2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering

Cross-Architecture Binary Semantics Understanding via Similar Code Comparison

Yikun Hu, Yuanyuan Zhang, Juanru Li, Dawu Gu
Department of Computer Science and Engineering
Shanghai Jiao Tong University
Shanghai, China

***Abstract*—With the prevailing of smart devices (e.g., smart phone, routers, cameras), more and more programs are ported from traditional desktop platform to embedded hardware with ARM or MIPS architecture. While the compiled binary code differs significantly due to the variety of CPU architectures, these ported programs share the same code base of the desktop version. Thus it is feasible to utilize the program of commodity computer to help understand those cross-compiled binaries and locate functions with similar semantics. However, as instruction sets of different architectures are generally incomparable, it is difficult to conduct a static cross-architecture binary code similarity comparison.**

To address, we propose a semantic-based approach to fulfill this target. We dynamically extract the signature, which is composed of conditional operations behaviors as well as system call information, from binaries on different platforms with the same manner. Then the similarity of signatures is measured to help identify functions in ported programs. We have implemented the approach in MOCKINGBIRD, an automated analysis tool to compare code similarity between binaries across architectures. MOCKINGBIRD supports mainstream architectures and is able to analyze ELF executables on Linux platform. We have evaluated MOCKINGBIRD with a set of popular programs with cross-compiled versions. The results show our approach is not only effective for dealing with this new issue of cross-architecture binary code comparison, but also improves the accuracy of similarity based function identification due to the utilization of semantic information.

I. INTRODUCTION

Detecting code with similar functionality contributes enormously to the understanding of programs, especially for binary executables of which the reverse engineering is extremely time-consuming. If programs share same code base (e.g., latest version and historical version of a software product) and one of them has been analyzed, the knowledge of this analyzed program can be transferred to the analysis of the others and helps save the time of analyzing similar or identical part of code. Nonetheless, even for binary code of one specific program, there exist different versions with signification variation. There are two main reasons leading to that variation. First, more and more programs nowadays are cross-compiled for different CPU architectures to support not only commodity computers but also smart devices such as smart phones and routers. Those cross-compiled binaries from different architectures vary in instruction sets, code offsets and function calling conventions. The wide diversity of different Instruction Set Architectures (ISA) makes it hard to compare binary code of one particular architecture to that of another architecture statically even if they are compiled with the same code base. Moreover, during the compilation process the same code base may be compiled with different compilers using different configurations (e.g., different optimization levels). This also brings significant change to the representations of binaries. In this situation, conducting static syntactic-based code similarity comparison is less effective or even infeasible.

To tackle the obstacle of syntactic difference and achieve accurate code similarity detection, analysts make use of code semantics to help compare different functions. Various semantic features are made use of to facility semantic-based comparisons. However, they are not suitable for above user case. The approach based on system call birthmarks [26], [27] suffers from insufficiency of system calls. Approaches leveraging symbolic formulas [14], [30] is difficult to compare binaries across architectures, as there is no mature framework for symbolic execution on different ISAs. The feature of core values [10], [29] requires source code of programs under test, which is infeasible for binary analysis.

In this paper, we address the problem by proposing a novel and generic semantic signature. Our proposed semantic signature is a sequence of two dynamic features: Comparison Operand Pairs (COP) and System Call Attributes (SCA). A COP is a pair of operand values belonging to a comparison instruction (e.g., the `CMP` x86 instruction) whose operation result directly decides the following control flow. SCAs consist of names and argument values of all system calls invoked in an execution, which reflect the behaviors of the program. These two features are both dynamic runtime information and are less influenced by the implementation variance or instruction set difference. Thus they are suitable for fingerprinting certain functionality. In addition, the generation of our semantic signature leverages `VEX-IR`, a RISC-like intermediate representation defined by the Valgrind [18] open-source dynamic binary instrumentation tool, to achieve multi-architecture analysis through representing instruction sets of different architectures with a uniform style. Thus we can instrument programs with different instruction sets and extract semantic signatures dynamically.

Suppose we have complete knowledge of a binary compiled for one platform from a code base, we can utilize this version as a template to help identify particular functions of a target binary compiled from the same code base (but with different instruction set, and compiled using different compilers with different configuration). Generally, although the source code is unavailable, it is feasible to execute the target binary on different platforms, which guarantees us the capability of semantic signature extracting. We first translate both template binary and target binary into `VEX-IR` and use Valgrind to instrument those executables. These two executables are then executed by given exactly the same input and executions are monitored by our instrumentation module, which is responsible for extracting COPs and SCAs. The extracted runtime information and the translated IR are finally analyzed to generate normalized semantic signatures for each function. With the generated signatures, we can compare the code similarity between template binary and target binary by calculating the distance of function signature sequence, and detect the most similar function pairs. In this way, we are able to locate the corresponding functions in target binary with the prior knowledge of the functions in template binary and therefore decrease the cost of unnecessary analysis.

We have built a prototype system called MOCKINGBIRD to fulfill the mentioned code similarity detection automatically. To show the effectiveness, we evaluate MOCKINGBIRD with 10 different utilities including Internet downloader, data compressor, code interpreter, e-mail client, and file format converter. These utilities are all cross-compiled to different architectures and are compiled with variant compiling configurations to generate multiple versions. MOCKINGBIRD is able to monitor their executions, extract semantics signatures and compare the code similarity, producing accurate matching results. Compared to the results in [19], which only achieves a 32.1% accuracy rate for the comparison between *OpenSSL* binaries for ARM and MIPS , our tool gives an accuracy rate of 82.1% when identifying *OpenSSL* relevant functions. While the accuracy rate of BLEX [6], another tool that employs the *blanket execution* technique, is 64% when it functions as a search engine, MOCKINGBIRD produces an average accuracy rate of 78.47%. These results indicate that our approach is promising on cross-architecture code similarity detection.

In summary, the contributions of this work are as follows:

- We propose a generic approach to handle binaries compiled for various architectures and fulfill code similarity detection. Our approach addresses the instruction set diversity through adopting `VEX-IR` and semantic signatures based analysis, which is suitable for searching similar functions among not only cross-compiled programs but also those with difference compilation configurations.

- We define a fine-grained semantic signature for function-level similarity detection. To improve the generality, we further introduce a series of techniques including *Pointer abstraction*, *Canary removal*, *Boundary unifying*, and *Loop compression* to normalize signatures and thus make our comparison more adaptive.

