# Accurate and Scalable Cross-Architecture Cross-OS Binary Code Search with Emulation

Yinxing Xue, *Non-member,* Zhengzi Xu, *Non-member,* Mahinthan Chandramohan, *Non-member,* Yang Liu, *Member, IEEE,* and Chia Yuan Cho, *Non-member,*

**Abstract**—Different from source code clone detection, clone detection (similar code search) in binary executables faces big challenges due to the gigantic differences in the syntax and the structure of binary code that result from different configurations of compilers, architectures and OSs. Existing studies have proposed different categories of features for detecting binary code clones, including CFG structures, n-gram in CFG, input/output values, etc. In our previous study and the tool BINGO, to mitigate the huge gaps in CFG structures due to different compilation scenarios, we propose a selective inlining technique to capture the complete function semantics by inlining relevant library and user-defined functions. However, only features of input/output values are considered in BINGO. In this study, we propose to incorporate features from different categories (e.g., structural features and high-level semantic features) for accuracy improvement and emulation for efficiency improvement. We empirically compare our tool, BINGO-E, with the pervious tool BINGO and the available state-of-the-art tools of binary code search in terms of search accuracy and performance. Results show that BINGO-E achieves significantly better accuracies than BINGO for cross-architecture matching, cross-OS matching, cross-compiler matching and intra-compiler matching. Additionally, in the new task of matching binaries of forked projects, BINGO-E also exhibits a better accuracy than the existing benchmark tool. Meanwhile, BINGO-E takes less time than BINGO during the process of matching.

**Index Terms**—Binary Code Search, Binary Clone Detection, Vulnerability Matching, Emulation, 3D-CFG

## 1 INTRODUCTION

WITH the prosperity of open source projects and open software under GPL, new software (even commercial software) can reuse the code of the existing related projects. However, extensive reuse of existing code leads to some legal and security issues. According to the report on 'gpl-violations.org' [1], over 200 cases of GNU public license violations have been identified in software products. Among them, VMWARE is the well-known software vendor that is facing the lawsuit regarding the GPL violation [2]. Meanwhile, code reuse brings risks that a vulnerability disclosed in reused projects can spread into the new commercial products as undisclosed vulnerability. For example, in Dec. 2014, vulnerabilities in the implementation of the network time protocol (NTP) *ntpd* affected the products that import it, such as Linux distribution, Cisco's 5900x and Apple's OSX switches.

When the source code of an executable or a library is not available, binary code search can facilitate the tasks of GPL violation detection [3], software plagiarism detection [4], [5], reverse engineering [6], semantic recovery [7], malware detection [8] and buggy (vulnerable) code identification [9], [10], [11] in various software components. However, binary code search is a fundamental problem that is much more challenging than the problem of source code search, due to the syntax gaps caused by the platform, the architecture and the compilation options.

- *This research is supported in part by the National Research Foundation, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-30).*
- *Y. Xue, Z. Xu, M. Chandramohanare and Y. Liu are with Nanyang Technological University, Singapore, 637553. E-mail:tslxuey@ntu.edu.sg, XU0001ZI@e.ntu.edu.sg, mahintha001@e.ntu.edu.sg, yangliu@ntu.edu.sg.*
- *C.Y. Cho is with DSO National Laboratories, Singapore, 118230. E-mail:cchiayua@dso.org.sg*

*Manuscript received April XX, 2017; revised XXXX XX, 2017.*

In source code search, the similarity of two code segments is measured based on some representations of source code, e.g., approaches based on token [12], abstract syntax tree (AST) [13], control flow graph (CFG) [14], or program dependency graph (PDG) [15]. All these representations capture the syntactic or structural information of the program, and yield accurate results for source code search. However, these approaches fail when applied to binary code search. The reason is that the same source code may be compiled into the assembly code with different structures due to the choice of architecture, platform or compilation options. Thus, the syntactic or structural information is not sufficient for accurate binary code search.

We propose three desirable characteristics for an accurate yet scalable cross-architecture and cross-OS binary code search tool.

- **P1.** Resilient to the syntactic and structural differences due to different configurations of compilers, architectures and OSs.
- **P2.** Accurate in capturing the abstract and complete function semantics, balancing the impact of compiler optimization levels.
- **P3.** Scalable to large-size real-world binaries, avoiding the overheads of the analysis based on dynamic executions or constraint solving.

The existing binary code search approaches adopt two types of analysis: namely, static analysis [10], [16], [4], [9] and dynamic analysis [17], [18], [19], [20]. Static approaches usually rely on the syntactical and structural information of binaries, especially CFG (i.e., a control flow of basic blocks within a function) to perform the matching. Dynamic approaches often inspect the invariants of input-output or intermediate values of program at runtime to check the equivalence of binary programs. In general, static approaches are good at P3, but suffer from P1 and P2. On the contrary, dynamic methods have merits in P2 and P3, while having drawbacks in P3.

TABLE 1: Comparison of existing techniques. Here, ✓ or ✗ represent whether it supports the corresponding feature or not, respectively.

| Tool | Type | Low-level semantic fea. | High-level semantic fea. | Structural fea. | Syntactic fea. | Constraint solving | Fast filtering | Selective inlining | Emulation |
|---|---|---|---|---|---|---|---|---|---|
| BLEX [20] | Dynamic | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| TRACY [10] | Static | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ |
| DISCOVRE [21] | Static | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ |
| COP [4], [5] | Static | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| Bug search [9] | Static | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| ESH [22] | Static | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |
| BINGO | Static | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ |
| BINGO-E | Static | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ |

In Table 1, we list the state-of-the-art binary code search tools in the literature. Among these tools, only BLEX [20] is the dynamic function matching tool that uses seven semantic features generated from running execution. All the other tools adopt static analysis, and hence suffer from the issues in P1 and P2. TRACY [10] uses the pure syntax information for function matching, and it uses $k$-tracelet (i.e., basic blocks of length $k$ along the control-flow path), which is architecture- and OS-dependent. DISCOVRE [21] proposes to find across-architecture bugs in binaries in a scalable manner, where it uses two filters (numeric and structural) to quickly locate the similar candidates for the target function. COP [4] is a plagiarism detection tool that adopts the theorem prover to search for semantically equivalent code segments. The bug search tool proposed in [9] supports cross-architecture analysis by translating the binary code to an intermediate representation, and solving the resulting assignment formulas of input and output variables. ESH [22], inspired by the idea of similarity by the composition for images, proposes to decompose each procedure to small code segments (so called *strands*), semantically compare the strands to identify similarity, and lift the results into procedures.

In Table 1, we list the major features[1] used for similarity scoring and the search optimization techniques in these tools. *Low-level semantic fea.* refer to the features of low level architectural information used for matching (e.g., symbolic analysis on the CPU flag and register values [23]). *High-level semantic fea.* refer to the features of API-relevant information used for matching (e.g., the names and sequence of invoked built-in APIs of a lib [20]). *Structural fea.* refer to the structural information used by the static approaches (e.g., basic-block structures used in [4], [9], [21] and fixed length tracelet in [10]). *Syntactic fea.* refer to the syntactic information used for matching (e.g., the function name, or string literals extracted from binary [3]).

In general, merely relying on structural and syntactic features cannot make the match across the boundary of architecture, OS or compiler options, and hence fail P1. Instead, semantic features help to achieve P1. Even with the semantic features, the existing static approaches may not capture the complete function semantics due to different function inlining options of candidate functions [9]. Hence, the proper inlining strategy is desirable for achieving P2. Besides, scalability is a real challenge for tools [4], [9] based on symbolic analysis (low-level semantic features) and constraint solving. For example, COP took half a day to find target functions in Firefox for a few signature functions [4]. Thus, to achieve P3, time-consuming constraint solving should be avoided.

In our previous work [23], we propose a binary code search tool, named BINGO. For P2, BINGO adopts a *selective inlining* (§5) to include relevant libraries and user-defined functions in order to capture the complete semantics of the function. For P3, a filtering technique is proposed to shortlist the candidate functions.

1. Refer to section 4 for details of each category of features.

Subsequently, after inlining and filtering, for P1, BINGO extracts the partial traces of various lengths from the candidate functions as function models, and then extracts *low-level semantic features* from the function models (§4.3). Last, BINGO measures the function similarity score by Jaccard containment similarity of features.

In our previous evaluation on a number of real-world binaries [23], BINGO outperforms the available state-of-the-art tools (i.e., TRACY [10] and BINDIFF [24]), for the same tasks in terms of search accuracy and performance. However, we found some false positive (FP) cases in using BINGO. When a function has a small size of assembly instructions, the low-level semantic features extracted from this function could be too generic to contain any real semantics. For example, Fig. 2 shows a false positive case of BINGO due to the same input/output values (see details in §3.2).

To address the FP cases, the new version of our tool, BINGO-E, supports high-level semantic features and structural features. High-level semantic features we use are the information of system calls or library calls (§4.2). The rationale is that such information is commonly used for malware detection [25], as they can reveal the semantics of the functions. On the other hand, we borrow the idea of fast bytecode clone detection based on the approach of projecting the CFG into a centroid of 3 dimension coordinate [26]. Inspired by this approach (3D-CFG for short [26]), the problem of graph isomorphism is reduced to a problem of calculating the distances among centroids. 3D-CFG implemented in BINGO-E adopts the following structural information (§4.1): basic block (BB) sequence, loop information, and in-degree/out-degree of BB.

To speed up the process, BINGO-E leverages on emulation rather than constraint solving that is used in extracting low-level semantic features. We adopt UNICORN [27], a lightweight multi-platform, multi-architecture CPU emulator framework, to virtually execute the partial traces that are extracted from the function models of a given function. The emulation step may take significantly less time than constraint solving. The challenge of integrating UNICORN [27] lies in the handle of function calls (§4.3.3). Last, similarity scores of two functions in terms of structures, low-level/high-level semantics are combined in the final matching (§6).

**Key contributions.** Beyond the contributions that have been made in our previous work [23], we make these new contributions:

- Our work is the first attempt to incorporate emulation with cross-architecture cross-OS binary code search.
- We extract structural features based on the idea of centroid based clone detection. Previously, 3D-CFG based matching was mainly applied for bytecode clone detection.
- We combine different categories of features listed in Table 1. We evaluate our approach under the scenarios of cross-architecture matching, cross-OS matching, and cross-compiler matching.
- We empirically compare BINGO-E with BINGO and the available state-of-the-art tools of binary code search in terms of search accuracy and performance.

