A Semantics-Based Hybrid Approach on Binary Code Similarity Comparison

Yikun Hu, Hui Wang, Yuanyuan Zhang, Bodong Li and Dawu Gu*

Abstract—Binary code similarity comparison is a methodology for identifying similar or identical code fragments in binary programs. It is indispensable in fields of software engineering and security, which has many important applications (e.g., plagiarism detection, bug detection). With the widespread of smart and IoT (Internet of Things) devices, an increasing number of programs are ported to multiple architectures (e.g. ARM, MIPS). It becomes necessary to detect similar binary code across architectures as well. The main challenge of this topic lies in the semantics-equivalent code transformation resulting from different compilation settings, code obfuscation, and varied instruction set architectures. Another challenge is the trade-off between comparison accuracy and coverage. Unfortunately, existing methods still heavily rely on semantics-less code features which are susceptible to the code transformation. Additionally, they perform the comparison merely either in a static or in a dynamic manner, which cannot achieve high accuracy and coverage simultaneously. In this paper, we propose a semantics-based hybrid method to compare binary function similarity. We execute the reference function with test cases, then emulate the execution of every target function with the runtime information migrated from the reference function. Semantic signatures are extracted during the execution as well as the emulation. Lastly, similarity scores are calculated from the signatures to measure the likeness of functions. We have implemented the method in a prototype system designated as BINMATCH which performs binary code similarity comparison across architectures of x86, ARM and MIPS on the Linux platform. We evaluate BINMATCH with nine real-word projects compiled with different compilation settings, on variant architectures, and with commonly-used obfuscation methods, totally performing over 100 million pairs of function comparison. The experimental results show that BINMATCH is resilient to the semantics-equivalent code transformation. Besides, it not only covers all target functions for similarity comparison, but also improves the accuracy comparing to the state-of-the-art solutions.

Index Terms—Binary code similarity comparison, reverse engineering, program analysis, code clone.

1 INTRODUCTION

Binary code similarity comparison is a fundamental methodology which identifies similar or identical code fragments in target binary programs with the reference code. It has numerous important applications in software engineering as well as security, for example, plagiarism detection [2], [3], [4], code searching [5], [6], [7], program comprehension [8], malware lineage inference [9], [10], [11], patch code analysis [12], [13], known vulnerability detection [14], [15], [16], [17], etc. In addition, with the development of smart and IoT (Internet of Things) devices, binary code similarity comparison is also required to be performed across multiple architectures considering above applications. Therefore, to improve the productivity and ensure the security of the software, it is necessary to effectively compare binary code similarity across architectures.

To fulfill the target, there exist two challenges. The first one is the *semantics-equivalent code transformation* (**C1**). It results from different compilation settings [18] (i.e., different compilers or optimization options), code obfuscation [19], [11], and varied instruction set architectures (ISAs) [15]. Because of the code transformation, even though two pieces of binary code are compiled from the same code base (i.e., semantically equivalent), they would differ significantly on the syntax and structure level, such as variant instruction sequences and control flow graphs, etc. The other challenge lies in the *trade-off* between comparison accuracy and coverage (**C2**) [20]. Dynamic methods procure rich semantics from code execution to guarantee the high accuracy of comparison, yet they analyze merely the executed code, leading to low code coverage. In contrast, static methods are able to cover all program components, while they heavily rely on syntax or structure-based code features which lacks semantics and thus produce less accurate results.

In the literature, it has drawn much attention to compare the similarity of binary code. However, existing solutions adopt either static methods which depend on semantics-less code features or dynamic methods which merely care about executed code. They cannot reach the compromise between comparison accuracy, which corresponds to **C1**, and coverage. Typically, static methods discovRE [16], Genius [17], and Kam1n0 [21] extract code features from control flow graphs, and measure the binary function similarity basing on graph isomorphism. Multi-MH [15], BinGo [7], and IMF-sim [20] capture behaviors of a binary function by sampling it with random values. Since the random input lacks semantics and is commonly illegal for the function, it could hardly trigger the core semantics of that function. Besides,

* *Corresponding author. E-mail: dwgu@sjtu.edu.cn*

- *Y. Hu is with the Department of Computer Science and Technology, SEIEE, Shanghai Jiao Tong University, Shanghai, China.*
- *H. Wang is with the Department of Computer Science and Technology, SEIEE, Shanghai Jiao Tong University, Shanghai, China.*
- *Y. Zhang is with the Department of Computer Science and Technology, SEIEE, Shanghai Jiao Tong University, Shanghai, China.*
- *B. Li is with the Department of Computer Science and Technology, SEIEE, Shanghai Jiao Tong University, Shanghai, China.*
- *D. Gu is with the Department of Computer Science and Technology, SEIEE, Shanghai Jiao Tong University, Shanghai, China.*

A preliminary version of this paper [1] appeared in the Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution (ICSME'18), *Madrid, Spain, September 23-29, 2018.*

Asm2Vec [22] leverages machine learning techniques to extract code features from the lexical relationships of assembly code tokens, while it is still syntax-based and suffers from **C1**. For dynamic methods, although Ming et al. [11], Jhi et al. [2], and Zhang et al. [3] adopt semantics-based code features, i.e., system calls and invariant values during execution, they perform detection merely on executed code. BLEX [18] pursues high code coverage at the cost of breaking normal execution of binary functions, distorting the semantics inferred from the collected features. Thus, it is necessary to propose a method which only depends on semantics and takes advantages of both static and dynamic techniques so as to achieve high accuracy and coverage for binary code similarity comparison.

In this paper, we propose BINMATCH, a semantics-based hybrid method, to fulfill the target. Given the reference function, BINMATCH aims to identify its match of similar semantics in the target binary program. BINMATCH firstly instruments the reference function, and executes it with available test cases to record its runtime information. It then migrates the runtime information to each function of the target program, and emulates the execution of that function. During the execution of the reference function and the emulation of the target functions, the semantic signature of each function is extracted simultaneously. Finally, BINMATCH compares the signature of the reference function with that of each target function in pairs to measure their similarity. Semantics describes the processes a computer follows when executing a program, which could be shown by describing the relationship between the input and output of a program [23]. To overcome **C1** of semantics-equivalent code transformation, BINMATCH only relies on signatures generated from the input/output and intermediate processing data collected during the (emulated) execution of the whole reference or target function. To address **C2** of the trade-off between comparison accuracy and coverage, BINMATCH adopts the hybrid method which captures signatures either in a static or in a dynamic manner. By executing the reference function and emulating the target functions, BINMATCH is able to extract their semantics-based signatures from the (emulated) executions. Because of the emulation, it is not necessary to really run the target program. BINMATCH emulates the target functions with the runtime information migrated from the reference function. Thus, it could cover all functions of the target program.

We have implemented a prototype of BINMATCH using the above method. We evaluate it with nine real-world projects compiled with various compilation settings, obfuscation configurations, and ISAs on the 32-bit Linux platform, totally performing over *100 million* pairs of function comparisons. The experimental results indicated that BINMATCH not only is robust to semantics-equivalent code transformation, but also outperforms the state-of-the-art solutions of binary code similarity comparison.

The paper makes the following contributions.

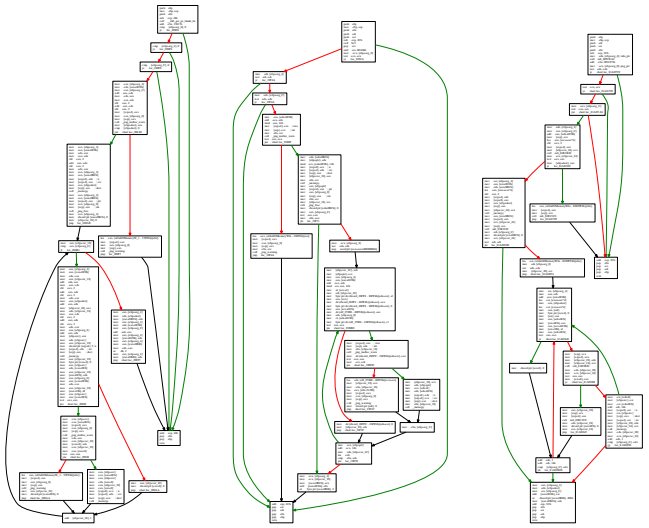
- We propose BINMATCH, a semantics-based hybrid method, to compare binary code similarity. It captures the semantic signature of a binary function either in a dynamic (execution) or in a static (emulation) manner. Thus, it could not only detect similar functions accurately with signatures of rich semantics, but also cover all target functions under analysis.
- BINMATCH emulates the execution of a function by migrating existing runtime information. To smooth the process of migration, we propose novel strategies to handle global variable reading, indirect jumping/calling, and library function invocation.
- We implement BINMATCH in a prototype system which supports cross-architecture binary code similarity comparison on the 32-bit Linux platform. BINMATCH is evaluated with nine real-world projects which are compiled with different compilation settings, obfuscation configurations, and instruction set architectures. The experimental results show that BINMATCH is robust to the semantics-equivalent code transformation. Besides, it covers all candidate target functions for similarity comparison, and outperforms the state-of-the-art solutions.

As this work is an extended version of our conference paper [1], we list below, the contributions of this extension:
(1) Effectiveness: We adopt Intel C++ Compiler (ICC) to compile the object projects, and leverage BINMATCH to compare the similarity of the resulting binary functions to those compiled by GCC and Clang as well as the obfuscated ones. Besides, we conduct experiments to evaluate the capacity of BINMATCH in comparing similar binary function across the mainstream architectures, i.e., x86, ARM and MIPS. In addition to Kam1n0 [21] and BinDiff [24], we compare the results of BINMATCH to those of Asm2Vec [22] and CACompare [25] as well. The experimental results further show the effectiveness of BINMATCH in handling semantics-equivalent code transformation which exists in the binary code.
(2) Practicability: Despite effective, BINMATCH is inefficient. To make the method more practical, we propose a strategy to prune the process of signature comparison. Additionally, we adopt a hash-based technology to efficiently estimate the similarity of function signatures.
(3) Investigation into thresholds: To reach the compromise between comparison accuracy and efficiency, we investigate the thresholds for applying the pruning strategy and the hash-based technology, and find suitable values for BINMATCH. The results indicate that it thus fulfills the comparison efficiently. Besides, it also outperforms the existing solutions from the perspective of accuracy.

The rest of this paper is organized as follows. Section 2 introduces a motivating example and presents the system overview of BINMATCH. Section 3 introduces how BINMATCH extracts semantics signatures of binary functions and compares their similarity. Section 4 presents several aspects to implement BINMATCH. The experimental results are shown and analyzed in Section 5. Some related issues are discussed in Section 6. Section 7 discusses the related work and the conclusion follows in Section 8.

2 MOTIVATION AND OVERVIEW

In this section, we firstly present a typical application of binary code similarity comparison, and illustrate the challenges of the topic with an example. Then, we explain the basic idea of BINMATCH and show the overview of its system.



(a) GCC -O0 (b) Clang -O2 (c) NConvert v6.17

Fig. 1: Control Flow Graphs of png_set_unknown_chunks

2.1 The Motivating Example

Known vulnerability detection is a typical application of binary code similarity comparison [7], [14], [15], [16], [17]. Given a piece of code which contains a known vulnerability, it is possible to locate its similar (or identical) match in other programs so as to check whether they are vulnerable or not.

