Privacy-preserving Program Analysis with Homomorphic Encryption

Yujie Wang*, Shuai Wang*,
*Hong Kong University of Science and Technology

Abstract—**Program analysis is a widely used technique in malware detection or other security aspects of computer programs. However, there is a concern that existing commercial program analysis services may leak user privacy given that users usually have to upload their software samples onto the service providers for program analysis. To prevent leakage of privacy, in this paper, we demonstrate the feasibility of performing analysis of encrypted programs by adopting Homomorphic Encyption to existing program analysis techniques. More specifically, we explore the feasibility of performing both static and dynamic analysis of encrypted programs. Moreover, we proposed a scheme called *privacy-preserving processor*, which enables execution on encrypted machine instructions, to dynamically analyzing encrypted programs.**

1. Introduction

Program analysis, a process of automatically analyzing the behavior of programs regarding properties such as safety, correctness and robustness, is widely used in malware detection or other security aspects of computer programs. To date, we have observed a number of (commercial) program analysis services hosted on cloud platforms and provide an API for user to upload their software samples for check. Typical services include VirusTotal [1], Scitools [2], VISUAL EXPERT [3]. However, we have a general concern that such program analysis services are prone to leaking user privacy, given that users usually have to upload their software samples, either in source code or binary code, onto the platforms for bug detection or other inspection. As a result, the implementation of software samples might be leaked to the platforms. Overall, we see a strong and urgent need to ensure the confidentiality of the analyzed software, as a piece of sensitive information uploaded to the cloud.

Motivated by this observation, this paper aims to demonstrate the feasibility of a privacy-preserving program analysis scheme which allows users and program analysis tools to do the followings: users preprocess codes, encrypt the preprocessed programs and upload the encrypted preprocessed programs to program analysis tools, then program analysis tools do computation entirely on the ciphertext without knowing the information from users uploaded programs.

In particular, we show the feasibility of privacy-preserving setting of two static analysis methods that are, *signature-based detection* and *code-embedding-based similarity analysis*. Moreover, we explore the feasibility of dynamic analysis of encrypted program and proposed a design of executing encrypted machine instructions which we call it *privacy-preserving processor*.

To adopt privacy-preserving setting to the two static analysis methods mentioned above, we envision that three phases are needed. The first phase, *program preprocessing*, processes software of either source code or binary code formats. The second phase, *encryption*, converts the preprocessed programs into cipher-text and uploads to the remote server for analysis. The third phase, deployed on the remote service, conduct program analysis directly over the encrypted cipher-text.

The first phase varies for different analysis methods. As for the second phase, we adopt recent breakthroughs in Homomorphic Encryption (HE). We report that latest research in this field has applied HE to various computation models and among these works, one noticeable ones is Fast Fully Homomorphic Encryption Scheme Over The Torus (TFHE) [4] which enables bit-by-bit encryption. In this paper, we refer to this HE method, and develop an end-to-end scheme for privacy-preserving program analysis. As for the third phase, we build computation models based on multiplication and addition operations, which supported by HE, to perform aforementioned static analysis methods entirely on the ciphertext. Moreover, we illustrate the proposed privacy-preserving static analysis by taking Cod2Vec [5], a code embedding model, as an example and similarity analysis, a type of program analysis, as the task to realize code-embedding-based similarity analysis of encrypted programs.

Beyond static analysis, we further explore the feasibility of dynamically analyzing encrypted programs. This task is more challenging than statically analyzing encrypted programs because dynamic analysis of encrypted programs requires to execute encrypted programs while the static one takes encrypted programs as input for computation models. Therefore, we proposed a scheme of executing encrypted machine instructions and we call it *privacy-preserving processor*. As the name implies, this scheme mimics a processor and it is built based on addition and multiplication operation supported by HE. To elaborate this scheme, we convert a simple single cycle CPU proposed by Patterson and Hennessy [6] to a privacy-preserving processor. In addition, this privacy-preserving processor can serve as a more general approach to perform any computation models given that computation models can be transformed into a computer program.

In this paper, our contributions can be summarized as below:

- Combine program analysis with Homomorphic Encryption to perform privacy-preserving program analysis.
- Show the feasibility of privacy-preserving setting of two static analysis methods that are signature-based detection and code-embedding-based similarity analysis.
- Explore the feasibility of dynamic analysis of encrypted program and proposed a scheme called privacy-preserving processor, which enable execution on encrypted machine instructions.

2. Background and Related Work

In this section, we review works related to program analysis and Homomorphic Encryption for later discussion about combining them two to realize privacy-preserving program analysis.

2.1. Program Analysis

Program analysis, a process of automatically analyzing the behavior of programs regarding properties such as safety, correctness and robustness, is widely used in malware detection or other security aspects of computer programs. Different program analysis techniques were proposed and those techniques can be divided into two categories: *Static Analysis*, which performs analysis without executing the program, and *Dynamic Analysis*, which analyzes the program during runtime execution.