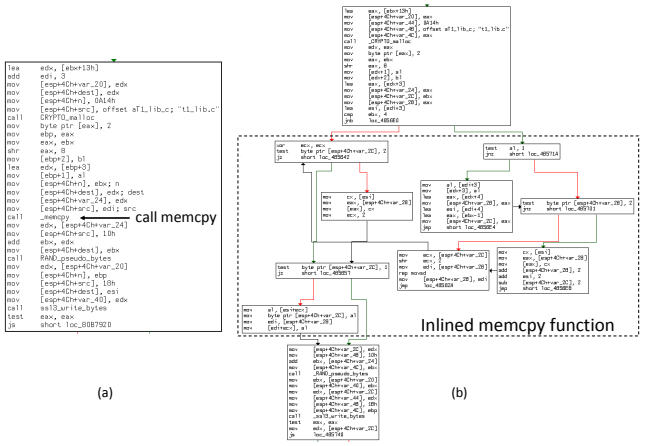



Fig. 1: Code segment responsible for Heartbleed vulnerability (CVE-2014-0160) appeared as in the binary (a) compiled with `gcc` 4.6, and (b) compiled with `mingw32`

**Experimental results.** We conduct various experiments to evaluate the effectiveness of BINGO-E. For cross-architecture matching on `coreutils` binaries, on average, the rate of best match for BINGO-E (65.6%) is significantly better than that of BINGO (35.1%) . For cross-compiler matching and intra-compiler matching on `coreutils` binaries, on average, the rate of best matching is improved from the range of 30%-60% (for BINGO) to the range of 70%-99% (for BINGO-E). For cross-OS matching on Windows binary `mscvrt` and Linux binary `libc`, on average, the rate of best matching is improved from 22% (for BINGO) to 51.7% (for BINGO-E). In additional experiments on binary code matching on forked projects, BINGO-E achieves a higher rate of best matching (93.6%) than that (88%) of the existing benchmark tool. Last, by virtue of emulation, the most time-consuming part of low-level semantic feature extraction is significantly accelerated, which allows us to introduce more features to improve the accuracy.

## 2 MOTIVATION

In this section, we state the challenges in the current research of cross-architecture and cross-OS binary code search with illustrative examples.

### 2.1 Motivating Example

A binary program consists of a number of functions, each of which can be represented by the CFG (control-flow graph), a (directed) graph of basic blocks (BBs). The assembly instructions in a function are systematically grouped into several BBs, which are considered as the building blocks of binary program. Thus, this BB-based representation is used by many static binary analysis tools.

The same source code may produce binary code of different BB structures, due to different compilation configurations. Taking the heartbleed vulnerability (CVE-2014-0160) for example, Fig. 1(a) and (b) show the BB structures of the same source code compiled with `gcc` and `mingw`, respectively. Apparently, these two binary code segments share no identical BB structures — with `gcc`, the vulnerable code is represented as a single basic block; with `mingw`, represented as several basic blocks. A detailed inspection suggests that library function `memcpy` is inlined in `mingw` version; while in `gcc`, it is not. Huge differences between the binaries in syntax and program structure make binary code search a challenging task.

### 2.2 Challenges for Syntactic Approaches

Syntax is the most direct information usable for code search. Most of existing approaches that rely on syntax information have attempted to use instruction patterns [28], [29], [16], [10]. As no consistent low-level syntax representation (i.e., assembly instructions) is available for cross-architecture search, these approaches fail for cross-architecture analysis. To make binary code search across the architecture and OS boundary, semantics-based matching has been proposed [4], [9], in which the machine state transition represents the semantics of a binary. Still, three challenges are faced in semantics-based matching.

### 2.3 Challenges for Semantics-based Matching

**C1: The trade-off challenge of deciding the granularity of function model.** The state-of-the-art tools COP [4] and bug search [9] assume that BB-structure is preserved across binaries, and a single BB can be matched with other BBs of similar semantics. Based on BB-structure, semantic features are extracted and built into the function model. However, as admitted by Pewny *et al.* [9], this approach is too sensitive to BB-structure differences, and hence problematic for smaller functions whose CFG structures are more susceptible to compilation options. For example, the BB-structure compiled with `gcc` 4.6 in Fig. 1(a) is significantly different from that compiled with `mingw32` in Fig. 1(b). In this case, the approach of BB-centric matching fails as no two BBs can match in Fig. 1.

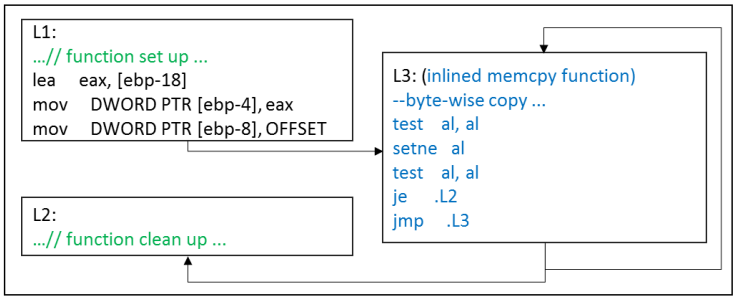
**C2. The accuracy challenge of using low-level semantic features.** Functions are considered in isolation by most of static tools, i.e., semantics of callee functions are not considered as part of the caller's semantics. This leads to partial semantics problem, especially when some common utility functions are implemented by the developers themselves (e.g., Adobe Reader has its own `malloc` implementation). *Blindly inlining* all the callee functions can be a remedy, as all the user-defined functions are inlined in [30]. Nevertheless, this approach fails in practice due to two main reasons: (1) heavy inlining may lead to code size explosion [31], and (2) not all the callee functions are semantically relevant to the caller function. Thus, an inlining strategy is needed to decide that: `memcpy` should be inlined in Fig. 1(a) for matching with the semantically relevant one in Fig. 1(b). On the other side, for functions not to be inlined, e.g., `ss13_write_bytes` in Fig. 1(a), features of library call information should not be ignored.

**C3. The performance challenge of using static symbolic analysis or dynamic execution.** Syntax based techniques, in general, are scalable [10]. As discussed earlier, they fail on cross-architecture analysis. To address this problem, semantics based approaches are preferred. However, extracting and solving low-level semantic features is a time-consuming job. Previously, we adopted an *efficient* and architecture-, OS- and compiler-*neutral* function filtering mechanism in [23]. With such mechanism, extracting input/output values of registers and flags of a function via constraint solving still takes 4,469s on average to extract low-level semantic features from 2,611 Linux `libc` functions — it takes 1.7s to extract semantic features from a `libc` function. Hence, this part still has much space for improvement. To scale up code search for large-size binaries in real world, an efficient method is desired.

Previously, BINGO has adopted $k$-tracelet for C1, selective inlining for C2, and filtering mechanism for C3 [23]. However, BINGO still suffers issues of false positive cases (see Fig. 2 and §3.2) and inefficient feature extraction. In this paper, BINGO-E adopts new features and emulation to address these issues (see §4).

```
...//function set up ...
mov    DWORD PTR [ebp-12], OFFSET
sub    esp, 8
push   DWORD PTR [ebp-12]
lea    eax, [ebp-26]
push   eax
call   strcpy
add    esp, 16
mov    DWORD PTR [ebp-16], eax
mov    eax, 0
...// function clean up ...
```

(a) String copy via calling function *strcpy*




(b) Memory copy in an iterative way

Fig. 2: A false positive case of BINGO

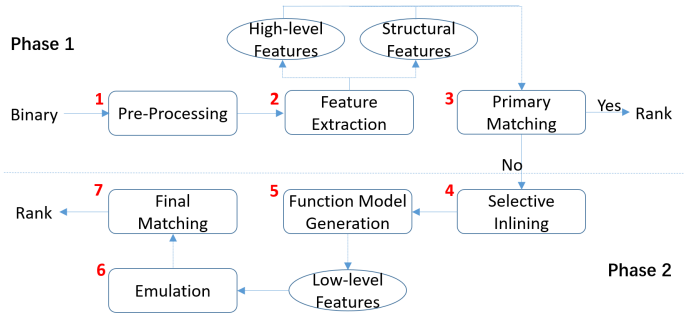



Fig. 3: BINGO-E work flow: rectangles represent processing steps, ovals represent features, arrows represent control/data flows

## 3 SYSTEM OVERVIEW OF BINGO-E

BINGO-E, as a new extension of BINGO, aims to be an accurate and scalable search engine for binary code. Given a binary function to be matched (*signature function* in search), BINGO-E returns the functions from the pool of analyzed binaries (*target function* in search), ranked based on their semantic similarity.

### 3.1 The Sketch of Work Flow

Fig. 3 shows the work flow of BINGO-E. The input is one signature binary function and the possible candidate target functions. The output is the ranked list of candidate target functions above the user-defined threshold of similarity score. The basic idea is the two-phase matching: first employing the structural and high-level semantic features to find similar target functions; if the similarity scores of all the returned target functions are below a certain threshold (*No* flow after primary matching in Fig. 3), adopting the low-level semantic features to compare the I/O values on $k$-tracelets of the signature function with those of candidate target functions.

We explain how each step in our approach works in brief. At step 1, given a signature function, it performs the preprocessing, i.e., disassembling and building the CFG for the signature function. At step 2, we extract these features from the high-level semantics and structure information in Table 2. As shown in Table 2, we extract 6 types of high-level semantic features (§4.2) and 3D-CFG structural features (§4.1). At step 3, a primary matching step is conducted to measure the similarity based on the Jaccard distance of high-level semantic features and the centroid distance of structural features (§6.1). If no results are above the similarity threshold, the process proceeds with comparing low-level semantic features (§6.2). At step 4, for each function in the target binaries, function calls are identified and selectively inlined according to the relevance of called libraries and other user-defined functions (§5). Note that selective inlining is for addressing C2 (§2.3). Selective inlining is transitive. Assume function $a$ calls function $b$ and $b$ calls function $c$. If $a$ selectively inlines $b$, then the decision of inlining $c$ into $b$ is according to Algorithm 1 (§5). If $a$ does not need to inline $b$, then it is certainly unnecessary to inline $c$ into $b$. At step 5, for the signature functions, length variant partial traces (up to $k$-tracelets) are generated and grouped to form the function models (§6.2.1), which low-level semantic features are extracted from (§6.2.2). Note that function model generation and feature extraction is a one-time job. At step 6, we rely on emulation to get the input/output values, flags, memory addresses of signature functions as low-level semantic features (§4.3.3). Finally, at step 7, combining Jaccard Distance for low-level semantic features and results of the primary matching, BINGO-E returns target functions that are above the similarity threshold in a ranked order (§6.3).

TABLE 2: Different categories of features used by BINGO-E

| Low-level Semantic Feature | High-level Semantic Feature | Structural Feature |
|---|---|---|
| mem. operations<br>flags<br>registers | op. type<br>system call tags<br>function call seq.<br>function parameter<br>local variable<br>op. code | BB sequence<br>loop information<br>In-degree of BB<br>out-degree of BB |

The approach is presented in the following way: in §4, different categories of features and emulation (step 2 and 6) are introduced; in §5, selective inlining (step 4) is elaborated; in §6, the key steps 3, 5 and 7 are explained with details.

### 3.2 Solutions to the Matching Challenges

Here, we highlight how BINGO-E overcomes the challenges aforementioned in §2.2 and improves BINGO.

**S1: *Flexible function model* via $k$-tracelet and 3D-CFG.** In BINGO and BINGO-E, to address C1, we borrow the idea of tracelet in TRACY [10], where David *et al.* extract a fixed-length partial trace of 3 BBs (3-tracelet). Then they check semantic equivalence between 3-tracelets based on data-dependence analysis and rewriting, in which constraining solving is applied to verify input/output invariants among the registers, flags and variables. In BINGO, we use a set of variant-length partial traces (i.e., a set of $k$-tracelet, $k$ is up to 4) to represent the function model. In BINGO-E, we complement the tracelet-based approach with function structural information (i.e., 3D-CFG). 3D-CFG is resilient to BB structure

changes due to optimization levels, as it uses structural information of a function as a whole.