- We implement MOCKINGBIRD reverse engineering tool to support program analysis and code comparison across three mainstream architectures (IA-32, ARM and MIPS). Our evaluation on daily use utilities shows that our approach not only improves the average accuracy for common application scenarios, but also provides analyst a new methodology on cross-architecture program analysis.

II. APPROACH

In this section, we introduce our approach to understand semantics of particular binary function by similar code comparison. Suppose we have understood the semantics of functions in a binary (the template binary). To understand particular functions of other binaries (the target binary) compiled from the same code base but possibly for other architectures and with different compiling configurations, we compare the target binary to the template to locate functions of similar semantics. The comparison mainly relies on our proposed semantic signatures (Section II-A), and starts from providing the template and target binaries with the same input to extract semantic signature for each function executed (Section II-B). Then, the extracted signatures are normalized considering the semantic differences resulted from architectures and compiling configurations (Section II-C). Afterwards, signatures of target functions are compared to those of template functions, and similarity scores are calculated (Section II-D). Finally, we obtain a list of scores for every target function, and the pair of functions with the highest score is treated as a match.

A. Semantic Signatures

We mainly define the semantic signature with the concepts of COP and SCA. At the assembly level, a COP is the value pair of operands of the comparison instruction, which introduces condition test in an execution and decides the jump target of following branch instruction. For instance, the corresponding IA-32 code in Table I for Line 2 of Figure I is `cmp [ebp+arg_0], 0`, and the value of those two operands (*[ebp+arg_0]* and *0*) is defined as a COP. These two values are compared by the `CMP` instruction, and the result of comparison decides whether the conditional jump (to *loc_804848A*) in the following `JLE` instruction is employed.

Comparison instructions convert control dependencies into data dependencies. When semantic-similar functions are executed with the same input, the possibility that observed behaviors are similar is high [6]. In our cases, if two binaries compiled from the same code base show equal behaviors with the same input, the similarity are reflected by comparisons in those two executions that decide control flows. Thus, we utilize COP sequence to detect similarity of dynamic behavior of program and collect COP sequence to help build semantic signatures. Take the code in Listing 1 as an example. If the value of parameter *num* is *5*, then for different assembly code compiled from the same source code in Table I, the extracted COP sequences are all $\{(5, 0), (1, 0)\}$.

In addition, our semantic signature contained SCA is composed of name and arguments of a system call invoked in an execution, which also indicates the semantics of that execution. Listing 2 presents the system calls related to function in Listing 1. Due to the *printf* function at Line 5 in Listing 1, *sys_write* is invoked at Line 4, indicating that the program performed a write operation in that execution. And the corresponding SCA is represented as ("sys_write", *Hash*("1, tag, 10")). Note that the second arguments of *sys_write*, *0x4038000*, is the address of variable *num* and such data pointers may differ from each execution because of address randomization. Thus we normalize these pointer values as *tag* (which is detailed in Section II-C), hash the argument list of

TABLE I: Assembly code of binaries of function in Listing 1 and corresponding IR

	IA-32	ARM	MIPS
Assembly Code	var_C = dword ptr -0Ch arg_0 = dword ptr 8 push ebp mov ebp, esp sub esp, 28h cmp [ebp+arg_0], 0 ; COP jle short loc_804848A mov eax, [ebp+arg_0] and eax, 1 mov [ebp+var_C], eax cmp [ebp+var_C], 0 ; COP jz short loc_804848A mov eax, [ebp+arg_0] mov [esp+4], eax mov dword ptr [esp], offset format call _printf loc_804848A: leave retn	var_C = -0xC var_4 = -4 PUSH {R7,LR} SUB SP, SP, #0x10 ADD R7, SP, #0 STR R0, [R7,#0x10+var_C] LDR R3, [R7,#0x10+var_C] CMP R3, #0 ; COP BLE loc_841C LDR R3, [R7,#0x10+var_C] AND.W R3, R3, #1 STR R3, [R7,#0x10+var_4] LDR R3, [R7,#0x10+var_4] CMP R3, #0 ; COP BEQ loc_841C MOV R3, #unk_84A4 MOV R0, R3 ; format LDR R1, [R7,#0x10+var_C] BLX printf loc_841C AND.W R7, R7, #0x10 MOV SP, R7 POP {R7,PC}	var_10 = -0x10 var_8 = -8 var_4 = -4 arg_0 = 0 addiu $sp, -0x28 sw $ra, 0x28+var_4($sp) sw $fp, 0x28+var_8($sp) move $fp, $sp sw $a0, 0x28+arg_0($fp) lw $v0, 0x28+arg_0($fp) blez $v0, loc_4006E0 # COP or $at, $zero lw $v0, 0x28+arg_0($fp) andi $v0, 1 sw $v0, 0x28+var_10($fp) lw $v0, 0x28+var_10($fp) beqz $v0, loc_4006E0 # COP or $at, $zero la $v0, unk_4008E0 move $a0, $v0 # format lw $a1, 0x28+arg_0($fp) jal printf or $at, $zero loc_4006E0: move $sp, $fp lw $ra, 0x28+var_4($sp) lw $fp, 0x28+var_8($sp) addiu $sp, 0x28 jr $ra or $at, $zero

VEX-IR	t10 = LDle:I32(t21) ; load *num*
	t31 = CmpLE32S(**t10,0x0**:I32) ; **COP**
	if (t31) { PUT(68) = 0x804848A:I32 }
	t1 = And32(t10,0x1:I32)
	t25 = CmpEQ32(**t1,0x0**:I32) ; **COP**
	if (t25) { PUT(68) = 0x804848A:I32 }
		t65 = LDle:I32(t44) ; load *num*
		t67 = CmpLE32S(**t65,0x0**:I32) ; **COP**
		if (t67) { PUT(68) = 0x841C:I32 }
		t7 = And32(t65,0x1:I32)
		t57 = CmpEQ32(**t7,0x0**:I32) ; **COP**
		if (t57) { PUT(68) = 0x841C:I32 }
			t20 = LDle:I32(t18) ; load *num*
			t21 = CmpLE32S(**t20,0x0**:I32) ; **COP**
			if (t21) { PUT(128) = 0x4006E0:I32 }
			t7 = And32(t6,0x1:I32)
			t15 = CmpEQ32(**t7,0x0**:I32) ; **COP**
			if (t15) { PUT(128) = 0x4006E0:I32 }

Listing 1: A function for printing positive odd number

```
void print_positive_odd(int num){
    if (num > 0){
        int flag = num & 1;
        if (flag != 0)
            printf("%d\n", num);
        return;
    }
}
```

Listing 2: System call log for code in Listing 1 on IA-32

```
sys_read (0, 0x4037000, 1024)
sys_fstat64 (1, 0xbeb86da0)
sys_mmap2 (0x0, 4096, 3, 34, -1, 0)
sys_write (1, 0x4038000, 10)
```

each system call using simple string hash function, and form the SCA along with the name of the system call.