2.2. Static Analysis

. There are many static analysis techniques developed to extract information in a program. A typical way is to analyze control-flow, data-flow, raw codes and other aspects of the program. Among existing static analysis techniques, we focus on converting signature-based detection and code embedding into a privacy-preserving mode.

2.2.1. Signature-Based Detection. Signature-based detection is a widely used approach to detect the presence of malware. Based on the assumption that the presence of malware can be described through patterns (also called signatures), the malware can be detected by identifying the corresponding signature. In the case of a virus scanner, the pattern may be a unique pattern of codes that attaches to a file, or it may be the hash of a known bad file.

A simple implementation of signature-based detection is to find a match of target patterns, which can be signatures of a virus, from a sequence, which can be the binary codes of a suspicious software. As malware became more sophisticated, the signature of a malware may change each time the object spread from one system to the next. As a result, various signature-based detection techniques were proposed to identify more sophisticated malware.

2.2.2. Code Embedding. We first present a general introduction about code embedding techniques. While our example will be focusing on machine code (since binary code embedding is generally more desired for security applications), we note that the procedure is indeed applicable to source code embedding as well. As depicted in Figure 1,

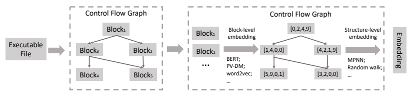

Figure 1. Code Embedding

modern software embedding techniques comprise a three-step procedure: the input of the pipeline is a binary executable, where the pre-process module first disassembles the binary representation into assembly instructions. Then, the control flow graph (CFG) is recovered over the assembly program and feed the graph representation into the second module. The two modules collaborate to convert instruction, basic block, and structure-level information into numerical representations. The final output of our DNN component is a graph embedding (i.e., numerical vectors), which delivers a numerical representation of each input program. Well-trained graph embedding subsumes rich information of the embedded graph into the numerical vectors; two graphs are deemed as “similar” when the cosine distance of their corresponding embedding vectors is short. We now elaborate on each step in details.

Block-Level Embedding. For each recovered basic block on CFG, the basic block-level embedding module strikes to convert the block into a numerical representation. The general idea behind basic block-level embedding, is to treat instructions within each basic block as a “paragraph” p, where each machine instruction deems a sentence i. Tokens, including both assembly instruction opcodes and operands in each basic block, are deemed as words. One de facto research tool leverages the PV-DM model [7]. It trains an embedding model by iteratively mapping the current paragraph p and a set of assembly instructions $i \in I$ into several vectors and use the average of these vectors to predict one target assembly instruction i_t. I is usually picked as several instructions surrounding i_t. The back-propagated prediction error will be used to update vectors extracted from p and $i \in I$. We also observed related research which leverages the standard BERT model [8], by masking tokens to pre-train on the masked language model task (MLM), and extract all neighboring blocks of each basic block to pretrain on the adjacency node prediction task (ANP). The ANP task, by subsuming a block’s neighboring information, is also conceptually similar to the next sentence prediction (NSP) task in the standard implementation of BERT.

Structure-Level Embedding. While the aforementioned procedure adequately extracts basic block-level embeddings, program structure-level information is not yet recovered. At this step, graph neural networks (GNN) is commonly adopted to extract program structure-level embedding from a holistic perspective [9]. GNN has been demonstrated to be highly effective for representation learning of graph structures (e.g., CFG).

For instance, the popular GNN design, message passing neural networks (MPNN) [10], has been shown as highly effective to compute the graph-level embeddings. The standard MPNN implementation has two phases, including a message passing phase and a readout phase. The message-passing phase updates the hidden states at each node of the graph based on its neighboring nodes. This process could take several iterations, where the extracted numerical representation at each node could be gradually updated until reaching a fixed point or a pre-defined threshold. Then, a readout phase (also referred to as aggregation phase) computes a whole graph feature vector with a predefined readout function and yield the final embedding output.

State-of-the-art GNN models have been proposed and have achieved promising results on graph classification tasks. Recent research has revealed that a well-trained GNN model exhibits expressive power comparable to Weisfeiler-Lehman graph isomorphism test [9]. Similarly, recent work has also adopted random walk methods to extract sequences longer than basic blocks to facilitate learning. It is shown that multiple random walks will put a higher probability to cover basic block that dominate others [7]. Considering that popular blocks can be the indicator of loop structures or cover important branching conditions, using random walks deem a natural way to prioritize basic blocks that dominate others and extract a holistic understanding of the program structures.