**S2: Completing function semantics via *selective inlining* and identifying the *high-level semantics*.** To mitigate the issue of incomplete semantics of the analyzed function, selective inlining is proposed in BINGO. However, several FP cases are observed during semantic matching, where low-level semantic features are insufficient to reveal the true functionality. For instance, two code segments in Fig. 2 cannot be distinguished by low-level semantic features as they have the same input and output results for any given string input. In fact, they have different functionalities, as the code in Fig. 2(a) copies the string array via calling function *strcpy*, and the code in Fig. 2(b) does the memory copy (not limited to string copy) in an iterative way. In BINGO-E, we consider not only low-level semantic features (e.g., symbolic features on register, flag, memory status), but high-level semantic features that are relevant to function calls, including system calls and library calls.

**S3: Efficient semantics extraction via binary code *emulation*.** In order to speed up the process of extracting low-level semantic features, we propose to apply Unicorn [27] to virtually execute the extracted partial traces and identify the input/output values from them. Unicorn is a lightweight multi-platform, multi-architecture CPU emulator framework, which is built on QEMU. According to the manual [27], for extracting low-level semantic features, emulation may boost the efficiency by one or two orders of magnitude, compared with constraint solving.

To sum up, in Fig. 3, step 2, 3, 6 and 7 are newly introduced into BINGO-E. Among them, step 2 and 3 are the implementation of **S1** and **S2**, while step 6 is the implementation of **S3** (§3.2). Note that in step 2, only the static analysis is required so that the process of feature extraction is scalable. In step 6, to avoid the overheads of actual execution or constraint solving for extracting low-level features, emulation is adopted to speed up the process. In step 7, to remove the false positive cases due to matching with low-level semantic features alone, the results of step 2 (similarity in high-level semantics and BB-structure) are also considered (§6.3).

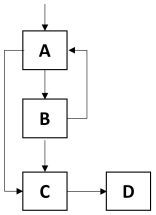
## 4 BINARY CODE MATCHING FEATURES

In this section, we elaborate the features that we adopt to measure the similarity of two binary code segments. As listed in Table 2, these features fall into three categories. For each feature in Table 2, the example and description are given in the following subsections. Features of structures and high-level semantics are extracted via static analysis, which is the functionality of step 2. Features of low-level semantics are extracted by emulation (step 6).

### 4.1 Structural Features

3D-CFG [32] depicts the CFG structure of a function at a high-abstraction level. It is used to measure the similarity between methods in/among bytecode of Android apps. The basic idea in [32] is to assign a structural 3D-coordinate value for each node in the control flow graph (CFG) of a method, then calculate the mass centroid of these coordinates as the centroid of the method. By calculating the distance between two methods' centroids, the similarity among these two methods can be measured.

According to [32], for a given method, each node in the CFG is a basic code block. Each node has a unique coordinate, denoted as a 3-tuple $(x, y, z)$, where $x$ is its sequential number in the CFG; $y$ is the number of its outgoing edges (out-degree of the node in CFG), and $z$ is the loop depth of the node.




**Structural features:**
BB sequence:
main: $A \rightarrow B \rightarrow C \rightarrow D$
the sequential number of BB (A): 1
In-degree of BB (A): 2
out-degree of BB (A): 2
Loop information:
The loop depth of BB (A): 1
involving number of BBs: 2

Fig. 4: The BB-structure of a function, and its structural features

Based on the nodes inside the function $f$, we define the centroid $\overrightarrow{c_f}$ of function $f$:

**Definition** 1. The centroid is a 4-tuple $(c_x, c_y, c_z, w)$ [32]:

- $w = \sum_{e(p,q)\in 3D-CFG}(w_p + w_q)$,
- $c_i = \frac{\sum_{e(p,q)\in 3D-CFG}(w_p i_p + w_q i_q)}{w}, where\ i \in \{x, y, z\}$

where $e(p, q)$ is the edge between the node $p$ and $q$, $(x_p, y_p, z_p)$ is the coordinate of the node $p$, and $w_p$ is the number of assembly instructions in the node of BB.

To emphasize the importance of function call instruction (relevant to high-level semantics), the weighted centroid is defined as $\overrightarrow{c_f}' = (c'_x, c'_y, c'_z, w')$ for function $f$, where $w' = w + N$, and $N$ is the number of function call instructions; $c'_x$, $c'_y$ and $c'_z$ are recalculated according to the new weight $w'$, respectively. Suppose each BB node in Fig. 4 contains only one instruction ($w_A = 1, w_B = 1, w_D = 1$), and node $C$ contains one function call instruction ($w_C = 2$). We have the following coordinates: $(1, 2, 1)$ for $A$, $(2, 2, 1)$ for $B$, $(3, 1, 0)$ for $C$ and $(4, 0, 0)$ for $D$. Then, the centroid is $(2.2, 1.5, 0.6, 10)$ and the weighted centroid is $(2.38, 1.38, 0.46, 13)$.

Given a function $f$, the function signature $\overrightarrow{f} = (\overrightarrow{c_f}, \overrightarrow{c_f}')$ is a feature vector containing the structural information of the function, where $\overrightarrow{c_f}$ is the centroid, $\overrightarrow{c_f}'$ is the weighted centroid.

The difference between the centroids of two functions $f_1$ and $f_2$ is the primary condition to evaluate their similarity:

**Definition** 2. The Centroid Difference Degree (CDD) [32] of two centroids $\overrightarrow{c_1} = (c_{1x}, c_{1y}, c_{1z}, w_1)$, $\overrightarrow{c_2} = (c_{2x}, c_{2y}, c_{2z}, w_2)$ (the distance of the weighted centroids $\overrightarrow{c_1}'$ and $\overrightarrow{c_2}'$) is

$$CDD(\overrightarrow{c_1}, \overrightarrow{c_2}) = max(\frac{|c_{1x}-c_{2x}|}{c_{1x}+c_{2x}}, \frac{|c_{1y}-c_{2y}|}{c_{1y}+c_{2y}}, \frac{|c_{1z}-c_{2z}|}{c_{1z}+c_{2z}}, \frac{|w_1-w_2|}{w_1+w_2})$$

Given two functions, the function-level difference degree is defined as the maximum value between their centroid distance (CDD) and their weight centroid distance (weighted CDD):

**Definition** 3. Function Difference Degree (FDD) of two functions $\overrightarrow{f_1} = (\overrightarrow{c_1}, \overrightarrow{c_1}')$ and $\overrightarrow{f_2} = (\overrightarrow{c_2}, \overrightarrow{c_2}')$ is

$$FDD(\overrightarrow{f_1}, \overrightarrow{f_2}) = max(CDD(\overrightarrow{c_1}, \overrightarrow{c_2}), CDD(\overrightarrow{c_1}', \overrightarrow{c_2}'))$$

The two code segments in Fig. 2 will not be matched as similar according to these structural features. Since the code segment in Fig. 2(a) has only one BB and the code segment in Fig. 2(b) has 3 BBs. Their CDD and weighted CDD are all 1 ($c_{1y}$ and $c_{1z}$ are all 0), which indicates zero similarity according to [32][33]. Thus, 3D-CFG captures the overall structure of a binary function; and it is effective in measuring the structural similarity of two given binary functions.

### 4.2 High-level Semantic Features

For the high-level semantics, we mainly make use of the operation types, system call tags (tags of function calls to system APIs), function call sequences, function call parameters, local variables

```
push  ebp
mov   ebp, esp
sub   esp, 56
sub   esp, 8
lea   eax, [ebp − 16]
push  eax
push  OFFSET FLAT : .LC0
call  scanf
add   esp, 16
mov   eax, DWORD PTR[ebp − 16]
sub   esp, 12
push  eax
call  strlen
add   esp, 16
mov   DWORD PTR[ebp − 12], eax
cmp   DWORD PTR[ebp − 12], 9
jg    .L2
```

(a) basic block $B_1$

```
mov   eax, 0
leave
ret
```

(b) basic block $B_2$

```
mov   edx, DWORD PTR[ebp − 12]
mov   eax, DWORD PTR[ebp − 16]
sub   esp, 4
push  edx
push  eax
lea   eax, [ebp − 56]
push  eax
call  memcpy
add   esp, 16
mov   eax, 10
sub   eax, DWORD PTR[ebp − 12]
mov   ecx, eax
mov   eax, DWORD PTR[ebp − 12]
lea   edx, [0 + eax ∗ 4]
lea   eax, [ebp − 56]
add   eax, edx
sub   esp, 4
push  ecx
push  32
push  eax
call  memset
add   esp, 16
```

(c) basic block $B_3$

Op. type:
$data\_transfer(13), stack(10),$
$arithmetic(11), call(4), logical(1),$
$control\_transfer(1), misc(1),$
System call tags:
$io, string, mem, mem$
Function call sequence:
$scanf \rightarrow strlen \rightarrow$
$memcpy \rightarrow memset$
Function parameter: Null
Local variable:
$ebp - 12(mov[ebp - 12], eax)$
Op. code:
$push \rightarrow mov \rightarrow ...$
$... \rightarrow leave \rightarrow ret$

(d) High-level semantic features of $B_1$, $B_2$ and $B_3$

```
int foo2()
{
 char ∗buf;
 char ∗p[10];
 scanf("%s", &buf);
 int len;
 len = strlen(buf);
 if(len < 10){
  memcpy(p, buf, len);
  memset(p + len,' ', 10 − len);
 }
 return0;
}
```

(e) the source code of these binary code segments

Fig. 5: the high-level semantic features of an exemplar function

and op. code. The basic idea is that function calls (especially system calls) carry the semantics of code and they can be used for semantic-based code matching tasks, e.g., system-call based malware detection [25]. Hence, the information of function calls (tags, parameters, sequences) is extracted as features. Additionally, we extract features from the numbers of different types of binary instructions. We first classify different binary instructions into 15 categories[2], including *data_transfer*, *arithmetic*, *stack*, *logical*, *shift_rotate*, *control_transfer*, *loop*, *string*, *flag*, *misc*, *sign*, *fstp*, *port*, *mmx*, *call*. Then we use the number of instructions in each category as a feature. Last, we also extract features from local variables. The reason is that local variables may represent the intermediate values in the execution, and the same intermediate values indicate the same or similar function semantics.

