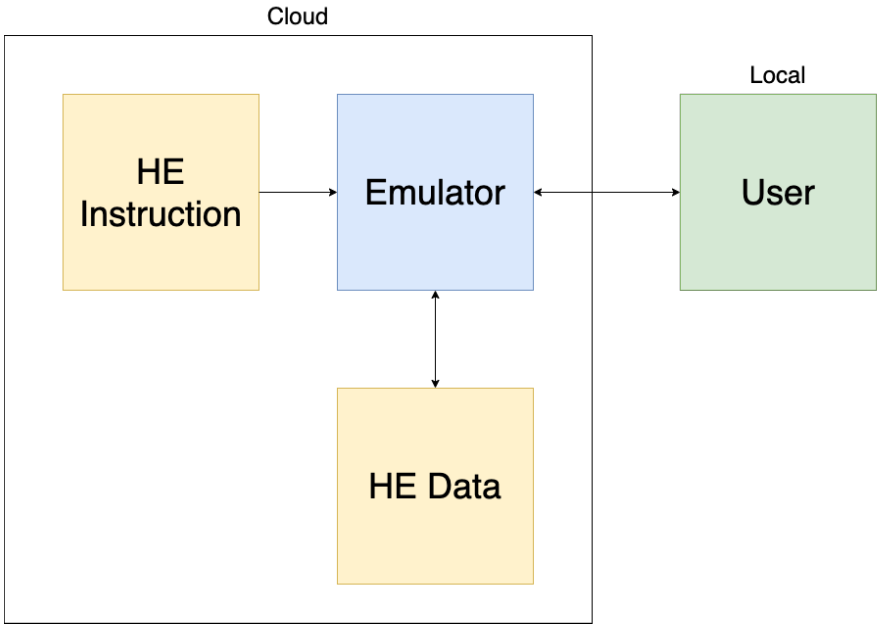


Figure 4: Cloud service model

## 4.1 Data Processing

In actuality, the `HE instruction` and `HE data` can be text files or arrays of data bits stored in buffers, if sufficient memory is available. CryptoEmu employs a load-store architecture. All computations occur on virtual registers (vReg), where a vReg is an array of 32 encrypted data bits. Depending on the memory available on the machine, the number of total vReg is configurable. However, it is the compiler's responsibility to recycle vRegs and properly generate read/write addresses. A snippet of possible machine instructions is as follows:

```
LOAD     R1   READ_ADDR1
LOAD     R2   READ_ADDR2
ADD      R0   R1, R2
STORE    R0   WRITE_ADDR
```

Above, to perform a homomorphic addition, CryptoEmu fetches the LOAD instruction from the instruction memory. Because the instruction itself is in cleartext, CryptoEmu decodes the instruction, loads a piece of encrypted data from HE data memory indexed by READ_ADDR1, and copies the encrypted data into vReg R1. Then, CryptoEmu increments its program counter by 4 bytes, reads the next LOAD instruction, and loads encrypted data from HE data memory into vReg R2. After the two operands are ready, CryptoEmu invokes a 32-bit adder and passes R1, R2 to it. The adder returns encrypted data in R0. Finally, CryptoEmu invokes the STORE operation and writes R0 data into the HE data memory pointed to by WRITE_ADDR. Under Assumption 2, the honest cloud could infer some user information from program execution because HE instructions are in cleartext. However, all user data stays encrypted and protected from malicious attackers. Vulnerabilities are discussed in §7.3.

### 4.2 Branch and Control Flow

CryptoEmu can perform a homomorphic comparison and generate N (negative), Z (zero), C (Unsigned overflow), and V (signed overflow) conditional flags. Based on conditional flags, the branch instruction changes the value of the program counter and therefore changes program flow. Because branches are homomorphically evaluated on the encrypted conditional flag, the branch direction is also encrypted. To solve this problem, CryptoEmu employs a client-server communication model from CryptoBlaze [15]. Through a secure communication channel, the cloud server will send an encrypted branch decision to a machine (client) that owns the user's private key. The client deciphers the encrypted branch decision and sends the branch decision encrypted with the server's public key to the server. The cloud server finally decrypts the branch decision, and CryptoEmu will move forward with a branch direction. Under assumption 2, the honest cloud will not use branch decision query and binary search to crack user's encrypted data, nor will the honest cloud infer user information from the user. In §7.3, the scenario that assumption 2 does not hold will be discussed.

## 5 Data Processing Units

Data processing units are subroutines that perform manipulation on encrypted data, including homomorphic binary arithmetic, homomorphic bitwise operation, and data movement. Under Assumption 1, data processing units are implemented with OpenMP [16] and are designed for parallel computing. If the data processing units exhaust all cores available, the rest of the operations will be serialized. We benchmarked the performance of data processing units with 16-bit and 32-bit vReg size. Benchmarks are based an computing node on Euler. The computing node has 2 NUMA nodes. Each NUMA nodes has two sockets, and each socket has a 12-core Intel Xeon CPU E5-2650 v3 @ 2.30GHz CPU. The 48-core computing node runs CentOS Linux release 8.2.2004 with 128 GB memory.

### 5.1 Load/Store Unit

CryptoEmu employs a load/store architecture. A LOAD instruction reads data from data memory; a STORE instruction writes data to data memory. The TFHE library [2] provides the API for load and store operations on FHE data. If data memory is presented as a file, CryptoEmu invokes the specific LD/ST subroutine, moves the file pointer to the right LD/ST address, and calls the appropriate file IO API, i.e.,

```
import_gate_bootstrapping_ciphertext_fromFile()
```

or

```
export_gate_bootstrapping_ciphertext_toFile()
```

Preferably, if the machine has available memory, the entire data file is loaded into a buffer as this approach significantly improves LD/ST instruction's performance. Table 2 shows LD/ST latency for 16-bit and 32-bit. LD/ST on a buffer is significantly faster than LD/ST on a file. The performance speedup is even more when the data file size is large because LD/ST on file needs to use *fseek()* function to access data at the address specified by HE instructions.

| | 16-bit (ms) | 32-bit (ms) |
|---|---|---|
| Load (file) | 0.027029 | 0.0554521 |
| Store (file) | 0.0127804 | 0.0276899 |
| Load (buffer) | 0.0043463 | 0.00778488 |
| Store (buffer) | 0.0043381 | 0.0077692 |

Table 2: LD/ST latencies.

## 5.2 Adder

CryptoEmu supports a configurable adder unit of variable width. As for the ISA that CryptoEmu supports, adders are either 16-bit or 32-bit. Operating under an assumption that CryptoEmu runs on a host that supports multi-threading, the adder unit is implemented as a parallel prefix adder [17]. The parallel prefix adder has a three-stage structure: pre-calculation of generate and propagate bit; carry propagation; and sum computation. Each stage can be divided into sub-stages and can leverage a multi-core processor. Herein, we use the OpenMP [16] library to leverage parallel computing.

**Stage 1: Propagate and generate calculation.** Let $a$ and $b$ be the operands to adder, and let $a[i]$ and $b[i]$ be the $i^{th}$ bit of $a$ and $b$. In carry-lookahead logic, $a[i]$ and $b[i]$ generates a carry if $a[i]$ $AND$ $b[i]$ is 1 and propagates a carry if $a[i]$ $XOR$ $b[i]$ is 1. This calculation requires an FHE AND gate and an FHE XOR gate, see §3 and Fig. 2 for gate bootstrapping. An OpenMP parallel region is created to handle two parallel sections. As shown in Fig. 5.2, CryptoEmu spawns two threads to execute two OpenMP sections in parallel.

For a 16-bit adder, *get_gp()* calculations are applied on every bit. This process is parallelizable: as shown in Fig. 5.2, CryptoEmu spawns 16 parallel sections [16], one per bit. Inside each parallel section, the code starts another parallel region that uses two threads. Because of nested parallelism, 32 threads in total are required to calculate every generation and propagate a bit concurrently. If there is an insufficient number of cores, parallel sections will be serialized, which will only affect the efficiency of the emulator.

**Stage 2: Carry propagation.** Let $G_i$ be the carry signal at $i^{th}$ bit, $P_i$ be the accumulated propagation bit at $i^{th}$ bit, $g_i$ and $p_i$ be outputs from propagate and generate calculation. We define operator $\odot$ such that

```
#pragma omp parallel sections num_threads(2)
{
    #pragma omp section
    {
        bootsAND(&g_o[0], &a_i[0], &b_i[0], bk);
    }

    #pragma omp section
    {
        bootsXOR(&p_o[0], &a_i[0], &b_i[0], bk);
    }
}
```

Figure 5: Parallel optimization for bitwise (g,p) calculation, *get_gp*()

```
#pragma omp parallel sections num_threads(N)
{
    #pragma omp section
    {
        get_gp(&g[0], &p[0], &a[0], &b[0], bk);
    }

    #pragma omp section
    {
        get_gp(&g[1], &p[1], &a[1], &b[1], bk);
    }

    ...

    #pragma omp section
    {
        get_gp(&g[14], &p[14], &a[14], &b[14], bk);
    }

    #pragma omp section
    {
        get_gp(&g[15], &p[15], &a[15], &b[15], bk);
    }
}
```

Figure 6: Parallel optimization for (g, p) calculation

```
#pragma omp parallel sections num_threads(2)
{
    // Compute carry out (G_i)
    #pragma omp section
    {
        // g(i) = g(i) + p(i) * g(i-1)
        bootsAND(g_tmp, p_1, g_0, bk);
        bootsOR(g_next, g_1, g_tmp, bk);
    }

    #pragma omp section
    {
        // p(i) = p(i) * p(i-1)
        bootsAND(p_next, p_1, p_0, bk);
    }
}
```

Figure 7: Parallel optimization for bitwise carry calculation, *get_carry*()

$$(g_x, p_x) \odot (g_y, p_y) = (g_x + p_x \cdot g_y, p_x \cdot p_y) \ .$$

Carry signal $G_i$ and accumulated propagation $P_i$ can be recursively defined as

$$(G_i, P_i) = (g_i, p_i) \odot (G_{i-1}, P_{i-1}), \text{ where } (G_0, P_0) = (g_0, p_0) \ .$$

The above recursive formula is equivalent to

$$(G_{i:j}, P_{i:j}) = (G_{i:n}, G_{i:n}) \odot (G_{m:j}, P_{m:j}), \text{ where } i \geq j \text{ and } m \geq n \ .$$

Therefore, carry propagation can be reduced to a parallel scan problem. In CryptoEmu, we defined a routine, *get_carry()* to perform operation $\odot$. As shown in Fig. 5.2 , the $\odot$ requires two FHE AND gate and an FHE OR gate. CryptoEmu spawns two threads to perform the $\odot$ operation in parallel.

For a 16-bit adder, we need 4 levels to compute the carry out from the most significant bit. As shown in fig 8, every two arrows that share an arrowhead represents one $\odot$ operation. The $\odot$ operations at the same level can be executed in parallel. In the case of a 16-bit adder, the maximum number of concurrent $\odot$ is 15 at level 1. Because of nested parallelism within the $\odot$ operation, the maximum number of threads required is 30. With a sufficient number of cores, parallel scan reduced carry propagation time from 16 times $\odot$ operation latency, to 4 times $\odot$ operation latency.