2.3. Dynamic Analysis

Compared with static analysis, dynamic analysis is performed by dynamically executing a target program. With the aim of identifying potential vulnerability or bugs in the target program, the target program is executed with sufficient test inputs to perform analysis by observing both the behaviours during execution and outputs produced in the final stage. Different techniques of dynamic analysis were developed to serve different purposes such as code coverage analysis, fuzz testing, dynamic taint analysis, etc. In general, dynamic analysis techniques reason over the run-time behavior of a program and record its dynamic state, which is also called as profile or trace. During program execution, a program profile measures events like lines of code, basic blocks, control edges, routines etc. Generally, there are two phases in the process of dynamic analysis. The first phase is program instrumentation and profile/trace generation while the second phase is analysis or monitoring. As for program instrumentation, additional statements are inserted into the program for the purpose of generating traces and these instrumented statements are executed during the running time. The analysis or monitoring can be performed either during execution or after execution for the purpose of finding potential vulnerabilities in the program.

2.4. Software Similarity Analysis

Software similarity analysis determines how similar two pieces of programs are, and it is critical in many software security applications. For instance, malware analysis identifies malware samples that exhibit similar behavior in order to reveal malware families, and, therefore, to eliminate the need for re-analyzing known malware executables [11], [12]. Patch-based software exploitation compares the pre-patch and post-patch software to pinpoint vulnerabilities fixed by the patch and automatically synthesize exploitations towards the pre-patched software [13]. In addition, software similarity analysis is also the basis of many software code comprehension tasks, including code cloning and plagiarism detections [14], [15].

Conventional techniques for software code similarity analysis leverage a set of program syntactic features for comparison, such as distributions of instructions, opcode sequences, system calls, and control-flow graph techniques [16], [17], [18], [19], [20]. The de facto industrial standard tool BINDIFF identifies similar functions mostly through graph isomorphism comparison [18]. This algorithm detects similar functions by comparing the control flow and call graphs. Several existing studies leverage dynamic analysis and software birthmarks for similarity analysis [21], [22], [23], [24], [25], [26], [27]. However, an obvious issue for existing dynamic analysis is the low coverage of test targets (e.g., functions), rendering it generally unsuitable for production environments.

To date, there are *two major lines* of research that are revamping software code similarity analysis: 1) **learning-based prediction**, which learns an embedding representation from software by reusing advanced artificial intelligence (AI) techniques in this new setting and uses the embedding for similarity prediction, and 2) **semantics-based proof**, which performs rigorous proof of program equivalence and similarity by using formal methods (e.g., symbolic execution).

One actively-developed line of research leverages AI techniques, including both machine learning and deep learning models for similarity analysis [7], [28], [29], [5], [30], [31], [19], [32]. The state-of-the-art techniques essentially learn *code embeddings* (e.g., a graph), continuous learnable vectors for representing source code or binary executables. The advantage of such deep learning-based techniques is that they can learn a (close-to) optimal embedding for a program, as a neural network can be trained end-to-end without having much domain knowledge. To train an embedding representation, most of the existing research leverages program syntactic information, as well as data and control dependencies [33], [34], [35], [36], and the learned embedding representation is further used for similarity prediction. Research works in this direction primarily takes a "fuzzy" reflection on the high-level source code or instructions, and, therefore, is usually flexible and scalable, promoting the efficient analysis of large-scale software samples, although usually only syntax-level information (e.g., number of stack operations, number of unconditional jumps), lightweight data and control dependencies are used for learning.[1] In other words, these approaches are fundamentally incapable of handling more diverse program representations due to compiler optimizations and program obfuscations which can largely change the program control structures (e.g., via control-flow flattening [40]). Although recent research has made certain improvements with respect to code obfuscation [7], there still exists considerable space for improvement. In summary, we interpret that learning-based techniques may miss many software samples with drifted syntax representations or control/data structures but that are indeed similar or identical with respect to program semantics.

Another line of research seeks to extract program semantics information as a rigorous proof of whether two pieces of programs are equivalent or not. Typically, formal methods such as symbolic execution techniques are used to reason out semantics information in terms of symbolic formulas [41], [42], [39], [43], [44], [45]. SMT-based constraint solving techniques are then used to check the equivalence of symbolic formulas, which serves as a strong and rigorous evidence for the equivalence of two programs. While the proposed technique gives rigorous proof of the equivalence of programs and can potentially defeat optimization and obfuscation to a certain degree (since compiler optimization and code obfuscation do not change program semantics), formal methods such as symbolic execution are usually quite *slow*, due to the path explosion problem and challenges in solving complex constraints. Therefore, while such formal methods can yield highly precise results even with respect to obfuscated and optimized code in the real world, their limited scalability is a key issue in production environments.

2.5. Homomorphic Encryption (HE)

This section gives a high-level introduction of HE. From a holistic perspective, HE denotes an encryption scheme which supports the following critical property:

$$\delta(f(\epsilon(x, k_{pub}), \epsilon(y, k_{pub})), k_{pri}) = f(x, y)$$

where ϵ is an encryption function, δ is the decryption function, k is key and f, either addition or multiplication, is the function to be evaluated.
There are two types of Homomorphic Encryption, Fully Homomorphic Encryption (FHE) and Leveled Homomorphic Encryption (LHE) [46]. The key difference between these two HE is the frequency of performing *bootstrapping*, a important step to guarantee the correctness of HE operations by reducing the noise in ciphertext. FHE performs bootstrapping every HE operation while LHE performs bootstrapping after a number of HE operations (Leveled). As a result, LHE is generally faster than FHE but requires to be monitored the state of operations. In this paper, we only use FHE to design our scheme and refer FHE as HE by default.