In Fig. 5, we give an example for the high-level semantic features. In Fig. 5(e), the semantics of the function $foo2$ is to do the string copy. Its binary code contains three BBs, in which $B_1$ has two out-edges to $B_2$ and $B_3$, and $B_3$ has one out-edge to $B_2$. The high-level semantic features in Fig. 5(d) are extracted from all the BBs of the function. Among all the instructions, we have 13 instructions of *data_transfer*, 10 of *stack*, 4 of *call*, 1 of *control_transfer*, and so on. Similar to the idea of mapping concrete system calls into abstract actions in [25], we add tags to the common system calls or library calls in order to capture the high-level semantics of the function. For example, the tag of function call *scanf* is *io*, the tag of function call *strlen* is *string*, and the tag of *memcpy* and *memset* is *mem*[3]. Similar to the idea of using API usage patterns for code recommendation [34] (source code search), we also extract the function call sequences on the longest program path: $scanf \rightarrow strlen \rightarrow memcpy \rightarrow memset$ on the control flow $BB_1 \rightarrow BB_3 \rightarrow BB_2$. Function $foo2$ takes no parameters, and has the value of $null$ for this feature. Local variables used in source code is $len$, which stores the return value of function call $strlen$ in Fig. 5(e). Hoverer, in the binary code ($B_1$ in Fig. 5(a)), the address at $ebp - 12$ stores the return value of function call $strlen$ (according to the instruction "`push eax`" that stores the return value of "`call strlen`" into `eax` and the instruction "`mov DWORD PTR[ebp − 12], eax`"). So we use the memory address $ebp - 12$ as the feature for local variable. Last, the instruction types for all instructions in all BBs are used as a feature.

The two code segments of the false positive case in Fig. 2 is not matched as similar according to these high-level features. Code segment in Fig. 2(a) has function call, while code segment in Fig. 2(b) does not. They also have different local variables and opcode. For example, local variables are `ebp − 12` and `ebp − 16` in Fig. 2(a), while they are `ebp − 4` and `ebp − 8` in Fig. 2(b). Therefore, high-level semantic features are very helpful in recovering function semantics.

### 4.3 Low-level Semantic Features

The basic idea of capturing low-level semantic features is to compare the input/output pair [4][9], or compare the frequencies of different types of memory operations [20]. In BINGO, features of input/output pairs are adopted for comparison (§ 4.3.1). In BINGO-E, as emulation is supported to improve the efficiency, we extend the approach to support features that are based on values (including intermediate values) read from (or written to) the program stack and heap (§ 4.3.2).

#### 4.3.1 *Features Used in* BINGO

Similar to [4] and [9], a set of symbolic formulas (called *symbolic expressions*) are extracted from each partial trace, where they capture the effects of executing the partial trace on the machine state. Here, machine state is characterized by a 3-tuple $\langle mem, reg, flag \rangle$ denoting the memory *mem*, general-purpose registers *reg* and the condition-code flags *flag* [9], [4]. For example, a machine state, before executing a partial trace, is given by $\mathcal{X} = \langle mem, reg, flag \rangle$, then just immediately after execution, the machine state is given by $\mathcal{X}' = \langle mem', reg', flag' \rangle$. Here, $\mathcal{X}$ and $\mathcal{X}'$ are referred to as *pre-* and *post-state*, respectively, and the symbolic expressions capture the relationship between these pre- and post-machine states.

In order to measure the similarity between the signature function and a target function, one can use a constraint solver, such as Z3 [35], to calculate the semantic *equivalence* between the symbolic expressions extracted from the signature and target functions. However, using a constraint solver to measure semantic equivalence is very costly [4] and not scalable for real-world cases, considering the complexity of pair-wise matching between the signature function and a possibly huge number of target functions. To tackle the scalability issue, in [9], a machine learning technique is applied. Specifically, Input/Output (or I/O) samples are randomly generated from the symbolic expressions and are fed into the machine learning module to find semantically similar functions. Here, I/O samples are the input and output pairs — assigning concrete values to the pre-state variables in the symbolic expressions and obtaining the output values of the post-state variables.

One of the drawbacks in randomly generating I/O samples is that the dependency among the symbolic expressions is ignored [9]. For example, consider the following 2 symbolic expressions:

$$\texttt{zf}' \equiv (\texttt{edx} < 2) \wedge (\texttt{edx} > 0) \quad (1)$$

$$\texttt{ecx}' \equiv \texttt{edx} + 4 \quad (2)$$

where symbolic expression 1 (Eq. 1) has one pre-state (i.e., `edx`) and one post-state variable (i.e., $\texttt{zf}'$), similarly, the symbolic expression 2 (Eq. 2) also has one pre- (i.e., `edx`) and one post- (i.e., $\texttt{ecx}'$) state variable. Thus these two symbolic expressions are mutually dependent through a common pre-state variable `edx`. However, in randomly

2. The instructions inside each category can be found from this link: https://sites.google.com/site/toolbingoe/instruction-category

3. A more complete mapping list for function call tagging can be found from this link: https://sites.google.com/site/toolbingoe/system-call-tags

```
X86:
mov  eax, 0xa
mov  [0x1100], eax
mov  ebx, 0xa
sub  eax, ebx
```

(a) X86 code segment 1

```
X86:
mov  eax, 0xa
push 0x4
push eax
push 0x1100
call memcpy
mov  ebx, 0xa
sub  eax, ebx
```

(b) X86 code segment 2

```
ARM:
mov  r0, #0xa
mov  r3, #0x1100
str  r0, [r3]
mov  r1, #0xa
subs r0, r0, r1
```

(c) ARM code segment

**Output:**
overflow : 0
carry : 0 for (a)(b), 1 for (c)
zero : 1
negative : 0
memory at 0x1100: 0x000a

(d) output of these segments

Fig. 6: An example of using emulation to extract low-level features

generated I/O samples (as in [9]), `edx` can be assigned to two different values in each symbolic expression, which ignores the dependency and may lead to inaccurate semantics.

In the above example, to satisfy the mutual dependency on pre-state `edx`, `edx` must be 1 and $\texttt{ecx}'$ must be 5 to ensure $\texttt{zf}'$ to be `true`. Hence, constraint solving is applied to find appropriate values for pre- and post-state variables, as it considers all the symbolic expressions from the same partial trace.

Note that in the above example, we compare the input/output value pair for the same registers, assuming they have the same initial state. However, for cross-architecture code search, input/output values of registers cannot be used, as no identical registers exist on different architectures. In BINGO, only input/output values of some flags and memory addresses are used. Thus, to further improve the accuracy, more features are needed for cross-architecture code search.

#### 4.3.2 *Features Newly Added in* BINGO-E

As aforementioned, the input/output values of registers cannot be directly used for cross-architecture code search. Owing to emulation, BINGO-E is now much faster and more convenient at extracting features of function arguments and intermediate values used by the function. Hence, to further improve the accuracy of low-level semantic features, we propose to incorporate the following features of intermediate values that are also used in BLEX [20]:

1) Values read from the program heap (denoted as "v1")
2) Values written to the program heap (denoted as "v2")
3) Values read from the program stack (denoted as "v3")
4) Values written to the program stack (denoted as "v4")

According to the results reported by Egele *et al.* [20], features of "v1" can achieve a matching accuracy of $40\%$, and features of other values can achieve the accuracy between $50\% \sim 60\%$. Hence, using these features together can help find code segments that own identical or similar function parameters and local variables (stack operations), intermediate objects (heap operations). Since we adopt emulation to extract the low-level semantic features, the above 4 types of features are extracted via the address analysis during the emulation of the assembly instructions.

The address analysis is based on the basic idea: before the emulation, we assume the starting address of the stack (`ebp`) as a certain hex value. Then in the emulation, we keep tracking the end address of the stack (`esp`) after each instruction is executed. If an operation falls in the address between `ebp` and `esp`, we can know the operation is on stack (belonging to "v3" or "v4"). Otherwise, we will consider it as the operation on heap (belonging to "v1" or "v2"). However, there are few cases where the instructions are operated on the global stack (the stack for global variables). In emulation, it is difficult to further distinguish those non-heap addresses, and hence we just consider these addresses as heap addresses.

#### 4.3.3 *Emulation*

In BINGO, for a code segment or a $k$-length partial trace (i.e., $k$-tracelet), the low-level semantic features (e.g., the symbolic expression in § 4.3) are extracted and constraint solving is applied to resolve these symbolic expressions. Executing code segments in BLEX [20] or solving them by SMT solver in BINGO lead to an intractable problem — the scalability problem especially when the code segment is long and the symbolic analysis becomes complicated.

Emulation is faster than actual execution and much faster than constraint solving. In BINGO-E, emulation is adopted for extraction of feature values introduced in § 4.3.1 and § 4.3.2. Given two code segments, we assume they have the same initial memory status (i.e., the same address and layout). For code matching on the same architecture, we check the values of registers, flags after emulation and the addresses of read/write operation on stack and heap during emulation; while for code matching across architectures, we check the values of flags after emulation and the addresses of read/write operation on stack and heap during emulation. For example, for three code segments in Fig. 6, with the same initial memory status, they have the same resulting memory status — the overflow flag (V-flag) is 0, the zero flag (Z-flag) is 1, the negative flag (N-flag) is 0, and the carry flag (C-flag) is consistent (0 for x86 in Fig. 6(a) and (b) has the same meaning as 1 for Arm in Fig. 6(c)). Being code matching on the same architecture, code segments in Fig. 6(a) and (b) even have the same resulting values for registers `eax` and `ebx`. In addition, we also use the low-level features on values read from or written to the program heap/stack in this case. For example, `0x1100` is the intermediate value written to the stack.

In emulation part, the major technique challenges are 1). handling the function call and 2). choosing input or initial value for registers, flags and memory addresses. Note that emulation does not conflict with selective inlining. The emulation step is after the selective inlining step, and hence many important function calls (e.g., system calls and library calls) have been inlined. For function calls that are not selected for inlining, we adopt two different strategies for different scenarios:

1) In matching binary code from the same code base, we adopt the strategy of inlining-none — that is none of these called functions will be inlined. The rationale is twofold: functions that are not inlined by selective inlining are less likely representing the semantics of the program (more potential of being utility functions); since signature function and target functions both have the called function (due to the same code base), for the same input, ignoring the called function should still produce the same results for the semantic clones.
2) In matching binary code from different code bases, we adopt the strategy of inlining-all, namely the strategy that all the called function will be inlined to emulate the execution of instructions. The reason is that binary segments compiled from different code bases usually call different system calls or library calls — ignoring them will bring errors and lead to different results for the same input, even if the two binary segments are semantic clones.

Regarding the emulation input, we adopt the strategy of using three totally different input values. For two binary functions to be matched via emulation, we feed it with three types of input: all values with `0x0000`, all values with `0x1111`, and all random values. If two binary segments are matched according to low-level semantic features for all three types of input, we considered they are matched.

## 5 ACCURATE: SELECTIVE INLINING

Inlining is a technique in compilers to optimize the binaries for maximum speed or minimum size [31]. During compilation, the strategy of inlining is different according to the configured optimization level, which creates the challenge for binary code search. To mitigate this issue, we propose a selective inlining strategy which is based on function invocation patterns. Note that our strategy has a different goal in contrast with compilation process — ours is for program semantics recovery, while compilers' is for speed or size optimization.

