Binary Code Clone Detection across Architectures and Compiling Configurations

Yikun Hu, Yuanyuan Zhang, Juanru Li, Dawu Gu
Shanghai Jiao Tong University, Shanghai, China
{yixiaoxian, yyjess, jarod, dwgu}@sjtu.edu.cn

***Abstract*—Binary code clone detection (or similarity comparison) is a fundamental technique for many important applications, such as plagiarism detection, malware analysis, software vulnerability assessment and program comprehension. With the prevailing of smart and IoT (Internet of Things) devices, more and more programs are ported from traditional desktop platforms (e.g., IA-32) to ARM and MIPS architectures. It becomes imperative to detect cloned binary code across architectures. However, because of incomparable instruction sets of different architectures as well as alternative compiling configurations, it is difficult to conduct a binary code clone detection with traditional syntax- or structure-based methods.**

To address, we propose a semantics-based approach to fulfill the target. We recognize arguments and indirect jump targets of each binary function, and emulate executions of those functions, extracting semantic signatures to measure the similarity of functions. The approach has been implemented in a prototype system named CACOMPARE to detect cloned binary functions across architectures and compiling configurations. It supports comparisons between mainstream architectures (IA-32, ARM and MIPS) and is able to analyse binaries on the Linux platform. The experimental results show that CACOMPARE not only is effective in dealing with binaries of different architectures and variant compiling configurations, but also improves the accuracy of binary code clone detection comparing to state-of-the-art solutions.

***Keywords*—binary program analysis; code clone detection; reverse engineering; static analysis;**

I. INTRODUCTION

Binary code clone detection (or similarity comparison) is an important technique which has a variety of applications in software engineering and security, for example, software or algorithm plagiarism detection [33], [34], malware families classification [1], [22], known vulnerabilities searching [13], [25] and binary program comprehension [15]. With the widespread of smart and IoT(Internet of Things) devices, programs on traditional desktop platforms (e.g., IA-32) are progressively ported to emerging architectures, such as ARM, MIPS. Therefor, it becomes necessary to propose solutions to binary code clone detection across architectures.

However, even for binaries compiled from identical code base, their representations could be significantly different. There are two main reasons, which are also the challenges of binary cloned code detection, leading to the problem. The first one is the diversity of *instruction set architectures* (ISA). Binaries of different ISAs differ in instruction sets, code offsets and calling conventions. Thus, those binaries are generally incomparable even though they are compiled from the same code base. The other one is the *structure gaps* introduced by different *compiling configurations*, including alternative compilers and compiling options (e.g., optimization levels). Basing on variant compiling configurations, different strategies are employed to optimize and generate the resulting binary code, which bring considerable changes to the structures (e.g., function inlining). Therefore, it is difficult or even infeasible for traditional syntax- or structure-based methods to detect binary clone code across architectures and compiling configurations.

In the literature, it has drawn much attention to detect cloned (or similar) binary code across architectures and compiling configurations. MULTI-MH [25] leverages input/output values of basic blocks to detect similar code. DISCOVRE [12] performs graph isomorphism algorithm on binary code CFG (Control Flow Graph) after filtering candidate functions with their syntax features. GENIUS [13] exploits reduced CFG to search for similar binary code via isomorphic graph comparison. BINGO [6] compares the similarity of binary functions with code signatures extracted via *n-grams* basing on CFG. However, above solutions all heavily depend on CFG which could be altered significantly because of different ISAs or variant compiling configurations. Besides, DISCOVRE and BINGO introduce filtering strategies to improve efficiency, which could cause false positives to the contrary. In some cases, the performance of DISCOVRE is worse than the version without pre-filtering [13]. Apart from above static approaches, MOCKINGBIRD [15] provides a dynamic solution to cross-architecture similar code comparison. Nonetheless, it requires legal inputs to trigger target functions for comparison, which is difficult to satisfy in real world situations.

In this paper, we propose CACOMPARE, a semantics-based system statically detecting binary cloned code across architectures and compiling configurations, performing on binary functions compiled from the same code base. Given a list of template functions and a target binary program , CACOMPARE aims to find the most similar function in the target program comparing to each template function. For a binary function (a template or target function), CACOMPARE firstly recognizes arguments it consumes and possible indirect jump targets of its switch statements. Then it unifies representations of binaries from different architectures with intermediate representations (IR) and emulates executions of those binaries to extract semantic signatures. Lastly, CACOMPARE computes similarity scores of each template function to every target

function, returning a list of target functions ranked by the scores. CACOMPARE solves the problem of different ISAs by adopting IR so that it is able to perform generic analysis on binaries across architectures. Additionally, CACOMPARE extracts semantic signatures of the whole functions by emulating their executions, not relying on CFG. Thus, it overcomes the second challenge of structure gaps introduced by compiling configurations. To show its effectiveness and capacity, we evaluate CACOMPARE on 9 different utilities, containing more than 10,000 functions. The experimental results indicate that CACOMPARE not only is effective to detect binary cloned code across architectures and compiling configurations, but also improves the accuracy of detection comparing to state-of-the-art solutions.

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

- We propose a semantics-based approach to detect binary cloned code across architectures and compiling configurations. We leverage IR to unify representations of binaries from various architectures and extract semantic signatures from the whole functions by emulating their executions, so that the approach is suitable for cross-architecture analysis and robust to representation and structure gaps of binaries.
- We introduce function argument recognition and switch indirect jump target detection to assist emulating executions of functions. Besides, we propose fine-grained semantic signatures for function-level binary code clone detection. Normalization strategies are employed as well to make signatures more general and adaptive for similarity comparison.
- We implement the approach in the prototype system CACOMPARE, which supports cross-architecture code clone detection of ELF (Executable and Linkable Format) files for three mainstream architectures (IA-32, ARM and MIPS). The experimental results show that CACOMPARE not only is effective for binary code clone detection across architectures and compiling configurations, but also improves the accuracy of detection comparing to state-of-the-art solutions in scenarios of this paper.

II. OVERVIEW AND BACKGROUND

In this section, we present the overview of CACOMPARE and introduce the necessary background knowledge for it.