B. Signatures Extraction

We extract COP from executed `VEX-IR` traces, which are obtained by instrumenting binaries. Because `VEX-IR` unifies assembly code for different architectures with uniform syntax, it becomes much easier to extract COP. As presented in Table I, at the assembly level, COPs of different architectures are expressed by distinct instructions. After the unification, they are all involved in comparison statements (`CmpLE32S` and `CmpEQ32`) of IR. We observe that COP is usually an operands pair of comparison statements: `Cmp` (integer comparison) and `Cmpf` (float comparison). So we just emulate the execution with IR trace, and locate those comparison statements whose operands are potential COPs. They are finally recorded as COP once encountering control flow related statements whose operation depends on values of them.

Control flow related statements include *conditional exit* statement, *guard store* statement, *guard load* statement and *if then else* statement. *Conditional exit* statement appears as exit statement of an IR code block. Its format is:

`if` (<*guard*>) `goto` <*dst*>

where *guard* is a conditional expression. If it is true, the program jumps to address stored in *dst*, or executes the next statement sequentially if it is false. *Conditional exit* statement is translated from conditional jumps of assembly code, such as *jz* of IA-32, *beq* of MIPS, etc. In Table I, all control flow related statements are *conditional exit* statements, and for IR translated from IA-32 code, (`t10, 0x0`) is recorded as COP only when the followed *conditional exit* statement is executed, as the statement depends on `t31`, the operation result of (`t10, 0x0`).

The formats of *guard store* and *guard load* statements are:

`if` (<*guard*>) `ST`(<*addr*>) = <*data*>
`if` (<*guard*>) (`LD`(<*addr*>)) `else` <*alt*>

where *guard* is a conditional expression as well. *addr* is the address to be written to or read from. *data* is the data to be stored, and *alt* is the value to be load if *guard* is false. Lastly, *if then else* statement is represented as:

`ITE (`<*cond*>, <*iftrue*>, <*iffalse*>`)`

The *iftrue* expression is performed if *cond* is true, or the *iffalse* expression is performed.

C. Normalization

For the sake of accuracy and efficiency, we introduce strategies to normalize semantic signatures into a more general form to ease comparisons in the next step. On one hand, pointer values may differ from each execution because of address randomization and different architectures. Like address randomization, stack canary value is a security mechanism and such value is also unrelated to semantics. Besides, `VEX-IR` also introduces instructions of the same semantics but different representations. Thus, we need to normalize above cases to ensure accuracy of sequence comparisons. On the other hand, COPs introduced by loops may enlarge the size of signatures tremendously, to improve efficiency, we normalize COPs of a loop into a reduced form. Details of normalization strategies are as followed.

- **Pointer Abstraction**. Pointer values appearing in COP commonly point to code in .text section or data in .data section of the executables. So we normalize values among intervals of above sections into tags, and equal values share the same tag. For example, Listing 3 presents a COP sequence with pointer values, which are extracted from the IA-32 architecture, and Listing 4 shows the sequence after the processing of pointer abstraction. Stack and heap addresses are normalized as well, because system call arguments may be variable addresses on the stack or heap.

Listing 3: A COP Sequence with Pointer Values

```
0x00000000    0x08143c48
0x00000000    0x08145400
0x00000000    0x00000080
0x00000000    0x00000000
0x00000000    0x08145400
0x00000000    0x00000100
0x00000000    0x08143c48
```

Listing 4: The COP Sequence After Pointer Abstraction

```
0x00000000    PTR_000000
0x00000000    PTR_000001
0x00000000    0x00000080
0x00000000    0x00000000
0x00000000    PTR_000001
0x00000000    0x00000100
0x00000000    PTR_000000
```

- **Canary Removal**. Canary values are checked before function returning. If a function ends with a branch that one target is a normal return and the other is unsuccessful termination, we recognize the code as canary checking and the last COP of the function is removed.
- **Boundary Unification**. `VEX-IR` simplifies comparisons into four kinds of operations: `CmpEQ` (equal), `CmpNE` (not equal), `CmpLT` (less than) and `CmpLE` (less or equal). While in some cases, `CmpLT` and `CmpLE` are equivalent. For example, `CmpLT(t0, 0xA:I32)` and `CmpLE(t0, 0x9:I32)` are of the same semantics. So we unify all `CmpLE` operations to `CmpLT` by adding `1` to the second operand.
- **Loop Compression**. COP sequence of a loop is of a simple pattern. For example, Listing 5 is a sequence of loop, where one operand value is fixed (0x00000027) and the other one increased with the same step. In a signature sequence, if at least three continuous elements match the pattern, we treat them as COPs of a loop. We only record the first COP of the loop followed with the number of times for looping. So in the example, the sequence is then compressed into the form described in Listing 6.

Listing 5: A COP Sequence of A Loop

```
0x00000001    0x00000027
0x00000002    0x00000027
0x00000003    0x00000027
0x00000004    0x00000027
0x00000005    0x00000027
```

Listing 6: The COP Sequence After Loop Compression

```
0x00000001    0x00000027
0x00000005    0x00000005
```

D. Sequence Similarity Comparison

After the above two steps, we obtain a sequence of signature for each function. Then we match functions by computing the similarity of these sequences. With a template binary and a target binary, we compare each executed function of the template binary to those in the target binary, and finally gain a list of target functions sorted by the similarity scores to the template function. The most similar match is at rank 1.