### 5.1 Function Invocation Patterns

In order to inline relevant functions, we use the function invocation patterns to guide the inlining decision. Based on our study, six commonly-observed invocation patterns are identified and summarised in Fig. 7. Incoming (outgoing) edges in Fig. 7 represent the incoming

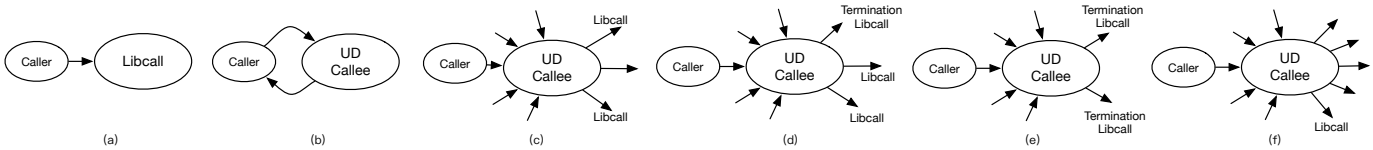



Fig. 7: Commonly observed function invocation patterns, where 'UD' denotes user-defined callee function. Here, all the incoming and outgoing edges (or calls) represent user-defined functions, unless specified as *Libcall* or *Termination Libcall* next to them

(outgoing) calls to (from) the function. Here, we elaborate the six patterns as follows.

**Case 1:** Fig. 7(a) depicts the direct invocation of standard C library function(s) by the caller function under investigation. To recover the semantics, it is essential to understand the semantics of called library function(s). Hence, the library function is inlined into the caller function. Currently, we only consider the most common standard C library functions — in total 60 functions, from both Linux (`libc`) and Windows (`msvcrt`), for inlining. This list can be further extended when necessary.

**Case 2:** Fig. 7(b) depicts the case of a recursive relationship between the caller and the UD (user-defined) callee function $f$. Hence, we inline $f$ into its caller. Note that the recursive functions are, unlike in compilers, inlined only once. For example, `gcc` has a default inlining depth of 8 for recursive functions.

**Case 3:** Fig. 7(c) depicts the common pattern of a *utility function* — e.g., the UD callee function $f$ is called by many other UD functions, while $f$ calls several *library functions* and a very few (or zero) UD functions. This suggests that $f$ is behaving as a utility function, as $f$ has some semantics that is commonly needed by other functions, and hence $f$ is likely to be inlined.

**Case 4:** The UD callee function $f$ in Fig. 7(d) is a variant of 7(c), where it has several references to library functions and *zero* reference to other UD functions. Such zero reference to UD functions makes $f$ an ideal candidate for inlining. Note that $f$ is inlined as the majority ($50\%$ or more) of its invoked library functions are not of termination type. Hence, we can safely assume that $f$ is doing much more than just facilitating program termination. Here, termination type refers to the library functions that lead to exceptions or program termination (e.g., `exit` and `abort`).

**Case 5:** In Fig. 7(e), The UD callee function $f$ is a variant of 7(d), which has only references to library functions. However, all the invoked library functions in $f$ are of termination type. Hence, we consider as a function that facilitates only program termination (or exception handling) and its semantics are of little interest to the caller, which should not be inlined.

**Case 6:** Fig 7(f) depicts the scenario of a *dispatcher function* where the UD function $f$ is called by (i.e., incoming calls) many other UD functions and $f$ itself calls many other UD and library functions. In this case, $f$ appears to be a dispatcher function without much unique semantics, and hence, in most cases, not inlined.

## 5.2 Inline Decision Algorithm

From the discussions above, in Fig. 7 (a), (b), (d) and (e) there is a clear criterion in deciding whether a callee should be inlined or not. However, for Fig. 7 (c) and (f), the most commonly observed invocation patterns, a systematic decision-making procedure is still needed. To identify the cases of commonly-used functions (i.e., utility functions) to be inlined, we borrow the *coupling* concept from the software quality and architecture recovery community. In software metrics, the coupling between two software packages is measured by the dependency between them, e.g., the software package instability metrics [36]. In this work, similarly, we measure the function coupling by the *function coupling score*.

**Definition** 4. ***Function Coupling Score*** refers to the complexity of the invocations involved in a function and is calculated as $\alpha = \lambda_e/(\lambda_e + \lambda_a)$, where $\lambda_a$ represents the number of UD functions that refers to the callee, and $\lambda_e$ represents the number of UD functions that is referred to by the callee.

**Algorithm 1:** Selective inlining algorithm

```
Data: caller F, set of callee functions C, set of termination lib. func. L_t, set
      of inlining lib. func. L_s
Result: inlined function F^I
1  Algorithm SelectiveInline(F, C, L_s, L_t)
2      foreach function f in C do
           // inline selected library functions
3          if f ∈ L_s then
4              F^I ⟵ F.inline(f)
5              return F^I
6          else if f ∉ L_s && isLibCall(f) then
7              return null
           // for all other user-defined callee functions
8          I_u^f ⟵ getIncomingCalls(f)
9          O_u^f, O_l^f ⟵ getOutgoingCalls(f)
10         O_u^f ⟵ O_u^f \ F                          // remove recursion
11         if |O_u^f| == 0 && (|O_l^f ∩ L_s| − |O_l^f ∩ L_t|) ⩽ 0 then
12             return null
13         else
14             α = λ_e/(λ_e + λ_a)  where λ_a = |I_u^f|, λ_e = |O_u^f|
               // lower the α, function f is likely to be inlined
15             if α > threshold t && notRecursive(F, f) then
16                 return null
17             else
18                 F^I ⟵ F.inline(f)
19                 if |O_u^f| > 0 then
20                     SelectiveInline(f, O_u^f, L_s, L_t)
21                 else
22                     return F^I
23     return F^I
```

The lower the value of $\alpha$, more likely the callee should be inlined. The rationale is that the low function coupling score implies the high independency of the functionality — the high possibility of being utility function. When calculating $\lambda_e$, we only consider the UD functions invoked by the callee, not the library functions. As mentioned in case 3 (Fig. 7(c)) and 6 (Fig. 7(f)), it is the invocations of UD functions that indicate the behavior of the callee as a dispatcher or a utility function, not the invocations of library functions.

The proposed selective inlining process is presented in Algorithm 1. As first, if the callee is one of the selected library functions (e.g., `memcpy` and `strlen` as in case 1), it is directly inlined into the caller (lines 3-5) and rest of the library functions are ignored. Next, the UD functions that refer to the callee is denoted as $I_u^f$; while the library and UD functions referred to by the callee are identified and denoted as $O_l^f$ and $O_u^f$, respectively at line 8-9. Callee that invokes only library functions that are of termination type is not inlined (lines 11-12). Finally, for the rest of the callee functions, the function coupling score is calculated and if it is below some threshold value $t$, the callee is inlined (lines 13-24). This recursive procedure is continued until all the related functions are analysed.

In our preliminary study on BusyBox compiled for x86 32bit, we identify that 14 UD utility functions (case 3) have more than 50 incoming calls with 1 outgoing call, while 12 UD dispatcher functions (case 6) have more than 50 outgoing calls with just 1 incoming call. Similarly, in BusyBox compiled for ARM, we identify such 15 UD utility functions but only 4 UD dispatcher functions. This clearly differentiates the function invocation patterns case 3 (Fig. 7(c)) and 6 (Fig. 7(f)). In this work, we adopt a lightweight static analysis to

make the inlining decision for the performance reason (as shown in Section 7). A more expensive program analysis can be used to improve the accuracy, but we strike for the balance between performance and accuracy in this work.

## 6 Scalable Function Matching

In BinGo and BinGo-E, similarity matching is done at the granularity of function and sub-function levels (i.e., we can match a part of a target function to the signature function).

### 6.1 Primary Matching

In primary matching, high-level semantic features and structural features are combined to measure the semantic similarity of the two code segments as the granularity of function. Let $\mathcal{SIM}_{\mathcal{H}}(sig, tar)$ denote the similarity score of using high-level semantic features and $\mathcal{SIM}_{\mathcal{S}}(sig, tar)$ denote the similarity score of using structural features. The former score is measured by Jaccard containment similarity [37], considering each function has a bag of high-level semantic features:

$$\mathcal{SIM}_{\mathcal{H}}(sig, tar) = \frac{\mathcal{H}_{sig} \bigcap \mathcal{H}_{tar}}{\mathcal{H}_{sig}} \tag{3}$$

Specifically, for each of six types of high-level semantic features, the Jaccard distance is calculated, and the overall $\mathcal{SIM}_{\mathcal{H}}(sig, tar)$ is the mean value of Jaccard distances of different types of high-level semantic features. Note that if both signature function and target function have no function calls, then features in terms of function call tags, function call sequence, function parameters are not available, and they are not counted in overall similarity.

The latter score is measured by calculating FDD (Definition 3 in §4.1), and the FDD value is already normalized into the range between 0 to 1. $\mathcal{SIM}_{\mathcal{S}}(sig, tar)$ is to convert the distance to the similarity:

$$\begin{aligned} \mathcal{SIM}_{\mathcal{S}}(sig, tar) &= Convert(FDD(sig, tar)) \\ &= 1 - FDD(sig, tar) \end{aligned} \tag{4}$$

The idea of convertion is that if these two functions' distance is closer to 0, their similarity is closer to 1. Vice versa, if their distance is as big as it could possibly be, their similarity is prone to be 0.

Once we have similarity scores in high-level semantics and structures, the primary matching result $\mathcal{SIM}'(sig, tar)$ is calculated as the weighted sum of $\mathcal{SIM}_{\mathcal{H}}(sig, tar)$ and $\mathcal{SIM}_{\mathcal{S}}(sig, tar)$, which is defined as follow:

$$\begin{aligned} \mathcal{SIM}'(sig, tar) = (1 - \mathcal{W}/2) * \mathcal{SIM}_{\mathcal{H}}(sig, tar) \\ +(\mathcal{W}/2) * \mathcal{SIM}_{\mathcal{S}}(sig, tar) \end{aligned} \tag{5}$$

where $\mathcal{W} = min(\frac{w'_{sig}}{w'_{tar}}, \frac{w'_{tar}}{w'_{sig}})$ and $\mathcal{W}$ means the ratio of these two functions' instruction number.

Hence, the idea of primary matching is straightforward — structural features are effective when the sizes of two functions are similar. If two functions have the similar size of instructions, structural features have the same weight as high-level semantic features. If not, high-level semantic features have more weight than structural features. According to this idea, a high value of $\mathcal{SIM}'(sig, tar)$ (e.g., $\geq 0.8$) indicates that $sig$ and $tar$ are similar in both structures and high-level semantics.

### 6.2 Low-level Semantic Features Matching

When the primary matching does not find any ranked results that are of high $\mathcal{SIM}'(sig, tar)$ values, low-level semantic features are used. Note that low-level semantics refer to not only the implementation details, but also the partial semantics of the function. To this end, we propose the function model consisting of partial traces of various lengths, which is more flexible in terms of comparison granularity, compared with the BB-centric function modelling [9], [4]. Then, from these partial traces we extract symbolic expressions. Based on I/O values for register, flags and memory addresses, we match the partial traces inside two functions. Last, we apply Jaccard containment similarity [37] to measure the similarity score of two function models.