2.5.1. Fast Fully Homomorphic Encryption Scheme over the Torus (TFHE). A milestone of HE is the encryption scheme proposed by Gentry [46], which introduced the important concept of bootstrapping and proved that Fully Homomorphic Encryption (FHE) is achievable in polynomial time. Fully Homomorphic Encryption is a scheme that supports computing addition and multiplication over encrypted data for arbitrary times. However, the first proposed FHE scheme suffers from low efficiency and high memory consumption. Later, variants of FHE are proposed to enhance the efficiency, gradually enabling a practical adoption of FHE in real-world scenarios [47], [48], [49], [50], [51], [52], [53]. In this research, we employ a recently-developed HE scheme, TFHE [4], to encrypt program embeddings. TFHE generalizes and primarily enhances the efficiency of FHE scheme on the basis of GSW and its ring variants. TFHE has been shown as highly efficient and has been used as the basis to do fast and accurate Convolutional Neural Network (CNN) inference over encrypted data [54]. Therefore, we choose TFHE to be our starting point in developing end-to-end privacy-preserving progress analysis tools.

3. Signature-Based Detection with Homomorphic Encryption

Signature-based detection is one of fundamental anti-malware approaches that detects the presence of a malware. Based on the assumption that malware can be described through patterns (also called signatures), this approach identifies the malware by matching code patterns of the software in question with the database of signatures of known malicious programs, also known as blacklists. One setting of the problem of detecting signatures assembles a pattern matching problem that aims to determine whether a pattern $\mathcal{P}$ of length m is contained in a given sequence $\mathcal{S}$ of length n.

3.1. Method

Consider a scenario that a user uploads his software to an online platform and the platform performs signature-based malware detection to find a match between the software (sequence $\mathcal{S}$) and malware signatures (pattern $\mathcal{P}$) stored in the database. To adopt pattern matching to a privacy-preserving mode, the user first encrypts programs, then the platform performs pattern matching over encrypted programs with Homomorphic Encryption.

1. From a rigorous perspective, "semantics" is slightly **overused** in this line of research. While some existing works state to capture "semantics" from input program, they are essentially control structures and intermediate data flow facts (e.g., two instructions that access the same memory location). Such information is indeed not resilient towards nontrivial program transformations and obfuscations. From a holistic perspective, program semantics are deemed as the *meaning* of a program, which can be well represented by the input/output relations of code fragments. For instance, we usually describe the "semantics" of an array sorting program as the output array is a sorted version of the input array, instead of noting some intermediate data flow facts which may have subtle changes across different implementations. Unfortunately, we only observed a few research works leveraging input/output relations for code similarity analysis [37], [38], [39].

There are various pattern matching algorithms and we list some of them in Table 1. Among these algorithms, some of them use tricks like hash function and tables to speed up the process. However, many of these tricks become inefficient when we adopt them to a privacy-preserving mode with Homomorphic Encryption. For instance, looking up a position given an index are usually inefficient because the index is ciphertext rather than plaintext. According to our knowledge, the naive string-search algorithm (or brute force method), whose matching time is $O(mn)$ where n and m are the length of sequence $\mathcal{S}$ and pattern $\mathcal{P}$ respectively, is the fastest pattern matching algorithm that is compatible with the multiplication and addition operations supported by Homomorphic Encryption.

TABLE 1. PATTERN MATCHING ALGORITHMS

Algorithm	Preprocessing time	Matching time
Naive string-search algorithm	none	$\Theta(mn)$
Rabin–Karp algorithm [55]	$\Theta(m)$	average $\Theta(n+m)$, worst $\Theta((n-m)m)$
Knuth–Morris–Pratt algorithm [56]	$\Theta(m)$	$\Theta(n)$
Boyer–Moore string-search algorithm [57]	$\Theta(m)$	best $\Omega(n/m)$, worst $O(mn)$
Bitap algorithm [58]	$\Theta(m)$	$O(mn)$
Two-way string-matching algorithm [59]	$\Theta(m)$	$O(n+m)$

4. Code Embedding Similarity Analysis with Homomorphic Encryption

In this section, we proposed a scheme of program similarity analysis with encrypted code embeddings. In detail, a program is first converted to numeric representation which are usually fixed-length vectors, then the numeric representations of programs are encrypted and similarity analysis are conducted on these encrypted numeric representations. To conduct similarity analysis on encrypted numeric representations, Homomorphic Encryption is used during computations over encrypted values. In the following sections, we take Cod2Vec, a code-embedding-based program analysis framework, as an example to demonstrate the feasibility of code embedding similarity analysis with Homomorphic Encryption.