`NConvert`[1] is a closed-source image processor which supports multiple image formats. It handles files of the PNG (Portable Network Graphics) format with the statically-linked open-source library `libpng`[2]. Unfortunately, `libpng` is found to suffer from an integer overflow vulnerability in the function *png_set_unknown_chunks* before the version of 1.5.14 (CVE-2013-7353[3]). The vulnerability allows attackers to cause a denial of service via a crafted image. To ensure whether `NConvert` suffers from the vulnerability, analyzers firstly need to locate the potential vulnerable function in it.

Since the source code of `libpng` is available, it is reasonable to locate the target function via code similarity comparison. `NConvert` is closed-source that only its executable is accessible, and the compilation setting of the executable is unknown. Even though executables are compiled from the same code base, different compilation settings would lead to *semantics-equivalent code transformation*, generating syntax and structure-variant binary code of equal semantics (**C1**). Figure 1 presents the CFGs (Control Flow Graphs) of *png_set_unknown_chunks*. Functions in Figure 1a and Figure 1b are compiled from the source code of `libpng` v1.5.12 with the setting of gcc -O0 and clang -O2 separately, while Figure 1c is extracted from the executable of `NConvert` v6.17 via manual reverse engineering. Because of the code transformation, despite the same semantics, the three functions differ in instruction sequences and CFGs. Thus, methods relying on syntax or structure code features (e.g., CFG isomorphism, binary code hashing) become ineffective.

Another problem is the *trade-off* between comparison accuracy and coverage (**C2**). Existing dynamic analysis-based methods only handle the executed code. However, *png_set_unknown_chunks* is statically-linked, mixing with the user-defined functions in the executable of `NConvert`. It requires huge extra work for dynamic methods to generate test cases in order to cover the target function, which is still an issue of binary code dynamic analysis [26]. In contrast, static analysis-based methods could cover all functions of `NConvert`. Nevertheless, they depend on semantics-less code features because they perform without actually executing the code. Therefore, the static methods cannot handle the semantics-equivalent code transformation.

2.2 System Overview of BINMATCH

We propose BINMATCH to compare the similarity of binary functions. Given a *reference* function, BINMATCH finds its match of similar semantics in the target binary program, returning a list of functions (the *target* functions) of the target program, which is ranked basing on the similarity of semantics.

Figure 2 presents the work flow of BINMATCH. Provided the reference function has been well analyzed or understood (*png_set_unknown_chunks*), BINMATCH dynamically instruments and executes it with available test cases, capturing its semantic signature (§3.1). Meanwhile, runtime information is recorded during the execution (§3.2). Then, BINMATCH emulates every function of the target program (`NConvert`) with the runtime information. During the emulation, signature of each target function is extracted as well. Afterward, BINMATCH compares the signature of the reference function to that of each target function in pairs, and computes their similarity score (§3.4). Finally, BINMATCH generates a list of target functions ranked by the similarity scores in descending order.

In Summary, to overcome **C1**, BINMATCH completely depends on the semantics-based signature which is generated from the input/output and the intermediate processing data during the (emulated) execution of a function. To address **C2**, BINMATCH captures function signatures in a hybrid manner. It extracts the signature of the reference function via *dynamically executing* its test cases. We assume that the reference function has been well studied and its test cases are available. In above example, the integer overflow of *png_set_unknown_chunks* has been known, and its test case could be found in the `libpng` project or the vulnerability database. Then, with the runtime information of the reference function, BINMATCH extracts the signature of each function of the target program (`NConvert`) via *static emulation*. Therefore, BINMATCH is able to cover all target functions and detect similar function with signatures of rich semantics.

3 METHODOLOGY

In this section, we firstly introduce the semantic signatures adopted by BINMATCH. Then, we discuss how it captures the signatures of binary functions and measures their similarity.

1. https://www.xnview.com/en/nconvert/
2. http://www.libpng.org/pub/png/libpng.html
3. https://www.cvedetails.com/cve/CVE-2013-7353/

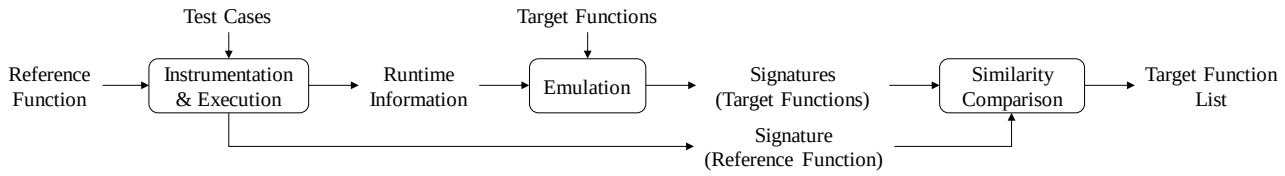


Fig. 2: System Architecture of BINMATCH

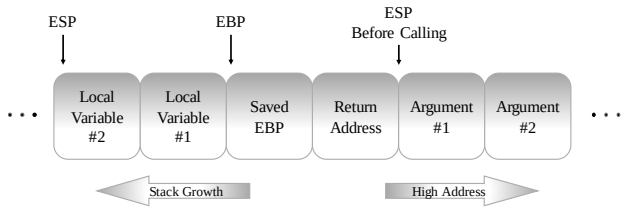


Fig. 3: Calling Stack of *cdecl*

3.1 Semantic Signature

The semantics describes the processes a computer follows when executing a program. It could be shown by describing the relationship between the input and output of the program [23]. Thus, given a specific input, we focus on two points to reveal the semantics of a binary function: i) *what* is the corresponding output after the function processing the input, and ii) *how* the function processes the input to generate the output. The signature adopted by BINMATCH consists of the following features:

- *Output Values:* For a binary function, the feature consists of the return value and the global variable values written to the memory. It covers the output of a function. When given the specific input, the feature directly shows the semantics of a function.
- *Comparison Operand Values:* The feature is composed of values for comparison operations whose results decide the following control flow of an (emulated) execution. A function might have numerous paths, while only one is triggered by the input to generate the output. The feature describes how an input chooses the path of a function to produce the corresponding output, indicating the relationship between the input and output. Therefor, it reflects the semantics of a function.
- *Invoked Standard Library functions:* Standard library functions provide fundamental operations for implementing user-defined functions. They have complete functionality, such as `malloc` meaning dynamic memory allocation, `fread` representing file reading, etc. Then the invocations of those library functions indicate the semantics of the (emulated) execution. Besides, the feature has been show to be effective for binary code similarity comparison [27], [28]. Thus, it is adopted as complement to the semantic signature of BINMATCH.

During the (emulated) execution of a binary function, BINMATCH captures the sequence of above features, and considers it as the signature of that function for latter similarity comparison.

Algorithm 1: Instrumentation

Input: Instruction under Analysis $\mathcal{I}$
Output: Instruction after Instrumentation $\mathcal{I}_r$

```
1  Algorithm Instrumentation (I)
2      I_r ← I
3      // capture features for the signature
4      if I outputs data then
5          I_r ← record_data_val (I_r)
6      if I performs comparison then
7          I_r ← record_oprd_val (I_r)
8      if I calls a standard library function then
9          I_r ← record_libc_name (I_r)
10     // record runtime information
11     if I reads an argument of the function then
12         I_r ← record_arg_val (I_r)
13     else if I uses global variable then
14         I ← record_var_val (I_r)
15     else if I calls a function indirectly then
16         I_r ← record_func_addr (I_r)
17     else if a subroutine returns then
18         I_r ← record_ret_val (I_r)
19     return I_r
```

3.2 Instrumentation and Execution

In this step, BINMATCH dynamically instruments the reference function R to extract its signature by executing the available test cases. Meanwhile, runtime information for Emulation (§3.3) is recorded as well.

Algorithm 1 presents the pseudo-code of instrumentation. For the instruction $\mathcal{I}$ of R, if it outputs data, performs comparison operations, or calls a standard library function, BINMATCH injects code before it to capture corresponding features, then generates the signature of R (Line 4-9).

Line 11-18 present the code for recording runtime information of R's execution. Similar functions should behave similarly if they are executed with the same input [7], [15], [18], [25]. Therefore, BINMATCH records the input of R's execution, which is provided for the Emulation in the next step. For a binary function, the input consists of the argument values, global values, and return values of its subroutines [13]. According to *cdecl*, the default calling convention of x86, function arguments are prepared by callers and passed through the stack, as shown in Figure 3. In contrast, 32-bit ARM and MIPS have specific argument registers (i.e., R0-R3 for ARM, $a0-$a3 for MIPS). When the function has arguments more than the registers, the surplus ones are passed by the stack, which is similar to *cdecl*. Thus, if $\mathcal{I}$ reads a variable from argument registers or

Algorithm 2: Emulation

```
Input: Emulated Memory Space of the Target Function M
Input: Runtime Value Set of the Reference Function S
1  Algorithm Emulation (M, S)
2      init_func_stack (M)
3      assign_func_arg (M, S)
4      foreach instruction I to be emulated do
5          if I uses a global variable then
6              if the variable is accessed for the first time then
7                  migrate_var_val (M, S)
8          if I calls a function indirectly then
9              addr ← get_tar_addr (I)
10             if addr ∈ S then
11                 migrate_ret_val (M, addr, S)
12             else if addr is an illegal function address then
13                 exit_emulation()
14         if I invokes a standard library function then
15             libf ← get_func_name (I)
16             if libf needs system supports then
17                 migrate_ret_val (M, libf, S)
18                 record_feat_val (M, I)
19                 continue
20         // capture features for the signature
21         if I contains features then
22             record_feat_val (M, I)
23         emulate_inst (M, I, S)
```

stack with the address higher than the stack pointer (ESP in Figure 3) before calling, BINMATCH considers the variable as a function argument and records its value (Line 11-12).

For ELF (Executable and Linkable Format) files, global data is placed in specific data sections, e.g., `.bss` for uninitialized global data. Therefore, if $\mathcal{I}$ uses data within those sections, BINMATCH considers the values as global data and records them along with the accessing addresses (Line 13-14). After this step, a record sequence of accessed global variables is generated. If a variable is accessed for multiple times, there would be the same number of records in the sequence as well. Besides, BINMATCH records the target addresses of subroutines indirectly invoked by R (Line 15-16). The return values of all subroutines are recorded as well, including user-defined functions and library functions (Line 17-18).

3.3 Emulation

For every target function T to be compared with the reference function R, BINMATCH emulates its execution with runtime information recorded in the last step. The semantic signature of T is captured simultaneously. The basic idea of the process is to emulate T with the same input (i.e., runtime information) of R as if it was executed in the memory space of R. If T is the match of R, then their generated signatures should be similar.

To fulfill the emulation, BINMATCH firstly needs to prepare the stack frame for T which is similar to dynamic execution (§3.3.1). Then, it provides T with the input of R to perform the emulation. From a function's perspective, the input consists of arguments, global variables, and return values of subroutines [13]. Therefore, we need to handle function argument assignment (§3.3.2) and global variable reading (§3.3.3). Since the targets of direct user-defined function calls are explicit, we then just focus on indirect calls (§3.3.5) and standard library function calls which might require the system support (§3.3.6). It is also necessary to consider indirect jumps whose target addresses are implicit for emulation (§3.3.4).

Algorithm 2 presents the pseudo-code of the process. BINMATCH firstly prepares the stack frame for the function emulation, including initializing the stack pointer values (Line 2) and providing T with the arguments of R (Line 3). Before emulating the instructions of T with the runtime intermediate data of R (Line 23), BINMATCH needs to handle global variable reading (Line 5-7), indirect function calling (Line 8-13), and standard library function invocation (Line 14-19) if necessary. If T is not the match of R, the emulation might access illegal memory addresses which have never been recorded in the last step. BINMATCH then stops the emulation. Additionally, BINMATCH records the features of T to generate its signature (Line 21-22). Next, we discuss the algorithm of emulation in more details.