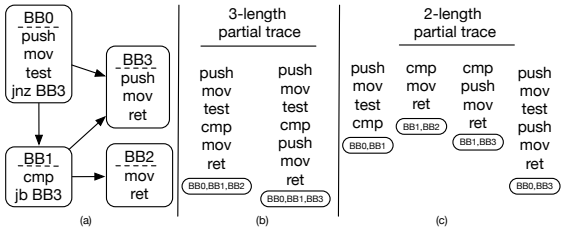



Fig. 8: A sample CFG (a) and the extracted 3-length (b) and 2-length (c) partial traces

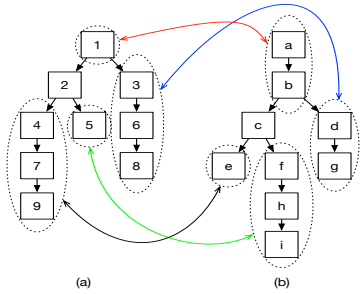



Fig. 9: A sample signature (a) and target (b) functions, where the lines indicate the matched partial traces and the nodes refer to the basic-blocks.

#### 6.2.1 K-length Partial Trace Extraction

Our partial trace extraction is based on the technique proposed by David *et. al.* [10]. We omit the algorithm and explain the results using one example. Fig. 8 depicts a sample CFG of a function and the extracted 2- and 3-length partial traces (i.e., for $k = 2, 3$). We can observe that the original control-flow instructions (`jnb`, and `jb`) are omitted as the flow of execution is already determined. Note that the feasibility of the flow of executions is not considered at this step.

#### 6.2.2 Function Model Generation

In BinGo, we leverage on length variant partial traces to model the functions. We generate partial traces with three different lengths (i.e., $k = 1, 2$ and 3) from each function, all of which collectively constitute the function model. For example, function models generated from the signature ($\mathcal{M}_{sig}$) and target ($\mathcal{M}_{tar}$) functions in Fig. 9 are listed as follows:

$\mathcal{M}_{\texttt{sig}} : \{\langle 1\rangle, \langle 2\rangle, \ldots, \langle 1, 2\rangle, \langle 2, 5\rangle, \ldots, \langle 1, 2, 4\rangle, \langle 2, 4, 7\rangle, \ldots\}$
$\mathcal{M}_{\texttt{tar}} : \{\langle \texttt{c}\rangle, \langle \texttt{b}\rangle, \ldots, \langle \texttt{a}, \texttt{b}\rangle, \langle \texttt{b}, \texttt{c}\rangle, \ldots, \langle \texttt{a}, \texttt{b}, \texttt{c}\rangle, \langle \texttt{b}, \texttt{c}, \texttt{f}\rangle, \ldots\}$

In BinGo and BinGo-E, we support the function model similarity matching in terms of *n-to-m*, *1-to-n*, *n-to-1* and *1-to-1* partial trace matching across the signature and target functions, mitigating the impact of program structural difference.

***n-to-m*** Partial traces of length $n(\in \mathbb{Z}_{>1})$ generated from the signature function are matched against the partial traces of length $m(\in \mathbb{Z}_{>1})$ generated from target function. In Fig. 9, matching partial traces $\langle 3, 6, 8\rangle$ and $\langle d, g\rangle$ is *3-to-2* matching.

***1-to-n*** A basic-block (i.e., a partial trace of length 1) in the signature function is matched against the partial traces of length $n(\in \mathbb{Z}_{>1})$ generated from target function. In Fig. 9, partial traces $\langle 1\rangle$ and $\langle 5\rangle$ are matched with $\langle a, b\rangle$ and $\langle f, h, i\rangle$, respectively.

***n-to-1*** Partial traces of length $n(\in \mathbb{Z}_{>1})$ generated from the signature function are matched against a basic-block in the target function. In Fig. 9, partial trace $\langle 4, 7, 9\rangle$ is matched with $\langle e\rangle$.

***1-to-1*** A basic-block in the signature function is matched against a basic-block in the target function. It is also known as pairwise comparison at the level of single basic-block. In Fig. 9, partial trace $\langle 2\rangle$ is matched with $\langle c\rangle$.

*1-to-n* matching addresses the issue of basic-block splitting — a single basic block in the signature function is split into several smaller basic blocks in the target function. Similarly, *n-to-1* matching addresses

the basic-block merging problem. In tracelet modeling [10], authors recommended that both the signature and target should be of the same size (i.e., $k = 3$), and hence only *n-to-n* matching is performed. In [9] and [4], pairwise comparison of single basic-block (i.e., *1-to-1* matching) is performed as an initial step to shortlist target functions. In contrast, in BINGO and BINGO-E, all the 4 types of function matching are needed to cover all possible BB-structure variances that arise due to architecture, OS and compiler differences.

For the similarity score of using low-level semantic features $\mathcal{SIM}_{\mathcal{L}}(sig, tar)$, considering the function model as a bag of partial traces, Jaccard containment similarity [37] is also used to measure the similarity of two different function models:

$$\mathcal{SIM}_{\mathcal{L}}(sig, tar) = \frac{\mathcal{M}_{sig} \bigcap \mathcal{M}_{tar}}{\mathcal{M}_{sig}} \tag{6}$$

where $\mathcal{M}_{sig}$ and $\mathcal{M}_{tar}$ refer to function models of low-level semantics that are generated from signature and target functions, respectively.

### 6.3 Final Matching

To remove false positives of using low-level semantics alone, $\mathcal{SIM}'(sig, tar)$ is also considered in the final matching.

Let $\mathcal{SIM}^*(sig, tar)$ denote the overall similarity score calculated in the final matching. $\mathcal{SIM}^*(sig, tar)$ is calculated as the weighted sum of $\mathcal{SIM}'(sig, tar)$ and $\mathcal{SIM}_{\mathcal{L}}(sig, tar)$. Here, we adopt two strategies for different matching scenarios.

If the matching is based on the same code base, we consider both $\mathcal{SIM}_{\mathcal{L}}(sig, tar)$ and $\mathcal{SIM}'(sig, tar)$ are equally important.

$$\begin{aligned}\mathcal{SIM}^*(sig, tar) = (1/2) * \mathcal{SIM}'(sig, tar) \\ +(1/2) * \mathcal{SIM}_{\mathcal{L}}(sig, tar)\end{aligned} \tag{7}$$

Otherwise, if the matching is based on different code bases (i.e., experiments on cross-OS matching), we ignore the structural features and give equal weight for low-level semantics and high-level semantics.

$$\begin{aligned}\mathcal{SIM}^*(sig, tar) = (1/2) * \mathcal{SIM}_{\mathcal{H}}(sig, tar) \\ +(1/2) * \mathcal{SIM}_{\mathcal{L}}(sig, tar)\end{aligned} \tag{8}$$

Still take the code segments in Fig. 2 as example. The two code segments are from different code bases, and hence the BB structure will not be similar. Low-level semantics and high-level semantics are used together to measure their semantic similarity. Previously, in BINGO, using low-level semantics alone suggests they have a similarly value of 0.7 — 70% of tracelets in the signature function match the target function on I/O value pairs. Now, in BINGO-E, since they have a low high-level similarity of 0.7 and no similarity in high-level features and structural features, their overall similarity is lower to 0.35. Hence, the false positive has a lower similarity score, and will be removed from the best-match or top-5-match list.

## 7 EXPERIMENTAL RESULTS

**Implementation.** In BINGO and BINGO-E, we use IDA Pro to disassemble and generate CFGs from the functions in binaries. Partial traces are generated using the TRACY plugin[4], where they are of three different lengths (i.e., $k$-tracelet with $k = 1, 2$ and 3). In BINGO, to facilitate analysis and feature extraction, similar to the preprocessing in [9] and [4], we lift the assembly instructions to an architecture and OS independent intermediate language, REIL [38]. In BINGO-E, no intermediate language is necessary as constraint solving based on REIL is not needed. All the experiments were conducted on a machine with Intel Core i7-4702MQ@2.2GHz with 32GB DDR3-RAM.

**Configuration.** For cross-architecture matching, experiments are conducted on the different executables (i.e., x86-32, x86-64, ARM version) of `coreutils`. For cross-compiler matching, experiments are conducted on the executables that are compiled from `coreutils` with the different combinations of adopted compilers (`clang` and `gcc`) and optimization levels (O0-O3). For cross-OS matching, we

4. https://github.com/tech-srl/tracy

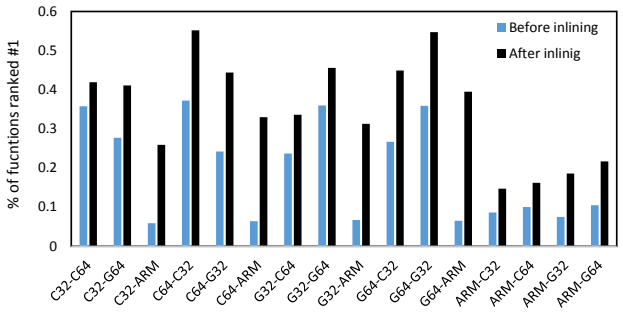



Fig. 10: In BINGO, cross-architecture function matching before/after selective inlining using `coreutils` binaries. Here, **C** and **G** represent compilers `clang` and `gcc`, respectively, and 32 and 64 denote x86 32bit and 64bit architectures.

are matching between functions between `msvcrt` (for Windows) and `libc` (for Linux), which are compiled from different code bases with `gcc` for x86-32. For these experiments, we previously conduct them to evaluate BINGO; we also conduct them to evaluate how BINGO-E improves. To evaluate the real-world application of BINGO, we use the code of open source project vulnerabilities as signature to search for the similar vulnerability in commercial products (e.g., Adobe PDF Reader). For BINGO-E, we search for similar functions in binaries of forked projects, and compare the results with COP [4].

### 7.1 Research Question

Through our experiments, compared with the results of BINGO, we aim at answering these four research questions (RQs):

**RQ1.** How robust is BINGO-E in detecting *semantically equivalent* functions *across architectures* with code optimization?
**RQ2.** How robust is BINGO-E in detecting *semantically equivalent* functions *across compilers* with code optimization?
**RQ3.** How robust is BINGO-E in identifying *semantically similar* functions, across OS, in *wild* binary executables?
**RQ4.** What are the key real-world applications of BINGO-E?
**RQ5.** How scalable is BINGO-E?

These RQs fall into three major topics, namely *robustness* (RQ1-3), *application* (RQ4) and *scalability* (RQ5). The robustness of BINGO-E lies in two aspects: (1) matching *semantically equivalent* functions, and (2) matching *semantically similar* functions. RQ1-2 are about the first aspect as we aim to match all functions in binary $A$ to all functions in binary $B$, where the same source code (i.e., syntactically identical at the source code level) is compiled for different architectures using different compilers and code optimization options. RQ3 focuses more on finding semantically similar functions across binaries that are not compiled from the same source code (i.e., syntactically different even at the source-code level). In RQ1-2, we evaluate the selective inlining algorithm to show its effectiveness. In RQ3, we compare BINGO-E with BINGO, BINDIFF [24] and TRACY [10], indirectly showing the merits of incorporating different categories of features. In RQ4, we evaluate BINGO in finding real-world vulnerabilities and compare it against DISCOVRE [21] and [9], and evaluate BINGO-E in matching binary functions of forked open-source projects. Finally, in RQ5, we demonstrate the performance of BINGO and BINGO-E.