4.1. Scenario and Threat Model

Before going into the detailed design of program similarity analysis with encrypted code embeddings, we first define the threat model, i.e., the capability of each active party involved in the context.

- **Platform providers**: *honest-but-curious* adversaries. The platform providers follow the protocol honestly, but would like to (stealthily) explore the content of the submitted software samples. In short, the platform providers only get access to the public key and take ciphertext (encrypted from the program embedding vectors) as the input of its analysis.
- **Users**: a *honest* user will follow the protocol to convert her sample programs into numerical vectors and encrypt with his public key. She will never reveal her private key to the public.

From a holistic view, this research will assure that at the end of the analysis, the platform providers would learn nothing beyond their own analysis outputs.

4.2. Method

We first briefly review the setting of Cod2Vec. Cod2Vec transforms a program in to a vector in $\mathbb{R}^{384}$ named *code embedding*, then use a `softmax` layer to output a probability vector in $\mathbb{R}^{214041}$. To convert the framework of Cod2Vec to a privacy-preserving mode, the $\mathbb{R}^{384}$ code embedding vector obtained in the first phase is encrypted and then is used to compare the similarity. We choose `cosine` as the metric to compare the similarity between two given code embedding vectors, which means we compute the following:

Given two code embeddings x and y, we measure

$$cos(x, y) = \frac{x}{||x||_2} \cdot \frac{y}{||y||_2}.$$

The above formula is easy to compute over HE because $\frac{x}{||x||_2}$ and $\frac{y}{||y||_2}$ could be computed locally without HE, so the only operations involved HE is the inner product which are multiplications and additions.

4.3. Observation

We first compared pair-wise similarity among code embeddings of 368445 programs, and each of programs belongs to one of 214041 groups. As shown in Table 2, we observe that the average intra-group similarity (0.567) is significantly greater than the average inter-group similarity (0.011), so the similarity between two programs is reflected from the similarity between corresponding code embeddings.

Next, we measure the computational cost of comparing the similarity between two code embeddings with a privacy-preserving mode. The privacy-preserving mode is supported by TFHE, and a value is encrypted into a n bit ciphertext while $\frac{n}{2}$ representing the integer part and $\frac{n}{2}$ representing the decimal part respectively. In Table 3, we list the computational cost of similarity comparison between two code embeddings for different bit length n. According to the result, the computational cost grows in the order of $O(n^2)$.

TABLE 2. COD2VEC SIMILARITY COMPARISON

	Intra-group	Inter-group
Average Similarity (cosine)	0.567	0.011

TABLE 3. COMPUTATIONAL COST OF DIFFERENT TFHE SETTINGS

Precision (bit length)	2	4	8	16
Bit length for integer	1	2	4	8
Bit length for decimal	1	2	4	8
Time (sec.) per dot product	155.4	614.3	2697.1	10919.1

TABLE 4. INSTRUCTIONS SUPPORTED BY THE SINGLE-CYCLE CPU IN FIGURE 2

NAME	INSTRUCTION	FORMAT	OPCODE/FUNCT(Hex)
Add	add	R	0 / 20
Add Immediate	addi	I	8
And	and	R	0 / 24
And Immediate	andi	I	c
Branch On Equal	beq	I	4
Jump	j	J	2
Load Word	lw	I	23
Or	or	R	0 / 25
Or Immediate	ori	I	d
Set Less Than	slt	R	0 / 2a
Store Word	sw	I	2b
Subtract	sub	R	0 / 22

5. Dynamic Analysis with Homomorphic Encryption

To perform dynamic analysis of encrypted programs, the problem of executing encrypted programs is the major challenge because the execution of a program involves more complicated operations compared with matrix multiplication and addition. Therefore, in this section, we explore the feasibility of executing encrypted programs and we proposed a scheme of executing encrypted machine instructions. In detail, this scheme simulates a single-cycle processor which manipulates registers and memory cache according to machine instructions encrypted with HE, therefore, we call this scheme as *privacy-preserving processor*. To explain the proposed scheme, we take a simple Patterson & Henness single-cycle processor [6] as shown in Figure 2 as an example, which supports instructions in Table 4. Other processors like *MIPS*, *ARM* and *x86* follow the similar idea.

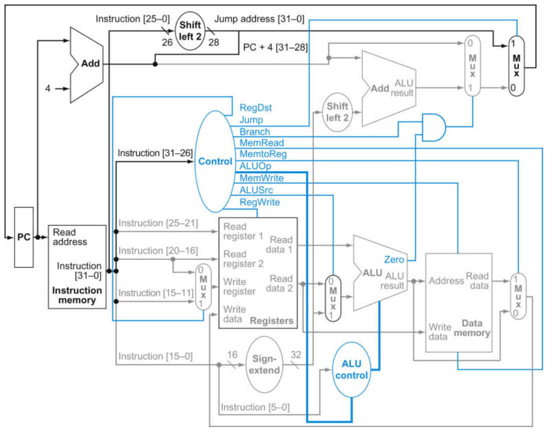

Figure 2. Patterson & Hennessy Single Cycle CPU

5.1. Scenario and Threat Model

To formulate the problem of dynamic analysis of encrypted programs, we first impose the following constrains on the program to be analyzed:

- The instructions of program to be analyzed are supported by Patterson & Henness single-cycle processor as shown in Figure 2, and the instructions supported are listed in Table 4.
- The addressing spaces of instructions and memory (which stores data) are isolated from each other and is contiguous with the size of power of 2. Therefore, we denote the range of the addressing space of *instructions* and *data memory* as $[0, 2^{L_i} - 1]$ and $[0, 2^{L_d} - 1]$ respectively.