3.3.1 Stack Frame Pointer Initialization

Similar to execution, every T for emulation has its own stack frame which is accessed by the stack pointer or the base pointer (e.g., ESP or EBP) with relative offsets. Before emulating, BINMATCH assigns the stack and base pointers with those initial values of the reference function. After assigning the argument values (§3.3.2), the arrangement of the stack frame is decided by the code of T, such as pushing or popping values, allocating memory for local variables, etc.

3.3.2 Function Argument Assignment

In our scenario, functions for similarity comparison are compiled from the same code base, i.e., they have identical interface with the same number and order of arguments. According the calling convention, BINMATCH recognizes the argument list of T. If the argument number of T equals to that of R, BINMATCH assigns the argument values of R to those of T in order. Otherwise, T cannot be the match of R, then BINMATCH skips the emulation. For example, R and T have the following argument lists:

```
R(rarg_0, rarg_1, rarg_2)
T(targ_0, targ_1, targ_2)
```

Provided BINMATCH has the values of `rarg_0` and `rarg_2` (R only accesses the two arguments in the execution), it assigns their values to `targ_0` and `targ_2` separately. To make the emulation smooth, arguments without corresponding values (`targ_1`) are assigned with a predefined value (e.g., `0xDEADBEEF`).

3.3.3 Global Variable Reading

In the execution of the reference function R, it might read global (or static) variables whose values have been modified by former executed code. For example, R accesses a global variable `gvar` whose initial value is *0*. During the execution, before R is invoked, its caller modifies `gvar` with the

```
1  mov   ecx, gvar1
2  test  ecx, ecx
3  mov   eax, gvar2
4  add   ecx, eax
```

(a) Reference Function

```
1  mov   ecx, gvar1'
2  mov   ebp, gvar2'
3  test  ebp, ebp
4  add   ebp, ecx
```

(b) Target Function

Fig. 4: Global Variable Value Migration

```
1  mov   edx, [ebp-0E4h] ; load a local variable
2  lea   eax, [edx-0Ah] ; get the index
3  cmp   eax, 2Ah
4  ja    loc_8052880 ; the default case
5  jmp   dword ptr [eax*4+808F630h]; indirect jump
```

Fig. 5: Indirect Jump of a Switch on x86

```
1  LDR     R1, [SP, #0x88+arg_0] ; load the index
2  CMP     R1, #8
3  LDRLS   PC, [PC, R1, LSL#2] ; indirect jump
4  B       loc_410F4 ; the default case
5  -------------------------------------------------
6  DCD loc_410F8
7  DCD loc_40FCC
8  ...
```

Fig. 6: Indirect Jump of a Switch on ARM

```
1  mov   eax, [ebp+arg_0] ; load the first argument
2  mov   eax, [eax]
3  ...
4  call  eax ; indirect call
```

Fig. 7: Indirect Call Decided by the Input on x86

value *1*. Then *R* processes with `gvar` of value *1*. To ensure the target function *T* is emulated with the same input as *R*, the modified global values should be assigned to the corresponding addresses which *T* reads from.

Global variables are stored in specific data sections of an executable file (e.g., `.data` section). The size of each variable is decided by the source code. The location of the variable is determined during the process of compilation and not changed afterward. Therefore, if the addresses of *T*'s global variable accessing are input-unrelated, their explicit values could inferred during the emulation. Recall that BINMATCH has generated a record sequence of accessed global variables during the execution of *R* (§3.2). Then, it migrates the unassigned values in the record sequence to the inferred addresses according to the usage order. If the address originates from the input, which is actually the one processed by *R* (e.g., pointer assigned as an argument of *T*), BINMATCH then directly reuses the first unassigned value of that address in the record sequence for the emulation. In addition, if the address is illegal for both *R* and *T*, BINMATCH stops and exits the emulation.

Figure 4 shows an example of two functions for global variable value migration which bases on the usage order. During the execution of *R*, two global variables `gvar1` and `gvar2` are read at Line 1 and Line 3 separately in Figure 4a. `gvar1` is used to test its value at Line 2, and `gvar2` is used for the addition operation at Line 4. So the usage order of the two variable is `[gvar1, gvar2]`. When emulating *T* in Figure 4b, BINMATCH identifies `ecx` and `ebp` are loaded with global variables `gvar1'` and `gvar2'` at Line 1 and Line 2. Then, it finds `ebp` is used for testing at Line 3, and `ecx` is used for the addition at Line 4 afterward. The usage order of the global variables in Figure 4b is `[gvar2', gvar1']`. Therefore, BINMATCH assigns the value of `gvar1` to `gvar2'`, and `gvar2` to `gvar1'` accordingly. If there are no enough global values to assign (e.g., *T* reads two global variables but *R* reads only one), BINMATCH stops and exits the emulation.

3.3.4 Indirect Jumping

An indirect jump (or branch) is implemented with a jump table which contains an ordered list of target addresses. For x86 and MIPS, jump tables are stored in `.rodata`, the read-only data section of an executable. Therefore, similar to reading a global data structure, a jump table entry is accessed by adding the offset to the base address of the jump table. The base address is a constant value, and the offset is computed from the input. Figure 5 shows an indirect jump of a switch structure on x86. At Line 2, the index value is computed with `edx`, a value of an input-related local variable, and stored in `eax`. If the index value is not above `0x2A`, which represents the default case, an indirect jump is performed according to the jump table whose base address is `0x808F630` (Line 5).

On ARM, jump tables are inlined into the code. They directly follow the code which accesses the tables. Figure 6 presents the indirect jump and jump table of a switch structure on ARM. It loads the index from the first function argument (`arg_0`), storing it in `R1` (Line 1). The index is compared with `0x8` (Line 2). If it is larger than `0x8`, the program directly jumps to the default case (Line 4). Otherwise, the program refers to the jump table and gets the corresponding target address (Line 3). Since the jump table is attached to the jumping code, `PC` (or `R15`, the Program Counter) is used as the base address. Note that the code in Figure 6 is compiled with the A32 instruction set of ARM, which has the fixed instruction length of 32 bits (4 bytes). Because of the processor's pipeline, the `PC` value is always 8-byte ahead the current executed instruction. When executing the loading instruction at Line 3, the `PC` in the right operand is pointing to the first entry of the jump table at Line 6. Therefore, on ARM, indirect jumps also access jump tables with the decided value as the base address.

During the compilation, entries of jump tables are sorted and placed in the resulting binary code. With the same input, code of identical semantics would jump to the same path to process the input. Thus, BINMATCH just follows the emulated control flow and has no need to do extra work for indirect jumps.

3.3.5 Indirect Calling

Similar to indirect jumping, targets of indirect calls are decided by the input at runtime as well. In some cases, target addresses directly come from the input, as shown in Figure 7. At Line 1, the first function argument (`arg_0`), which is the pointer of a data structure, is loaded to `eax`. After the first member is fetched, which is a function address, the code calls the function indirectly. Since the target function *T* is emulated with the memory space of the reference function *R*, if *T* is the match of *R*, targets of the indirect calls in above cases should be those invoked during the execution of *R*.

```
1   test    eax, eax ; input realted
2   jnz     short loc_806E0D4
3   mov     ds:dword_808C810, 808157Bh ; the false branch
4   jmp     loc_806E0E9
5 loc_806E0D4:
6   mov     ds:dword_808C810, 80815FCh ; the true branch
7 loc_806E0E9:
8   mov     eax, ds:dword_808C810
9   ...
10  call    eax ; indirect call
```

Fig. 8: Indirect Call Affected by the Control Flow on x86

Algorithm 3: Function Similarity Comparison

Input: Signature of the Reference Function $\mathcal{S}_r$
Input: Signature of the Target Functions $\mathcal{S}_t$
Input: Length Threshold $\mathcal{L}$
Output: Similarity Score $\mathcal{S}$
1 **Algorithm** *Comparison* ($\mathcal{S}_r$, $\mathcal{S}_t$, $\mathcal{L}$)
2 $L_r \leftarrow$ **length**(S_r)
3 **if** $L_r < \mathcal{L}$ **then** $F \leftarrow$ **jaccard_with_lcs**
4 **else** $F \leftarrow$ **hd_with_simhash**
5 $\mathcal{S} \leftarrow F(\mathcal{S}_r, \mathcal{S}_t)$
6 **return** $\mathcal{S}$

BINMATCH then migrates the return values of those calls to the corresponding ones of *T* (Line 10-11 in Algorithm 2).

In some cases, varied execution paths would generate different target addresses for a function call. That is decided by the input. Figure 8 presents an example of the case. At Line 1, the input-related value stored in `eax` is tested. If it is not zero, the branch is taken at Line 2 (`jnz`: jump if not zero), jumping to `0x806E0D4`. Then a function address `0x80815FC` is stored into data section at `0x808C810` (Line 6). Otherwise, another function address `0x808157B` is stored (Line 3). Afterward, the program jumps to the stored address indirectly (Line 10). Thus, the whole process merely depends on the input. It is not necessary to do other work for indirect calls in such case. In other cases, indirect calls are implemented with jump tables as well, such as virtual function tables. BINMATCH handles such indirect calls the same way as that for indirect jumps.

When the address is not a legal function address of either *R* or *T*, *T* cannot be the match of *R*. BINMATCH just stops the process and exits (Line 12-13 in Algorithm 2).

3.3.6 Standard Library Function Invocation

If the target function *T* calls a standard library function which requests the system support (e.g., `malloc`), BINMATCH skips its emulation and assigns it with the result of the corresponding one invoked by the reference function *R* (Line 16-19 in Algorithm 2). For example, *R* and *T* calls following library functions in order:

```
R: malloc_0, memcpy, malloc_1
T: malloc_0', memset, malloc_1'
```

BINMATCH assigns return values of `malloc_0`, `malloc_1` to `malloc_0'`, `malloc_1'` separately, and skips the emulation. In contrast, `memset` is emulated normally, because it has no need for the system support. Afterward, when *T* accesses the memory values on the heap, i.e., via the return values of `malloc_0'` or `malloc_1'`, it would be assigned with those of *R* for the emulation.

3.4 Similarity Comparison

BINMATCH has captured the semantic signature (the feature sequence) of the reference function via execution, and those of target functions via emulation. In this step, it compares the signature of the reference function to that of each target function in pairs, and calculates their similarity score, as shown in Algorithm 3. BINMATCH adopts two solutions to measure the signature similarity. One is the Jaccard Index [29] with Longest Common Subsequence (LCS) [30], the other is Hamming Distance (HD) [31] with SimHash [32]. The former solution is relatively more accurate but slow, while the latter one is fast but less accurate. Thus, a length threshold ($\mathcal{L}$) is specified to select the suitable method. When the lengths of the reference signatures are short, i.e., less than $\mathcal{L}$, BINMATCH performs the comparison with the accurate matching (Line 3). Otherwise, it leverages the fuzzy matching to fulfill the target (Line 4). In such way, we aim to reach a compromise between comparison accuracy and efficiency. We will discuss the value of $\mathcal{L}$ in Section 5.2.1.

After the comparison, BINMATCH generates a list of target functions along with similarity scores, which is ranked by the scores in descending order. Next, we discuss the details of the solutions adopted by BINMATCH for similarity comparison.