**Stage 3: Sum calculation.** The last stage for parallel prefix adder is sum calculation. Let $s_i$ be the sum bit at $i^{th}$ bit, $p_i$ be the propagation bit at $i^{th}$ bit, $G_i$ be the carry signal at $i^{th}$ bit. Then

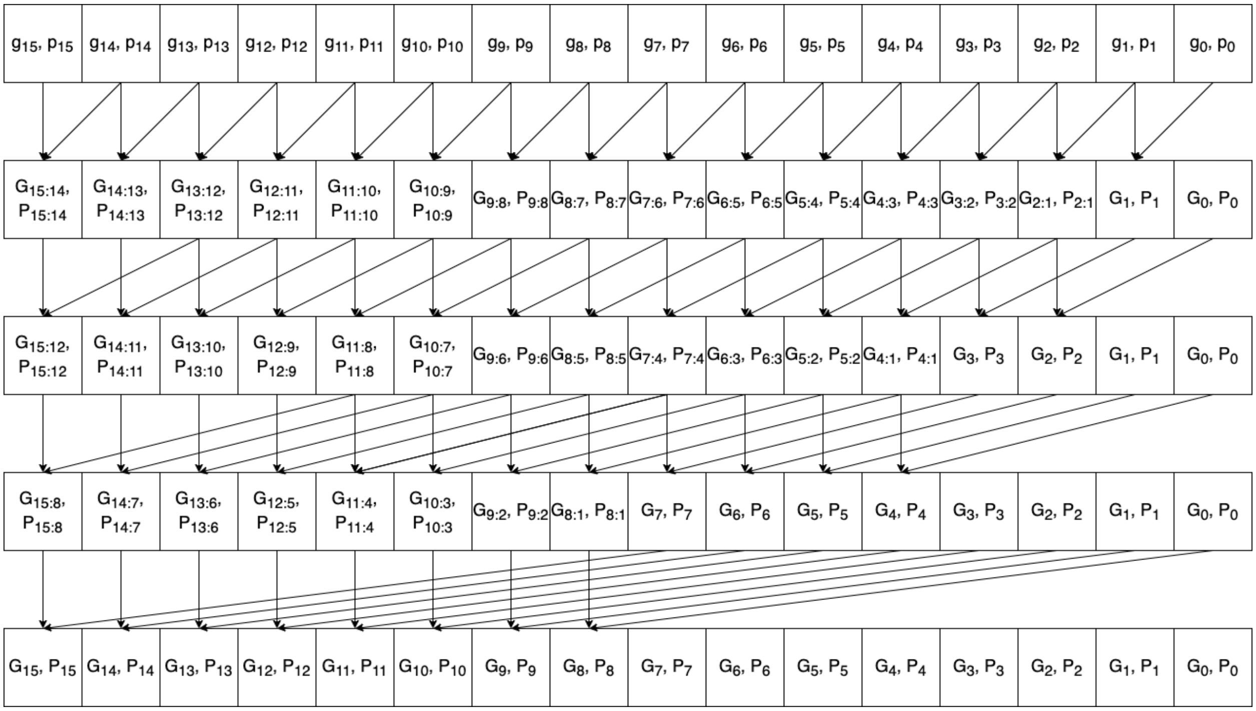


Figure 8: Parallel scan for carry signals

$$s_i = p_i \ XOR \ G_i \ .$$

One FHE XOR gate is needed to calculate 1-bit sum. For 16-bit adder, 16 FHE XOR gates are needed. All FHE XOR evaluation are independent, therefore can be executed in parallel. In total 16 threads are required for the best parallel optimization on sum calculation stage.

**Benchmark: the 16-bit adder.** Table 3 shows benchmarking results for a 16-bit adder unit executed on the target machine describe earlier in the document. If parallelized, the 1-bit *get_gp()* shown in Fig. 5.2 has one FHE gate latency around 25ms as shown in Table 1. Ideally, if sufficient cores are available and there is no overhead from parallel optimization, (g,p) calculation should run 16 *get_gp()* concurrently, and total latency should be 25ms. In reality, 16-bit (g,p) calculation uses 32 threads and takes 51.39ms to complete due to overhead in parallel computing.

For carry propagation calculation, the 1-bit *get_carry()* shown in Fig. 5.2 has two FHE gate latency of around 50ms when parallelized. In an ideal scenario, each level for carry propagation should run *get_carry()* in parallel, and total latency should be around 50ms. In reality, the 16-bit carry propagation calculation uses 30 threads on level 1 and takes 93.42ms. A collection of 28 threads are used on carry propagation level 2; the operation takes 93.58ms. A collection of 24 threads are used on carry propagation level 3; the operation takes 80.34ms. Finally, 16 threads are used on carry propagation level 4; the operation takes 70.85ms.

| Operation | Latency (ms) |
|---|---|
| (g,p) calculation | 51.3939 |
| Carry propagation (Level 1) | 93.4178 |
| Carry propagation (Level 2) | 93.5273 |
| Carry propagation (Level 3) | 80.342 |
| Carry propagation (Level 4) | 70.8481 |
| Sum calculation | 34.2846 |
| Total latency, including overhead | 528.482 |

Table 3: 16-bit adder latency

For sum calculation, 1-bit sum calculation uses one FHE XOR gate, with latency around 25ms. Ideally, if CryptoEmu runs all 16 XOR gates in parallel without parallel computing overhead, the latency for 16-bit sum calculation should be around 25ms. In reality, due to OpenMP overhead, the 16-bit sum calculation uses 16 threads and takes 34.28ms to complete.

In total, a 16-bit adder's latency is 486.66ms. This result includes latency for all stages, plus overheads like variable declaration, memory allocation, and temporary variable manipulation.

**Benchmark: the 32-bit adder.** Table 4 shows benchmarking results for a 32-bit adder unit executed on the target machine describe earlier in the document. Note that *get_gp()* and *get_carry()* have the same performance as the 16-bit adder. If sufficient cores are available, in the absence of OpenMP overhead, the (g,p) calculation should run 32 *get_gp()* concurrently for 32-bit adder at a total latency of 25ms. In reality, the 32-bit (g,p) calculation uses 32 threads and takes 94.92ms to complete.

| Operation | Latency (ms) |
|---|---|
| (g,p) calculation | 94.9246 |
| Carry propagation (Level 1) | 147.451 |
| Carry propagation (Level 2) | 133.389 |
| Carry propagation (Level 3) | 127.331 |
| Carry propagation (Level 4) | 112.268 |
| Carry propagation (Level 5) | 91.5781 |
| Sum calculation | 49.0098 |
| Total latency, including overhead | 941.12 |

Table 4: 32-bit adder latency

For carry propagation calculation, if sufficient cores are available, each level for carry propagation should run *get_carry()* in parallel. Without parallel computing overhead, total latency should be around 50ms. Level 0 of 32-bit carry propagation calculation uses 62 threads. Because the platform on which CryptoEmu is tested has only 48 cores, level 0 carry propagation calculation is serialized and takes 147.45ms to complete. Level 1 carry propagation calculation uses 60 threads, and similar to level 0, its calculation is serialized. Level 1 carry propagation calculation takes 133.39ms. Level 3 carry propagation calculation that uses 56 threads is serialized and takes 127.33ms to complete. Level 4 carry propagation calculation uses 48 threads, and it is possible to run every *get_carry()* in parallel on our 48-core workstation. Level 4 carry propagation calculation takes 112.27ms. Level 5 carry propagation calculation uses 32 threads. It is able to execute all *get_carry()* concurrently; level 5 takes 91.58ms to complete.

For sum calculation, the 32-bit adder spawns 32 threads in parallel to perform FHE XOR operation if sufficient cores are available. The latency for the 32-bit sum calculation should be around 25ms. In reality, the 32-bit sum calculation uses 32 threads and takes 49ms to complete.

In total, 32-bit adder's latency is 941.12ms. This result includes latency for all stages, plus overheads like variable declaration, memory allocation, and temporary variable manipulation.

## 5.3 Subtractor

The subtractor unit supports variable ALU size. CryptoEmu supports subtractors with 16-bit operands or 32-bit operands. Let $a$ be the minuend, $b$ be the subtrahend, and $diff$ be the difference. Formula for 2's complement subtraction is:

$$a + NOT(b) + 1 = diff \ .$$

As shown in Fig. 9, CryptoEmu reuses adder units in §5.2 to perform homomorphic subtractions. On the critical path, extra homomorphic NOT gates are used to create subtrahend's complement. For a subtractor with N-bit operands, N homomorphic NOT operations need to be applied on the subtrahend. While all FHE NOT gates can be evaluated in parallel, in §3 we showed that FHE NOT gates do not need bootstrapping, and is relatively fast (around 0.0007ms latency per gate) comparing to bootstrapped FHE gates. Therefore, parallel execution is not necessary. Instead, the homomorphic NOT operation is implemented in an iterative for-loop, as shown below:

```
for(int i = 0; i < N; ++i)
    bootsNOT(&b_neg[i], &b[i], bk);
```

In addition to homomorphic NOT operation on subtrahend, the carry out bit from bit 0 needs to be evaluated with an OR gate because carry in is 1. Therefore, the adder in §5.2 is extended to take a carry in bit. When sufficient cores are available on the machine, a subtractor units adds negation and carry bit calculation to the adder unit's critical path.

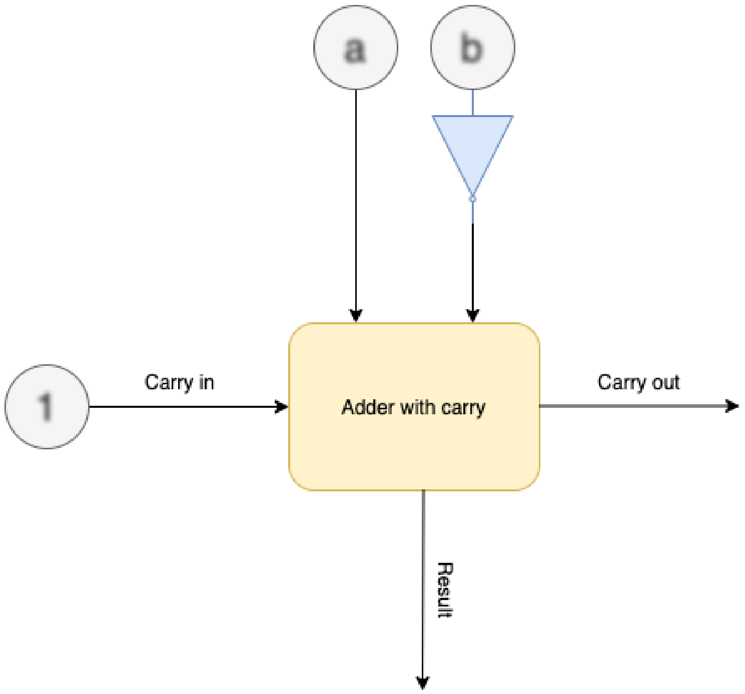


Figure 9: Subtractor architecture. Blurry text in the figure denotes encrypted data.

**Benchmark.** Tables 5 and 6 report benchmark results for the 16-bit and 32-bit subtractors on the target machine. Negation on the subtrahend takes a trivial amount of time to complete. The homomorphic addition is the most time-consuming operation in the subtractor unit. The homomorphic addition is a little slower than the homomorphic additions in §5.2 because the adder needs to use extra bootstrapped FHE gates to process carry in and calculate carry out from sum bit 0.

| Operation | Latency (ms) |
|---|---|
| Negation | 0.0341106 |
| Add with carry | 680.59 |
| Total latency, including overhead | 715.015 |

Table 5: 16-bit subtractor latency

## 5.4 Shifter

CryptoEmu supports three types of shifters: logic left shift (LLS), logic right shift (LRS), and arithmetic right shift (ARS). Each shifter type has two modes: immediate and register mode. In immediate mode, the shift amount is in cleartext. For example, the following instruction shifts encrypted data in R0 to left by 1 bit and assigns the shifted value to R0.

| Operation | Latency (ms) |
|---|---|
| Negation | 0.0347466 |
| Add with carry | 1058.65 |
| Total latency, including overhead | 1115.25 |

Table 6: 32-bit subtractor latency

```
LLS    R0    R0    1
```

This instruction is usually issued by the cloud to process user data. Shift immediate implementation is trivial. The shifter calls *bootCOPY()* API to move all data to the input direction by the specified amount. The LSB or MSB will be assigned to an encrypted constant using the *bootCONSTANT()* API call. Because neither *bootCOPY()* nor *bootCONSTANT()* need to be bootstrapped, they are fast operations, see Table 3. Therefore, an iterative loop is used for shifting. Parallel optimization is unnecessary.