After making the above assumptions, the problem of dynamic analysis of encrypted programs can be formulated as follows:

- **The user** first zero-pads the data memory and instructions into sizes of 2^{L_d} and 2^{L_i}, respectively. Secondly, he encrypts instructions, data memory separately, and encrypts the initial values of registers. Thirdly, the encrypted instructions, data memory and registers are sent to the platform.
- **The platform** executes the encrypted programs with different inputs, analyzes (in a way according to different tasks) and returns the results (or running states) to the user.
- **The user** decrypts the received results (or running states).

5.2. Privacy-Preserving Processor

To execution an encrypted program, we proposed a scheme called *privacy-preserving processor* that simulates Patterson & Henness single-cycle processor. Similar as Patterson & Henness single-cycle processor, our privacy-preserving processor contains the following components:

1) **Program Counter (PC)**.Fetch the instruction in term of the (encrypted) value of *PC*.
2) **Instruction Cache**. Store read-only (encrypted) instructions.
3) **Registers**.Read or write the (encrypted) values of registers according to an (encrypted) instruction.
4) **Arithmetic-Logic Unit (ALU)**. Performs arithmetic and bitwise operations on (encrypted) binary numbers.
5) **Data Memory Cache**. Read or write (encrypted) data given an input value and address.
6) **Control Unit**. Generate (encrypted) control signal.

We observe that a Patterson & Henness single-cycle processor can be regarded as a combination of `AND, OR` and `NOT` gates. Given that TFHE supports both multiplication and addition over encrypted inputs, we can build encrypted `AND, OR` and `NOT` gates that take encrypted bits as inputs and output the encypted result, which we call privacy-preserving gates (PPG). Therefore, to build a privacy-preserving processor, we can replace the normal boolean gates with encrypted gates built by TFHE multiplication and addition. Next, we illustrate details of the processor.

5.2.1. Memory (cache) and Registers. In normal processor, a memory unit or register can be implemented through flip flops as shown in Figure 3. A value in a flip flop is the output of the gate which is held by latches and controlled by input signals. Given that our PPGs are simulated, holding a value in our scheme does not require latches. Instead, only input signals are incoorperated into a memory unit or register, which can be implemented as a combination of boolean circuits. Hence, the registers and memory units in our privacy-preserving processor can be simulated by PPGs. Moreover, there is no concept of memory hierarchy, which is widely incorporated into modern processors for the purpose of fast memory access, because, according to our knowledge, we find that such structure does not contribute to the efficiency of the process. As a result, all encrypted instructions and data are stored in instruction (cache) and memory (cache) in the processor.

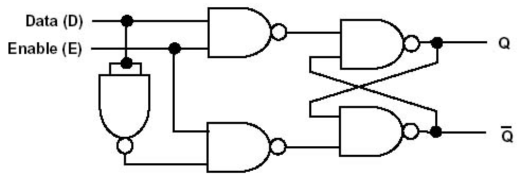

Figure 3. Flip Flop

5.2.2. Access Memory and Registers. To access or write a memory unit or register in term of a given address, a typical pipeline is to first use a multiplexer to decode the given address, as shown in Figure 4, then the write or read control signal is sent to the target register or memory unit so that values are read from or written to the desired location. A multiplexer or decoder is also a combination of gates. Therefore, the above pipeline can be simulated by PPGs, as a result, memory and registers can be accessed given a encrypted address.

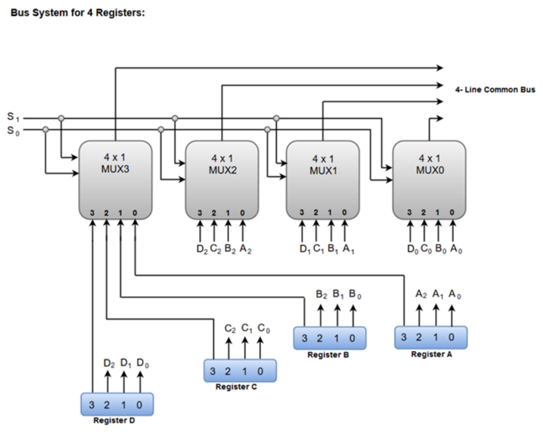

Figure 4. Access Registers

5.2.3. Arithmetic-Logic Unit (ALU). ALU is a combinational digital circuit that performs arithmetic and bitwise operations on integer binary numbers. To perform an operation, an ALU typically first computes all kinds of operations, then use a demultiplexer to select the target operation. Given that ALU is a combination of logic gates, the ALU in our proposed privacy-preserving processor has the similar structure as the ALU in a Patterson & Henness single-cycle processor as shown in Figure 5 with logic gates being replaced with PPGs.