3.4.1 Jaccard Index with Longest Common Subsequence

Jaccard Index is a statistic used for measuring the similarity of sets. Given two sets S_r and S_t, the Jaccard Index is calculated as followed:

$$J(S_r, S_t) = \frac{|S_r \cap S_t|}{|S_r \cup S_t|} = \frac{|S_r \cap S_t|}{|S_r| + |S_t| - |S_r \cap S_t|} \quad (1)$$

$J(S_r, S_t)$ ranges from 0 to 1, which is closer to 1 when S_r and S_t are considered to be more similar.

To better adapt to the scenario of BINMATCH, we utilize the Longest Common Subsequence (LCS) algorithm to the Jaccard Index. On one hand, a signature is captured from the (emulated) execution of a function. The appearance order of each entry in the signature is a latent feature as well. The order reflects how the input is processed to generate the output, thus it is semantics-related. On the other hand, the signature might be captured from an optimized or obfuscated binary function that it would contain diverse or noisy entries in the sequence. LCS not only considers the element order of two sequences for comparison, but also allows skipping non-matching elements, which tolerates semantics-equivalent code transformation. Hence, the LCS algorithm is suitable for signature similarity comparison of BINMATCH. With LCS, in Equation 1, S_r and S_t represent the signatures of the reference and target functions. $|S_r|$ and $|S_t|$ are their lengths, and $|S_r \cap S_t|$ is the LCS length of the two signatures.

3.4.2 Hamming Distance with SimHash

SimHash is a solution which quickly estimates the similarity of two sets. The basic idea of SimHash is similar items are hashed to similar hash values, i.e., with small bitwise hamming distances. Assuming that the hash values have the size of *F*, the similarity of two sets S_r and S_t is computed as followed:

$$Sim(S_r, S_t) = 1 - \frac{HD[SH_F(S_r), SH_F(S_t)]}{F} \quad (2)$$

Here, $SH_F(S)$ means the SimHash value of set S, ranging from 0 to 2^F. Then, the hamming distance (HD) of the two SimHash values ranges from 0 to F. As a result, $Sim(S_r, S_t)$ ranges from 0 to 1 as well. The larger the value is, the more similar the two sets are considered to be.

In such way, comparing to the high time complexity $O(n^2)$ of the LCS algorithm, SimHash only has the time complexity of $O(n)$. However, SimHash treats the feature sequences (signatures) as sets. It ignores the order of sequence elements, which is considered to be semantics-related. Thus, it is less accurate in handling signatures extracted from optimized or obfuscated functions. Consequently, hamming distance with SimHash computes the similarity of signatures more efficiently, while Jaccard Index with LCS has higher accuracy.

Algorithm 4: Pruning Similarity Comparison

Input: Signature of the Reference Function $\mathcal{S}_r$
Input: Signature of the Target Functions $\mathcal{S}_t$
Input: Length Threshold $\mathcal{L}$
Input: Pruning Threshold $\mathcal{P}$
Output: Similarity Score $\mathcal{S}$

```
1 Algorithm pruningComparison (S_r, S_t, L, P)
2     L_r ← length(S_r)
3     L_t ← length(S_t)
4     // pruning strategy
5     if max(L_r, L_t) / min(L_r, L_t) > P then S ← −1
6     // Algorithm 3
7     else S ← comparison(S_r, S_t, L)
8     return S
```

4 IMPLEMENTATION

Currently, BINMATCH supports binary function similarity comparison of ELF (Executable and Linkable Format) files. It performs analysis on 32-bit Linux platform of three mainstream ISAs, i.e., x86, ARM, and MIPS. Next, we discuss the key aspects of the implementation.

4.1 Binary Function Boundary Identification

BINMATCH performs comparison on the function level. It requires the address and length information of each binary function under analysis. Given an ELF file, `IDA Pro` v6.6[4] is adopted to disassemble it and identify the boundaries of its binary functions. The plugin of `IDA Pro`, `IDAPython`, provides interfaces to obtain addresses of functions. For example, `Functions(start, end)` returns a list of function start addresses between address `start` and `end`. As a result, we develop a script with IDAPython to acquire function addresses of binary programs automatically. Besides, function arguments and switch structures are identified as well to assist in assigning argument values (§3.3.2) and emulating indirect jumps (§3.3.4). Although the resulting disassembly of `IDA Pro` is not perfect [33], [34], it is sufficient for the scenarios of BINMATCH.

4.2 Instrumentation and Emulation

We implement the instrumentation module of BINMATCH with `Valgrind` [35], a dynamic instrumentation framework. `Valgrind` unifies binary code under analysis into VEX-IR, a RISC-like intermediate representation (IR), and injects instrumentation code into the IR code. Then, it translates the instrumented IR code into binaries for execution. IR translation unifies the operations of binary code and facilitates the process of signature extraction. For example, memory reading and writing instructions are all unified with `Load` and `Store`, the opcodes defined by VEX-IR. Hence, we just concentrate on the specific operations of IR and ignore the complex instruction sets of different architectures.

The step of emulation is implemented basing on `angr` [36], a static binary analysis framework. `angr` borrows VEX-IR from `Valgrind`, and translates binary code to be analyzed into IR statically. Given a user-defined initial state, it provides a module named `SimProcedure` to emulate the execution of IR code. SimProcedure allows injecting extra code to monitor the emulation of the IR code. It actually emulates the process of instrumentation. Besides, `angr` maintains a database of standard library functions to ease the emulation of those functions (§3.3.6). Thus, we develop a script of monitoring code, which is similar to the instrumentation code developed with `Valgrind`, to capture semantic signatures during the emulation with `angr`.

4.3 Function Inlining and Signature Inlining

Function inlining is an operation which expands a callee to its caller. It eliminates the calling and returning of the callee, improving the efficiency of code execution. Then, it is adopted as a strategy for code optimization [37]. Function inlining also might be used as an obfuscation technique that modifies boundaries of functions [38], posing difficulties to reverse engineering.

Since BINMATCH works on the function level, function inlining would affect its accuracy of comparison. For example, the reference function M_r invokes the subroutine N_r during the execution. The corresponding function N_t is inlined into M_t in the target binary program, becoming M_tN_t. Because the signature of M_tN_t is actually extracted from two functions, while that of M_r only contains one, BINMATCH might miss the match of $[M_r, M_tN_t]$ finally. To alleviate the side effects of function inlining, BINMATCH inlines the signature of a callee to its caller, which is similar to the process of function inlining. In above example, the signature of N_r is then expanded in that of M_r, becoming the signature of M_rN_r for the similarity comparison. Note that BINMATCH only inlines the signatures of user-defined functions, not counting those of standard library functions.

4.4 Pruning Strategy of Similarity Comparison

The code features adopted by BINMATCH are semantics-related. Intuitively, because of signature inline, signature lengths of similar functions should be close. Thus, we propose a signature length-based pruning strategy to improve the efficiency of similarity comparison. As presented at Line 5 in Algorithm 4, given a pre-defined pruning threshold $\mathcal{P}$ (>1), BINMATCH skips the comparison when the difference between two signature lengths is sufficiently

4. https://www.hex-rays.com/products/ida/

large, i.e., the division of their lengths is lager than $\mathcal{P}$ or less than $\frac{1}{\mathcal{P}}$. The two functions are considered to be dissimilar under that condition. We will discuss the value of $\mathcal{P}$ in Section 5.2.2.

5 Evaluation

We conduct empirical experiments to evaluate the effectiveness and capacity of BinMatch. We firstly discuss thresholds ($\mathcal{L}$ in §3.4 and $\mathcal{P}$ in §4.4) adopted by BinMatch, which balance the accuracy and efficiency of comparison (§5.2). Then, BinMatch is evaluated with binaries compiled with different compilation settings, including variant optimization options and compilers (§5.3). We also evaluate the effectiveness of BinMatch in handling obfuscation by comparing binary functions with their obfuscated versions (§5.4). Lastly, we leverage BinMatch to compare the similarity of binary code compiled with different ISAs, across x86, ARM and MIPS (§5.5). The results of above experiments are all compared to those of existing solutions.

5.1 Experiment Setup

The evaluation is performed in the system of Ubuntu 16.04 which is running on an Intel Core i7 @ 2.8GHz CPU with 16G DDR3-RAM.

5.1.1 Dataset

We adopt programs of nine real-world projects as objects of the evaluation, as listed in Table 1. The object programs have various functionalities, including data transformation (`convert`, `ffmpeg`), data compression (`gzip`), code parsing (`lua`), email posting (`mutt`), etc. With those objects, the effectiveness of BinMatch is shown to be not limited by the types of programs and functions under analysis.

For cross-compilation-setting comparison (§5.3), the objects are compiled with different compilers (i.e., GCC v4.9.3, Clang v4.0.0, and ICC v16.0.4) and variant optimizations (i.e., `-O3` and `-O0`). For comparison with obfuscated code (§5.4), we adopt Obfuscator-LLVM (`OLLVM`) [39] to obfuscate the object programs. OLLVM provides three widely used techniques for obfuscation, including *Instruction Substitution*, *Bogus Control Flow*, and *Control Flow Flattening*. We use the three techniques to handle the object programs optimized with `-O3` and `-O0` respectively. Then, for cross-architecture comparison (§5.5), the object programs are compiled for three architectures, i.e., x86, ARM, and MIPS, separately, with the compiler GCC and optimization option `-O3`. As a result, we totally compile *142* unique executables for the evaluation.

For each experiment, we select two executables of an object program, i.e., E_r (the reference executable) and E_t (the target executable). BinMatch executes E_r with the test command presented in Table 1, and considers each executed function as a reference function. Then, it compares every reference function to all target functions of E_t in pairs to compute similarity scores. Consequently, BinMatch performs over *100 million* pairs of function comparisons in all the experiments.

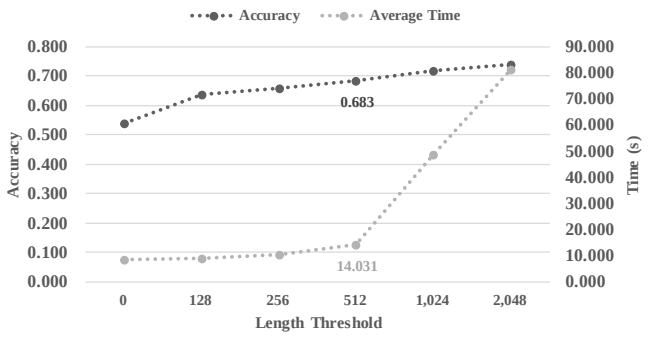


Fig. 9: Accuracy and Time versus Length Threshold

5.1.2 Ground Truth

All the executables for the evaluation are stripped that their debug and symbol information is discarded. To verify the correctness of the experimental results, we compile their *extra unstripped copies*, and establish the ground truth with the symbol information.

For each reference function, BinMatch generates a list of target functions ranked by the similarity scores in descending order (§3.4). According to the ground truth, if the reference function name exists in the Top *K* entries of the resulting target function list, we consider the match of the reference function could be *found* by BinMatch in E_t. Given the reference function, BinMatch is designed for assisting analyzers in looking for similar matches in target binaries. Thus, it is reasonable to assume the analyzers could further identify the correct match with acceptable amount of effort when provided with *K* candidates. In this paper, we assign *K* with values of *1*, *5*, and *10* respectively.