In register mode, the shift amount is an encrypted data stored in the input register. For example, the following instruction shifts encrypted data in R0 to left by the value stored in R1 and assign shifted value to R0.

```
LLS    R0    R0    R1
```

Because the shifting amount stored in R1 is encrypted, the shifter can't simply move all encrypted bits left/right by a certain amount. The shifter is implemented as a barrel shifter, with parallel computing enabled.

**Logic left shift.** Figure 10 shows the architecture for the 16-bit LLS. In the figure, numbers in the bubbles denote encrypted data in the shift register and the shift amount register. Numbers in the diamond denote an encrypted constant value generated by the *bootsCONSTANT()* API. The 16-bit LLS has four stages. In each stage, based on the encrypted shift amount, the FHE MUX homomorphically selects an encrypted bit from the shift register. In the end, the LLS outputs encrypted shifted data. FHE MUX is an elementary logic unit provided by TFHE library [2]. FHE MUX needs to be bootstrapped twice, and its latency is around 50ms. Therefore, it is reasonable to spawn multiple threads to execute all MUX select in parallel in each stage. For the 16-bit LLS, each stage needs 16 threads to perform a homomorphic MUX select, as shown in the following code:

```
 #pragma omp parallel sections num_threads(16)
{
    #pragma omp section
    {
        bootsMUX(&out[0], &amt[0], &zero[0], &in[0], bk);
    }
```

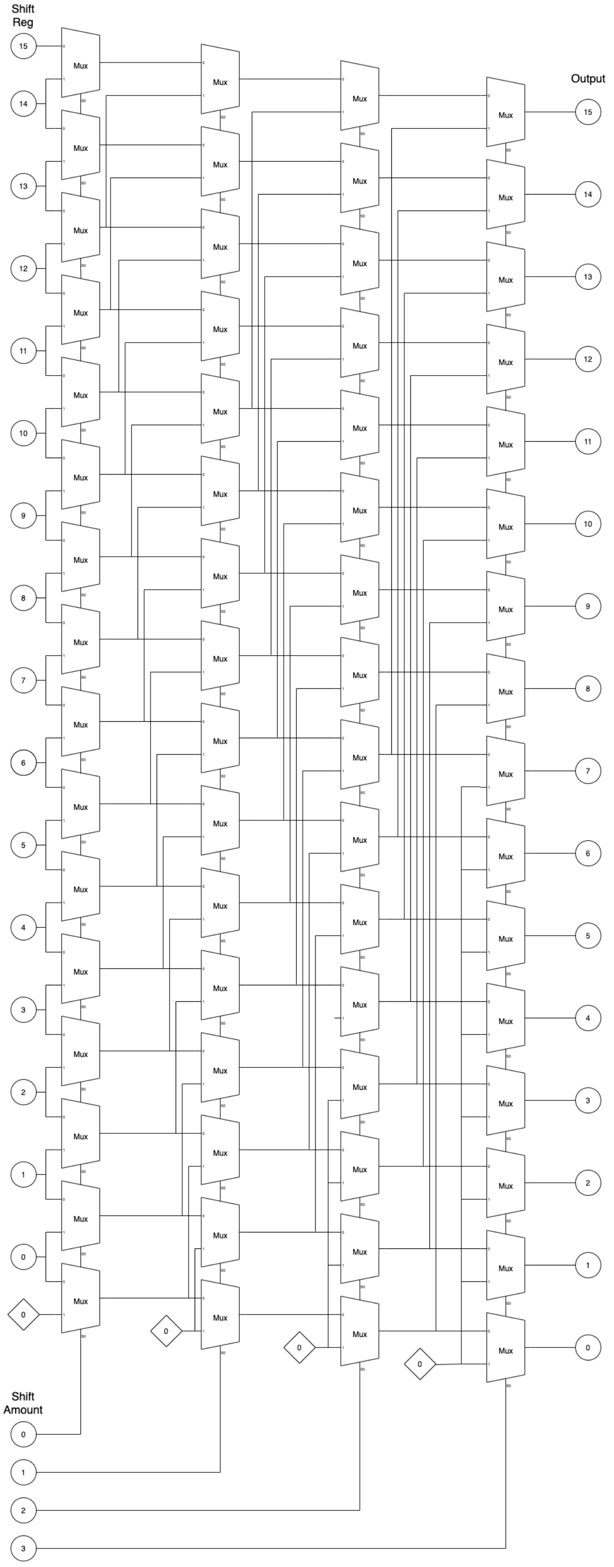


Figure 10: LLS architecture

```
        #pragma omp section
        {
            bootsMUX(&out[1], &amt[0], &in[0], &in[1], bk);
        }

        ...

        #pragma omp section
        {
            bootsMUX(&out[14], &amt[0], &in[13], &in[14], bk);
        }

        #pragma omp section
        {
            bootsMUX(&out[15], &amt[0], &in[14], &in[15], bk);
        }
    }
```

In a parallel implementation with zero overhead, each stage should have one FHE MUX latency of around 50ms. Therefore, in an ideal scenario, four stages would have a latency of around 200ms.

**Benchmark: Logic left shift.** Table 7 shows the benchmark results for the 16-bit LLS on the target platform. Each stage spawns 16 threads to run all FHE MUX in parallel with latency from 50-90ms, a latency that is in between 1 FHE MUX latency to 2 FHE MUX latency, due to parallel computing overhead. In total, it takes around 290ms to carry out a homomorphic LLS operation on 16-bit encrypted data.

| Operation | Latency (ms) |
|---|---|
| Mux select (Stage 1) | 87.3295 |
| Mux select (Stage 2) | 82.2656 |
| Mux select (Stage 3) | 76.6871 |
| Mux select(Stage 4) | 55.5396 |
| Total latency, including overhead | 287.92 |

Table 7: 16-bit LLS latency

Table 8 shows the benchmark result for the 32-bit LLS. Each stage spawns 32 threads and takes around 90-100ms to complete. In total, it takes around 450-500ms for a homomorphic LLS operation on 32-bit encrypted data.

| Operation | Latency (ms) |
|---|---|
| Mux select (Stage 1) | 101.92 |
| Mux select (Stage 2) | 89.7695 |
| Mux select (Stage 3) | 98.8634 |
| Mux select(Stage 4) | 91.3939 |
| Mux select (Stage 5) | 91.8491 |
| Total latency, including overhead | 474.739 |

Table 8: 32-bit LLS latency

**Logic right shift/Arithmetic right shift.** LRS has an architecture that is similar to the architecture of LLS. Figure 11 shows the architecture for the 16-bit LRS. Compared to Fig. 10, the only difference between LLS and LRS is the bit order of the input register and output register. To reuse the LRS architecture for ARS, one should simply pass MSB of the shift register as the shift-in value, shown as the numbers in the diamond in Fig. 11. The LRS and ARS shifter implementation is similar to that of LLS. For 16-bit LRS/ARS, CryptoEmu spawns 16 parallel threads to perform a homomorphic MUX select at each stage.

Table 9 shows the benchmark result for the 16-bit LRS and ARS. LRS and ARS have similar performance. At each stage, LRS/ARS utilizes 16 threads, and each stage takes 50-90ms to complete. Single stage latency is between 1 FHE MUX latency to 2 FHE MUX latency, due to parallel computing overhead. In total, 16-bit LRS/ARS latency is around 290-300ms.

| Operation | LRS Latency (ms) | ARS Latency (ms) |
|---|---|---|
| Mux select (Stage 1) | 88.1408 | 87.7689 |
| Mux select (Stage 2) | 85.5154 | 79.6416 |
| Mux select (Stage 3) | 75.0295 | 75.2938 |
| Mux select(Stage 4) | 55.9639 | 54.9246 |
| Total latency, including overhead | 290.517 | 296.124 |

Table 9: 16-bit LRS, ARS latency

Table 10 shows the benchmark result for the 32-bit LRS and ARS. The 32-bit LRS/ARS has five stages. Each stage creates 32 parallel threads to evaluate FHE MUX and takes 90-100ms to complete. Single stage latency is around 2 FHE MUX latency. In total, the 32-bit LRS/ARS takes around 470-500ms to complete a homomorphic LRS/ARS operation on 32-bit encrypted data.

## 5.5 Multiplier

**Design consideration.** Binary multiplication can be treated as a summation of partial products [18], see Fig. 12. Therefore, adder units mentioned in §5.2 can be reused for partial

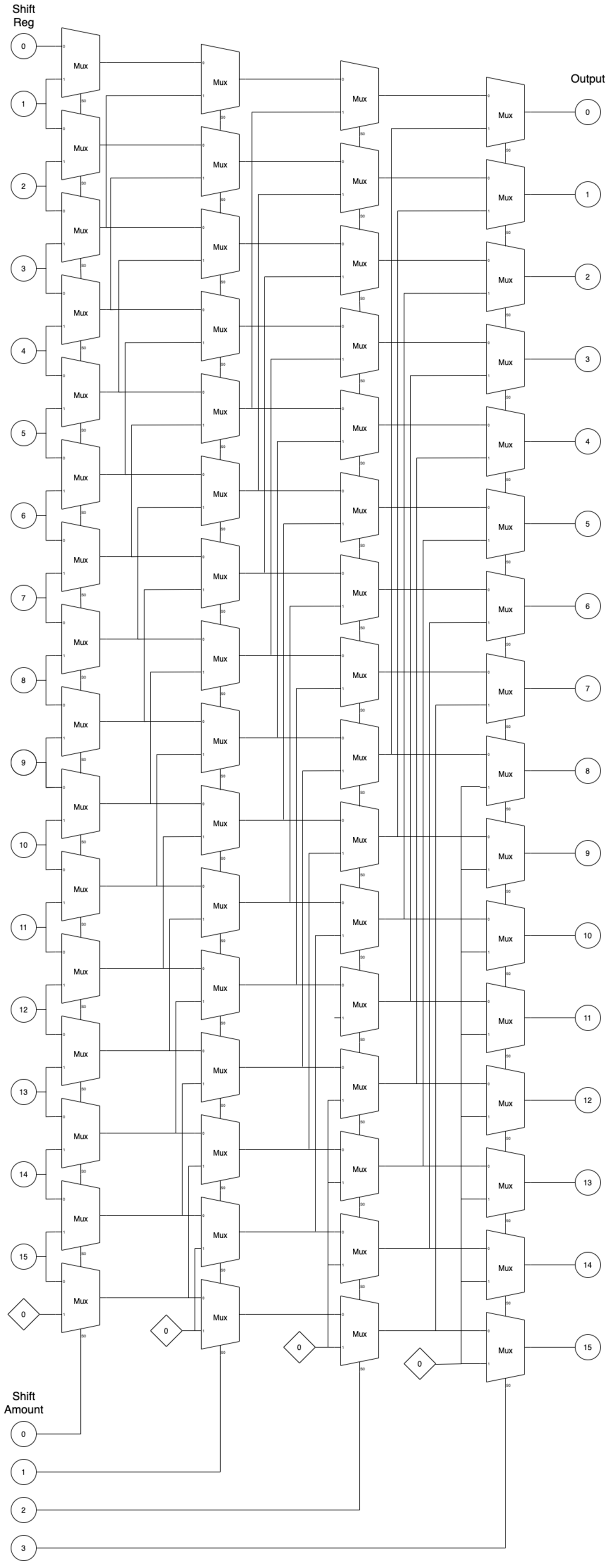


Figure 11: LRS architecture

| Operation | LRS Latency (ms) | ARS Latency (ms) |
|---|---|---|
| Mux select (Level 1) | 104.33 | 106.132 |
| Mux select (Level 2) | 104.75 | 95.3209 |
| Mux select (Level 3) | 90.2377 | 90.1364 |
| Mux select(Level 4) | 89.7183 | 90.5383 |
| Mux select(Level 5) | 90.6315 | 94.4654 |
| Total latency, including overhead | 472.896 | 491.787 |

Table 10: 32-bit LRS, ARS latency

sum computation. Summing up all partial products is a sum reduction operation, and therefore can be parallelized.