A. System Overview

Given a template function and a target binary program, CACOMPARE returns a function list of the target program, ranked by the similarity scores comparing to the template function. Figure 1 displays the architecture of CACOMPARE. For each binary function, CACOMPARE firstly pre-processes the binary code, including disassembling the binary code, generating CFG (Control Flow Graph), collecting information of basic blocks and edges in the CFG, etc (Pre-processing). Then CACOMPARE traverses the CFG to recognize the arguments needed for the execution (Argument Recognition) and detects the switch statements of the function, collecting all possible

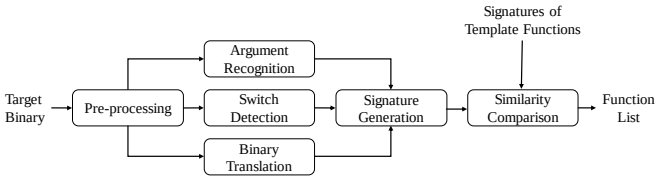


Fig. 1: CACOMPARE System Overview

destination addresses of the indirect jumps (Switch Detection). Meanwhile, it converts binaries into a uniform format with intermediate representations (Binary Translation). Next, with the arguments and switches information, CACOMPARE emulates executions of the function on the unified representation with random values as inputs to generate semantic signatures (Signature Generation). Finally it compares the signature similarity of each target function to a template function which has passed the above processes as well, and returns a function list ranked basing on their similarity scores (Similarity Comparison).

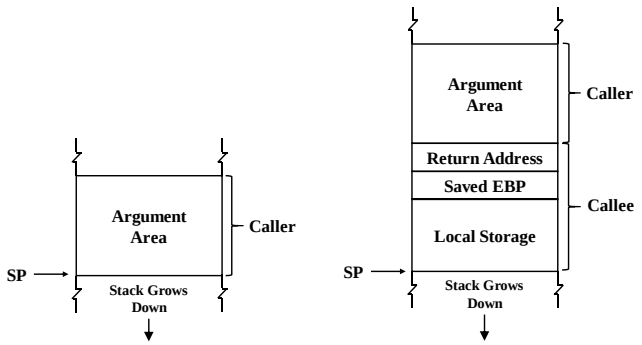
Generally, incomparable instruction sets of different architectures and structure gaps introduced by variant compiling configurations are two challenges of cross-architecture binary cloned code detection. To solve the problem of different instruction set architectures, CACOMPARE leverages unified intermediate representations to facilitate cross-architecture analysis. To overcome the challenges of structural gaps, it extracts semantic signatures from execution emulation of the whole function, which avoids relying on syntax or structure information of binaries.

B. Calling Convention

Calling convention is the scheme for how subroutines receive arguments from their caller and how they return a result. It is the basis of Argument Recognition (§III-A).

Figure 2 shows the stack layout of *cdecl*, the default calling convention for C programs on IA-32, before and after a function calling. In Figure 2a, the argument area stores the arguments required by the callee. The caller fulfills the area before entering the subroutine, and the stack pointer (SP) points at the bottom address of the argument area. After calling (Figure 2b), the return address is saved on the stack. The callee is responsible for preserving values of registers and allocating memory for local variables. Therefore, variables whose addresses are higher than the SP value in Figure 2a are the arguments of the callee.

Unlike IA-32, which lacks architecture registers, ARM and MIPS provide core registers (R0-R3 on ARM and $a0-$a3 on MIPS) for passing arguments to subroutines. If callees take a small number of arguments, only registers are used to reduce the overhead of function calling. If the parameter registers have been used up, the left arguments are passed on the stack, which is similar to *cdecl*. However, it does not mean if a function has arguments on the stack, the parameter registers are all employed as argument holders because of 64-bit values. For ARM, only registers of even numbers along with the following registers (R0, R1 and R2, R3) can hold 64-bit values. For example,



(a) Before Calling (b) After Calling

Fig. 2: Stack Layout of *cdecl* Before and After Calling

```
static uerr_t
getftp (struct url *u, wgint passed_expected_bytes,
wgint *qtyread, wgint restval, ccon *con,
int count, FILE *warc_tmp)
```

above is the prototype of function `getftp` from `wget 1.15` at ftp.c:245. The type `wgint` has a length of 64 bits. So the second argument *`passed_expected_bytes`* is not passed with R1 and R2, but with R2 and R3, and R1 is left unused. The convention of MIPS requires the first four 32 bits of the parameter area on the stack are reserved for $a0-$a3 and the size of a stack frame should be aligned to 8. So parameter registers may be unused as well to ensure the alignment. Therefor, arguments of a function can only be recognized through the consumption of corresponding variables.

III. Approach

In this section, we discuss each step of CACompare in details to explain how it does the preparation and compares the similarity of function semantic signatures to detect cloned binary functions.

A. Function Argument Recognition

In this step, CACompare recognizes arguments consumed by a function, returning the number of parameters and their positions, including relative offsets if they are passed on the stack or register numbers if passed by registers. The results provide input information for signature generation (§III-C).

Algorithm 1 presents the algorithm for function arguments recognition basing on calling conventions discussed in §II-B. For each instruction along a path of a function CFG, if it accesses a stack variable whose address is larger than the initial stack pointer value (SP value in Figure 2a), the variable is an argument, and its offset relative to $\mathcal{S}$ is recorded (Line 7-10). Because all the offsets in $\mathcal{A}^{\mathcal{S}}$ are relative values, the initial SP value $\mathcal{S}$ could be arbitrary. If the instruction accesses a parameter register which is not defined beforehand, the register is an argument as well (Line 11-13). The local set *LocReg* records the registers which have been written along the path (Line 20-21), and those registers are impossible to carry the arguments. For completeness, ECX and EDX of IA-32 are also treated as parameter registers, because calling conventions