5.1.3 Evaluation Metrics

Similar to previous research [8], [18], we measure the performance of BinMatch with $Accuracy$, the ratio of executed reference functions which could be *found* in the Top *K* entries of the resulting target function lists. The formula is as followed:

$$Accuracy = \frac{|\text{Found Matches}|}{|\text{Reference Functions}|} \tag{3}$$

5.2 Parameter Settings

We leverage either *Jaccard Index with LCS* (accurate but less efficient) or *Hamming Distance with SimHash* (efficient but less accurate) to measure the similarity of function signatures. We propose the signature length threshold $\mathcal{L}$ to select suitable method for the measurement (§3.4). Besides, we introduce the ratio threshold $\mathcal{P}$ to prune unnecessary comparison to improve the efficiency of similarity measurement (§4.4). In this section, we attempt to find the best values of $\mathcal{L}$ and $\mathcal{P}$ for the following experiments, which balance the accuracy and efficiency of BinMatch.

5.2.1 Length Threshold for Similarity Comparison

We execute the reference executables, totally obtaining *14,207* reference functions. We randomly select *4,000* of

TABLE 1: Object Projects of Evaluation

Program	Version	Description	Test Command
convert	6.9.2	Command-line interface to the ImageMagick image editor/converter	`convert` *sample.png* -background black -alpha remove *sample.jpg*
curl	7.39	Command-line tool for transferring data using various protocols	`curl` -O *http://ftp.gnu.org/gnu/wget/wget-1.13.tar.xz*
ffmpeg	2.7.2	Program for transcoding multimedia files	`ffmpeg` -f image2 -i *sample.png sample.gif*
gzip	1.6	Program for file compression and decompression with the DEFLATE algorithm	`gzip` --best --recursive --force *sample_directory*
lua	5.2.3	Scripting parser for Lua, a lightweight, multi-paradigm programming language	`lua` *sample.lua*
mutt	1.5.24	Text-based email client for Unix-like systems	`mutt` -s *"hello" user@domain.com* < *sample.txt*
openssl	1.0.1p	Toolkit implementing the TLS/SSL protocols and a cryptography library	`openssl` s_server -key *key.pem* -cert *cert.pem* -accept *44330* -www
puttygen	0.70	Part of PUTTYGEN suit, a tool to generate and manipulate SSH public and private key pairs	`puttygen` -P *sample.pem* -o *key.pem*
wget	1.15	Program retrieving content from web servers via multiple protocols	`wget` *http://ftp.gnu.org/gnu/wget/wget-1.13.tar.xz* --no-cookies

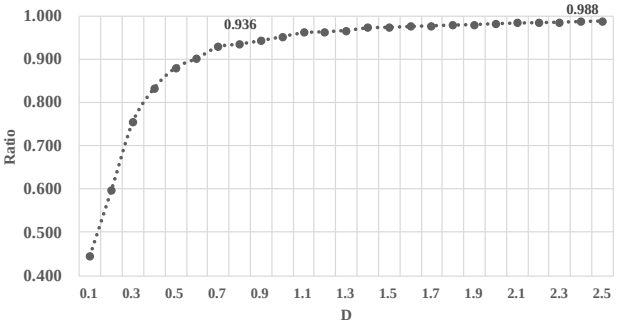


Fig. 10: Accumulative Function Ratio versus Pruning Threshold

them, and investigate the performance of BINMATCH with variant values of the length threshold $\mathcal{L}$. The pruning strategy is disabled in this part of the experiments, i.e., $\mathcal{P} = +\infty$.

The results are shown in Figure 9. The black line represents the Top *1* accuracy, and the gray line is the average time for processing each reference function. As $\mathcal{L}$ increases exponentially, the corresponding time rises in a similar manner, while the accuracy grows much slower in a linear-like form. When BINMATCH performs the similarity comparison only with the Jaccard Index, i.e., $\mathcal{L} = +\infty$, it achieves the Top *1* accuracy of *85.4%*. That could be considered as the upper bound capacity of BINMATCH. Nevertheless, the time consumption of each function reaches *262.637* seconds. Considering the average time, note that $(512, 14.031)$ is the turning point of the line. When $\mathcal{L} = 512$, although the Top *1* accuracy is only *68.3%*, the Top *5* accuracy rises to *86.4%* which is comparable to the Top *1* accuracy when $\mathcal{L} = +\infty$. Therefore, in the following experiments, BINMATCH adopts:

$$\mathcal{L} = 512. \tag{4}$$

5.2.2 Ratio Threshold for Pruning

Considering the reference and target signatures S_r and S_t with the lengths of L_r and L_t, where $L_r \neq L_t$, we define

$$D = \ln \frac{max(L_r, L_t)}{min(L_r, L_t)} = |\ln L_r - \ln L_t| \tag{5}$$

According to Equation 1, the possible maximum similarity score of (S_r, S_t) computed by Jaccard Index is

$$J_M(S_r, S_t) = \frac{min(L_r, L_t)}{max(L_r, L_t)}, \text{ if } S_r \subset S_t \text{ or } S_r \supset S_t \tag{6}$$

As presented at Line 5 in Algorithm 4, when

$$\frac{min(L_r, L_t)}{max(L_r, L_t)} < \frac{1}{\mathcal{P}}, \tag{7}$$

the pair of comparison (S_r, S_t) is pruned. Combining Equation 7 with Equation 5 and 6, we have

$$J(S_r, S_t) \leq \frac{min(L_r, L_t)}{max(L_r, L_t)} = \frac{1}{e^D} < \frac{1}{\mathcal{P}} \tag{8}$$

Therefore, in this section, we aim to find the acceptable maximum Jaccard Index of similarity comparison, i.e, the minimum value of D, to fulfill the pruning.

We randomly select *1,000* reference functions, and leverage BINMATCH to perform similarity comparison with the Jaccard Index, i.e, $\mathcal{L} = +\infty$. Since we merely consider at most Top *10* candidates of the results, we investigate the 10th similarity scores for all the target functions in the resulting lists. Then, we find the average value of all the 10th similarity scores is *0.450*, and the minimum value is *0.094*. As a result, we obtain the candidate values of D denoted as $D_{avg} = \ln \frac{1}{0.450} = 0.798$ and $D_{min} = \ln \frac{1}{0.094} = 2.364$.

We further select another *3,000* reference functions randomly to test D_{avg} and D_{min}. According to the ground truth, we find the corresponding matches of the reference functions in target executables. Then, we compare the signature length of each reference function to that of its corresponding target function. We set $D \in (0, 2.5]$ with the step of 0.1, obtaining the accumulative ratio of function pairs versus D as presented in Figure 10. When $D = 0.8$,

93.6% of function pairs are correctly covered, and there exist *193* samples which are incorrectly pruned. We observe the following reasons leading to the incorrectness:

- **Duplicated Functions.** Compilers might create several copies of a function for the resulting executable. They ensure the jumping distance from a caller to its callee is less than the memory page size (e.g., `0x1000` bytes for x86), avoiding page faults when calling functions and improving execution efficiency.
  BINMATCH collects function signatures from (emulated) executions. The signature of the original function would be divided into parts for its copies. Then the signature length of a duplicated function might be much less than that of the original one.
- **Compiler-created Functions.** A Compiler would replace standard library functions with its own efficient ones during compilation. Typically, we find ICC generates binaries inlined with `__intel_fast_*` functions, e.g, `__intel_fast_memcmp`. After stripping symbol names, it is difficult to distinguish those functions from user-defined ones. BINMATCH then records their code features as well which enlarge the signature lengths of their callers.
- **Transformation between `switch` and `if` structures.** In some cases, the `switch` and `if` structures could be transformed between each other equivalently. It is also an optimization strategy of compilers. Comparing to a `switch`, an `if` structure contains more condition comparisons for branches, which corresponds to the code feature of Comparison Operand Values (§3.1), and generates longer signatures.

When $D = 2.4$, 98.8% of samples are correctly handled. We then use BINMATCH to perform similarity comparison for the left *37* samples ($\mathcal{L} = 512$), and find that the differences between the reference and target signatures are so large that none of their scores could be ranked within Top *10* in the resulting lists.

Among the above reasons, *duplicated functions* could be handled by combining their signatures. Since they are exactly identical, it is possible to find all the duplicated functions with static analysis, such as binary function hashing. Then, BINMATCH records code features of duplicated functions as one function signature which could be considered as that of the original one. However, it is difficult to decide whether a function is the compiler-created one in a stripped executable. It is also challenging to unify the representations of `switch` and `if` structures of binary programs. Although we cannot perform pruning correctly for all cases, the above experiments indicate that the possibility is low to make the mistakes when $D = 2.4$. Therefore, in following experiments, BINMATCH adopts

$$\mathcal{P} = e^{2.4} \approx 11.023. \tag{9}$$

5.3 Analysis across Compilation Settings

5.3.1 Cross-optimization Analysis

In this section, we leverage BINMATCH to match binary functions compiled with different optimizations. For a compiler, higher optimization options contain all strategies specified by lower ones. Taking GCC v4.9.3 as an example[5], the option `-O3` enables all the *68* optimizations of `-O2`, and turns on another *9* optimization flags in addition. `-O2` also covers all the *32* strategies specified by `-O1`. Thus, we only discuss the case of `-O3` (E_r) versus `-O0` (E_t), which has larger differences than any other pair of cross-optimization analysis.

Figure 11 shows the accuracy of cross-optimization comparisons between each object program compiled by GCC, Clang, and ICC separately. In Figure 11a, the average accuracy of Top *1*, *5*, and *10* is *68.9%*, *82.5%*, and *87.0%*. The performance of BINMATCH increases notably regrading the Top *5* and *10* target functions in the resulting lists. Thus, the possibility is high for analyzers to find the real match by further considering the first five or ten candidates in a target function list.

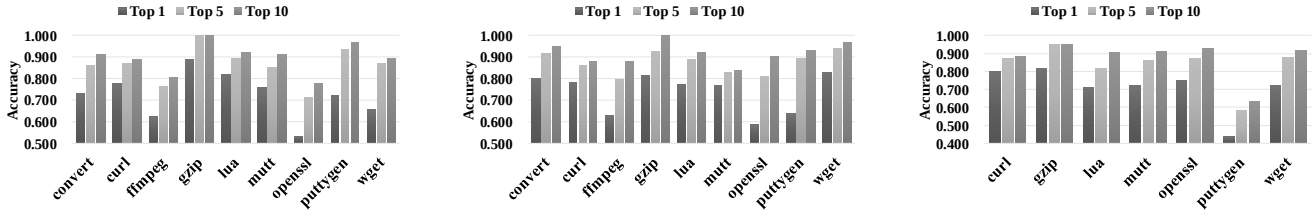
The results of Clang- and ICC-compiled programs are similar. The average accuracy of Top *1*, *5*, *10* is *71.8%*, *85.8%*, *90.9%* in Figure 11b, and *72.6%*, *84.6%*, *89.5%* in Figure 11c. Since ICC fails to generate executables for `convert` and `ffmpeg` with the corresponding compilation settings, we only conduct experiments with the left seven object programs for ICC. In Figure 11c, the results of `puttygen` are much worse than those of other object programs. The Top *1* accuracy of `puttygen` is *43.6%*, while that of every other object exceeds *70.0%*. When generating the executable of `puttygen` optimized with -O3, ICC inlines its own library functions to replace the standard ones, while such optimization is not applied to the -O0 version. The test command of `puttygen` (as presented in Table 1) triggers the reference functions which frequently invoke those inlined by ICC, leading to huge differences in signatures for comparison. BINMATCH then produces the relative low accuracy.

5.3.2 Cross-compiler Analysis

In this section, BINMATCH is evaluated with binaries compiled by different compilers. Similar to the cross-optimization analysis, only the case of `-O3` (E_r) versus `-O0` (E_t) is considered. The results are presented in Figure 12. BINMATCH performs well in most cases. The average Top *1*, *5* and *10* accuracy of all experiments is *65.0%*, *79.4%*, and *84.6%* separately.