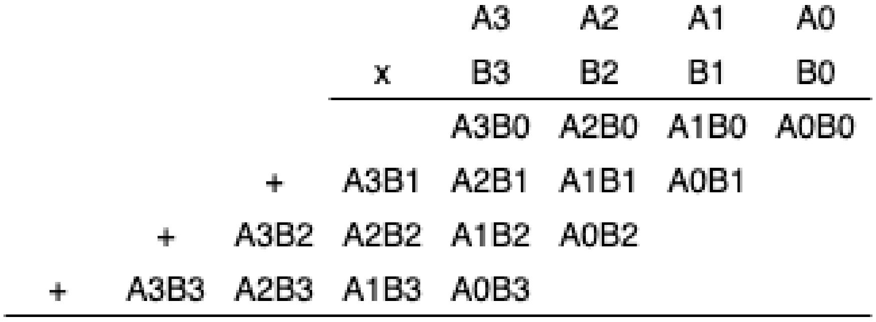


Figure 12: Binary multiplication

However, the best parallel optimization cannot be achieved on our 48-core computing node. For a 16-bit wide multiplier, the product is a 32-bit encrypted value. Therefore, a 32-bit adder is required to carry out the homomorphic addition. Each 32-bit adder has peak thread usage of 64 threads: 31 threads with nested parallel (g,p) calculation that uses two threads. Thus, on the server used (with 48 cores), the 32-bit adder has to be partially serialized. For the 16-bit multiplier's parallel sum reduction, at most eight summation occur in parallel and each summation uses a 32-bit adder. The peak thread usage is 512 threads. For a 32-bit multiplier, maximum thread usage is 2048 threads. Thus, because homomorphic multiplication is a computationally demanding process, the server used does not have sufficient resources to do all operations in parallel. Homomorphic multiplication will thus show suboptimal performance on the server used in this project.

Based on the design consideration above, CryptoEmu implements a carry-save multiplier [19] that supports variable ALU width. Carry-save multiplier uses an adder described in 5.2 to sum up partial products in series.

### 5.5.1 Unsigned multiplication

Figure 13 shows the multiplier's architecture. For a 16-bit multiplier, $A$ and $B$ are 16-bit operands stored in vRegs. $P$ is an intermediate 16-bit vReg to store partial products. Adder is a 16-bit adder with a carry in bit, see §5.2.

On startup, vReg $P$ is initialized to encrypted 0 using TFHE library's bootCONSTANT() API. Next, we enter an iterative loop and homomorphically AND all bits in vReg $A$ with LSB of vReg $B$, and use the result as one of the operands to the 16-bit adder. Data stored in vReg $P$ is then passed to the 16-bit adder as the second operand. The adder performs the homomorphic addition to output an encrypted carry out bit. Next, we right shift carry out, vReg $P$ and vReg $B$, and reached the end of the iterative for-loop. We repeat the for-loop 16 times, and the final product is a 32-bit result stored in vReg $P$ and vReg $B$. The pseudo-code below shows the 16-bit binary multiplication algorithm.

```
P = 0;
for 16 times:
    (P, carry out) = P + (B[0] ? A : 0)
    [carry out, P, B] = [carry out, P, B] >> 1;
return [P, B]
```

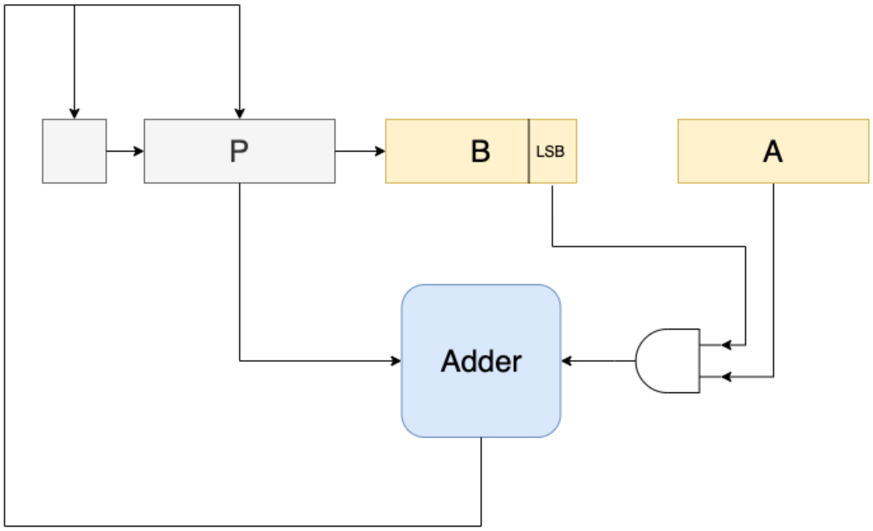


Figure 13: Carry save multiplier

For implementation, an N-bit multiplier uses N threads to concurrently evaluate all the FHE AND gates. The adder is already parallel optimized. The rest of the multiplication subroutine is executed sequentially. Therefore, the multiplier is a computationally expensive unit in CryptoEmu.

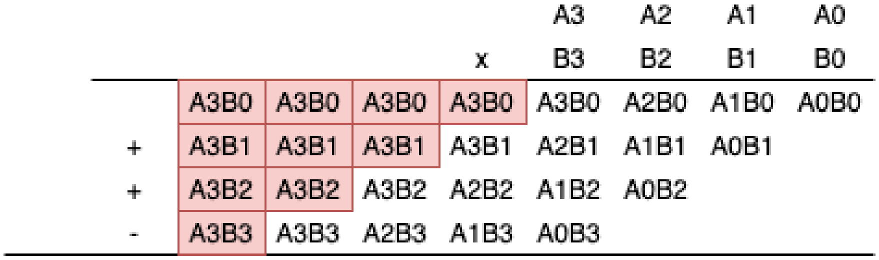
**Benchmark: Unsigned multiplication.** Table 11 shows benchmark for 16-bit unsigned multiplication and 32-bit unsigned multiplication. A single pass for partial product summation takes around 715ms and 925ms, respectively. Total latency is roughly equal to N times single iteration's latency because summation operations are in sequence.

| | 16-bit multiplier (ms) | 32-bit multiplier (ms) |
|---|---|---|
| Single iteration | 715.812 | 926.79 |
| Total latency, including overhead | 11316.8 | 36929.2 |

Table 11: Unsigned multiplication latency

### 5.5.2 Signed multiplication

Signed multiplication is implemented using the carry-save multiplier in Figure 13, with slight modifications. For N-bit signed multiplication, partial products need to be signed extended to 2N bit. Figure 14 shows the partial product summation for 4-bit signed multiplication. This algorithm requires a 2N-bit adder and a 2N-bit subtractor for N-bit signed multiplication. The algorithm is further simplified with a "magic number" [19]. Figure 15 shows the simplified signed multiplication. Based on this algorithm, the unsigned carry-save multiplier is modified to adopt signed multiplication.



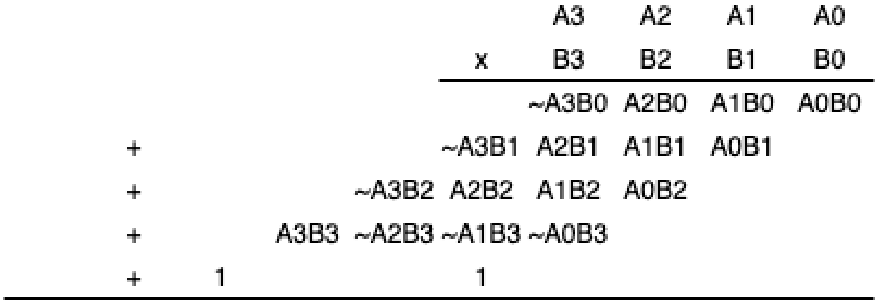
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | A3 | A2 | A1 | A0 |
| | | | | x | B3 | B2 | B1 | B0 |
| | A3B0 | A3B0 | A3B0 | A3B0 | A3B0 | A2B0 | A1B0 | A0B0 |
| + | A3B1 | A3B1 | A3B1 | A3B1 | A2B1 | A1B1 | A0B1 | |
| + | A3B2 | A3B2 | A3B2 | A2B2 | A1B2 | A0B2 | | |
| - | A3B3 | A3B3 | A2B3 | A1B3 | A0B3 | | | |

Figure 14: Signed binary multiplication

**Benchmark: Signed multiplication.** The unsigned multiplication was modified for signed multiplication based on the simplified algorithm outlined above. Table 12 shows benchmark results for the 16-bit and 32-bit signed multiplications. A single pass for partial product summation takes roughly 745ms and 1155ms, respectively. The total latency for signed multiplication is around N times that of the single iteration.

## 5.6 Divider

CryptoEmu supports unsigned division over encrypted data. Figure 16 shows the flow chart for a non-restoring division algorithm for unsigned integer [20].



```
                               A3     A2     A1     A0
                        x      B3     B2     B1     B0
                               ~A3B0  A2B0   A1B0   A0B0
+                       ~A3B1  A2B1   A1B1   A0B1
+                ~A3B2  A2B2   A1B2   A0B2
+         A3B3   ~A2B3  ~A1B3  ~A0B3
+  1                    1
```

Figure 15: Simplified signed binary multiplication

|  | 16-bit multiplier (ms) | 32-bit multiplier (ms) |
|---|---|---|
| Single iteration | 742.8 | 1154.77 |
| Total latency, including overhead | 13120.6 | 34826 |

Table 12: Signed multiplication latency

The division algorithm is based on sequential addition/subtraction. In every iteration, the MSB of vReg $A$ decides whether the divider takes an addition or subtraction. This function is implemented building off the adder in §5.2. The MSB of vReg $A$ is passed as the $SUB$ bit into the adder. In the adder, the second operand $b$ is homomorphically XORed with the $SUB$ bit and then added with the first operand $A$ and $SUB$ bit. If FHE XOR gates are evaluated in parallel and parallel computing overhead is ignored, the adder with add/sub select is about 1 FHE XOR gate slower than regular adders. Note that, the integer divide instruct does not care about the value of remainder because the result will be rounded down (floor) to the closest integer, and therefore remainder calculation is skipped to save computation time.