Algorithm 1: Arguments Recognition Algorithm

Input: Initial SP Value $\mathcal{S}$
Input: Parameter Register Set $\mathcal{R}$
Input: Target Function CFG $\mathcal{C}$
Output: Stack Argument Offset Set $\mathcal{A}^{\mathcal{S}}$
Output: Register Argument Set $\mathcal{A}^{\mathcal{R}}$
1 **Algorithm** *getArgument* ($\mathcal{S}$, $\mathcal{R}$, $\mathcal{C}$)
2 $S_t \leftarrow \mathcal{S}$
3 $LocReg \leftarrow \varnothing$
4 $\mathcal{A}^{\mathcal{S}} \leftarrow \varnothing$, $\mathcal{A}^{\mathcal{R}} \leftarrow \varnothing$
5 **foreach** *Path* P *in* $\mathcal{C}$ **do**
6 **foreach** *Instruction* I *along* P **do**
7 **if** I *reads a value from stack* **then**
8 $addr \leftarrow$ **`getReadAddress`**(I)
9 **if** $addr > \mathcal{S}$ **then**
10 $\mathcal{A}^{\mathcal{S}} \leftarrow \mathcal{A}^{\mathcal{S}} \cup \{addr - \mathcal{S}\}$
11 **else if** I *reads a register value from* R **then**
12 **if** $R \in \mathcal{R}$ *AND* $R \notin LocReg$ **then**
13 $\mathcal{A}^{\mathcal{R}} \leftarrow \mathcal{A}^{\mathcal{R}} \cup R$
14 **if** I *modifies SP* **then**
15 $S_t \leftarrow$ **`modifySP`**(S_t)
16 **else if** I *calls a function* f **then**
17 $C_t \leftarrow$ **`getCFG`**(f)
18 $A_t^S, A_t^R \leftarrow$ **`getArgument`**$(S_t, \mathcal{R}, C_t)$
19 $\mathcal{A}^{\mathcal{R}} \leftarrow \mathcal{A}^{\mathcal{R}} \cup A_t^R \backslash LocReg$
20 **if** I *writes a register* R **then**
21 $LocReg \leftarrow LocReg \cup R$
22 **return** $\mathcal{A}^{\mathcal{S}}$, $\mathcal{A}^{\mathcal{R}}$

fastcall and *thiscall* leverage them to pass integral parameters. During the traversal of the CFG, operations modifying the stack pointer value would be taken into consideration and the SP value is updated correspondingly (Line 14-15), because the stack could be accessed by the frame pointer as well as the stack pointer. If the instruction calls a function, register parameters which are unchanged in the caller but used in the callee are arguments of the caller as well (Line 16-19).

It should be mentioned that the target of Algorithm 1 is **not** to recover the prototype of a function. Because the number and type of an argument in the resulting binary does not follow the definition in the source code strictly. In the example of `getftp` (§II-B), although the function only has 6 arguments in the prototype, the resulting number it consumes is 9 (1+2+1+2+1+1+1), because 64-bit parameters are broken up into two 32-bit variables for accessing on 32-bit architectures.

B. Switch Indirect Branch Targets Detection

A switch is a selection control mechanism allowing the value of a variable or expression to change the control flow of program execution via a multiway branch. In this step, CACompare detects the switch statements and their corresponding destination addresses of binary functions to facilitate the emulation for signature generation.

Compilers typically generate a jump table (or branch table), containing a serial list of target addresses for the switch and an indirect jump leveraging runtime values as offsets to index

```
1  sll     $v0, 2         # compute the relative offset
2  la      $a0, 0x446A10  # load address (jump table start)
3  addu    $v0, $a0, $v0  # address of jump table entry
4  lw      $v0, 0($v0)    # get the target address
5  jr      $v0            # indirect jump to the target
6  or      $at, $zero
```

Fig. 3: Indirect Jump for Switch from `wget` on MIPS

the jump table. In the literature, accurate targets recognition of an indirect jump still remains an important issue for binary analysis. But with the jump table, it is possible to collect targets of an indirect jump for a switch statement.

To detect indirect jump targets of a switch, CACOMPARE needs identify jump tables of the binary program firstly, then matches each jump table to the corresponding indirect jump in a function. The process of identifying jump tables is based on the technique proposed by Wang et. al [31]. For 32-bit architecture, it traverses data sections of the binary program, and considers the address of a 32-bit value as the first jump table entry (jump table start) if it is referred to by an instruction as the operand. Then if a 32-bit value follows a jump table entry and refers to instructions within the same function, it is treated as a jump table entry as well.

With jump tables identified, CACOMPARE then finds their corresponding indirect jumps. For a switch, the jump target is obtained by the offset relative to the jump table start. So if an indirect jump serves a switch, an access to the jump table start would appear beforehand. CACOMPARE traverses each path of a function CFG. If an instruction refers to a jump table start, then the following indirect jump is the corresponding one. For example, Figure 3 displays the process of computation for a switch target from `wget` on MIPS. It accessed the jump table start address (0x446A10) at Line 2. With the relative offset stored in $v0, the jump table entry address which holds the target address is computed at Line 3. Finally, the control flow transfers to the target address with an indirect jump in Line 5. Thus, the corresponding indirect jump of jump table starting at 0x446a10 is `jr $v0` at Line 5.

C. Semantic Signature Generation

Similar code must have semantically similar execution behavior, whereas different code must behave differently [10]. Therefore, for functions compiled from the same code base, given the same input, they must execute the same path and generate the same output, even though they are different in instruction sets or compiled with variant compiling configurations. In this step, CACOMPARE provides functions with the same random input, then emulates their executions and records semantic signatures during the emulation for future comparison.

1) Semantic Signature: Semantic signatures recorded by CACOMPARE are listed as followed:

- **Input and Output Values.** Apart from randomly generated argument values, inputs are consist of data read from data sections (e.g., .data, .rodata). Output values include the return value of the function and memory write values whose written addresses are beyond the range of local stack frame. Inputs and outputs are the most straightforward features of function behaviors which indicate semantics of executions. For example, Figure 4 presents a function printing positive numbers and its corresponding IA-32 assembly code. Argument `num` is an input value. If `num` is positive, offset of the format string `%d\n` (Line 3 and 14) in the .rodata section is an input value as well. Register `eax` holds the return value (Line 16 and 19), so it is an output value.
  Variables on the local stack frame will be discard when the function returns. So CACOMPARE would not treat those local variables as signatures.
- **Comparison Operands and Condition Codes.** Comparison operands are values which introduce condition tests in an execution to decide branch targets of the following conditional jumps. Condition codes are the conditions of those comparisons test for. Comparison operands reveals what values for comparing, and the following condition code informs how to compare. In Figure 4, operand values of the comparison instruction at Line 10 are the comparison operands (value of `arg_0` and `0`) which corresponds to `num` and `0` at Line 2. The condition code `LE` (less or equal) is obtained from the conditional jump at Line 11.
  Comparison operations convert control dependencies into data dependencies [30]. When binary functions of the same code base are provided with the same input, their execution path must be the same as well. Comparison

```
1  bool print_positive (int num){
2      if (num > 0){
3          printf("%d\n", num);
4          return true;
5      }
6      return false;
7  }
```

```
8       arg_0   = dword ptr  8
9        sub    esp, 28h
10       cmp    [esp+28h+arg_0], 0 ; comparison operands
11       jle    short loc_8048482  ; condition code: le
12       mov    eax, [esp+28h+arg_0]
13       mov    [esp+4], eax
14       mov    dword ptr [esp], offset format ; "%d\n"
15       call   _printf
16       mov    eax, 1
17       jmp    loc_8048487
18      loc_8048482:
19       mov    eax, 0
20      loc_8048487:
21       add    esp, 28h
22       retn
```