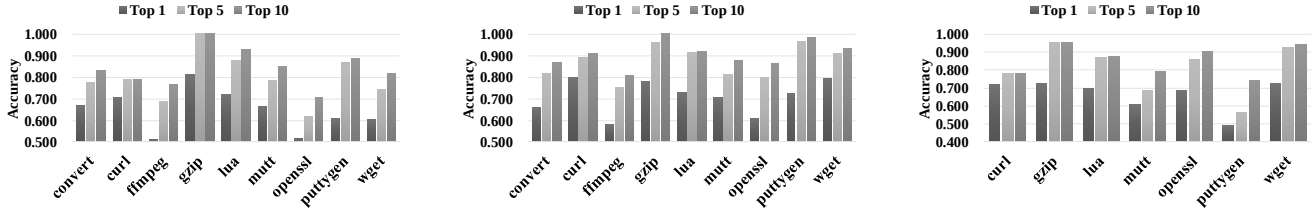
Compiler-created functions of ICC still constitute the reason that affects the performance of BINMATCH. Specifically, the average Top *1* accuracy displayed in Figure 12c and 12f is *67.0%*, while that of ICC -O3 vs. -O0 in Figure 11c is *72.6%*. Additionally, we find *floating-point number* is another reason decreasing the accuracy. GCC leverages x87 floating-point instructions to implement corresponding operations, while Clang and ICC uses the SSE (Streaming SIMD Extensions) instruction set. x87 adopts the FPU (floating point unit) stack to assist in processing floating-point numbers. The operations deciding whether the stack is full or empty insert redundant entries to the semantic signature with comparison operand values (§3.1). In contrast, SSE directly operates on a specific register set (i.e., XMM registers) and has no extra operations. Besides, x87 could handle single precision, double precision, and even 80-bit double-extended precision floating-point calculation, while SSE mainly processes

5. https://gcc.gnu.org/onlinedocs/gcc-4.9.3/gcc/Optimize-Options.html

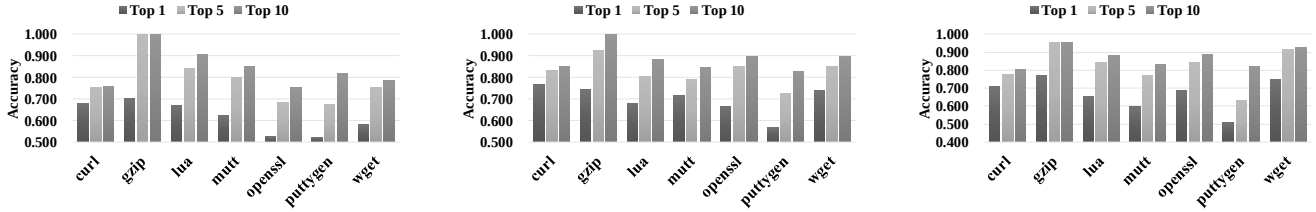


(a) GCC -O3 vs. -O0 (b) Clang -O3 vs. -O0 (c) ICC -O3 vs. -O0

Fig. 11: Accuracy of Cross-optimization Comparison



(a) GCC -O3 vs. Clang -O0 (b) Clang -O3 vs. GCC -O0 (c) ICC -O3 vs. GCC -O0



(d) GCC -O3 vs. ICC -O0 (e) Clang -O3 vs. ICC -O0 (f) ICC -O3 vs. Clang -O0

Fig. 12: Accuracy of Cross-compiler Comparison

single-precision data. Due to the different precision of representations, even though the floating-point numbers are the same, their values generated by different instruction sets are not equal, therefore affecting the accuracy. As a result, when processing executables compiled by GCC (Figure 12a-12d), the average Top *1* accuracy is *63.9%*. In contrast, when the comparisons are performed between Clang and ICC (Figure 12e and 12f), the corresponding accuracy is *68.5%*.

5.3.3 Comparison with Existing Work

In this section, we compare BINMATCH to the state-of-the-art methods Asm2Vec [22], Kam1n0 [21], and the industrial tool BinDiff [24] supported by Google, which are all open for public use. Thus, we could use them to detect similar binary functions with the same settings as BINMATCH. Since BINMATCH is evaluated with the executed reference functions, to make fair comparison, we investigate the performance of the three solutions with those reference functions as well. We configure the three solutions with their default settings, and the results are displayed in Table 2. Similar to BINMATCH, Asm2Vec returns a list of target functions for each reference function. Thus, we also present its accuracy of Top *1*, *5*, *10* respectively. The last two rows show the average accuracy for all the experiments and the processing time of each function on average. Obviously, BINMATCH performs much better than other three from the perspective of accuracy. Specifically, on average, its Top *1* accuracy even outperforms the Top *10* accuracy of Asm2Vec. Besides, benefiting from the adoption of SimHash and the pruning strategy, its average processing time of each function is around *4* seconds. Although BINMATCH is still slower than the other three, considering the accuracy, it deserves the time.

Asm2Vec adopts machine learning techniques for binary function similarity comparison. It treats each path of a function as a document, and leverage the PV-DM model [40] to encode the function into a feature vector. Asm2Vec explores the co-occurrence relationships among assembly code tokens, aiming to describe a binary function with the most representative (or the unique) instructions. However, binary code might be implemented with semantics-equivalent but different kinds of instructions, especially when the code is generated with variant compilation settings.

Kam1n0 and BinDiff are typical solutions which rely on syntax and structure features to detect binary similar functions. Kam1n0 captures features of a function from its control flow graph (CFG), and encodes the features as a vector for indexing. Thus, essentially, it detects similar functions

TABLE 2: Comparison with the state-of-the-art methods `Asm2Vec`, `Kam1n0`, and the industrial tool `BinDiff`. *@K* represents the Top *K* accuracy.

Reference	Target	BINMATCH			Asm2Vec			Kam1n0	BinDiff
		@1	@5	@10	@1	@5	@10		
GCC -O3	GCC -O0	0.689	0.825	0.870	0.444	0.623	0.674	0.288	0.338
	Clang -O0	0.614	0.748	0.808	0.417	0.580	0.629	0.212	0.273
	ICC -O0	0.603	0.756	0.811	0.370	0.553	0.619	0.209	0.277
Clang -O3	Clang -O0	0.718	0.858	0.909	0.425	0.567	0.620	0.251	0.461
	GCC -O0	0.667	0.826	0.871	0.412	0.577	0.627	0.271	0.457
	ICC -O0	0.696	0.825	0.877	0.386	0.622	0.694	0.224	0.400
ICC -O3	ICC -O0	0.726	0.846	0.895	0.323	0.546	0.627	0.276	0.332
	GCC -O0	0.666	0.816	0.865	0.343	0.399	0.465	0.182	0.219
	Clang -O0	0.673	0.808	0.853	0.315	0.533	0.628	0.191	0.212
Average Accuracy		0.672	0.813	0.863	0.395	0.574	0.635	0.240	0.328
Time (s) / Function		4.151			1.255			1.332	0.210

by analyzing graph isomorphism of CFG. The relatively low accuracy of Kam1n0 indicates that compilation settings indeed affect representations of binaries, even though two pieces of code are compiled from the same code base. In addition to measuring the similarity of CFG, BinDiff considers other features to compare similar functions, such as function hashing which compares the hash values of raw function bytes, call graph edges which match functions basing on the dependencies in the call graphs, etc. By carefully choosing suitable features to measure the similarity of functions, BinDiff becomes resilient towards code transformation resulting from different compilers or optimization options to an extent. Therefore, it performs better than Kam1n0, but is still at a disadvantage comparing to BINMATCH.

5.4 Analysis on Obfuscated Code

In this section, we conduct experiments to compare normal binary programs (E_r) with their corresponding obfuscated code (E_t). We adopt OLLVM to obfuscate binary code which is optimized with -O3 and -O0 separately (OLLVM bases the compilation on Clang). Because obfuscation would insert much redundant code, resulting in huge length differences of signatures for comparison, we disable the pruning strategy in this part of experiments, i.e., $\mathcal{P} = +\infty$.

The experimental results are shown in Table 3. Results of Asm2Vec and BinDiff are also presented as references. OLLVM provides three techniques to fulfill the obfuscation. *Instruction substitution* (SUB) replaces standard operators (e.g., addition operators) with sequences of functionality-equivalent, but more complex instructions. It obfuscates code on the *syntax* level, affecting the comparison accuracy of Asm2Vec which treats binaries as documents, but posing fewer threats to BINMATCH which is semantics-based.

Bogus control flow (BCF) adds opaque predicates to a basic block, which breaks the original basic block into two. *Control flow flattening* (FLA) generally breaks a function up into basic blocks, then encapsulates the blocks with a selective structure (e.g., the switch structure) [41]. It creates a state variable for the selective structure to decide which block to execute next at runtime via conditional comparisons. BCF and FLA both change the *structure* of the original function, i.e., modifying the control flow. They insert extra code which is irrelevant to the functionality of the original function, generating redundant semantic features which are indistinguishable from normal ones (e.g., comparison operand values of opaque predicates). Thus, they affect the comparison accuracy of BINMATCH. For the settings of GCC/Clang/ICC -O3 vs. OLLVM -O0, when comparing with functions obfuscated by BCF, the average Top *1* accuracy is *51.3%*, and *48.1%* for FLA, while that of GCC/Clang/ICC -O3 vs. Clang -O0 is *66.8%*. However, BINMATCH still achieves more than *1.5* times the average Top *1* accuracy of Asm2Vec, and *1.7* times of BinDiff, i.e., *63.8%* of BINMATCH, *41.0%* of Asm2Vec, and *37.4%* of BinDiff. On average, the Top *1* accuracy of BINMATCH still outperforms the Top *10* accuracy of Asm2Vec.

Additionally, because the pruning strategy is disabled, BINMATCH spends *21.678* seconds on average to process each reference function, while that of comparisons with pruning is *4.151* seconds. The results indicate the importance of the pruning strategy for improving the efficiency.

5.5 Analysis across Architectures

In this section, we evaluate the capacity of BINMATCH to compare the similarity of binary functions of variant ISAs, across x86, ARM and MIPS. All the executables for comparison are compiled with GCC -O3. The ARM and MIPS binaries are generated or executed in the environments emulated by `QEMU` [42].

The experimental results are presented in Table 4, which are compared to those of CACompare [25], the state-of-the-art cross-architecture similar binary function detector. The average Top *5* accuracy of BINMATCH is comparable to that of CACompare. Besides, for each reference function, BINMATCH is 1.2 seconds faster than CACompare on average. Since there are *3,078* reference functions, BINMATCH then saves about *1* hour for the experiments. In fact, when all the similarity comparisons are performed with Jaccard Index and LCS, i.e., $\mathcal{L} = +\infty$, BINMATCH is able to achieve the average Top *1* accuracy of *86.2%* which indicates the upper bound capacity of BINMATCH. Nevertheless, it needs to spend *137.760* seconds on average in processing each reference function. Thus, BINMATCH has the ability to become

TABLE 3: Accuracy of comparing with obfuscated code. The target executables are obfuscated with OLLVM (BCF: Bogus Control Flow, FLA: Control Flow Flattening, SUB: Instructions Substitution). @*K* represents the Top *K* accuracy.