### 7.2 Answer to RQ1: Robustness in Cross-architecture Matching

#### 7.2.1 BINGO's Answer to RQ1

We conduct two experiments. First experiment is to match all functions in `coreutils` binaries compiled for one architecture (i.e., x86 32bit, x86 64bit and ARM) to the semantically equivalent functions in binaries compiled for another architecture. For example, binaries compiled for ARM are matched against the binaries compiled for x86 32bit and x86 64bit architectures, and *vice versa*. In all three architectures, the

TABLE 3: Total no. of functions selectively inlined in `coreutils` binaries compiled using `gcc` and `clang` for ARM, x86 32bit and 64bit architectures.

| | ARM | C64 | G64 | C32 | G32 |
|---|---|---|---|---|---|
| Total functions | 27,820 | 23,782 | 24,489 | 24,002 | 24,792 |
| Inlined functions | 1,620 | 776 | 1,002 | 823 | 1,188 |

compilation uses `gcc` (v4.8.2) and `clang` (v3.0) with the optimization level `O2` (default settings in many Linux distributions).

Fig. 10 summarizes the average results (before and after selective inlining) obtained for `coreutils` binaries, where we report the percentage of functions ranked one (i.e., best match). In the plot, the bar `G64-G32` (after inlining) should be read as, when functions compiled, using `gcc`, for x86 64bit architecture are used as signature, around 55% of the functions compiled, using `gcc`, for x86 32bit architecture achieve rank one. It is observed that when ARM binary functions are used as signatures, the overall results significantly degrade — only 18% of the functions are ranked one, compare to x86 32bit and 64bit architectures, where it is around 41%. We also notice that matching(after inlining) between binaries compiled for ARM and x86 64bit yields higher ranking compared to ARM and x86 32bit binaries. The rationale is that ARM and x86 64bit binaries are register intensive compared to x86 32bit binaries — in ARM and x86 64bit architectures, there are 16 general-purpose registers, and hence registers are used more frequently than used in x86 32bit binaries which have only 8.

Considering the fact that `coreutils` binaries are relatively small with around 250 functions (on average) per binary, we conduct the second experiment using `BusyBox` (v1.21.1), which contains 2,410 functions. We observe that the obtained results are comparable with the results of `coreutils` binaries — 41.3% of the functions are best-match in `BusyBox`, while it is 35% for `coreutils` binaries on average across 16 experiments shown in Fig. 10. This is an improvement of 27.5% over the accuracy achieved by [9] for `BusyBox`. However, for top-10 match (i.e., ranking 1-10), `coreutils` binaries achieve better results than `BusyBox` (78.4% vs. 67%). This result might be due to function size of the binary — in `BusyBox`, a function needs to be compared against on average 3,250 functions, and in `coreutils` binaries around 250 functions need to be compared.

**Impact of selective inlining.** As shown in Fig. 10, our selective inlining significantly improves the overall matching accuracy by 150% on average. In particular, when ARM architecture is used as target, selective inlining improves the total matching accuracy on average by about 400%. Table 3 summarizes the total number of functions inlined for each architecture. We notice that `coreutils` compiled for ARM contains more functions (27,820 across 103 binaries) than that compiled for x86 32bit (avg. 24,397) and x86 64bit (avg. 24,136). From the second row of Table 3, it can be seen that 61% and 82% more functions are selectively inlined by BINGO in ARM than that in x86 32bit and 64bit architectures, respectively. In ARM binaries, the larger number of compiled functions suggests that compiler inlining does not happen as frequently as it does in the other two architectures. Hence, selective inlining is required to capture the complete semantics in binary analysis, which results in the improved matching accuracy.

**Summary.** On `coreutils` and `BusyBox` binaries, BINGO achieves the best match (i.e., rank #1) for 35%-42% of the functions on average, for cross-architecture analysis, and selective inlining improves the result by 150% on average.

### 7.2.2 BINGO-E's Answer to RQ1

For BINGO-E, we also match the binary functions in `coreutils` binaries compiled for one architectures (i.e., x86 32bit, x86 64bit and ARM) to that for another. We use the two compilers `gcc` (v4.8.2) and `clang` (v3.0) with the optimization level `O2` (default settings in many Linux distributions).

Fig. 11 shows the average results after selective inlining on `coreutils` binaries for BINGO and BINGO-E. Note we only count the percentage of functions ranked 1st (i.e., best match in the case of mapping the binary functions from the same source code). In Fig. 11,

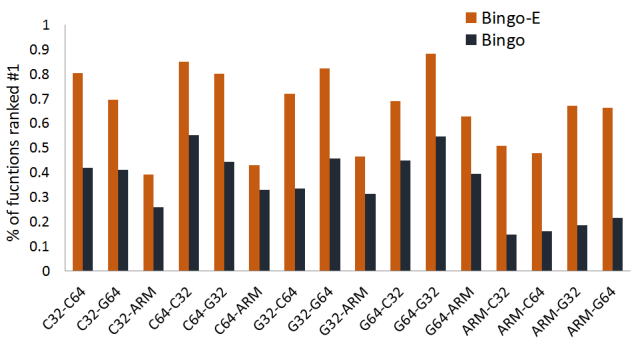



Fig. 11: Cross-architecture function matching after selective inlining using `coreutils` binaries for BINGO and BINGO-E. Here, **C** and **G** represent compilers `clang` and `gcc`, respectively, and 32 and 64 denote x86 32bit and 64bit architectures.

we list 16 possible scenarios of cross-architecture code matching. For example, in the plot, the brown bar `G64-ARM` denotes the result that 62.7% of the functions (signatures) compiled by `gcc` for x86-64 are the best matches of those functions (targets) compiled by `gcc` for ARM architecture according to BINGO-E.

Previously, we observed that in Fig. 10 that when binary functions compiled for ARM are used as signatures, the matching results significantly degrade — only 18% of the functions are ranked 1st, based on the low-level semantic features in BINGO. The reason is that I/O values of registers in BINGO are not effective for cross-architecture matching. Hence, in BINGO-E, the newly added high-level semantic and structural features are complement to improve the matching accuracy. The results in Fig. 11 prove the effectiveness of these new features. For most cases, the new features improve the matching accuracy by at least 50%, compared to the results of BINGO. Especially, in the cases of code for ARM as signatures (`ARM-C32`, `ARM-C64`, `ARM-G32`, `ARM-G64`), the improvement of best match rate is above 200%. This indicates the ineffectiveness of low-level semantic features used by BINGO for matching the code for ARM to the code for other architectures.

Currently, for BINGO-E, we get more similar results for many cases — the best match rate is within the range of $70\% \sim 90\%$ for the mapping from x86-32 to x86-64 (or the other way around), while the best match rate drops to $40\% \sim 60\%$ if the ARM code is used as signatures or targets. The rationale is that the BB structure of code for x86-32 or x86-64 is highly similar; even in code for ARM, the basic BB structure is also retained during the cross-architecture compilation. Hence, the structural features and high-level semantic features are the effective complements, since the CFG structures and function call information are retained during the compilation (no matter we use `gcc` or `clang`). We find that the high-level semantic features, if they are available, are even more helpful than structural features. The reason is that after selective inlining the information on function call in binary code does not change with the compilation options of architecture.

Another interesting observation is that the results for BINGO-E are much more symmetric than the results of BINGO. For example, the best match rate for BINGO-E in `C64-ARM` and `ARM-C64` is 0.431 and 0.479, respectively. Hence, reversing signatures and targets does not affect as much as that in BINGO. In contrast, for BINGO in `C64-ARM` and `ARM-C64` it is 0.33 and 0.162, respectively. The unsymmetrical results for BINGO are due to the step of function model generation, in which $k$-tracelet is sampled and generated from the CFG. As applying graph isomorphism for CFG matching, the matching of CFG $g_1$ to $g_2$ does not mean the matching of $g_2$ to $g_1$ — $g_1$ may be a sub-graph of $g_2$ and using $g_2$ as signatures leads to a similarity score lower than the similarity threshold. Similarly, in BINGO, low level semantic features extracted from $k$-tracelet (i.e., sampled partial CFG) also have this issue of unsymmetrical results.

To mitigate this, the function similarity (i.e., FDD in § 4.1) for 3D-CFG is defined to produce symmetrical results, according to Definition 3. In addition, the high-level semantic features (e.g., function call information) are retained in cross-architecture compilation. Hence, the results of BINGO-E are more symmetrical than that of BINGO.

In this experiment, on average, our approach significantly improves the accuracy of best match rate from 35.1% (for BINGO) to 65.6% (for BINGO-E) on `coreutils`. For cross-architecture matching on the same code base, structural features and high-level semantic features are effective, being good complement to low-level semantic features. Further, BINGO-E's results are more symmetrical than BINGO's.

## 7.3 Answer to RQ2: Robustness in Cross-compiler Matching

### 7.3.1 BINGO's Answer to RQ2

**Inter-compiler Matching.** Table 4 summarizes the results obtained by BINGO for different compilers (`gcc` and `clang`) with various optimization levels (O0, O1, O2 and O3) for x86 32bit architecture. Here, each row and column of the table represents the compiler (including the level of optimization) used to compile the signature and target functions, respectively. For example, the cell denoted by the second row and fifth column (`C1G0`) represents the result for which signature functions are compiled using `clang` with optimization level `O1` and the target functions are compiled using `gcc` with `O0`. From the table, we observe that BINGO is good, as 41.5% of the functions achieve rank one (on average across all the experiments for binaries compiled for x86 32bit) while this increases to 84% for top 10 matches, which is very promising. We observed similar behaviour in binaries compiled for x86 64bit architecture and due to space constraints, the results are reported in our tool webpage [39].

Interestingly, we find that regardless of the compiler type, matches between the binaries that are compiled with high code optimization levels (i.e., `O2` and `O3`) yield better results than matches between binaries with no code optimization (i.e.,`O0`). That is, high code optimization levels lead to the similar binary code even with different compilers, whereas without optimization the influence of the compiler in the binary is very evident. This observation is consistent with [9]. Further, within one compiler type (e.g., `gcc`), the matching accuracy achieved by optimization level `O2`, across all other compiler types and optimization levels, is always better or consistent with the results obtained by `O3` (relevant rows are shaded in light gray in Table 4). This behaviour is observed in x86 64bit architecture too.

**Intra-compiler Matching.** In addition, we also conduct the experiments across different versions of the same compiler (`gcc` v4.8.3 vs. `gcc` v4.6) and the results (for x86 32bit architecture) are summarized in Table 5. As expected, the highest accuracy is achieved when the signature and target functions are of the same optimization level, except when the signature functions are compiled with optimization level `O1`, where on such occasions, target functions compiled with optimization level `O2` yield better accuracy. The best results obtained for each signature binary is highlighted in light gray in Table 5. Further, we also observe that the overall accuracy for intra-compiler analysis (42.7%) is slightly better than the inter-compiler analysis (41.5%). Finally, in intra-compiler analysis, around 76.3% functions ranked within top 5 positions, whereas it is only 68.5% for inter-compiler analysis, an 11.8% improvement. Details on the results of intra-compiler analysis for x86 64bit architecture are reported in [39].