Fig. 4: Function Printing Positive Numbers and Corresponding IA-32 Assembly Code

```
1  IO   0000beef   // input value read at Line 10
2  CC   0000beef   00000000   LE // comparison operands
3                                 // and condition code
4                                 // at Line 10, 11
5  IO   0000beef   // input value read at Line 12
6  IO   25640a00   // format string "%d\n" at Line 14
7  LC     printf   // library function call at Line 15
8  IO   00000001   // return value at Line 16
```

Fig. 5: Signature Sequence of *print_positive* (*num=0xbeef*)

Algorithm 2: Argument Values Distribution for a Function

Input: Random Value Stream List $\mathcal{L}$
Input: Initial Stack Pointer Value $\mathcal{S}$
Input: Sorted Register Number List of Register Arguments $\mathcal{L}^{\mathcal{R}}$
Input: Sorted Offsets of Stack Arguments $\mathcal{L}^{\mathcal{S}}$
Output: Initial Program State before Emulation $\mathcal{P}$

1 **Algorithm** *argsDistribution ($\mathcal{L}$, $\mathcal{S}$, $\mathcal{L}^{\mathcal{R}}$, $\mathcal{L}^{\mathcal{S}}$)*
2 $ArgCnt \leftarrow 0$
3 **foreach** R *in* $\mathcal{L}^{\mathcal{R}}$ **do**
4 $\mathcal{P}[R] \leftarrow \mathcal{L}[ArgCnt]$
5 $ArgCnt \leftarrow ArgCnt + 1$
6 **foreach** O *in* $\mathcal{L}^{\mathcal{S}}$ **do**
7 $\mathcal{P}[\mathcal{S} + O] \leftarrow \mathcal{L}[ArgCnt]$
8 $ArgCnt \leftarrow ArgCnt + 1$
9 **return** $\mathcal{P}$

operands and condition codes are the checkpoints along the path, thus such signatures also reflect the semantics of one execution.

- **Library Function Calls.** Library functions records have been proven to be an effective semantic signature for code similarity comparison if they are sufficient in number [10], [32]. So CACOMPARE employs library function calls as a supplement to above signatures. For the example in Figure 4, library function `_printf` would be recorded if `num` is positive.

Figure 5 shows the signature sequence of the example function in Figure 4 when the input value *num=0xbeef*. The first column indicates types of the signature entries in a row (IO: I/O values, CC: comparison operands and condition codes, LC: library calls).

2) Execution Emulation: The key points of emulation are: i) Argument Values Distribution that all functions' parameters are provided with the same values, and ii) Control Flow Arrangement that introduces strategies for indirect jumps and function calling.

- **Argument Values Distribution:** Before emulation, CACOMPARE generates a stream of random values. According to arguments information collected in §III-A, CACOMPARE assigns values of the stream to parameters of each function sequentially. The process is presented in Algorithm 2. If a function has register parameters, CACOMPARE would give values to those registers first (Line 3-5), then initializes stack argument values subsequently (Line 6-8). The algorithm is applicable to *fastcall* and *thiscall* of IA-32 as well.
- **Control Flow Arrangement:** CACOMPARE has obtained knowledge of indirect jump targets of switch statements in §III-B. When it emulates switch statements, CACOMPARE always transfers the control flow to the target of a pre-defined offset (e.g., always branches to the first case), because jump targets are generated by input values, the possibility is low that random inputs can produce a meaningful offsets for a switch jump table. Besides, compilers generate switch jump table entries according to the sorted sequence of cases. Thus, it ensures the same execution path by selecting the target of a fixed offset. For other indirect jumps and calls with unknown targets, CACOMPARE just steps over and continues the emulation.

  For library function calls, CACOMPARE ignores their emulation as well. If a library function has a return value, CACOMPARE assigns a pre-defined constant to the return value register according to calling conventions. Additionally, because the input values are meaningless, loops and recursions may become infinite. A threshold is set to limit the times for executions of loops and recursions.

To enrich the semantic information, if a function is short in the length of signature sequence, CACOMPARE would emulate the function for several more times with other random values in the input stream until its signature length reaches a pre-defined threshold.

3) Signature Normalization: After the emulation, CACOMPARE generates a signature sequence for each function. It then normalizes the signatures considering the accuracy of comparison. There are two strategies for normalization:

- **Pointer Value Normalization**. During the emulation, addresses in ranges of data and code sections may be accessed then appear in signatures. However, addresses of above sections of variant binaries are different. Thus, values in ranges of those sections are considered as pointers and normalized into a pointer tag.
- **Comparison Boundary Unification**. For integer comparison, $a \leq b$ is equivalent to $a < b + 1$. So CACOMPARE normalizes comparison operands basing on their condition codes that the less (greater) or equal cases are all unified to less (greater) than.

D. Signature Comparison

CACOMPARE computes the similarity of two signature sequences with the Jaccard Index. The formula is as followed:

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

$|A|$, $|B|$ is the length of sequence A and B. $|A \cap B|$ represents the length of their Longest Common Subsequence (LCS). The time complexity of LCS algorithm is $O(mn)$ and could not be reduced currently. To improve the performance, when comparing long sequences whose lengths are larger than a pre-defined threshold, CACOMPARE adopts MinHash [5] to quickly estimate the Jaccard Index without explicitly computing the intersection and union sets. MinHash makes a trade-off between efficiency and accuracy. For CACOMPARE, we select 400 hash functions to approximate the Jaccard Index. The expected error is less than 0.05, which is accurate enough in this scenario.

IV. IMPLEMENTATION

Currently, CACOMPARE supports comparisons between binary functions from ELF (Executable and Linkable Format) files of three mainstream architectures: IA-32, ARM and

MIPS. Next, we introduce several aspects of the implementation in more details.

A. Function Information Extraction

We leverage `IDA Pro` v6.6 [7] to disassemble binary code and extract control flow graphs. Then argument recognition (§III-A) and switch indirect jump targets detection (§III-B) are completed automatically with `IDAPython`. Although the resulting disassembly of `IDA Pro` is not perfect [2], it is sufficient in our scenario.