There are numerous algorithms to compute similarity of sequences, such as *Levenshtein distance*, *Needleman-Wunsch algorithm*, *Longest Common Subsequence* (LCS), etc. We adopt LCS, because it requires no prior knowledge and have no limitations on the elements of sequences. In comparison, Levenshtein distance requires the definition of costs to insert, delete and modify elements, which are difficult to ensure suitable values for all cases. Needleman-Wunsch algorithm is used in bioinformatics to align protein or nucleotide sequences, which are composed of limited elements. But values of COPs and SCAs have no determined ranges. Besides, like Levenshtein distance, Needleman-Wunsch algorithm also requires to define the scores for match, mismatch and gap.

We compute their similarity of sequences A and B by Jaccard Index. The formula is as followed:

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

Listing 7: Instrumented Assembly code of IA-32 from Table I

```
1        ; memcpy(&mem_val, num_addr, 4)
2        ; printf("t10=%08x\n", mem_val)
3        ; printf("t31=CmpLE(t10,0x0)\n")
4        cmp [ebp+arg_0], 0 ; COP
5        ; printf("if(t31){PUT(68)=0x804848A}\n")
6        jle short loc_804848A
7        mov eax, [ebp+arg_0]
8        ; printf("t1=And(t10,0x1)\n")
9        and eax, 1
10       mov [ebp+var_C], eax
11       ; printf("t25=CmpEQ(t1,0x0)\n")
12       cmp [ebp+var_C], 0 ; COP
13       ; printf("if(t25){PUT(68)=0x804848A}\n")
14       jz short loc_804848A
```

Listing 8: IR trace of assembly code in Listing 7

```
1        t10=num_val
2        t31=CmpLE(t10,0x0) // COP
3        if(t31){PUT(68)=0x804848A}
4        t1=And(t10,0x1)
5        t25=CmpEQ(t1,0x0) // COP
6        if(t25){PUT(68)=0x804848A}
```

$|A \cap B|$ represents the length of their LCS. $|A|$, $|B|$ is the length of sequence A and B, separately.

III. Implementation

We implement our approach in a system named MockingBird. Currently, MockingBird supports comparisons between 32-bit Linux ELFs from three mainstream architectures: IA-32, ARM and MIPS.

A. Function Information Extraction

As our approach is at the function level, we leverage IDA Pro and IDAPython [3] to obtain the entry address of each function automatically. Then signatures are extracted according to those function ranges.

B. Instrumentation

We adopt Valgrind to implement the process of instrumentation. As we aim to gain the IR trace of one execution, the function of injected code is to output the corresponding IR of executed assembly code. For example, for IA-32 code in Table I, after instrumentation, the COP-related assembly code are presented in Listing 7, and injected code is described as comments. Then the IR trace is obtained as presented in Listing 8, where *num_val* in Line 1 is the concrete value stored in address *num_addr* in Line 1 of Listing 7.

For system calls, Valgrind provides the *–trace-syscalls* option, which enables the output of all system names and values of arguments. Listing 2 presents part of system call log for function in Listing 1 on IA-32.

C. Signature Extraction & Signature Sequence Comparison

Our signature extraction and signature sequence comparison are implemented in Python. MockingBird extract COPs by emulating the execution with the IR trace to compute the values of COPs. The workflow of Valgrind is that it firstly translates binaries into IR, then instruments on IR, and finally converts instrumented IR into binaries to execute. IR is accessible only in the step of instrumentation, but during that time, COPs are symbols not concrete values. While Valgrind allows us to access values of registers and memory read/write at runtime. For example, Line 4 in Listing 7 reads the memory at address `ebp+arg_0`, so in the injected code at Line 1, *num_addr* is replaced with that address value by Valgrind at runtime, and the value is read and recorded in the IR trace as *num_val* at Line 1 in Listing 8. Then with the IR trace, MockingBird computes other variable values and extract COP. Meanwhile, MockingBird extracts SCAs from the system call log produced by Valgrind as described in II.

Because lengths of signature sequences may exceed 10,000, In the sequence comparison step we adopt Hirschberg's Algorithm [9], of which the memory complexity is $O(min(m,n))$, to avoid running out memory when computing the LCS.

D. Optimization

In this section, we introduce the optimization strategies adopted in MockingBird. When profiling programs, the I/O operation is the performance bottleneck. Especially each binary instruction may have several corresponding IR statements. In some cases, the slow I/O may even fail the whole analysis process. For instance, when transferring a file through HTTPS with *wget*, the I/O operations during the SSL handshake phase greatly slow down communications between client and server. Then the server will discard the session because of no response from client for a period of time. Thus, we utilize *minilzo*, a lightweight subset of the *LZO* library for data compression and decompression, to reduce the overhead. Each time MockingBird allocates 1 MB of space for buffering the records. When the buffer is full, the contents are compressed and written to disk.

Additionally, sequence comparison is the most expensive process in our approach. We propose several pruning mechanisms to improve the performance. For a template sequence, each target sequence is compared to for similarity score. MockingBird records the maximum score of previous comparisons, and the value is updated if a higher score appears. Then, before comparing two sequences *A* and *B*, their possible maximum Jaccard Index is computed first. The formula is

$$\begin{aligned} & J_{psb\_max}(A,B) \\ = & \frac{min(|A|,|B|)}{max(|A|,|B|)+min(|A|,|B|)-min(|A|,|B|)} \\ = & \frac{min(|A|,|B|)}{max(|A|,|B|)} \end{aligned} \tag{2}$$

Jaccard Index gains the maximum value if one set is a subset of the other. In cases here, that is

$$|A \cap B| = |LCS(A,B)| = min(|A|,|B|)$$

If the such value is less than the current maximum score, the process of comparison will be skipped.

Another mechanism is to compute entropy of each sequence. Low entropy indicates the sequence poses little information. So we abandon such sequences for comparison. In our implementation, we only filter sequences whose entropy is 0. In practice, the threshold could be defined as any meaningful value.

IV. Evaluation

We conduct empirical evaluation with MockingBird: firstly, binaries compiled from different architectures are compared. Then, we compute the similarity of binaries with variant compiling configurations. Lastly, to test the upper capacity of MockingBird, we compare binaries of the same code base but provide them with different inputs.

A. *Experiment Setup*

We evaluate our approach on three mainstream architectures: IA-32, ARM and MIPS. The experimental environment for IA-32 is a virtual machine with 2G RAM allocated, while for ARM and MIPS we emulate them using QEMU. Because of the limitations of the architecture, memory of MIPS environment is only 256M. IR traces are obtained in above guest environments with Valgrind, then analyzed in the host system, which is running on an Intel Core i5-2320 @ 3GHz CPU with 8G DDR3-RAM.