Reference	Target	Obf.	BINMATCH			Asm2Vec			BinDiff
			@1	@5	@10	@1	@5	@10	
GCC -O3	OLLVM -O3	SUB	0.755	0.873	0.912	0.530	0.710	0.760	0.678
		BCF	0.615	0.722	0.773	0.509	0.676	0.718	0.311
		FLA	0.521	0.629	0.705	0.358	0.529	0.565	0.430
	OLLVM -O0	SUB	0.692	0.821	0.859	0.336	0.489	0.568	0.385
		BCF	0.504	0.561	0.580	0.302	0.473	0.535	0.211
		FLA	0.452	0.495	0.551	0.166	0.280	0.323	0.312
Clang -O3	OLLVM -O3	SUB	0.890	0.977	0.985	0.766	0.874	0.897	0.805
		BCF	0.647	0.752	0.842	0.624	0.777	0.814	0.334
		FLA	0.560	0.616	0.682	0.470	0.633	0.680	0.541
	OLLVM -O0	SUB	0.758	0.901	0.942	0.351	0.525	0.590	0.497
		BCF	0.532	0.592	0.616	0.356	0.524	0.567	0.053
		FLA	0.485	0.546	0.611	0.241	0.398	0.441	0.280
ICC -O3	OLLVM -O3	SUB	0.621	0.750	0.805	0.482	0.632	0.691	0.609
		BCF	0.718	0.848	0.896	0.398	0.558	0.622	0.208
		FLA	0.473	0.523	0.609	0.292	0.414	0.469	0.272
	OLLVM -O0	SUB	0.730	0.847	0.876	0.264	0.409	0.482	0.248
		BCF	0.501	0.570	0.605	0.254	0.368	0.423	0.095
		FLA	0.507	0.556	0.631	0.148	0.239	0.296	0.160
Average Accuracy			0.638	0.736	0.784	0.410	0.560	0.613	0.374

TABLE 4: Performance of cross-architecture comparison. All executables are compiled with GCC -O3, but with different instruction set architectures. @*K* represents the Top *K* accuracy.

Settings	BINMATCH			CACompare
	@1	@5	@10	
x86 vs. ARM	0.628	0.743	0.798	0.807
x86 vs. MIPS	0.721	0.827	0.866	0.770
ARM vs. MIPS	0.703	0.816	0.867	0.810
Average Accuracy	0.667	0.789	0.839	0.795
Time (s) / Function	3.451			4.694

more accurate than CACompare, but it also requires more time. According to the scenarios and requirements, users could choose the suitable $\mathcal{L}$ for it to balance accuracy and efficiency.

CACompare samples a function with random values as inputs, and extracts the code features via emulation as well. Illegal memory accessing is also tackled by providing random values. However, the random values lack semantics. They could hardly bypass the input checks of a function, and usually trigger paths which handle exceptions. In contrast, BINMATCH captures the signatures of target functions by emulating them with runtime values migrated from real executions. As a result, in some cases, BINMATCH is more robust than CACompare. Besides, it could generate results with higher accuracy if there is no strict time limit.

5.6 Threats to Validity

We construct the dataset of the experiments by compiling binaries from nine open-source projects (§5.1.1). Besides, we conduct experiments to infer suitable parameter values for BINMATCH (§5.2). Although the dataset consists of various types of programs, it cannot cover all cases in the real world, neither can the corresponding parameter values of $\mathcal{L}$ and $\mathcal{P}$.

BINMATCH adopts `IDA Pro` to acquire information of binary functions (§4.1). However, function boundary identification of `IDA Pro` is not perfect, which is actually still an issue of reverse engineering [43], [44], [45]. Additionally, BINMATCH is implemented with `Valgrind` and `angr` which both adopt VEX-IR as the intermediate representation (§4.2). However, VEX-IR is not perfect that 16% x86 instructions could not be lifted, although only a small subset of instructions is used in executables in practice and VEX-IR could handle most cases [46]. The incompleteness of VEX-IR might affect the accuracy of semantics signature extraction, while BINMATCH still produces promising results in above experiments.

6 DISCUSSION AND FUTURE WORK

6.1 Application Scope and Scenarios

In this paper, BINMATCH is implemented to process 32-bit code. The solution could be applied to 64-bit code as well. To fulfill the target, there might exist the following problems:

- *Calling Conventions:* BINMATCH identifies and assigns the arguments of a binary function according to its calling convention (§3.2 and §3.3.2). 64-bit instruction set architectures commonly prepare arguments with specific registers, e.g., RDI, RSI, RDX, RCX, R8, R9 of x86-64 on Linux, and additional ones are passed via the stack. Thus, we need to consider those specific registers firstly, then analyze the stack if necessary.
- *Floating-point Numbers:* Different instruction set architectures employ instructions of various precision to process floating-point numbers, such as x87 and SSE

of x86, SSE2 of x86-64. That would affect the detection accuracy of BINMATCH (§5.3.2). A possible solution is to unify the precision of floating-point values, representing the high-precision value with lower precision, e.g., representing double-precision values with single-precision. That would be left as future work.

Because BINMATCH needs to execute the reference functions, it is more suitable for scenarios where the reference functions have available test cases, such as known vulnerability detection (as shown in §2.1), patch analysis [13], [47]. In contrast, static methods are applicable to cases which require the high coverage of the reference code, such as plagiarism detection [2]. Dynamic methods are appropriate for the situation where the code behaviors are emphasized or the capacity of deobfuscation is required, such as malware lineage analysis [11].

6.2 Obfuscation

In the evaluation, BINMATCH is shown to be effective in analyzing obfuscated binary code which is generated by `OLLVM` (§5.4). The robust of BINMATCH is due to the nature of dynamic analysis and the adoption of semantics-based signatures. However, that does not mean BINMATCH could handle all kinds of obfuscations. Besides, the `OLLVM` code actually affects the accuracy of BINMATCH in the experiments. When analyzing benign code, BINMATCH achieves better results. For GCC/Clang/ICC -O3 vs. Clang -O0, the average Top *1*, *5*, and *10* accuracy is *66.8%*, *80.4%* and *85.8%*, while for GCC/Clang/ICC -O3 vs. OLLVM -O0, the corresponding ratio is *60.9%*, *70.1%*, and *74.0%* respectively. In the literature, deobfuscation has been well studied [48], [49], [50], [51]. Therefore, if BINMATCH fails to detect an obfuscated function, it is a better choice to deobfuscate it firstly, then perform further analysis.

6.3 Function Interfaces

In this paper, we assume a pair of matched functions shares the same interface, i.e., the same argument number and order. When the interface of the target function is modified, e.g., by obfuscation, BINMATCH becomes ineffective. For example, the reference function *R* has the interface

```
R(rarg_0, rarg_1, rarg_2),
```

while the interface of its corresponding match (target function *T*) is

```
T(targ_2, targ_1, targ_0, targ_3).
```

Note that not only a redundant argument `targ_3` is added to *T* via obfuscation, but also the first three arguments are disordered. For `targ_3`, as described in the previous section, extra analysis is necessary, such as deobfucation. That is out of the scope of this paper. For the disordering, a possible solution is to provide *T* with the permutation of *R*'s argument list. In the example, after `targ_3` is removed, BINMATCH generates the permutation of *R*'s argument list, overall *6* ($= P_3^3$) cases, then assigns them to *T* and computes the similarity score separately. The largest one among the six values, theoretically when the order is (`rarg_2`, `rarg_1`, `rarg_0`), is considered as the final similarity score of (*R*, *T*). It is left as future work.

6.4 Accuracy vs. Efficiency

It is a classical issue of program analysis. In this paper, since the lengths of signatures extracted via (emulated) executions are huge, we propose the hybrid method, combining LCS and SimHash for similarity comparison, to reach the compromise between efficiency and accuracy (§3.4). To improve the accuracy, it is possible to execute the reference function with different inputs to capture more semantics information and generate the signature. Furthermore, we could adopt the metrics from testing to evaluate each run of the reference function, such as delta code coverage [52]. Specifically, BINMATCH only records the signature extracted from the execution covering enough new code of the function, which is not executed before. It is left as future work.

7 RELATED WORK

Binary code similarity comparison (or clone detection) has many important applications in fields of software engineering and security, typically including plagiarism detection [19], [53], bug detection [15], malware analysis [11], etc.

Syntax and structural features are widely adopted to detect binary clone code. Sæbjørnsen et al. [54] detect binary clone code basing on opcode and operand types of instructions. Hemel et al. [55] treat binary code as text strings and measure similarity by data compression. The higher the compression rate is, the more similar the two pieces of binary code are. Khoo et al. [5] leverage n-gram to compare the control flow graph (CFG) of binary code. David et al. [6] measure the similarity of binaries with the edit distances of their CFGs. BinDiff [24] and Kam1n0 [21] extract features from the CFG and call graphs to search binary clone functions.

As discussed earlier in this paper, the main challenge of binary code similarity comparison is semantics-equivalent code transformation resulting from link-time optimization, obfuscation, etc. Because of the transformation, representations of binary code are altered tremendously, even though the code is compiled from the same code base. Therefore, syntax and structure-based methods become ineffective, and semantics-based methods prevail. Jhi et al. [2] and Zhang et al. [3] leverage runtime invariants of binaries to detect software and algorithm plagiarism. Ming et al. [11] infer the lineage of malware by code similarity comparison with the system call traces as the semantic signature. However, those solutions require the execution of binary programs and cannot cover all target functions. Egele et al. [18] propose blanket execution to match binary functions with full code coverage which is achieved at the cost of detection accuracy. Luo et al. [19], [53] and Zhang et al. [4] detect software plagiarism by symbolic execution. Although their methods are resilient to code transformation, symbolic execution is trapped in the performance of SMT/SAT solvers which cannot handle all cases, e.g., indirect calls. David et al. propose Esh [56] which decomposes the CFG of a binary function into small blocks and measures the similarity of the small blocks basing on a statistical model. However, the boundaries of CFG blocks would be changed by code transformation, affecting the accuracy of the method. BinSequence [57] explores the control flow graphs of binary functions and aligns the paths for similarity measurement.

Then, it would suffer from control flow transformation, e.g., control flow flattening.

More recently, with the prevalence of IoT devices, binary code similarity comparison is proposed to perform on ARM and MIPS, or even across architectures. Multi-MH [15], discovRE [16], Genius [17], and Xmatch [58] are proposed to detect known vulnerabilities and bugs in multi-architecture binaries via code similarity comparison. BinGo [7], CACompare [25], and GitZ [59] are proposed to analyze the similarity of binary code across architectures as well. However, discovRE and Genius still heavily depend on the CFG of a binary function. Xmatch extracts symbolic expressions of a binary function as the features, and treats them as sets for similarity comparison, ignoring the relative order (semantics information) of the expressions. Multi-MH, BinGo and CACompare sample a binary function with random values to capture corresponding I/O values as the signature, while the random values are meaningless that they merely trigger limited behaviors of the function. Thus, it is difficult for them to cover the core semantics of a binary function. Similar to Esh, GitZ bases the analysis on blocks of functions as well. It lifts LLVM-IR from function blocks, and counts on the optimization strategies of Clang to normalize the representations of the IR code. Then, it measures the similarity of IR code syntactically. Following their previous work (Esh and GitZ), David et al. propose FirmUp [60] to detect known vulnerabilities of firmware. They consider the similarity comparison as a back-and-forth game which further ensures the accuracy of the detection. BinArm [61] is propose to detect known vulnerabilities of firmware as well. It introduces a multi-stage strategy which firstly filters functions with the syntax and structure information, then performs graph matching of control flow graphs.