B. Binary Translation

Representations of binaries from different architectures are unified with `VEX-IR`, which is a RISC-like intermediate representation (IR) defined by `Valgrind` [24], an open-source dynamic analysis framework. We employ `PyVEX` [28], a Python project providing bindings to VEX-IR, to translate binaries with its API statically. The resulting VEX statements are stored in our own data structures, which are used for signature generation (§III-C).

C. Signature Generation

The random values for emulation input are generated from the range $[-1000, 1000]$, which is found to be sufficiently large to avoid collisions and small enough to cover many possible paths [25]. During emulation, if a function requires values from undefined addresses, we feed the function with a pre-defined constant (e.g., 0) to force the continuance of the process.

D. Similarity Score Computation

Considering the trade-off between accuracy and efficiency, a threshold is defined for adopting MinHash to compute the Jaccard Index. If lengths of both signature sequences for comparison is smaller than that value, then CACOMPARE uses the LCS algorithm.

As sequence comparison is the most costly process of CACOMPARE, we propose a pruning strategy to improve the performance. Each time, before two sequences are compared, their possible maximum Jaccard Index is computed. If the value is less than the maximum score of previous comparisons, the process of these two sequences is skipped. The equation of possible maximum score is shown as followed:

$$J_{max}(A, B) = \frac{min(|A|, |B|)}{max(|A|, |B|)} \quad (2)$$

The Jaccard Index obtains the maximum value if one element is a subset (subsequence) of the other one.

V. EVALUATION

We conduct empirical experiments to evaluate the effectiveness and capacity of CACOMPARE. Firstly, binaries compiled from different architectures are compared. Then, we compute the similarity scores of binaries compiled with variant compiling configurations. Finally, we compare CACOMPARE to state-of-the-art solutions and attempt to explain why it outperforms others.

TABLE I: Object Programs and Projects for Experiments

Program	Version	Description
busybox	1.25.1	Software providing several stripped-down Unix tools in a single executable file
convert	6.9.2	Command line interface to the ImageMagick image editor/converter
curl	7.39	Multi protocols supported data transferor with URL syntax
lua	5.2.3	Command line scripting parser for lua, a lightweight scripting language
mutt	1.5.24	Text-based email client for Unix-like systems
openssl	1.0.1p	Toolkit implementing the TLS/SSL protocols and a cryptography library
puttygen	0.64	Part of PUTTYGEN suit, a tool to generate and manipulate SSH public and private key pairs
siege	3.0.1	Http load testing and benchmarking utility
wget	1.15	Multi protocols supported file retriever. GNU command line project

A. Experiment Setup

Table I lists object programs and projects for the evaluation. We compile them on three architectures: IA-32, ARM and MIPS, with different compilers (`gcc` v4.7.3 and `clang` v3.0) and variant optimization levels (-O3, -O2 and -O0), generating overall 72 binaries.

The IA-32 binaries are compiled in Ubuntu 12.04 (i386), a virtual machine with 2G RAM allocated. ARM and MIPS binaries are compiled in QEMU emulation environments, whose systems are Debian 7.0. Analysis processes are performed in the host system, which is running on an Intel Core i5-2320 @ 3GHz CPU with 8G DDR3-RAM.

B. Ground Truth

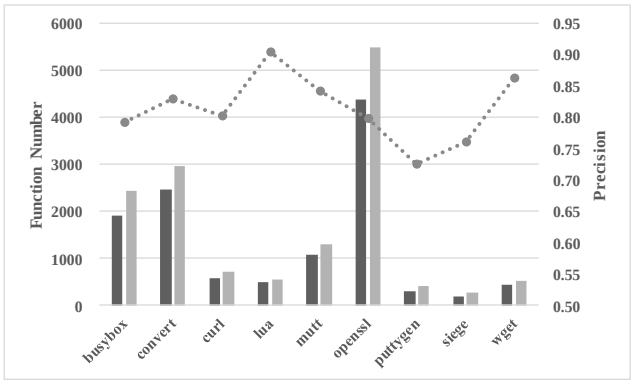
CACOMPARE performs comparisons on stripped binaries. To verify the accuracy of results, we also compile copies of those object programs with the `-g` option to establish ground truth basing on debug symbols. According to the symbol names, if the Rank 1 function in the resulting list holding the same name with the template function, then the match is correct.

Besides, compilers do function duplication to copy functions in resulting binaries, which ensure the jump distance from a caller to its callee is less than the page size (0x1000 bytes for 32-bit virtual address space). It avoids a page fault when calling a function to improve code efficiency. As the duplicated functions are exactly identical, if a match is two duplicated instances of the same function, it is also considered as a correct one.

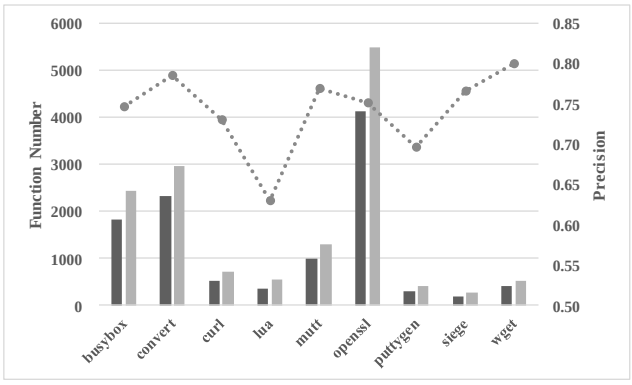
C. Accuracy

For each following experiment, we compare two binary programs *A* and *B* from the same code base. We treat every function of *A* as templates and attempt to detect the corresponding one in *B* (the target function). The accuracy is measured by the ratio of correct matches ranked at 1st in the resulting function lists basing on the ground truth.