Table II presents the objects of the evaluations. The programs are all from open-source projects. With different compilers (`gcc` v4.7.3 and `clang` v3.0) and variant optimization levels (-O3, -O2 and -O0), overall 80 binaries are compiled for the three architectures. The principle to select test input is to trigger the main or common functions of each program. Test inputs used in the evaluations are listed in Table II as well.

B. *Ground Truth*

Although the approach does not rely on debug symbols, to facilitate verifying results, we compiled all samples with the `-g` option to establish ground truth based on the symbol names. According to the symbol names, if the Rank 1 target function shares the same name with the template function, the match is correct. This process could be completed automatically.

Extra manual check is also needed because of function inlining and duplication. Function inlining resulting from code optimization removes the inlined functions from the target binary. The callee is inlined into caller to decrease function calling and improve efficiency. So signature of callee is involved in caller as well. For example, in template binary there exist a function *A* that calls function *B*, while in target binary the corresponding function B' is inlined into *A'* as *A'B'*. Thus, no matter the match is (*A*, *A'B'*) or (*B*, *A'B'*), we treat it as a correct match. Function duplication copies functions in final binaries to ensure the jump distance from caller to callee is less than 0x1000 bytes to avoid a page fault. As the duplicated functions are exactly identical, if a match is two duplicated instances of the same function, it should be treated correct.

C. *Effectiveness and Capacity*

1) Experiments Across Architecture: In this experiment, we aim to match all functions executed of the template binary *A* to those of the target binary *B*. Those two binaries are generated from the same code base, and therefore perform identical functionalities if given the same input, but are compiled on different architectures or with different compiling configurations. For each function executed in *A* (template function), we compute the similarities to functions in *B* (target function), finally gaining a list of target functions sorted by the similarity scores to the template function. The most similar match is at Rank 1. We then assess whether it is a correct match according to the Ground Truth.

We compile the 10 object programs for each architecture. Compiler is the `gcc` of each environment, and optimization option is *-O3*. Results are presented in Figure 1.

Except for the experiment of *siege* in ARM vs MIPS, the accuracy of all other comparisons are over 60%, and the average accuracy of all experiments is 77.7%. The accuracy of ARM vs MIPS is 80.3% on average, and the average accuracy of x86 compared to other architectures is 76.8%. Reasons leading to more differences from x86 to other architectures are as followed:

- *Library function inlining*. On x86, `gcc` inlines simple and common library functions, such as *printf*, *strlen*, but compilers on AMR and MIPS don't. It inserts signatures of library functions to callers, which increases the length of sequence and decreases the similarity score. MockingBird concentrates on user code, such optimization is a kind of obfuscation in some sense, which inserts junk data.
- *Float point number processing*. On x86, float point numbers are processed with FPU stack, whose push and pop operations include condition comparisons and are indistinguishable from other COPs in `VEX-IR`. While for ARM and MIPS, they both have specific floating point registers. Thus, for binaries with float point number processing, results of ARM vs MIPS are much better than those comparing to x86. Typical example is *ffmpeg*. Its accuracy rate of ARM vs MIPS is 82.5%, while for x86 vs ARM, the rate is 68.5%, and for x86 vs MIPS, the accuracy is 74.2%. In other experiments, floating point operations are small parts of the whole execution, their side-effects on MockingBird are limited.

In the three groups of experiments, *siege* gave the worst results of all on average, which is 61.2%, because of the differences of environments that include network connections, system configurations, etc. For *siege*, we limited test time of *siege* to 10 seconds, during which it constructed, sent packages and waited for responses. All above processes were instrumented by Valgrind, which gave extra overhead. On IA-32 with 2G RAM, *siege* received five responses in 10s, while for MIPS with 256M RAM, only receiving one response. Thus, performance of *siege* on different architectures influenced the results.

To our knowledge, Pewny et al. [19] propose the unique work on detecting similar code across architectures currently. Their approach is sensitive to CFG and segmentation of the basic blocks, which may be influenced by function inlining or the differences of architectures, while MockingBird

TABLE II: Programs used in the experiments

Program	Version	Description	Test Input
wget	1.15	Multi protocols supported file retriever. GNU command line project.	`wget http://ftp.gnu.org/gnu/wget/wget-1.13.tar.xz`
gzip	1.6	Data compression utility based on the DEFLATE algorithm. GNU command line project.	`gzip sample.txt`
lua	3.2.3	Command line scripting parser for lua, a lightweight scripting language.	`lua hello.lua`
puttygen	0.64	Part of PUTTYGEN suit, a tool to generate and manipulate SSH public and private key pairs.	`puttygen -P sample.in -o sample.out`
curl	7.39	Multi protocols supported data transferor with URL syntax.	`curl -O http://ftp.gnu.org/gnu/wget/wget-1.13.tar.xz`
siege	3.0.1	Http load testing and benchmarking utility.	`siege -c3 -t10S www.sample_address.com`
mutt	1.5.24	Text-based email client for Unix-like systems.	`mutt -s "hello" user@domain.com <hello_world.txt`
openssl	1.0.1p	Toolkit implementing the TLS/SSL protocols and a cryptography library.	`openssl dgst -md5 file.txt`
convert	6.9.2	Command line interface to the ImageMagick image editor/converter.	`convert sample.png -background black -alpha remove sample.jpg`
ffmpeg	2.7.2	Audio and video recorder, converter.	`ffmpeg -i sample.avi -vn -ar 44100 -ac 2 -ab 192 -f mp3 sample.mp3`

relies only on semantics signatures. Hence the accuracy of MOCKINGBIRD outperforms the approach in [19]. For example, the accuracy (Rank 1) of *openssl* comparison (ARM vs MIPS) is 32.1% and 80.0% in top 100 in [19], while the rate of Rank 1 detected by MOCKINGBIRD is 82.1%.