The unsigned division algorithm is a sequential algorithm. Within each iteration, a parallel optimized adder subroutine is invoked. Like multiplication, the unsigned division algorithm is computationally expensive. Pseudocode for the 16-bit unsigned division is shown below.

```
Non-restoring Division:
    A = 0;
    D = Divisor
    Q = Dividend

    for 16 times:
        [A, Q] = [A, Q] << 1
        if A < 0:
            A = A + D
        else:
```

```
        A = A - D
    if A < 0:
        Q[0] = 0;
    else:
        Q[0] = 1;

return Q
```

**Benchmark.** Table 13 provides benchmark results for the 16-bit and 32-bit unsigned integer divisions. Performance for a single iteration is slightly slower than homomorphic addition because one iteration uses an adder-subtractor unit. The division algorithm is sequential, and the total latency for N-bit division is around N times that of a single iteration's latency.

| | 16-bit divider (ms) | 32-bit divider (ms) |
|---|---|---|
| Single iteration | 769.063 | 1308.67 |
| Total latency, including overhead | 11659.5 | 38256.5 |

Table 13: Unsigned division latency

## 5.7 Bitwise operations and data movement

CryptoEmu supports several homomorphic bitwise operations and data movement of configurable data size. The instruction set supports 16-bit and 32-bit bitwise operation and data movement.

The collection of bitwise operations includes homomorphic NOT, AND, OR, XOR, and ORN, in both immediate mode and register mode. All homomorphic bitwise operations except bitwise NOT are implemented with parallel computing optimizations. For the N-bit bitwise operation, CryptoEmu spawns N threads to carry out all FHE gate evaluation in parallel. Bitwise NOT operation is an exception because the FHE NOT gate does not need to be bootstrapped and is relatively fast. Parallel computing cannot be justified because of its overhead. The following code shows a generic way to implement N-bit parallel bitwise operations using the TFHE and OpenMP libraries.

```
#pragma omp parallel sections num_threads(16)
{
    #pragma omp section
    {
        bootsAND(&out[0], &a[0], &b[0], bk);
    }
```

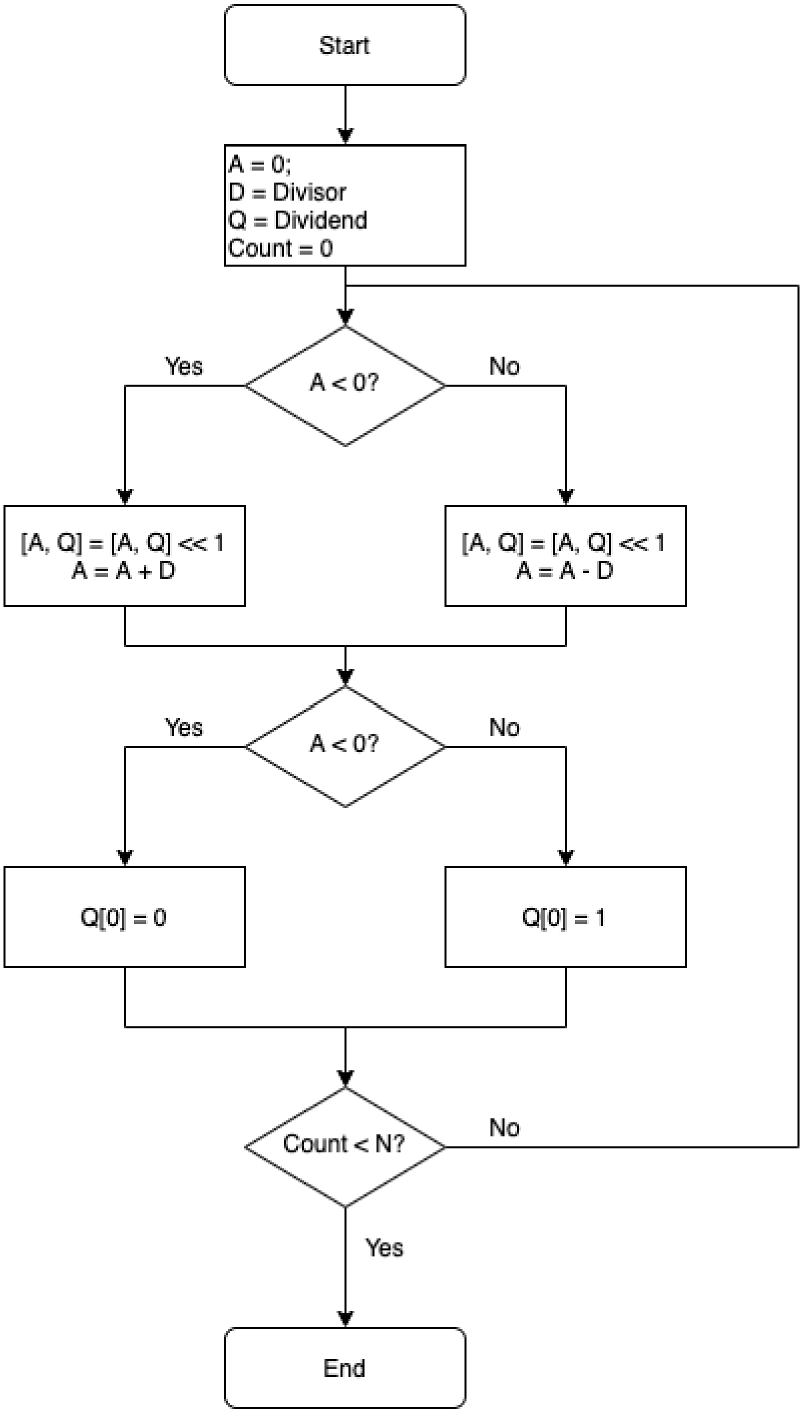


Figure 16: Binary division algorithm

```
        #pragma omp section
        {
            bootsAND(&out[1], &a[1], &b[1], bk);
        }

        ...

        #pragma omp section
        {
            bootsAND(&out[N-2], &a[N-2], &b[N-2], bk);
        }

        #pragma omp section
        {
            bootsAND(&out[N-1], &a[N-1], &b[N-1], bk);
        }
    }
```

Data movement instructions support homomorphic operations such as bit field clear, bit field insert, bit-order reverse, and byte-order reverse. Because data movement uses TFHE library's non-bootstrapped APIs like *bootsCOPY()* and *bootsCONSTANT()*, they are not implemented via OpenMP parallel sections.

**Benchmark.** Table 14 provides benchmark results for 16-bit and 32-bit bitwise operation and data movement instructions. When parallel optimized, an N-bit bitwise operation spawns N threads. For bitwise operation with bootstrapped FHE gate, latency is between 1-2 FHE gates when all threads are in parallel. The performance of the N-bit bitwise NOT and data movement instructions that are implemented sequentially is proportional to N times the corresponding TFHE API's latency.

# 6 Control Units

A control unit decides and/or changes the value of the program counter (PC), which in CryptoEmu is an integer in cleartext. CryptoEmu supports conditional execution that generates encrypted conditional flags defined in the ARM ISA. Based on conditional flags, the branch unit decides the branch direction to take. The PC is set to a cleartext that points to an HE instruction address when branch is taken, or increased by 4 if the branch is not taken.

## 6.1 Conditional Flags

The conditional unit handles NZCV flags defined in the ARM architecture. The N (negative) flag is set if an instruction's result is negative. The Z (zero) flag is set if an instruction's

| Operation | Category | Parallel? | 16-bit latency (ms) | 32-bit latency (ms) |
|---|---|---|---|---|
| AND(imm) | Bitwise operations | Yes | 26.9195 | 27.1288 |
| AND(reg) | Bitwise operations | Yes | 47.7228 | 55.4597 |
| OR(imm) | Bitwise operations | Yes | 28.1923 | 27.8774 |
| OR(reg) | Bitwise operations | Yes | 47.6171 | 50.089 |
| XOR(imm) | Bitwise operations | Yes | 29.2375 | 27.9389 |
| XOR(reg) | Bitwise operations | Yes | 47.1986 | 50.3683 |
| ORN(imm) | Bitwise operations | Yes | 27.8467 | 27.776 |
| ORN(reg) | Bitwise operations | Yes | 47.6422 | 57.0639 |
| NOT(imm) | Bitwise operations | Yes | 0.0072794 | 0.0116968 |
| NOT(reg) | Bitwise operations | No | 0.006089 | 0.01232 |
| BFC | Bitwise operations | No | 0.0028094 | 0.0026892 |
| BFI | Bitwise operations | No | 0.003278 | 0.0033974 |
| RBIT | Bitwise operations | No | 0.0104582 | 0.0216222 |
| REV | Bitwise operations | No | 0.0097926 | 0.0187618 |

Table 14: Bitwise operation and data movement latency

result is zero. The C (carry) flag is set if an instruction's results in an unsigned overflow. The V (overflow) flag is set if an instruction's results in an signed overflow. NZCV values are stored as encrypted data in a vReg. An instruction can be conditionally executed to update NZCV's value. The following code shows HE assembly for a while loop. Note that all data in this example is encrypted.

```
C++:
int i = 42;
while(i != 0)
    i--;

HE assembly:
MOV    R0    R0    42
Loop_label:
    SUBS   R0    R0    1
    B_NE   Loop_label
```

After the SUBS instruction is executed, the NZCV vReg is updated with results in R0. Based on the value of the Z flag, the program counter either updates its value to SUBS' instruction address (branch taken) or increases by 4 (branch non-taken).

Because the homomorphic evaluation of a conditional flag is computationally expensive, an ordinary data processing unit does not have a mechanism to update a conditional flag. In reality, instructions like ADDS, SUBS, and MULS are treated as micro-ops and completed in two steps: homomorphic data processing and homomorphic conditional flag calculation.

The N flag calculation is straight forward. CryptoEmu takes the MSB of result vReg and assigns it to the NZCV vReg. For the C flag, CryptoEmu takes the carry out result from previous unsigned computation and assigns it to the NZCV vReg. For the Z flag, the conditional unit performs an OR reduction on the input vReg, and assigns a negated result to the NZCV vReg. OR reduction can be parallelized. For a 16-bit conditional unit, OR reduction has four stages and maximum thread usage is eight. For a 32-bit conditional unit, OR reduction has five stages and maximum thread usage is 16. Finally, for the V flag, we take the sign bit (MSB) of two operands and the result for a signed computation, denoted as $a$, $b$, and $s$, respectively. Overflow is then evaluated as

$$ov = (\bar{a} \cdot \bar{b} \cdot s) + (a \cdot b \cdot \bar{s}) \ .$$

Note that the conditional unit calculates overflow in parallel by executing $(\bar{a} \cdot \bar{b} \cdot s)$ and $(a \cdot b \cdot \bar{s})$ concurrently.

**Benchmark.** Table 15 shows benchmark results for the 16-bit and 32-bit conditional units. The N flag and C flag calculations are fast because they do not have bootstrapped FHE gates. The Z flags and V flags are calculated at the same time. For conditional flag calculation on N-bit result, the maximum thread usage is $N/2 + 2$ threads. Z flag and V flag's latency dominates the total latency.

| | 16-bit (ms) | 32-bit (ms) |
|---|---|---|
| N flag | 0.0214161 | 0.0204952 |
| C flag | 0.0007251 | 0.0002539 |
| Z flag and V flag | 115.811 | 168.728 |
| Total latency, including overhead | 115.901 | 169.843 |

Table 15: Conditional flag latency

## 6.2 Branch

CryptoEmu has a vReg virtual register reserved for the program counter (PC). The PC stores a cleartext value that points at an HE instruction address. CryptoEmu loads an HE instruction from PC, decodes the instruction, invokes the data processing unit to execute the instruction, and repeats. Normally when using the ARM A32 ISA, the next instruction address that CryptoEmu is at the current PC plus 4. However, branch instructions can modify the PC value based on conditional flags, and therefore decide the next instruction address CryptoEmu fetches from. For example, the following code branches to ADDRESS_LABEL because SUBS instruction sets the Z flag.

```
MOV    R0   R0   1
SUBS   R0   R0   1
B_NE    ADDRESS_LABEL
```

§6.1 discussed conditional flag calculation. The NZCV flag is stored in a vReg as encrypted data. The cloud needs to know the value of the NZCV flags to decide which branch direction to take. However, the cloud has no knowledge of NZCV value unless it is decrypted by the user. CryptoEmu adopts a server-client communication model from CryptoBlaze [15]. As demonstrated in Fig. 17, the cloud (server) first homomorphically evaluates an HE instruction and updates the NCZV vReg. Next, the cloud sends the encrypted NCZC value and branch condition to the client. User deciphers the encrypted NCZC value, and sends a branch taken/non-taken decision back to the cloud through a secure channel. Once the cloud learns the branch decision, the PC will either be updated to PC+4, or to the branch address.