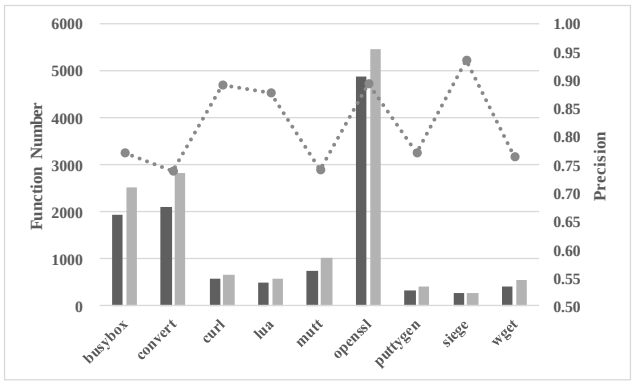
1) Analysis across Architectures: The 9 object programs are compiled on the three architectures separately, with `gcc v4.7` as the compiler and the optimization option is `-O3`. The results are displayed in Figure 6. Except for `lua`



(a) IA-32 vs ARM



(b) IA-32 vs MIPS



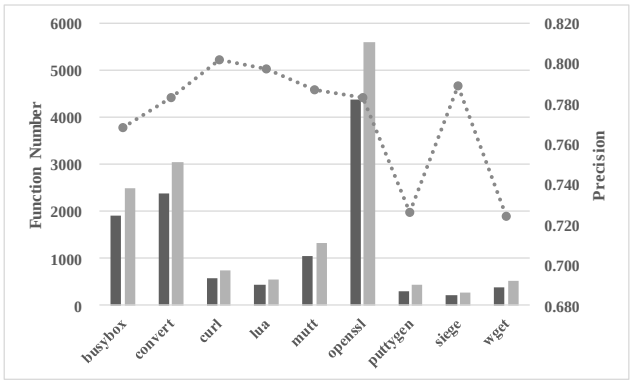
(c) ARM vs MIPS

Fig. 6: Results of cross-architecture comparisons. The gray bar represents the number of template functions and the black bar is the correct match number. Each dot on the dash line specifies the precision of that comparison.

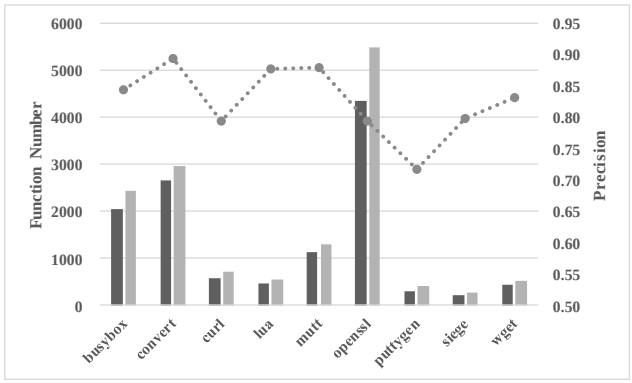
in Figure 6b whose precision is 63.0%, accuracy of other comparisons are all over 70%. The average precision of experiments in Figure 6 is 80.1%. The accuracy of IA-32 versus other architectures is 78.0% (81.1% vs ARM and 74.7% vs MIPS), while the average precision of ARM vs MIPS is 85.3%. Reasons causing more differences from IA-32 binaries to other architectures are as followed:

- **Library Function Inlining.** Although the compiler and optimization level is the same for all binaries, `gcc` on IA-32 trends to adopt more radical strategies to optimize code, including inlining library functions, while we observe that it hardly does so on ARM or MIPS even though the `-O3` option is set. CACOMPARE provides a pre-defined constant for library function on ARM or MIPS, but emulates executions of inlined ones for IA-32 binaries. If the following control flow depends on the result of a library function, the executed paths are then different, leading to false positives.
- **Float Point Processing.** For IA-32 binaries in the evaluation, float point numbers are processed with the FPU register stack. Operations with the float point stack generate signature of comparison operands as well, such as checking whether the stack is full or empty. They are noisy data but distinguishable from other normal comparison operands, while ARM and MIPS both have specific float point registers. Thus the results of comparison between them are better than those compared to IA-32.
- **Switch Implementation.** Apart from indirect jumps with branch tables, there also exist other methods to implement switches, such as binary search. For switches with indirect jumps, CACOMPARE chooses the specific target to continue the emulation, while for others, the target is selected through a set of conditional branches, which results in different paths of execution. Take `OpenSSL` as an example, the ARM and MIPS versions of binaries both have over 100 switches implemented with indirect jumps, while the IA-32 binary only has 76 switch instances with indirect branches. Therefore, the precision of `OpenSSL` comparisons in Figure 6a and 6b is 79.8% and 75.2% separately, but is 87.8% in Figure 6c, better than previous ones.

2) Analysis across Compiling Configurations: In this section, we evaluate the capacity of CACOMPARE detecting binary functions compiled with different configurations, in-



(a) GCC 4.7 vs Clang 3.0



(b) -O3 vs -O0

Fig. 7: Results of Different Compiling Configurations on IA-32

cluding variant compilers and optimization options.

In the experiments of different compilers, binaries are compiled with `gcc` v4.7 and `clang` v3.0, both with optimization level `-O2`. For comparisons of variant optimization options, we only discuss cases of comparisons between `-O3` and `-O0`, because high optimization levels contain all strategies of lower ones. Binaries compiled with above two options have the largest differences than any other pair of optimization levels.

Figure 7 displays the results of different compiling configurations on IA-32. In Figure 7a, the precision of all comparisons is over 70.0% and its average value is 78.2%. A reason causes the false positives is the two compilers select different instructions to process **float point values**. `GCC` chooses x87 floating-point instructions to assist the processing, while `Clang` uses SSE (Streaming SIMD Extensions). The former solution leverages the float point stack, and the latter one provides specific float point registers (e.g., XMM0-XMM7).

For comparisons of binaries with different optimization levels, the precision of all cases is over 70.0% as well (Figure 7b). The average precision is 82.6%. The main reason introducing false positives is **function inlining**. For example, function *B* is a subroutine of *A*. In an optimized version of the binary, *B* is inlined into *A* becoming *AB*. If *B* is the template function, the matching of *B* and *AB* would fail.

Overall, apart from the function inlining, cases of different optimization options are simpler than other comparisons which are faced with various optimization strategies of different compilers or variant instruction sets. So the performance of CACOMPARE for optimization differences (average 82.6%) is better than others (average 78.2% for `GCC` vs `Clang`, 78.0% for IA-32 versus other architectures).Results of different compiling configurations on ARM and MIPS is similar to IA-32, so we omit the analysis of those experiments.

Figure 8 shows the results of comparisons of binaries compiled with different optimization options across architectures. Results of cross-architecture comparisons are also presented as references. Obviously, the accuracy of the experiments is lower than former ones. The average precision of comparisons with `ARM_O0` is 77.8% and 67.7% for `MIPS_O0`. But the difference is small that the precision of the solid line in Figure 8a is only 3.3% lower than the dash line on average, and for MIPS instances in Figure 8b, the average difference is 7.0%.