2) Experiments Across Compiling Configurations: Previous experiments have shown the capacity of our solution to function matching if they are compiled for different architectures. Next, we evaluate how the choice of compilers and optimization levels affects the accuracy of our solution. For different compilers, we select `gcc` v4.7.3 and `clang` v3.0. While for comparisons of variant optimization levels, we only discuss the *-O3* case versus the *-O0* case. Because high optimization levels cover all strategies of lower ones. Take `gcc` as an example, *-O0* turns off all optimizations, *-O1* enables 40 strategies, *-O2* turns on another 34 flags, while *-O3* employs 10 new optimizations on the base of -O2. Overall, -O3 enables 84 optimization strategies, and -O0 enables none. So differences of binaries compiled with above two options are the largest.

Results of different compiler comparison on IA-32 are shown in Figure 2. Notice that the average accuracy rate is 82.8%, better than that across different architectures (77.7%). Take the case of *siege* as an example, the accuracy of comparison, which is 81.6%, improved remarkably compared to that in experiments across architectures (61.2%) because of the same environment (IA-32 2G RAM),

Comparisons of binaries compiled with different optimization levels were performed on IA-32 first. The average accuracy is 72.8%. The results also indicate that optimizations are main reasons resulting in function inlining. After manual checking, the ratio of inline functions of other experiments is only 2.45%, while for -O3 vs -O0, the rate is 13.3%.

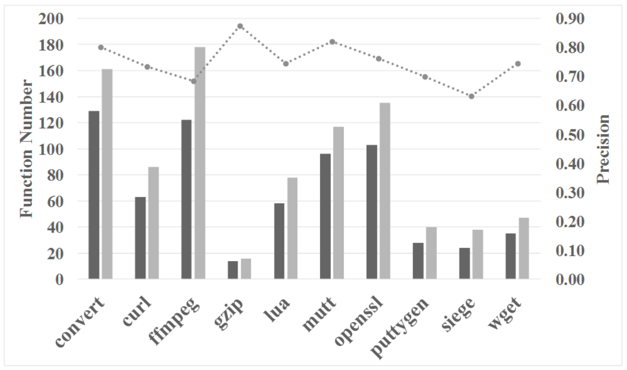
Results for comparisons of different optimization levels across architectures are described in Figure 3. Effects of different optimization levels are small against MOCKINGBIRD. For comparisons between x86 and ARM, the accuracy of `x86_O3` vs `ARM_O0` is only 2.90% lower on average than `x86_O3` vs `ARM_O3`. For x86 vs MIPS, the average difference is 7.03%.

The main reason leading to false positive in above experiments is the function inlining. Just like library function inlining, it enlarges lengths of signature sequences so as to introduce redundant data. After manual checking, we find the ratio of inlined functions that the factor of architecture introduces is 18.2%.

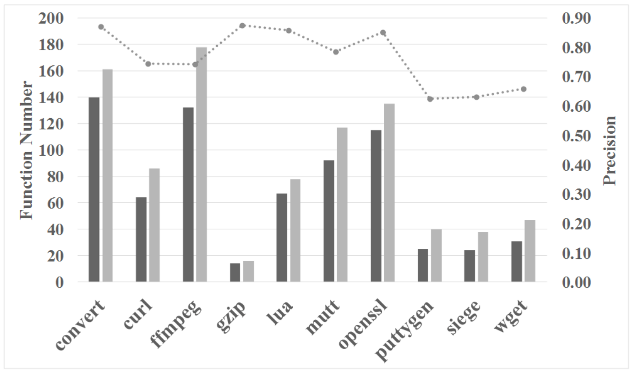
BLEX [6] and CoP [14] are able to detect similar binaries compiled with different compilers and variant optimization levels as well. BLEX mainly exploits read/write contents to search binaries, and ranked the correct match at Rank 1 in 64% of all queries. In comparison, the average accuracy of all experiments on IA-32 detected by MOCKINGBIRD is 82.8%. CoP is a tool based on symbolic execution. Although it achieved an average score of 88% when comparing x86 binaries of different optimization levels (*-O0* vs *-O2*), which is slightly higher than MOCKINGBIRD (83.1%), it cannot compare binaries across architectures.

3) Experiments with Different Inputs: The above experiments illustrate the true positive of MOCKINGBIRD. Next, we aim to measure the false positive produced. However, as long as given signature sequences of template and target binaries, MOCKINGBIRD returns lists of similarity scores. If the two binaries hold distinct semantics, it is difficult to judge whether the Rank 1 match is indeed correct or not. Besides, according to our user case which is understanding semantics of binaries by similarity detection, analysts have the prior knowledge of the target binary, including basic functionality (e.g. data compression, email client) and input format. It is meaningless to compare binaries with distinct semantics. So, in the following experiments, we merely compare binaries with similar semantics, which is to further test the capacity of MOCKINGBIRD.

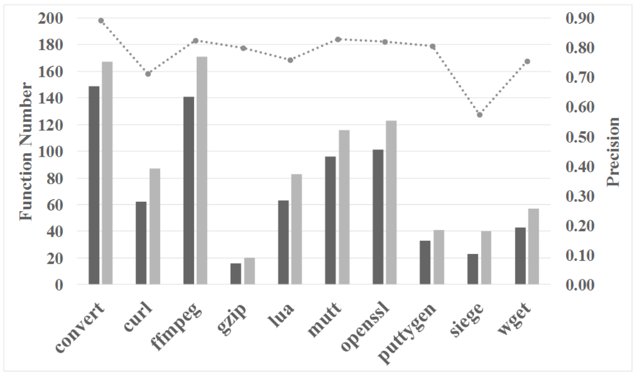
We use *wget* to retrieve the same file through different protocols (http and ftp), respectively. Differences of the two executions are functions related to protocols, such as *http_loop*/*ftp_loop*, *gethttp*/*getftp*, etc. Other functions including initialization, local file writing are all the same. Results show `wget_http` executed 60 functions overall after flitting, and 11 functions were http related. The precision compared to `wget_ftp` is 63.3% (38 / 60), and the recall is 77.6% (38 / 49). For the 11 specific functions, their average similarity score is merely 0.245, while that of 38 true positives is 0.504, and for 11 false positives, the average score is 0.155. For those true positives, the difference of similarity score between Rank



(a) x86 vs ARM



(b) x86 vs MIPS



(c) ARM vs MIPS

Fig. 1: Results of comparisons across architectures. The black bar represents the number of correct matching. The gray bar is the number of functions executed. The dot is precision of each experiment.