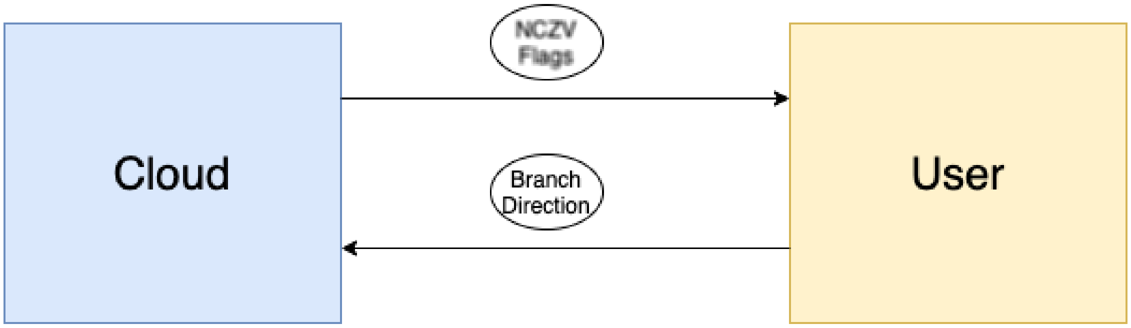


Figure 17: Server-client model for branch resolution
Blurry text in the figure denotes encrypted data.

# 7 Results

This section reports on (*i*) the scalability with respect to bit count when the core count is a fixed number; (*ii*) the scalability with respect to core count when the data size is a fixed number; (*iii*) on CryptoEmu's potential vulnerabilities; and (*iv*) comparison against the state of the art.

## 7.1 Scalability with respect to bit count

Table 1 shows the latency of the TFHE library API. Non-bootstrapped gates such as CONSTANT, NOT, and COPY are three to four orders of magnitude faster than bootstrapped gates. Therefore, compared to bootstrapped gates, non-bootstrapped gates are considered as negligible in the CryptoEmu's performance equation. Functional units without bootstrapped gates are not included in our scalability analysis.

The latency of a single bootstrapped gate is denoted as $G$. In the rest of the subsection, we analyzed the scalability of functional unit subroutines with respect to data size N. Two scenarios are considered: single core mode, and multi-core mode with a sufficient number of cores to do all intended parallel computation concurrently. CryptoEmu uses all available cores to concurrently perform homomorphic computation on encrypted data with multi-core mode latency. When CryptoEmu exhausted all core resources, the rest of the program will be serialized with single-core mode latency.

**Adder.** An adder has 3 stages: Generate and propagate calculation, carry calculation, and sum calculation. Let the available core number be 1 (single core mode); then, the time complexity for each stage is:

*Generate and propagate calculation*: To compute generate and propagate for one bit, two bootstrapped FHE gates evaluations are required. With N-bit operands, total latency is $2 \cdot G \cdot N$.

*Carry calculation*: Three bootstrapped FHE gates evaluations are required to calculate the carry out bit. With N-bit operands, total latency is $3 \cdot G \cdot N$.

*Sum calculation*: One bootstrapped XOR gate is needed to calculate one sum bit. With N-bit operands, total latency is $G \cdot N$.

*Total latency*: The latency for adder with single core is $6 \cdot G \cdot N$. Time complexity is $O(N)$.

Table 16 summarizes the discussion above for the N-bit adder with one core available.

| Operation | Latency | Time complexity |
|---|---|---|
| (g,p) calculation | $2GN$ | $O(N)$ |
| Carry calculation | $3GN$ | $O(N)$ |
| Sum calculation | $GN$ | $O(N)$ |
| Total latency | $6GN$ | $O(N)$ |

Table 16: Adder scalability, single core

Assume a sufficient but fixed number of processors are available. Then, the latencies become:

*Generate and propagate*: Calculation for generate and propagate for one bit can be executed in parallel, with a latency of one bootstrapped FHE gate. Given N-bit operands, N calculations can be carried out concurrently. Therefore, the latency for the (g,p) calculation is $G$.

*Carry calculation*: Carry out computation for one bit can be executed in parallel, with a latency of two bootstrapped FHE gates. For N-bit operands, carry calculation is divided into $log(N)$ stages. Operations in the same stage are executed in parallel. Therefore, latency for carry calculation is $2 \cdot G \cdot log(N)$.

*Sum calculation*: Each sum bit needs one bootstrapped XOR gate. For N-bit operands, N computations can be executed in parallel. Total latency for sum calculation is $G$.

*Total latency*: The total latency for an adder with sufficient yet fixed number of cores is $2 \cdot G \cdot log(N) + 2G$. Time complexity is $O(log(N))$.

Table 17 summarizes the scalability analysis discussion for the N-bit adder with sufficient, yet fixed number of cores available.

| Operation | Latency | Time complexity |
|---|---|---|
| (g,p) calculation | $G$ | $O(1)$ |
| Carry calculation | $2G \cdot log(N)$ | $O(log(N))$ |
| Sum calculation | $G$ | $O(1)$ |
| Total latency | $2G \cdot log(N) + 2G$ | $O(log(N))$ |

Table 17: Adder scalability, multiple core

**Subtractor.** A subtractor performs homomorphic negation on subtrahend, and homomorphically adds subtrahend's complement to minuend using an adder with carry in of one.

Negation uses TFHE library's non-bootstrapped NOT gate, and its latency is negligible. Adding a carry to the adder uses two extra bootstrapped gates. Therefore, the total latency for subtractor is $Latency(Adder) + 2 \cdot G$.

Table 18 summarizes key scalability aspects for the N-bit subtractor with single core and with sufficient yet fixed number of cores available.

| # of cores | Latency | Time complexity |
|---|---|---|
| Single core | $6GN + 2G$ | $O(N)$ |
| Multi-core | $2Glog(N) + 4G$ | $O(log(N))$ |

Table 18: Subtractor scalability

**Shifter.** For shifting with cleartext immediate, the operation is essentially a homomorphic data movement operation using the TFHE library's COPY and CONSTANT API. Shifting with immediate latency is therefore negligible compared to the rest of function units in CryptoEmu.

Shifting with encrypted vReg is implemented as a barrel shifter. For vReg N-bit shift, the operation has $log(N)$ stages. In each stage, individual bits are selected by bits in the shifting amount vReg moves to the next stage through a MUX. The TFHE MUX has latency around $2 \cdot G$.

When a single core is used, each stage has N MUXs. Latency in one stage is $2{\cdot}G{\cdot}N$. There are $log(N)$ stages, therefore total shifter latency for a single core processor is $2 \cdot G \cdot N \cdot log(N)$.

When sufficient cores are given, MUXs of the same stage can be evaluated in parallel, resulting in a latency of $2 \cdot G$. There are $log(N)$ stages, therefore total shifter latency for processor with sufficient cores is $2 \cdot G \cdot log(N)$.

Table 19 summarizes key scalability aspects for the shifter with respect to bit count N, in single core and multi-core mode.

**Multiplier.** The multiplier is implemented with iterative multiplication algorithms. A multiplier with N-bit operands requires N times iterations.

| # of cores | Latency | Time complexity |
|---|---|---|
| Single core | $2GNlog(N)$ | $O(Nlog(N))$ |
| Multi-core | $2Glog(N)$ | $O(log(N))$ |

Table 19: Shifter scalability

N-bit unsigned multiplication contains an N-bit FHE AND evaluation and invokes an N-bit adder for every iteration. When running unsigned multiplication with a single core, sequential latency for N-bit FHE AND is $N \cdot G$. Sequential latency for the N-bit adder is $6 \cdot G \cdot N$. Latency for one iteration is $7 \cdot G \cdot N$. Total latency for the N-bit unsigned multiplication $7 \cdot G \cdot N^2$.

When sufficient number of cores are provided to run subroutines in parallel, parallel latency for the N-bit FHE AND is $G$. Parallel latency for N-bit adder is $2 \cdot G \cdot log(N) + 2G$. One iteration takes $2 \cdot G \cdot log(N) + 3G$ to complete. Total latency for N-bit unsigned multiplication is $2 \cdot G \cdot N \cdot log(N) + 3 \cdot G \cdot N$.

Signed multiplication has the same latency per iteration as in unsigned multiplication. N-bit signed multiplication involves an N-bit addition that adds an offset to the upper half of the 2N-bit product. This operation has sequential latency of $6 \cdot G \cdot N$ and parallel latency of $2 \cdot G \cdot log(N) + 3G$. Total latency for signed multiplication is therefore $7 \cdot G \cdot N^2 + 6 \cdot G \cdot N$ for single core, and $2 \cdot G \cdot N \cdot log(N) + 3 \cdot G \cdot N + 2 \cdot G \cdot log(N) + 3G$ if a sufficient number of cores is available.

Table 20 summarizes key scalability aspects (latency and time complexity) for the unsigned and signed multiplication with respect to bit count N, in single core and multi-core mode.

| unsigned? | # of cores | Latency | Time complexity |
|---|---|---|---|
| Yes | Single core | $7GN^2$ | $O(N^2))$ |
| Yes | Multi-core | $2GNlog(N) + 3GN$ | $O(Nlog(N))$ |
| No | Single core | $7GN^2 + 6GN$ | $O(N^2))$ |
| No | Multi-core | $2GNlog(N) + 3GN + 2Glog(N) + 3G$ | $O(Nlog(N))$ |

Table 20: Multiplier scalability

**Divider.** The divider is implemented with iterative non-restoring division algorithm. A divider with N-bit operands requires N iterations. Every iteration involves negligible non-bootstrapped gates and an adder-subtractor unit. The adder-subtractor selectively inverts the second operand based on the value in the SUB bit. Then the subroutine invokes an adder to add the first operand, processes the second operand, and SUB bit together to form a result.

For computation on single core, selective inversion for an N-bit operand uses N FHE XOR gates, and therefore has a latency of $G \cdot N$. Adder with carry in has $6 \cdot G \cdot N + 2 \cdot G$ latency. Latency per iteration is $7 \cdot G \cdot N + 2 \cdot G$, and total latency is $7 \cdot G \cdot N^2 + 2 \cdot G \cdot N$ for single core.

If enough cores are available, the N-bit selective inversion can be executed in parallel, resulting in $G$ latency. Adder with carry in parallel computing mode has $2 \cdot G \cdot log(N) + 4 \cdot G$ latency. Latency per iteration is $2 \cdot G \cdot log(N) + 5 \cdot G$, and total latency is $2 \cdot G \cdot N \cdot log(N) + 5 \cdot G \cdot N$.