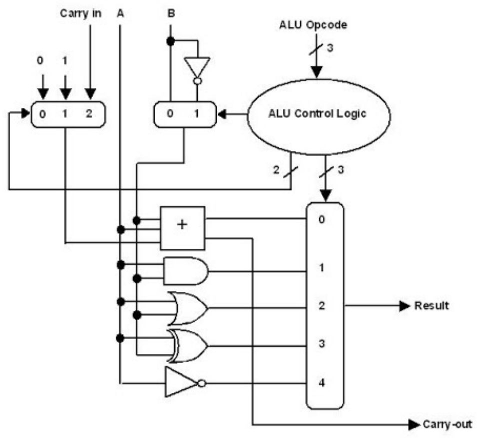

Figure 5. Arithmetic-Logic Unit

5.2.4. Control Unit. Since there are multiple datapaths in the pipeline of a processor, to select the correct datapath in term of the given instruction, a control unit is needed. In the case of MIPS32 processor, control unit is a combination of logic gates that takes `op` and `function` code, as shown in Figure 6, as input and generates control signals. Therefore, in our privacy-preserving processor, the control unit that takes encrypted instruction as input and generates encrypted control signals has the same structure as the one in a Patterson & Henness single cycle processor.

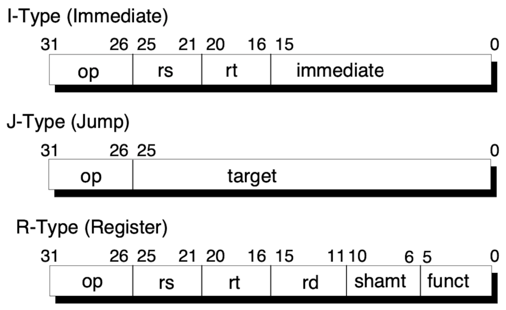

Figure 6. MIPS32 Instruction Formats

5.3. Optimization of Privacy-Preserving Processor

Given that Homomorphic Encryption is inherently computational expensive, simulating one privacy-preserving gate takes much longer time than a normal logic gate. Since one processor consists of a large amount of gates, optimization of our proposed privacy-preserving processor is demanding. Therefore, we propose the following suggestions.

- **Instruction Pipelining**. Modern processors are typically pipelining instructions to keep every part of the processor busy with some instructions by dividing incoming instructions into a series of sequential steps. However, instruction pipelining carries data dependence related problems (hazards), to solve harzards, another unit called harard detection unit is added to the processor, which is a combination of logic gates that stalls the pipeline until the hazard has passed. Therefore, a harzards detection unit built by PPGs can be added to our privacy-preserving processor to make the privacy-preserving processor pipeline instructions.
- **Parallelization**. To speed up the privacy-preserving processor, we can create hardware threads for each privacy-preserving gate. Plus, hardware supported optimization can be investigated such that the HE algorithms which simulates PPGs are supported by carefully designed hardwares.

5.4. Privacy-Preserving Processor as a Generalized Computation Model

Beyond executing encrypted instructions, privacy-preserving processor can act as a generalized computation model. This is because many computation models and algorithms are transferable to computer programs. Therefore, a large number of tasks that concern about leakage of privacy, besides analysis of encrypted programs, can be performed on the privacy-preserving processor.

6. Conclusion

In this paper, we combine program analysis with Homomorphic Encyption to perform analysis of encrypted program and we investigate both static analysis and dynamic analysis. However, there is still limitations in our research. The most significant one is computational cost. As shown above, operations performed with HE is much more expensive than operations without HE. Part of the reasons is that, only Fully Homomorphic Encryption is incorporated while Leveled Homomorphic Encryption is not incorporated in our proposed scheme. Therefore, future research can investigate adopting Leveled Homomorphic Encryption to accelerate computation. Additionally, parallelization can be implemented to make the process faster given that some operations are independent on others. As for our proposed privacy-preserving processor, specially-designed hardware may further improve the performance and domain-specific instruction pipelining is also another direction.

References

[1] V. Total, "Virustotal-free online virus, malware and url scanner," *Online: https://www. virustotal. com/en*, 2012.

[2] "Static analysis tools for fast code comprehension."

[3] "Visual expert pricing."

[4] 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.

[5] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[6] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design ARM Edition: The Hardware Software Interface*. Morgan kaufmann, 2016.

[7] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 472–489, IEEE, 2019.

[8] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," 2020.

[9] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?," *arXiv preprint arXiv:1810.00826*, 2018.