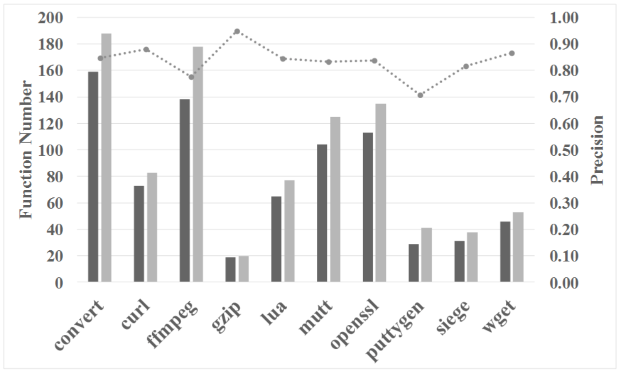


Fig. 2: Results of experiments of binaries compiled by different compilers (`gcc` vs `clang`) on IA-32

1 and Rank 2 is 0.211 on average. The difference of those 11 false positives is 0.014. The results indicates the confidence for a pair of match: *larger difference, more confidence*.

V. Discussion

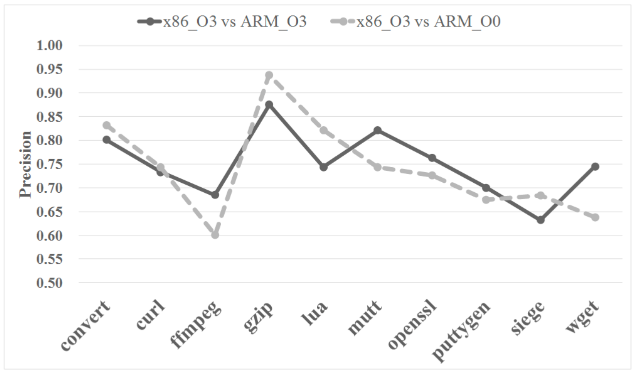
This section discusses the remaining challenges and possible future work of our system.

A. *Scope of Application*

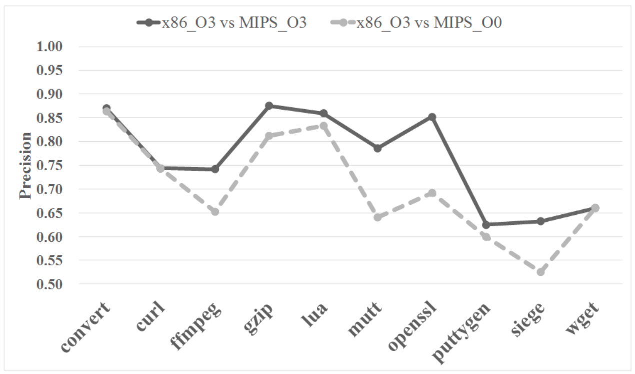
Since MockingBird relies on determined runtime values of an execution, it is difficult to handle cases with randomized algorithms. For parallelized programs, the execution of each thread can be monitored to extract semantic signatures and compared for similarity scores. That is left as future work. Currently, MockingBird is able to compare the same library functions of different programs, but the input for those functions should be identical. In the future, we plan to analyze binaries with symbolized input. Then signatures could be extracted from arbitrary template and target functions.

B. *Robust of Semantic Signatures*

COPs and SCAs both have their own limitations. COPs cannot handle cases short of conditional operations (e.g., functions only with sequential structures). Meanwhile, they are vulnerable to control flow obfuscation, such as flattening CFG and opaque predicate, as such techniques insert massive



(a) x86 -O3 vs ARM -O0



(b) x86 -O3 vs MIPS -O0

Fig. 3: Results of experiments of different optimization levels across architectures

redundant comparison operations, making large differences in the two signature sequences and result in false positives. While [24], [7] introduce corresponding approaches of deobfuscation. it is better to deobfuscate target binaries first before conducting our test since MOCKINGBIRD aims to infer semantic information of binaries, not to employ deobfuscation. SCAs also have troubles in tackling cases with insufficient system calls, such as functions only with arithmetic operations. Thus, we adopt the hybrid semantic signatures to make the signature more robust.

C. Dynamic Analysis Issues

Dynamic analysis introduces more runtime information and are more robust against minor code variance. However, it suffers from issues of input construction and code coverage. For our test cases, analysts have the prior knowledge of binaries under test, including basic functionalities and input formats. So it is not a problem to construct legal inputs. Besides, we focus on locating functions with specific semantics, and therefore code coverage is not the primary problem. While in future work, techniques of concolic execution [15] and fuzzing testing [2], [20] could be introduced for specific input construction and higher code coverage.

D. Dynamic Instrumentation Framework

Valgrind is the most well-developed cross-architecture dynamic binary instrumentation framework currently. However, comparing to that for IA-32, its support for ARM and MIPS is insufficient. On one hand, the ARM instruction set is incomplete. For instance, `SETEND` sets the endianness bit in the CPSR on ARM. But this instruction is not supported by Valgrind. Since the instruction is usually adopted in some implementations of library functions (e.g. `memcmp`). Analysts could supply their own libraries without such instructions to Valgrind if they just aim to analyze user code. On the other hand, Valgrind is unstable for MIPS in outputting logs. It occasionally outputs incomplete IR traces. As a result, we have to run the analysis process repeatedly for complete results.

E. Performance and Scalability

The instrumented code merely prints hard-coding IR statements. The overhead is low. Emulating IR traces and transmitting data dependences are both processed sequentially with time complexity of $O(n)$. The most expensive process of our approach is sequence comparison, whose time complexity is $O(mn)$ and could not be reduced currently. To improve performance, in future, we plan to adopt MinHash, which aims to estimate Jaccard Indexes quickly. On that occasion, it is, therefore, not necessary to compute the intersection (LCS) and union of two sequences.

VI. RELATED WORK

A. Code Similarity Detection

Similar code (also known as cloned code) reflects the existence of code copying and pasting during software development process, minor modification or forking from the same code base. Identifying similar or duplicated code is a common requirement for software maintenance. Main applications of similar code identification include software plagiarism detection and code clone detection.