D. Efficiency

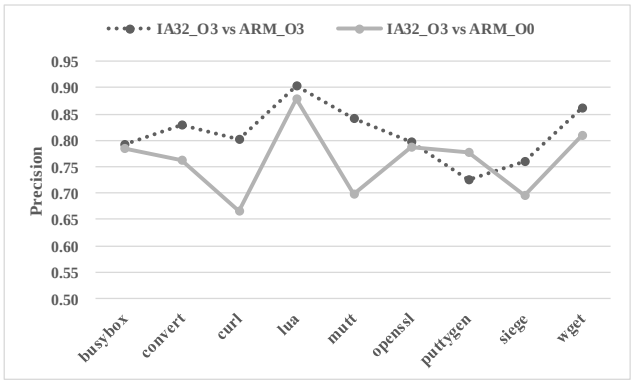
As described in §III-D, because of the adoption of MinHash, the process of signature comparison, which is the most costly process of CACOMPARE, only requires 5.2 seconds on average for comparison of each template function. For the preparation processes of CACOMPARE, it takes less than 90 seconds for each object binary to complete. Taking `OpenSSL` as an example, which has the largest number of functions in object programs (5,780 function on average), the average time of preprocess along with argument recognition and switch indirect jump detection is 55.2 seconds. The translation time of an `OpenSSL` binary is 32.9 seconds on average.

E. Comparison with State-of-the-Art Solutions

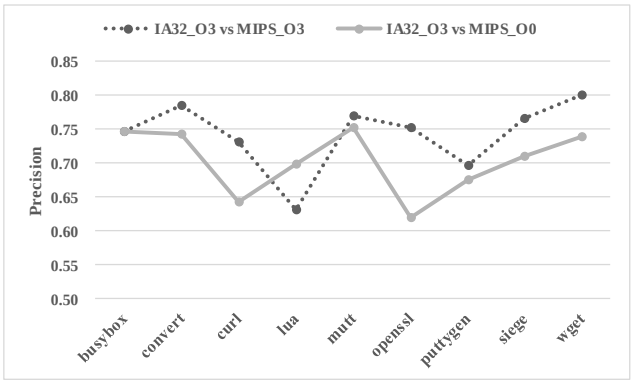
In this section, we compare CACOMPARE to state-of-the-art similar binary code comparison solutions from the perspective of accuracy. In the literature, MULTI-MH [25], GENIUS [13] and BINGO [6] provide solutions for detecting similar binary code across architectures. Although the source code is unavailable, those solutions are all conducted experiments on `BusyBox` / `Coreutils` and `OpenSSL`. So we compare results of CACOMPARE on those projects to above solutions' with the same settings.

MULTI-MH ranks 32.4% of functions in `BusyBox` (ARM vs x86) at Rank 1, while the precision of CACOMPARE is 83.4%. Besides, MULTI-MH obtains the precision of 32.1% for `OpenSSL` (ARM vs MIPS), and that of CACOMPARE is 87.8%.

GENIUS randomly selects 1,000 functions from `BusyBox`, `OpenSSL` and `Coreutils` as the dataset, which are compiled by `GCC` and `Clang` with `O0-O3`. Finally, it ranks 27% of the functions at top 1, whereas the average precision of all



(a) IA-32 vs ARM



(b) IA-32 vs MIPS

Fig. 8: Results of Different Optimization Levels across Architectures

comparisons performed by CACOMPARE on `BusyBox` and `OpenSSL` is 76.9%. `Coreutils` provides basic utilities of the GNU operating system. It is similar to `BusyBox` but much smaller in size (`Coreutils` has around 250 functions while `BusyBox` has more than 2,000). So the results of `Busybox` can be treated as a reference to `Coreutils`.

BINGO gives the precision of 41.3% for `BusyBox` across architectures on average, while it is 79.2% of CACOMPARE. For comparisons of different compiling configurations, BINGO ranks 41.5% of the functions at top 1, and CACOMPARE achieves the precision of 78.7%.

The reason that CACOMPARE outperforms other solutions is CACOMPARE depends on semantic signatures from function executions, whereas the others rely heavily on CFG (control flow graph). MULTI-MH leverages CFG to compute the similarity of basic blocks. GENIUS directly uses reduced CFG to detect isomorphic graphs. BINGO extracts code signatures basing on n-grams of CFG. However, CFGs of binaries from the same source code could be different because of different instruction sets. For example, on IA-32, there exists an instruction `REP`, which repeats a string instruction several times. To fulfill the same function, a loop structure would be generated on ARM and MIPS, adding several basic blocks to the CFG. Besides, optimizations performed by compilers alters CFG as well, such as function inlining, loop unrolling, etc. CACOMPARE avoids the representation and structure gaps introduced by different instruction sets and compiling configurations. Therefore it is much more robust to above factors of code transformations.

Additionally, MULTI-MH and BINGO employs read/write values as a code signature which includes intermediate values of executions. However, those values cannot be reserved because of compiler optimization, then they become noise date. CACOMPARE does **not** depends on intermediate values during emulation, but relies on input/output values which are strongly related to semantics. The problem of noisy intermediate values also affects BLEX, which is another state-of-the-art solution for single architecture (x86-64). It detects similar binary code of `Coreutils` and ranks 64% of functions at top 1, while the average precision of all comparisons of `Busybox` on IA-32 by CACOMPARE is 78.7%.

VI. DISCUSSION AND FUTURE WORK

In this section, we discuss remaining challenges and future work of the system.

A. Obfuscation

CACOMPARE depends on determined valued during the emulation, so it cannot handle programs implemented with randomized algorithms. Besides, CACOMPARE is vulnerable to code obfuscation, especially control flow obfuscation (e.g., CFG flattening, opaque predicates), which brings huge side effects to the signature of comparison operands and conditional codes. It is an important issue for static analysis as well. Thus, if a binary to be analyzed is obfuscated, it needs to be deobfuscated firstly. Papers [14], [29] introduce corresponding techniques.

B. Signature Sequence Length

CACOMPARE concentrates on the detection of complex functions which have large numbers of memory accesses, logic branches and library function callings. Those functions are also the main targets of compiler optimizations which lead to structure gaps, whereas for simple functions which may only have a few instructions, structure transformation is commonly small. Therefor, it is possible to detect simple functions whose semantic signature is insufficient with classical methods (e.g., leveraging abstract syntax tree [17]).