[10] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *ICML*, 2017.

[11] J. Jang, M. Woo, and D. Brumley, "Towards automatic software lineage inference," in *USENIX Security*, 2013.

[12] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, 2008.

[13] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pp. 143–157, 2008.

[14] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection," *IEEE Trans. Softw. Eng.*, vol. 43, pp. 1157–1177, Dec. 2017.

[15] F. Zhang, Y.-C. Jhi, D. Wu, P. Liu, and S. Zhu, "A first step towards algorithm plagiarism detection," in *ISSTA*, 2012.

[16] X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," CCS, 2009.

[17] T. Dullien and R. Rolles, "Graph-based comparison of executable objects," *SSTIC*, 2005.

[18] H. Flake, "Structural comparison of executable objects," DIMVA, 2004.

[19] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient cross-architecture identification of bugs in binary code," NDSS, 2016.

[20] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," ISSTA, 2009.

[21] G. Myles and C. Collberg, "Detecting software theft via whole program path birthmarks," ISC, 2004.

[22] D. Schuler, V. Dallmeier, and C. Lindig, "A dynamic birthmark for Java," ASE, 2007.

[23] X. Wang, Y. C. Jhi, S. Zhu, and P. Liu, "Detecting software theft via system call based birthmarks," in *ACSAC*, 2009.

[24] Y.-C. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu, "Value-based program characterization and its application to software plagiarism detection," ICSE, 2011.

[25] S. Cesare and X. Yang, *Software Similarity and Classification*. Springer Science, 2012.

[26] Y. C. Jhi, X. Jia, X. Wang, S. Zhu, P. Liu, and D. Wu, "Program characterization using runtime values and its application to software plagiarism detection," *IEEE Transactions on Software Engineering*, 2015.

[27] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Cross-architecture binary semantics understanding via similar code comparison," SANER, 2016.

[28] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *CCS*, 2017.

[29] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *NDSS*, 2019.

[30] S. Wang and D. Wu, "In-memory fuzzing for binary code similarity analysis," in *ASE*, 2017.

[31] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αdiff: Cross-version binary code similarity detection with DNN," in *ASE*, 2018.

[32] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary," ASE, 2018.

[33] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *CoRR*, vol. abs/1711.00740, 2017.

[34] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," *CoRR*, vol. abs/1904.12787, 2019.

[35] S. Luan, D. Yang, K. Sen, and S. Chandra, "Aroma: Code recommendation via structural code search," *CoRR*, vol. abs/1812.01158, 2018.

[36] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," NIPS 2018, 2018.

[37] D. McKee, N. Burow, and M. Payer, "Software ethology: An accurate and resilient semantic binary analysis framework," *CoRR*, vol. abs/1906.02928, 2019.

[38] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, S&P '15, pp. 709–724, IEEE Computer Society, 2015.

[39] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *FSE*, 2014.

[40] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *IEEE S&P*, 2014.

[41] D. Gao, M. K. Reiter, and D. Song, "BinHunt: Automatically finding semantic differences in binary programs," ICICS, 2008.

[42] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan, "Binary function clustering using semantic hashes," ICMLA, 2012.

[43] J. Ming, D. Xu, Y. Jiang, and D. Wu, "BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking," USENIX, 2017.

[44] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," PLDI, 2016.

[45] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "BinGo: Cross-architecture cross-OS binary search," FSE, 2016.

[46] 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.

[47] 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.

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

[49] M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 24–43, Springer, 2010.

[50] N. Howgrave-Graham, "Approximate integer common divisors," in *International Cryptography and Lattices Conference*, pp. 51–66, Springer, 2001.

[51] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 1–23, Springer, 2010.

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

[53] D. Stehlé, R. Steinfeld, K. Tanaka, and K. Xagawa, "Efficient public key encryption based on ideal lattices," in *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 617–635, Springer, 2009.

[54] Q. Lou and L. Jiang, "SHE: A fast and accurate deep neural network for encrypted data," in *Advances in Neural Information Processing Systems*, pp. 10035–10043, 2019.

[55] G. Plaxton, "String matching: Rabin-karp algorithm," *Theory in Programming Practice. University of Austin, Texas*, 2005.

[56] M. Régnier, "Knuth-morris-pratt algorithm: an analysis," in *International Symposium on Mathematical Foundations of Computer Science*, pp. 431–444, Springer, 1989.

[57] A. Ojugo and A. Eboka, "Signature-based malware detection using approximate boyer moore string matching algorithm," *International Journal of Mathematical Sciences and Computing*, vol. 5, no. 3, pp. 49–62, 2019.

[58] R. Baeza-Yates and G. H. Gonnet, "A new approach to text searching," *Communications of the ACM*, vol. 35, no. 10, pp. 74–82, 1992.

[59] M. Crochemore and D. Perrin, "Two-way string-matching," *Journal of the ACM (JACM)*, vol. 38, no. 3, pp. 650–674, 1991.