Table 21 shows latency and time complexity results for the unsigned division with respect to bit count N, in single core and multi-core modes.

| # of cores | Latency | Time complexity |
|---|---|---|
| Single core | $7GN^2 + 2GN$ | $O(N^2))$ |
| Multi-core | $2GNlog(N) + 5GN$ | $O(Nlog(N))$ |

Table 21: Divider scalability

**Bitwise operations.** Bitwise operation evaluates all input bits with one FHE gate. Latency per bit is $G$. For N-bit bitwise operation in single core mode, total latency is $G \cdot N$. In multi-core mode, because all FHE gate evaluations are in parallel, the total latency is $G$.

Table 21 shows latency and time complexity data for bitwise operation with respect to bit count N, in single core and multi-core mode.

| # of cores | Latency | Time complexity |
|---|---|---|
| Single core | $GN$ | $O(N)$ |
| Multi-core | $G$ | $O(1)$ |

Table 22: Bitwise ops scalability

**Conditional Flags.** Conditional flags calculation involves four sub-calculations: N flag, Z flag, C flag, and V flag. N flag and C flag calculations are trivial. Z flag calculation contains an OR reduction operation. In single core mode, the N-bit Z flag calculation has a latency of $G \cdot N$. In multi-core mode, the OR reduction is executed in parallel. N-bit OR reduction is broken down into $log(N)$ stages, and in each stage, the FHE OR gates are evaluated in parallel. The N-bit Z flag latency for parallel computing is $Glog(N)$.

The V flag calculation uses five gates regardless of the size of the input vReg. In single core mode, V flag calculation has $5 \cdot G$ latency. In multi-core mode because V flag calculation can be partially optimized with parallel computing, the latency is $3 \cdot G$. Therefore, total latency for single core mode is $G \cdot N + 5 \cdot G$, and total latency for multi-core mode is $G \cdot log(N) + 3 \cdot G$.

Table 23 shows latency and time complexity data for conditional flag calculation with respect to the bit count N, in single core and multi-core mode.

| # of cores | Latency | Time complexity |
|---|---|---|
| Single core | $GN + 5G$ | $O(N)$ |
| Multi-core | $Glog(N) + 3G$ | $O(log(N))$ |

Table 23: Conditional flags scalability

## 7.2 Scalability with respect to core count

CryptoEmu's performance is dictated by the extent to which it can rely on parallel computing. In ideal scenario, the processor has enough cores to enable any collection of tasks that can be performed in parallel to be processed as such. Sometimes, a part of the execution of an instruction takes place in parallel, but the rest is sequential because all CPU core resources are exhausted. In the worst case scenario, the processor has only one core and all computations are sequential/serialized.

Assume that the size of the functional units is 16-bit. Peak thread usage is then 32 threads, where each thread performance an FHE gate evaluation. On the server used in this study, which has 48 cores distributed over four CPUs, we vary the number of cores available to carry our emulation from 1 core to 48 cores. This amounts to a strong scaling analysis. We created scenarios where (*i*) the instruction set can draw on one core only and therefore the computation is sequential; (*ii*) the instruction set has insufficient cores and parallel computation is mixed with sequential computation; and (*iii*) the instruction set can count on sufficient cores to do all emulation in parallel.

**Adder.** Figure 18 shows the 16-bit adder's compute time with respect to core count. From the chart, the adder's latency decreases when the core count increases from 1 to 32. The adder reached its optimal speed at around 32 core count and has a slight decrease in speed when the core count exceeds 32. Adder's maximum speedup from parallel computing is around 7.78x.

**Subtractor.** Figure 19 shows the 16-bit subtractor's compute time with respect to core count. The 16-bit subtractor uses at most 32 cores to compute in parallel. The subtractor's latency decreases as the core count increases from 1-32, and reaches its maximum performance at around 32 core count. At and beyond 32 core count, all parallel optimizations are enabled. When the core counts saturate the subtractor's parallel computing requirement, the latency of the subtractor is around 700ms. The subtractor's maximum speedup from parallel computing is about 8.49x.

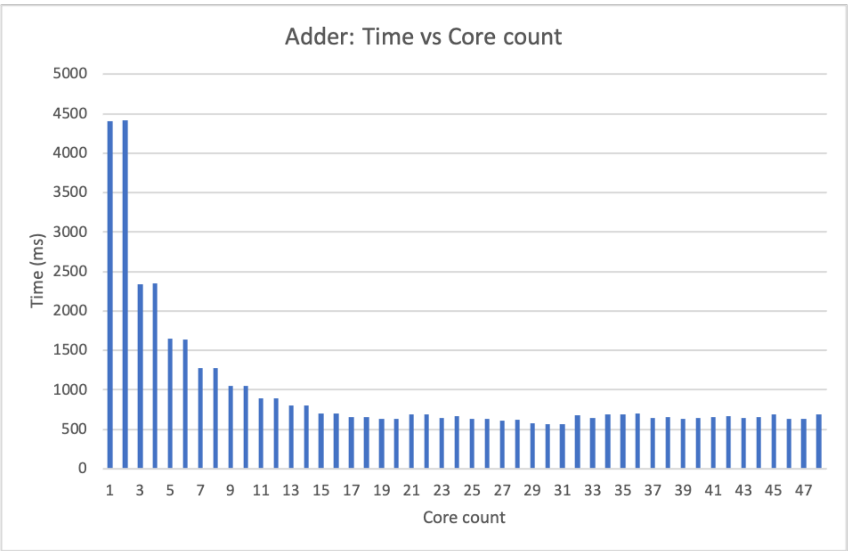


Figure 18: Time vs Core count

**Shifter.** Figure 20 shows the 16-bit shifter's compute time with respect to core count. A 16-bit shifter has a maximum core usage of 16. The 16-bit shifter's latency decreases as the core count increases from 1-16. The shifter's performance reaches its peak at core count 16. For core counts higher than 16, its stays stable at around 300ms. The maximum speedup yielded by parallel computing is around 10.94x.

**Multiplier.** Figure 21 shows the 16-bit multiplier's compute time with respect to core count. A 16-bit multiplier requires 32 cores to enable all parallel computing optimization. From the chart, both unsigned and signed multiplication's latency decrease as the number of cores increases from 1 to 32. The multiplier reached its top speed at around 32 cores. Beyond that, the multiplier's latency stays around 11000-12000ms. Unsigned multiplication has a maximum parallel computing speedup of 8.3x; signed multiplication has a maximum parallel computing speedup of 8.4x.

**Divider.** Figure 22 shows the 16-bit divider's compute time with respect to core count. A 16-bit divider requires 32 cores to achieve the best parallel computing performance. From the chart, divider's latency decreases as the core count increases from 1 to 32. From 32 cores on, the divider's latency stays at around 12000ms. The 16-bit divider has a maximum parallel computing speedup of 8.43x, reached when it can draw on 32 cores.

**Bitwise operations.** Figure 23 shows 16-bit bitwise operation's compute time with respect to the core count. 16-bit bitwise ops requires 16 cores to have the best parallel computing performance. From the graph, at core count 16 the bitwise operation reached its top speed. At 16 cores and beyond, the latency stays around 50ms. 16-bit bitwise op has a maximum

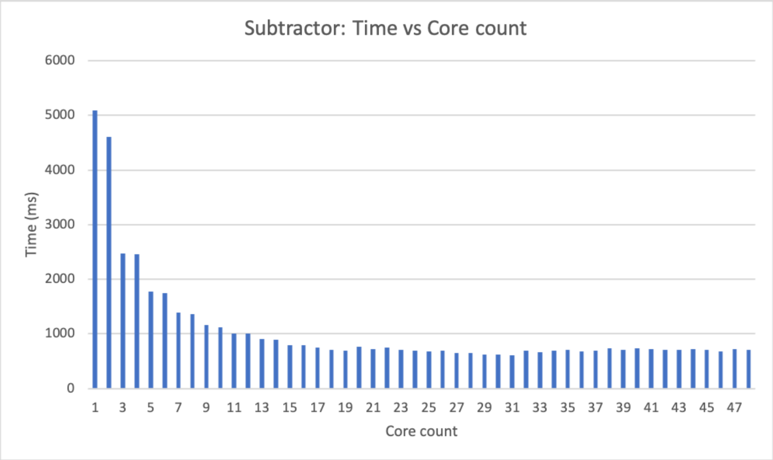


Figure 19: Time vs Core count

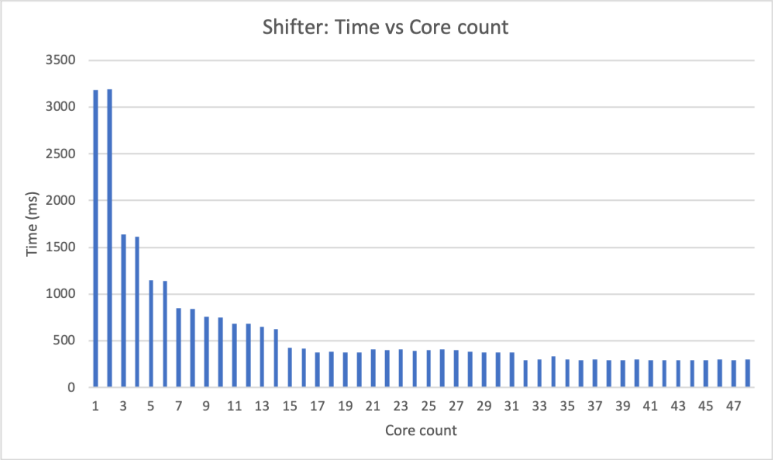


Figure 20: Time vs Core count

parallel computing speedup of 8.63x.

**Conditional Flags.** Figure 24 shows the 16-bit conditional flag unit's compute time with respect to core count. A 16-bit conditional flag unit runs a maximum of 18 threads in parallel. On the chart from core count 15 onward, the latency of the conditional flag unit hovers at around 120-130ms. The maximum parallel computing speedup is 4.7x.

## 7.3 Vulnerability

There are two source of vulnerability associated with the current CryptoEmu implementation: (*i*) the unencrypted HE instruction memory; and (*ii*) the branch resolution process.

Because HE instruction memory is in cleartext, the cloud knows exactly what assembly

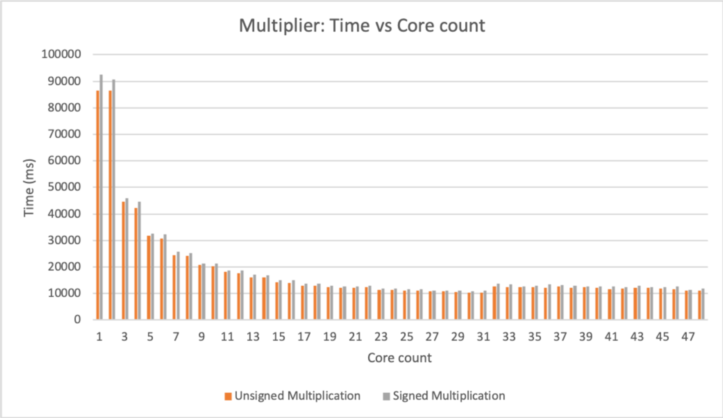


Figure 21: Time vs Core count

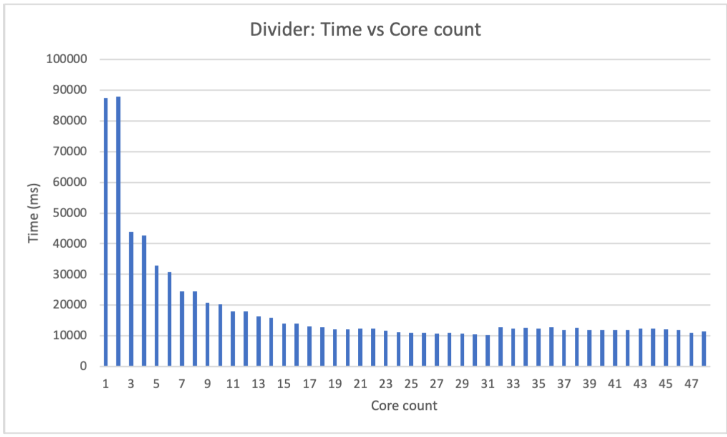


Figure 22: Time vs Core count

code is being executed. Assumption 2 assumes that the cloud is honest, so cloud visible instructions to the cloud do not pose a security issue. However, it is possible that the cloud's execution pipeline is compromised by attackers. If an attacker obtains control over the instruction set emulator, or performs a side-channel attack, user data can be breached.