Roy et al. [21] defines similarity of syntax and semantics on source code. Syntax similarity is based on program text (e.g. code layout, comments), while semantic similarity is based on algorithm's functionality. To tackle the problem of syntactic similarity detection, many techniques have been proposed over the last decade. Among them, some tools such as DECKARD [11], CCFinder [12], and CloneDR [1] are quite well developed for locating source code level syntactic similarity detection. CCFinder does lexical analysis through source code tokens. CloneDR detects similar code by comparing similarity of Abstract Syntax Tree (AST). DECKARD also leverages AST, but it depends on feature vectors extracted from AST. Bertillonage [5] introduces methods detecting clone code in JAVA with names of classes, methods and interfaces, etc. McMillan et al. [16] leverage JDK API calls invoked in JAVA code to detect similar applications.

Much more complex cases exist in binary code similarity comparison because most syntactic information has lost for binaries. Sæbjørnsen et al. [22] describe the first practical clone detection algorithm for binary executables. Their work extends an existing tree similarity framework to normalize assembly instructions and models structural information. Hemel et al. [8] employ data compression rate as a measure of the similarity. In [13], Khoo et al. combine *n*-grams with graphlets for structural matching. In [4], David et al. measure code similarity with edit distances between two functions.

Above approaches are able to detect similar binary code on syntax level, but they are generally infeasible to semantic detection. As compiler may produce variant binaries with different compiling options even with the same source code, detecting similar binaries is usually the problem of semantic similarity detection, which results in the popularity of semantic features. Jhi et al. [10] proposes to use core values as birthmark to detect software plagiarism. Zhang et al. [29] further developed the approach and discussed the problem of algorithm plagiarism. They point out that there exist critical runtime values, named core values, which are irreplaceable for all implementations of the same algorithm. However, to detect similar binaries compiled with different optimization options, it requires source code to extract core values. In [6], Egele et al. propose Blanket Execution (BLEX) for full code coverage to compare binary code. They leverage memory access and function calling as features, which are core values in essence. Unlike [10], [29], BLEX even have no process to refine those features. Luo et al. [14] and Zhang et al. [30] exploit symbolic execution to compare binary code similarity. They treat basic blocks (BB) as black boxes and represent output of BBs as functions of input. BBs are identical if their functions are equivalent, and similarity of functions are measured by the rate of clone BBs. However, there is no symbolic execution platform that can deal with multi-architecture binaries. So approaches based on symbolic execution are not able to detect similar binaries across architectures. In [19], Pewny et al. firstly proposed the approach to detect known bugs in binaries for different architectures via code similarity comparison. They use `VEX-IR` to unify representations of binaries, and extract semantic signatures by sampling with random input. With those signatures, similar BBs could be first detected then combined with CFGs, and the similarity of binaries is computed. As their approach requires information of CFG, it is sensitive to CFG variances. Our experiments show that both compilation's optimization options and architectures variations change CFGs. Thus their approach is not adaptive.

Code similarity detection is important for Android app analysis as well (e.g., app repackaging detection). In [28], the feature view graph of Android App is extracted as the birthmark. Wang et al. [25] detect Android App clone by counting number of times each variable used and defined.

B. *Sequence Alignment and String Metrics*

Apart from Needeman-Wunsh [17], Smith-Waterman [23] is another algorithm to find subsequences locally in bioinformatics. It compares subsequences of all possible lengths to optimize the similarity measure. So it is suitable for cases that large parts are missing in sequences. Sequence alignment techniques are also widely used in similar code comparison. For core values [10], [29], LCS is adopted to align the core value sequences. Luo [14] treats CFG as sequences of basic blocks, and align the basic blocks to compute the similarity of CFG.

String metrics are used to measure the similarity between text strings and are used heavily on fingerprint analysis, plagiarism detection, etc. They could be combined to any algorithm to compare similarity of sequences. Besides Jaccard Index and Levenshtein Distance described above, Hamming Distance and Sørensen-Dice coefficient are popular as well. Hamming Distance requires the same length of two sequences and conducts exclusive-or (xor) operation on them directly. Sørensen-Dice coefficient computes the similarity of two samples as Jaccard Index does, while it does not satisfy the triangle inequality, and is considered as the symmetric version of Jaccard Index.

VII. Conclusion

Understanding binaries via reverse engineering is a tedious task. In this work, we address the problem by automatic similar semantics comparison. We propose a semantic signature, which is composed of COP and SCA, to describe the binaries. The approach is implemented in a system called MockingBird. Our evaluations show although representations of binaries are quite different because of variant architectures and compiling configurations, MockingBird is more accurate than approaches based on similar semantics comparison. With the rise of programs cross-compiled for different architectures (e.g., Android on ARM, routers of MIPS), our approach can greatly assist understanding semantics of binaries for multiple architectures.

VIII. Acknowledgements

We would like to thank the reviewers for their insightful comments which greatly helped to improve the manuscript.

This work was supported in part by the National Key Technology Research and Development Program of China under Grants No.2012BAH46B02, the Major Program of Shanghai Science and Technology Commission under Grants No.15511103002, and the Research Funding Program of the Ministry of Education of PRC and China Mobile under Grants No. MCM20150301.

References

[1] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the 6th International Conference on Software Maintenance (ICSM)*, 1998.

[2] S. K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (SP)*, 2015.

[3] R. Data. Ida pro disassembler. https://www.datarescue.com/idabase/.

[4] 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 (PLDI)*, 2014.

[5] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle. Software bertillonage. *Empirical Software Engineering*, 18(6):1195–1237, 2013.

[6] 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.

[7] F. Gabriel. Deobfuscation: recovering an ollvm-protected program.

[8] 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.

[9] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[10] 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.

[11] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, 2007.

[12] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[13] 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 (MSR)*, 2013.

[14] 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.

[15] R. Majumdar and K. Sen. Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, 2007.

[16] C. McMillan, M. Grechanik, and D. Poshyvanyk. Detecting similar software applications. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012.

[17] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[18] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[19] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (SP)*, 2015.

[20] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley. Optimizing seed selection for fuzzing. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, 2014.

[21] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[22] 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.

[23] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[24] 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.

[25] H. Wang, Y. Guo, Z. Ma, and X. Chen. Wukong: a scalable and accurate two-phase approach to android app clone detection. In *Proceedings of the 24th International Symposium on Software Testing and Analysis (ISSTA)*, 2015.

[26] 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)*, 2009.

[27] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Detecting software theft via system call based birthmarks. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)*, 2009.

[28] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. View-droid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 7th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2014.

[29] 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.

[30] 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.