Additionally, machine learning techniques are also adopted for binary code similarity comparison. Xu et al. [62] leverage the neural network to encode CFGs of binary code into vectors. αDiff [63] trains the convolutional neural network with raw bytes of binary code, generating the best parameter values for the model. Then they apply the model to comparing the similarity of binary code. Thus, essentially, the two solutions still process with syntax and structure features. Asm2Vec [22] considers the assembly code disassembled from the binary code as documents. It attempts to discover the semantics hidden in the co-occurrence relationships among the assembly tokens, and adopts the most representative instructions as the feature of a function. However, Asm2Vec still originates in the text of assembly code as well. It is not robust enough to the semantics-equivalent code transformation, as indicated in the experiments (§5.3.3). Inspired by the work of Luo et al. [19], Zuo et al. [64] firstly compute the likeness of basic blocks, then align the basic blocks of a path, and finally infer the similarity of code components with multiple paths. They regard binary code as natural language, embedding instructions basing on word2vec [65], then adopting the neural machine translation model to compare basic blocks in a deep learning manner. Comparing to the previous basing on symbolic execution, the solution becomes much more efficient.

To sum up, the topic of binary code similarity comparison mainly focuses on two points: i) what signature to adopt, such as opcodes and operand types (syntax), CFG (structure) and system calls (semantics); ii) how to capture the signatures, such as statically disassembling, sampling, or dynamically running, etc. BINMATCH leverages the combination of output values, comparison operand values, and invoked standard library functions as the signature which is able to better reveal the semantics of a binary function. Besides, it captures the signature via both execution and emulation, which not only ensures the richness of semantics, but also covers all target functions to be analyzed.

8 CONCLUSION

Binary code similarity comparison is a fundamental methodology which has many important applications in fields of software engineering and security. In this paper, we propose BINMATCH to compare the similarity of binary code. BINMATCH completely relies on semantics-based signatures which are extracted either in a static or in a dynamic manner, via (emulated) executions. Thus, it is able to achieve high comparison accuracy and coverage at the same time. Besides, to balance accuracy and efficiency, in addition to the longest common subsequence algorithm, the accurate string matching method, BINMATCH also adopts the approximate matching technique SimHash for the function signature similarity measurement. The experimental results show that BINMATCH not only is robust to the semantics-equivalent code transformation caused by different compilation settings, commonly-used obfuscations, and variant ISAs, but also fulfills the function comparison efficiently. Additionally, BINMATCH also achieves better performance than the state-of-the-art solutions as well as industrial tool of binary code similarity comparison.

REFERENCES

[1] Y. Hu, Y. Zhang, J. Li, H. Wang, B. Li, and D. Gu, "Binmatch: A semantics-based hybrid approach on binary code clone analysis," in *2018 34th International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018.

[2] 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," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, 2011.

[3] F. Zhang, Y.-C. Jhi, D. Wu, P. Liu, and S. Zhu, "A first step towards algorithm plagiarism detection," in *Proceedings of the 21th International Symposium on Software Testing and Analysis (ISSTA)*, 2012.

[4] F. Zhang, D. Wu, P. Liu, and S. Zhu, "Program logic based software plagiarism detection," in *Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2014.

[5] W. M. Khoo, A. Mycroft, and R. Anderson, "Rendezvous: A search engine for binary code," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. IEEE Press, 2013, pp. 329–338.

[6] Y. David and E. Yahav, "Tracelet-based code search in executables," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI'14, 2014.

[7] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: Cross-architecture cross-os binary search," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2016.

[8] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Cross-architecture binary semantics understanding via similar code comparison," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 57–67.

[9] W. Andrew and L. Arun, "The software similarity problem in malware analysis," in *Duplication, Redundancy, and Similarity in Software*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[10] M. Lindorfer, A. Di Federico, F. Maggi, P. M. Comparetti, and S. Zanero, "Lines of malicious code: Insights into the malicious software industry," in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*. ACM, 2012.

[11] J. Ming, D. Xu, Y. Jiang, and D. Wu, "Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking," in *Proceedings of the 26th USENIX Security Symposium. USENIX Association*, 2017, pp. 253–270.

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

[13] H. Zhang and Z. Qian, "Precise and accurate patch presence test for binaries," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 887–902.

[14] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14. ACM, 2014, pp. 406–415.

[15] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 709–724.

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

[17] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *23rd ACM Conference on Computer and Communications Security (CCS)*, 2016.

[18] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: dynamic similarity testing for program binaries and components," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, 2014.

[19] 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 *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014.

[20] S. Wang and D. Wu, "In-memory fuzzing for binary code similarity analysis," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 319–330.

[21] S. H. Ding, B. Fung, and P. Charland, "Kam1n0: Mapreduce-based assembly clone search for reverse engineering," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 461–470.

[22] 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)*. IEEE, 2019, pp. 38–55.

[23] R. Harper, *Practical Foundations for Programming Languages*, 2nd ed. New York, NY, USA: Cambridge University Press, 2016.

[24] H. Flake, "Structural comparison of executable objects," *Proceedings of the First International Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2004.

[25] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Binary code clone detection across architectures and compiling configurations," in *Proceedings of the 25th International Conference on Program Comprehension*, ser. ICPC'17. IEEE Press, 2017, pp. 88–98.

[26] U. Kargén and N. Shahmehri, "Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 782–792.

[27] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Behavior based software theft detection," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2009.

[28] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei, "Can i clone this piece of code here?" in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 170–179.

[29] L. Hamers *et al.*, "Similarity measures in scientometric research: The jaccard index versus salton's cosine formula." *Information Processing and Management*, vol. 25, no. 3, pp. 315–18, 1989.

[30] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*. IEEE, 2000, pp. 39–48.

[31] R. Hamming, "Error Detecting and Error Correcting Codes," *Bell System Techincal Journal*, vol. 29, pp. 147–160, 1950.

[32] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '02. ACM, 2002, pp. 380–388.

[33] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *25th USENIX Security Symposium (USENIX)*. USENIX Association, 2016.

[34] X. Meng and B. P. Miller, "Binary code is not easy," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. ACM, 2016, pp. 24–35.

[35] N. Nethercote and J. Seward, "Valgrind: a framework for heavy-weight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[36] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 138–157.

[37] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W.-M. W. Hwu, "Profile-guided automatic inline expansion for c programs," *Software: Practice and Experience*, vol. 22, no. 5, pp. 349–369, 1992.

[38] A. Balakrishnan and C. Schulze, "Code obfuscation literature survey," *CS701 Construction of compilers*, vol. 19, 2005.

[39] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, B. Wyseur, Ed. IEEE, 2015, pp. 3–9.

[40] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proceedings of the 31st International Conference on International Conference on Machine Learning*, ser. ICML'14. JMLR.org, 2014, pp. II–1188–II–1196.

[41] T. László and Á. Kiss, "Obfuscating c++ programs via control flow flattening," *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, vol. 30, pp. 3–19, 2009.

[42] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41.

[43] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to recognize functions in binary code," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 845–860.

[44] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks." in *USENIX Security Symposium*, 2015, pp. 611–626.

[45] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries." in *USENIX Security Symposium*, 2016, pp. 583–600.

[46] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha, "Testing intermediate representations for binary analysis," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 353–364.

[47] R. Duan, A. Bijlani, Y. Ji, O. Alrawi, Y. Xiong, M. Ike, B. Saltaformaggio, and W. Lee, "Automating patching of vulnerable open-source software versions in application binaries," in *The Network and Distributed System Security Symposium*, 2019.

[48] S. K. Udupa, S. K. Debray, and M. Madou, "Deobfuscation: Reverse engineering obfuscated code," in *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE)*, 2005.

[49] B. Yadegari and S. Debray, "Symbolic execution of obfuscated code," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 732–744.

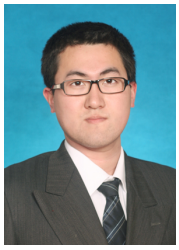
[50] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 674–691.

[51] D. Xu, J. Ming, Y. Fu, and D. Wu, "Vmhunt: A verifiable approach to partially-virtualized binary code simplification," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 442–458.
[52] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *The Network and Distributed System Security Symposium*, 2017.
[53] 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 Transactions on Software Engineering*, vol. 43, no. 12, pp. 1157–1177, 2017.
[54] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA)*, 2009.
[55] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, "Finding software license violations through binary code clone detection," in *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR)*, 2011.
[56] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI'16. New York, NY, USA: ACM, 2016, pp. 266–280.
[57] H. Huang, A. M. Youssef, and M. Debbabi, "Binsequence: fast, accurate and scalable binary code reuse detection," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 155–166.
[58] Q. Feng, M. Wang, M. Zhang, R. Zhou, A. Henderson, and H. Yin, "Extracting conditional formulas for cross-platform bug search," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 346–359.
[59] Y. David, N. Partush, and E. Yahav, "Similarity of binaries through re-optimization," in *ACM SIGPLAN Notices*, vol. 52, no. 6. ACM, 2017, pp. 79–94.
[60] ——, "Firmup: Precise static detection of common vulnerabilities in firmware," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 392–404.
[61] P. Shirani, L. Collard, B. L. Agba, B. Lebel, M. Debbabi, L. Wang, and A. Hanna, "Binarm: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2018, pp. 114–138.
[62] 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 *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS'17. New York, NY, USA: ACM, 2017, pp. 363–376.
[63] 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 *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE'18. New York, NY, USA: ACM, 2018, pp. 667–678.
[64] F. Zuo, X. Li, Z. Zhang, P. Young, L. Luo, and Q. Zeng, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *The Network and Distributed System Security Symposium*, 2019.
[65] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proceedings of the International Conference on Learning Representations*, 2013.

Yikun Hu received the BS degree in Computer Science and Engineering from South China University of Technology, Guangzhou, China. He is currently working toward the PhD degree in the Department of Computer Science and Engineering at Shanghai Jiao Tong University, Shanghai, China. His main interests include binary program analysis and software repair.

Hui Wang received the BS degree in Software Engineering from Harbin Institute of Technology, Harbin, China, in 2013. He is currently working toward the PhD degree in the Department of Computer Science and Engineering at Shanghai Jiao Tong University, Shanghai, China. His current research interests include protocol security and mobile security.

Yuanyuan Zhang received her BS and MS degrees from Wuhan University, and the PhD degree from Shanghai Jiao Tong University. She is currently an associate professor in the Department of Computer Science and Engineering at Shanghai Jiao Tong University. Before joining Shanghai Jiao Tong University, she worked in INSA de Lyon as a postdoctoral fellow. Her research interests include network security, architecture security and system security.



Bodong Li received his BS degree in Computer Science and Engineering from Shanghai Jiao Tong University, Shanghai, China. He is currently working toward the PhD degree in the Department of Computer Science and Engineering at Shanghai Jiao Tong University. His current research interests include mobile security, security assessment and malware analysis.

Dawu Gu received the BS degree in applied mathematics in 1992, and the MS and PhD degrees in cryptography in 1998, both from Xidian University of China. He is currently a distinguished professor in the Department of Computer Science and Engineering at Shanghai Jiao Tong University (SJTU) of China. His research interests include cryptography, side channel attack and software security. He leads the Laboratory of Cryptology and Computer Security (LoCCS) at SJTU. He has published over 150 scientific papers in academic journals and conferences, and has owned 30 innovation patents. He was the winner of Chang Jiang Scholars Program by the ministry of education of China. He won the prize of national scientific and technological progress of China in 2017. He serves as executive member of the council of China Association of Cryptologic Research and member of the council of China Computer Federation. He also serves as several technical editors for China Communications, J. Cryptologic Research, and Information Network Security, etc.