If Assumption 2 is lifted, then the unencrypted HE instruction memory becomes a security loophole.Note that, we still assume the cloud will not start a denial of service attack on user. For example, if user send an encrypted number such as age, credit score or salaries, the cloud can also use binary search to actively query the value of an encrypted data using conditional unit implemented in CryptoEmu, and eventually find out the cleartext. User can counter the attack by sending HE instructions with anti-disassembly techniques, making it difficult for side-channel attack. However, this approach does not eliminate an active attack from cloud.

The cloud is able to actively find encrypted data's value because the cloud can abuse the branch resolution process. The problem can be solved by having the user assume control

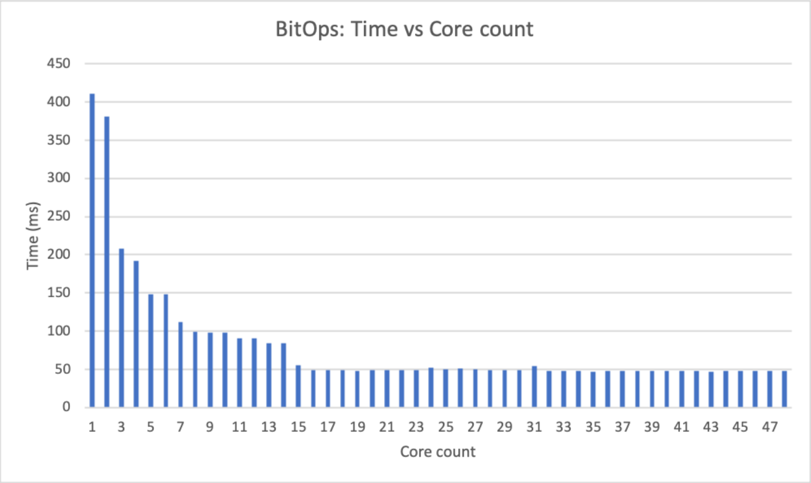


Figure 23: Time vs Core count

over the branch resolution process [15]. The user can ask for encrypted data from cloud, decrypt and process the data, send data back to cloud and inform the cloud which is the next instruction to fetch. With some extra overhead in data transmission, the user can make branch resolution unknown to the cloud and has full control over branch resolution.

## 7.4 Comparison against the state of the art

HELib [4] is one of the most widely adopted FHE software libraries. HELib implements the Brakerski-Gentry-Vaikuntanathan (BGV) scheme and supports basic homomorphic operations [21]. HELib has native support for binary negation, addition/subtraction, multiplication, binary condition, shift immediate, bitwise operation and binary comparison. CryptoEmu supports all these operation and a few more: left shift/right shift with encrypted amount, unsigned binary division, and NZCV condition flag computation. This section compares the performance of the operations supported by both HELib and CryptoEmu.

As reported in Table 24, for addition and subtraction, CryptoEmu's single core performance is about two times faster than HELib's. With sufficient cores, CryptoEmu yields maximum 18x speed up compare to HELib. For both signed and unsigned multiplication, CryptoEmu in single core mode is slightly slower than HELib. CryptoEmu yields a maximum 7x speed up in multiple core mode compare to HELib.

HELib supports logic left shift (LLS) with immediate. This operation is not computationally intensive compared to addition/subtraction. LLS with immediate is not implemented without parallel computing optimizations on CryptoEmu, and therefore single core latency is the same as multi-core latency. Because of the excellent support provided by TFHE library, LLS with immediate on CryptoEmu is about 132x faster than on HELib.

For bitwise operation, HELib's bitwise XOR is in fact 31x faster than CryptoEmu. This is expected because the implementation for HELib's bitwise XOR is not computationally

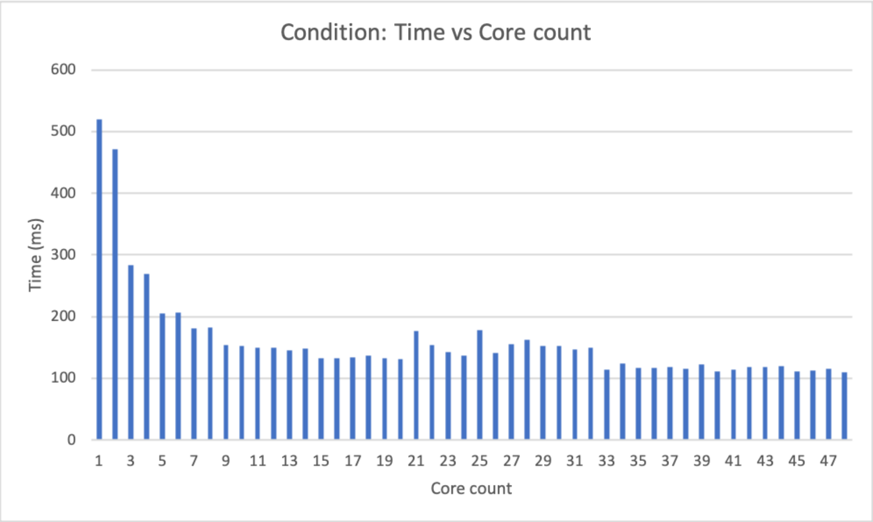


Figure 24: Time vs Core count

expensive. For bitwise OR and bitwise AND, CryptoEmu's single core speed is faster than HELib's. When maximum cores are enabled, CryptoEmu yields 16x speed up on bitwise OR and bitwise AND compared to HELib.

Bitwise NOT is not a computationally expensive operation on both CryptoEmu and HELib. Because bitwise NOT is implemented without parallel computing techniques, single core and multi-core performance are the same on CryptoEmu. By comparison, CryptoEmu is 151x faster than HELib on bitwise NOT operation.

For comparison/conditional operations, we benchmarked HELib's *CompareTwo()* function and CryptoEmu's conditional unit. HELib's compare routine compares two encrypted values and returns the greater/lesser number and comparison result. Our CryptoEmu compares two encrypted data and returns N, Z, C, and V flags. Because the Z-flag computation is on conditional unit's critical path, it is justified to compare latencies of two functionally equivalent operations that are on both unit's critical paths. CryptoEmu's comparison is much faster than HELib's in single core mode. When multiple cores are enabled, CryptoEmu yields maximum speedup of 42x.

# 8 Conclusion and future work

CryptoEmu successfully supports multiple homomorphic instructions on a multi-core host. With function units built upon the TFHE bootstrapped gate library, CryptoEmu is reconfigurable and reusable for general purpose computing. Through parallel computing algorithms that significantly reduces time complexity, CryptoEmu has a scalable implementation and achieves parallel computing speedup from 4.7x to 10.94x on functional units. Compare to HELib, CryptoEmu has maximum 18x speedup on addition/subtraction, 7x speedup on multiplication, 16x speed up on bitwise AND/OR, 42x speedup on comparison, 130x speedup

| Operation | HELib | CryptoEmu<br>(single core) | CryptoEmu<br>(multi-core) | Speedup |
|---|---|---|---|---|
| Addition | 10484.1ms | 4400.67ms | 566.149ms | 18.518x |
| Subtraction | 10962.2ms | 5088.57ms | 599.396ms | 18.289x |
| Multiplication (unsigned) | 69988.9ms | 86389.8ms | 10396.1ms | 6.732x |
| Multiplication (signed) | 81707.2ms | 92408.4ms | 10985.4ms | 7.438x |
| LLS (immediate) | 0.534724ms | 0.0040335ms | 0.0040335ms | 132.571x |
| Bitwise XOR | 1.52444ms | 416.013 ms | 47.1986ms | -30.9613x |
| Bitwise OR | 771.12ms | 416.146 ms | 47.6171ms | 16.194x |
| Bitwise AND | 756.641ms | 411.014ms | 47.6001ms | 15.90x |
| Bitwise NOT | 1.8975ms | 0.012508 ms | 0.012508 ms | 151.703x |
| Comparison | 4706.07ms | 519.757ms | 110.416ms | 42.62x |

Table 24: HELib vs CryptoEmu

on left shift with immediate, and 151x speedup on bitwise NOT.

Scalability of CryptoEmu can be further improved. The current design only supports single instruction set emulator process that runs on multiple cores. Therefore maximum core count is bounded by number of cores on one CPU. With more cores available, CryptoEmu can draw on more parallelism through pipelining, multiple instruction issue and dynamic instruction scheduling. Another aspect that could provide further speed improvements is the use of AVX instructions.

# References

[1] J. Clement, "Cyber crime: number of breaches and records exposed 2005-2020." `https://www.statista.com/statistics/273550`, 2020. Accessed on 2020-Dec-01.

[2] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption library," August 2016, Accessed on 2020-Dec-01. https://tfhe.github.io/tfhe.

[3] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *international conference on the theory and application of cryptology and information security*, pp. 3–33, Springer, 2016.

[4] S. Halevi and V. Shoup, "An implementation of homomorphic encryption," 2013, Accessed on 2020-Dec-17. https://github.com/shaih/HElib.

[5] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A survey on homomorphic encryption schemes: Theory and implementation," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–35, 2018.

[6] C. Gentry, “Fully homomorphic encryption using ideal lattices,” in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pp. 169–178, 2009.

[7] 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.

[8] 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*, pp. 617–640, Springer, 2015.

[9] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, “Faster packed homomorphic operations and efficient circuit bootstrapping for tfhe,” in *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 377–408, Springer, 2017.

[10] S. Halevi and V. Shoup, “Design and implementation of a homomorphic-encryption library,” *IBM Research (Manuscript)*, vol. 6, pp. 12–15, 2013.

[11] 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.

[12] 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.

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

[14] N. G. Tsoutsos and M. Maniatakos, “Heroic: homomorphically encrypted one instruction computer,” in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, IEEE, 2014.

[15] F. Irena, D. Murphy, and S. Parameswaran, “Cryptoblaze: A partially homomorphic processor with multiple instructions and non-deterministic encryption support,” in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 702–708, IEEE, 2018.

[16] OpenMP, “Specification Standard 5.1.” Available online at `http://openmp.org/wp/`, 2020.

[17] K. Vitoroulis, “Parallel prefix adders,” *Concordia university*, 2006.

[18] A. D. Booth, “A signed binary multiplication technique,” *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951.

[19] C. Terman, "6.004 computation structures." MIT OpenCourseWare, https://ocw.mit.edu, Spring 2017.

[20] geeksforgeeks.org, "Non-restoring division for unsigned integer." https://www.geeksforgeeks.org/non-restoring-division-unsigned-integer, 2018, Accessed on 2020-Dec-09.

[21] S. Halevi and V. Shoup, "Algorithms in helib," in *Annual Cryptology Conference*, pp. 554–571, Springer, 2014.