C. Library Function Inlining

As described in §V-C1, library function inlining may result in false positives. It is possible to adopt the idea of *selective inlining* proposed in [6] which inlines library functions. Firstly, a database of standard libraries is established, including IR of library functions and their parameter information. When a library function is invoked during execution emulation, CACOMPARE collects the argument values and passes them to

the corresponding one in the database to generate signatures and return values.

D. Input Check Bypassing

In some cases, at the beginning of a function, it may check values of inputs, requiring specific values which are legal for execution. CACOMPARE provides functions with random values whose possibility is low to pass the checks, leading to short lengths of signature sequences. Thus, techniques such as fuzzing testing [26] needs to be introduced to pass those checks in the future.

VII. RELATED WORK

A. Code Clone Detection

Cloned (or similar) code sources from the code reuse during software development. Identifying such code automatically is helpful for software maintenance (e.g., bug fixing). The technique of code clone detection is transplanted for other applications as well, such as malware analysis, program comprehension, etc. In the last decade, the focus of this technique migrates gradually from source code to binary, and the methodologies do from syntax-based to semantic-based.

CCFinder [19] detects cloned source code in large scales basing on lexical code tokens. CloneDR [4] leverages AST (Abstract Syntax Tree) to compute the similarity of cloned code. DECKARD [17] further extracts feature vectors from AST, improving the efficiency and accuracy of the detection. Cases of binary code clone detection are more challenging because of the lack of symbol information. In [27], Sæbjørnsen et al. propose the first practical code clone detection algorithm for binary executables. They normalize assembly instructions and model binaries with structural information to compute the similarity of binaries. Khoo et al. [20] employ n-grams with graphlets to detect cloned code with structural matching. In [9], David et al. measure similarity of binaries with edit distances of their control flow graphs.

As binaries may compiled with different configurations (e.g., variant compilers and optimization levels), there are huge differences in their representations, even though they are compiled from the same code base. So detecting similar binaries is actually the problem of semantic similarity detection, while above syntax- and structure-based methods cannot handle such cases. Jhi et al. [16] and Zhang et al. [33] leverage core values, which are irreplaceable runtime values of program executions, to detect software plagiarism. Luo et al. [23] and Zhang et al. [34] exploit symbolic execution to compare binary code similarity. In [10], Egele et al. propose Blanket Execution (BLEX) for full code coverage to compare binary code with memory access and function calling as features. David et al. [8] employ inputs and outputs of basic blocks to solve the comparisons of binary functions compiled with different compilers.

Apart from variant compiling configurations, solutions to cross-architecture comparisons need to tackle generally incomparable instruction sets. In [25], Pewny et al. propose the first solution to detect known bugs in binaries for different architectures via code similarity comparison. Afterwards, Eschweiler et al. [12] combine the syntax and structure features of binaries to compare their similarity. Feng et al. [13] continue the work and improve the efficiency to detect similar binaries of firmware images with reduced control flow graphs. In [6], Chandramohan et al. further leverage the technique of selective inlining to improve the accuracy of detection. Additionally, Hu et al. [15] run programs to extract code signatures of executed functions then compare their similarity.

B. Function Prototype Recovery

Function prototype recovery is a classical issue of binary program analysis. It is widely applied in fields of binary decompilation, security auditing, binary rewriting, etc. Balakrishnan et al. [3] exploit value set analysis to recovery variables in executables. Lee et al. [21] reconstruct data types of binary variables according to how the data is used. In [18], Caballero et al. identify and extract interfaces of binary functions with dynamic taint analysis. In [11], the authors not only identify memory arguments but also propose algorithms to recognize register arguments. In this paper, it is not necessary to recover function prototypes or argument types. CACOMPARE identifies parameters consumed by binary functions statically according to calling conventions of various architectures.

VIII. CONCLUSION

In this paper, we propose a semantics-based approach to detect binary clone functions and implement it in a prototype system named CACOMPARE. CACOMPARE first recognizes arguments and switch indirect jump targets of each function, then converts the binary into IR and emulates the execution with random values as input to extract semantic signatures. Finally, CACOMPARE detects cloned function by computes similarity of those signatures. The experimental results indicate that CACOMPARE is effective for cross-architecture and cross-compiling-configuration comparisons of binaries. Further we show that CACOMPARE outperforms the state-of-the-art solutions to binary similarity comparison across architectures.

IX. ACKNOWLEDGMENTS

We would like to thank the reviewers for their insightful comments which greatly helped to improve the manuscript. This work is partially supported by the Key Program of National Natural Science Foundation of China (Grants No.U1636217), the Major Project of the National Key Research Project (Grants No.2016YFB0801200), and the Technology Project of Shanghai Science and Technology Commission under Grants No.15511103002.

REFERENCES

[1] 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.

[2] 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.

[3] G. Balakrishnan and T. Reps. Divine: Discovering variables in executables. In *Proceedings of the 8th international conference on Verification, model checking, and abstract interpretation (VMCAI)*, 2007.

[4] 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.

[5] A. Z. Broder. On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of Sequences (SEQUENCES)*. IEEE, 1997.

[6] 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.

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

[8] 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 (PLDI)*, 2016.

[9] 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.

[10] 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)*, 2014.

[11] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua. Scalable variable and data type detection in a binary rewriter. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[12] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. In *The Network and Distributed System Security Symposium (NDSS)*, 2016.

[13] 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.

[14] F. Gabriel. Deobfuscation: recovering an ollvm-protected program. http://blog.quarkslab.com/deobfuscation-recovering-an-ollvm-protected-program.html.

[15] 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)*. IEEE, 2016.

[16] 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.

[17] 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.

[18] C. Juan, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. In *The Network and Distributed System Security Symposium (NDSS)*, 2010.

[19] 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*, 2002.

[20] 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.

[21] J. Lee, T. Avgerinos, and D. Brumley. Tie: Principled reverse engineering of types in binary programs. In *The Network and Distributed System Security Symposium (NDSS)*, 2011.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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 (NDSS)*, 2017.

[27] 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.

[28] 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 *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016.

[29] 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.

[30] M. J. G. Vikram Adve and P. Petersen. *Languages and Compilers for Parallel Computing*. Springer, 2007.

[31] S. Wang, P. Wang, and D. Wu. Reassembleable disassembling. In *24th USENIX Security Symposium (USENIX)*. USENIX Association, 2015.

[32] 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.

[33] 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.

[34] 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.