**Impact of selective inlining.** We observe that selective inlining improves the average matching accuracy by around 140% for cross-compiler analysis. Many functions are inlined when the binaries are compiled with the optimization level `O0` compared to `O3`. In particular, 1,473 functions (on average across `gcc` and `clang`) are inlined in for optimization level `O0`, whereas, it is only 831 for `O3`.

**Summary.** For different types and versions of compiler, in most cases, BINGO attains the best match for around 42% of the functions, and top-5 match for around 72% of the functions on average.

TABLE 4: In BINGO, percentage of functions ranked #1 in inter-compiler (x86 32bit) comparison for `coreutils`. Here, **C** and **G** represent `clang` and `gcc` compilers, respectively and 0-3 represents optimization O0-O3.

| | **C0** | **C1** | **C2** | **C3** | **G0** | **G1** | **G2** | **G3** |
|---|---|---|---|---|---|---|---|---|
| **C0** | - | 0.402 | 0.373 | 0.372 | 0.371 | 0.4 | 0.451 | 0.332 |
| **C1** | 0.392 | - | 0.456 | 0.455 | 0.396 | 0.429 | 0.503 | 0.388 |
| **C2** | 0.345 | 0.426 | - | 0.576 | 0.344 | 0.411 | 0.488 | 0.478 |
| **C3** | 0.344 | 0.425 | 0.575 | - | 0.343 | 0.41 | 0.488 | 0.477 |
| **G0** | 0.344 | 0.354 | 0.334 | 0.333 | - | 0.374 | 0.398 | 0.302 |
| **G1** | 0.37 | 0.409 | 0.41 | 0.409 | 0.376 | - | 0.517 | 0.395 |
| **G2** | 0.412 | 0.462 | 0.474 | 0.473 | 0.428 | 0.514 | - | 0.48 |
| **G3** | 0.326 | 0.372 | 0.438 | 0.439 | 0.331 | 0.42 | 0.5 | - |

TABLE 5: In BINGO, percentage of functions ranked #1 in intra-compiler (x86 32bit) comparison for `coreutils`. Here, **G** and **G'** represent `gcc` v4.8.3 and v4.6 compilers, respectively.

| | G0 | G1 | G2 | G3 | | G'0 | G'1 | G'2 | G'3 |
|---|---|---|---|---|---|---|---|---|---|
| G'0 | 0.43 | 0.36 | 0.36 | 0.27 | G0 | 0.44 | 0.4 | 0.41 | 0.42 |
| G'1 | 0.37 | 0.46 | 0.48 | 0.36 | G1 | 0.38 | 0.47 | 0.48 | 0.45 |
| G'2 | 0.4 | 0.44 | 0.52 | 0.41 | G2 | 0.4 | 0.49 | 0.52 | 0.5 |
| G'3 | 0.4 | 0.44 | 0.49 | 0.49 | G3 | 0.32 | 0.39 | 0.43 | 0.52 |

### 7.3.2 BINGO-E's answer to RQ2

**Inter-compiler Matching.** Still using `coreutils` binaries, Table 6 summarizes the results obtained by BINGO-E for different compilers (gcc and clang) with various optimization levels (O0, O1, O2 and O3) on `coreutils` for x86 32bit architecture. The row represents the optimization level of signature functions and the column represents that of target functions. We observe that the best results are usually achieved when O1 or O2 is adopted. The worst results (below 80%) are achieved when O3 is used, no matter for `gcc` or `clang`. Even C2-to-G1 or G2-to-C1 can achieve an accuracy above 88%. Hence, for cross-compiler matching, we find the impact of compiler differences matters less than that of the optimization levels.

In this experiment, BINGO-E (best-match rate ranging from 70.1% to 99.7% in Table 6) exhibits significant improvement over BINGO (best-match rate ranging from 32% to 58% in Table 4) for cross-compiler analysis. The improvement is attributed to the use of features of BB structure and high-level semantics, as BB structure can be retained for different compilers under the same or similar optimization levels.

**Intra-compiler Matching.** We also conduct intra-compiler matching for BINGO-E. Table 7 summarizes the searching results obtained by BINGO-E for differen versions of the compiler `gcc` with various optimization levels (O0, O1, O2 and O3) for x86 32bit architecture. Here, each row and column of the table represents the compiler (including the level of optimization) used to compile the signature and target functions, respectively. For example, for the cell at row G'0 and column G0 in the left part of Table 7, the value 0.91 means that 91% of functions successfully match if we use the version compiled by G'0 as signature and use the function compiled by G0 as target. It can be observed that usually the best-match results are achieved when the signature and target functions are combined by the same optimization level (O0-O2, except O3). The reasons lie that BINGO-E adopts the structural and high-level semantic features for intra and cross compiler matching. When the same optimization level (except O3) is used for signature and target functions, the structural information is well-retained. However, when O3 is used, the structure of the binary is so over-optimized that less information is retained for structural matching. The two over-optimized program may have less structural similarity in G'3-to-G3 or G3-to-G'3 matching.

TABLE 6: In BINGO-E, percentage of functions ranked #1 in inter-compiler (x86 32bit) comparison for `coreutils`. Here, **C** and **G** represent `clang` and `gcc` compilers, respectively and 0-3 represents optimization O0-O3.

| | **C0** | **C1** | **C2** | **C3** | **G0** | **G1** | **G2** | **G3** |
|---|---|---|---|---|---|---|---|---|
| **C0** | - | 0.865 | 0.791 | 0.785 | 0.799 | 0.840 | 0.836 | 0.701 |
| **C1** | 0.840 | - | 0.838 | 0.833 | 0.891 | 0.913 | 0.887 | 0.756 |
| **C2** | 0.814 | 0.889 | - | 0.996 | 0.862 | 0.926 | 0.858 | 0.757 |
| **C3** | 0.809 | 0.885 | 0.997 | - | 0.857 | 0.924 | 0.855 | 0.759 |
| **G0** | 0.801 | 0.888 | 0.853 | 0.846 | - | 0.884 | 0.856 | 0.754 |
| **G1** | 0.786 | 0.878 | 0.857 | 0.852 | 0.850 | - | 0.928 | 0.857 |
| **G2** | 0.806 | 0.882 | 0.814 | 0.808 | 0.848 | 0.963 | - | 0.732 |
| **G3** | 0.732 | 0.773 | 0.754 | 0.753 | 0.770 | 0.863 | 0.751 | - |

TABLE 7: In BINGO-E, percentage of functions ranked #1 in intra-compiler (x86 32bit) comparison for `coreutils`. Here, **G** and **G'** represent `gcc` v4.8.3 and v4.6 compilers, respectively.

| | G0 | G1 | G2 | G3 | | G'0 | G'1 | G'2 | G'3 |
|---|---|---|---|---|---|---|---|---|---|
| G'0 | 0.91 | 0.86 | 0.80 | 0.70 | G0 | 0.98 | 0.88 | 0.84 | 0.76 |
| G'1 | 0.86 | 0.97 | 0.88 | 0.82 | G1 | 0.84 | 0.94 | 0.90 | 0.84 |
| G'2 | 0.85 | 0.94 | 0.97 | 0.76 | G2 | 0.80 | 0.92 | 0.97 | 0.81 |
| G'3 | 0.79 | 0.88 | 0.89 | 0.75 | G3 | 0.72 | 0.83 | 0.78 | 0.74 |

In the experiment, the results of BINGO-E (best-match rate ranging from 70% to 98% in Table 7) are significantly better than the results of BINGO (best-match rate ranging from 27% to 52% in Table 5). For the same code base with intra-compiler matching, the structural and high-level semantic features are more effective than the low-level semantic features, relying on the retained structural information.

## 7.4 Answer to RQ3: Robustness in Cross-platform (OS) Matching

### 7.4.1 BINGO's Answer to RQ3

To evaluate BINGO as a search engine for binary programs (i.e., matching against wild binaries that do not share the same code), we conduct an experiment with open-source and close-source binaries. In this experiment, we choose Windows `mscvrt` as the binary (closed-source) from which the search queries are obtained, and Linux `libc` as the target binary (open-source) to be searched on.

**Building Ground Truth.** In conducting this experiment, we face a challenge in obtaining the ground truth. Ideally, for each function in `msvcrt`, we want to identify the corresponding semantically similar function in `libc`, so that we can faithfully evaluate BINGO. To reduce the bias in obtaining the ground truth through manual analysis, we rely on the function names and the function description provided in the official documentation, e.g., `strcpy` function in `msvcrt` is matched with `strcpy` (or its variants) from `libc`. Besides, several variants of the same functions are grouped into one family. In `libc`, there are 15 variants of `memcpy` functions are present (e.g., `memcpy_ia32`, `memcpy_ssse3`, and so on). Hence, if `memcpy` from `mscvrt` is matched to any of these 15 variants from `libc`, we still consider the match is correct.

**BINGO's Results.** BINGO reports a total of 60 matched cases, among which 50 matched cases are true positive, according to the built ground truth between `msvcrt` and `libc`. Table 8 summarizes the details of the 50 matched cases: BINGO finds the best match for 11 functions (22%), top 2-5 match for 26 functions (52%), and top 6-10 match for 13 functions (26%). Since binary code matching on different code bases is much more challenging than matching on the same code base, such results have demonstrated the robustness of BINGO in identifying functions that have similar semantics but different implementations.

**Summary.** BINGO outperforms two available state-of-the-art tools BINDIFF and TRACY on finding the semantically similar functions in cross-OS analysis. It attains an accuracy of 51.7% and 83.3% for rankings within top 5 and 10 positions, respectively.

### 7.4.2 BINGO-E's Answer to RQ3

To evaluate the capability of BINGO-E on matching binaries from different code bases, we repeat this experiment for cross-OS matching. In this experiment, functions from Window binary `mscvrt` are used as signature functions, while functions from Linux binary `libc` are as target functions to be matched.

**BINGO-E's Results.** To compare with BINGO, we also analyze the same number (i.e., 60) of matched cases, which have higher similarity scores than others among results reported by BINGO-E. After examining the matching results, 56 functions are found to be true positive cases, in which one function in `mscvrt` correctly finds its counterpart in `libc` in the top-10 matching. More specifically, BINGO-E finds the best match for 32 functions (51.7%), top 2-5 match for 12 functions (21.4%), and top 6-10 match for 12 functions (21.4%).

After inspecting the results, we report the possible false positive and false negative cases. For false positive cases, some of the functions are too simple so that accidental clones are detected. For example, if both two functions have only one BB and no function calls, and are similar in the beginning checksum. Actually they have different logics after checksum, somehow, they are indistinguishable by all categories of features. For false negative cases, the `mbstowcs` function in `msvcrt` calls 6 other functions within 3 BBs. But the same function in `libc` only contains 1 BB with 2 function calls. They have similar semantics but low similarity in function calls, BB structures; emulation somehow shows different results for the two functions. In this case, BINGO-E does match the two functions as it was supposed. Thus, such semantic binary code clones are still difficult to detect.

**Comparison with the State-of-the-art Tools.** To compare BINGO-E with the state-of-the-art tools, we repeat the aforementioned experiment using the industry standard binary comparison tool BINDIFF[5] and the publicly available academic tool TRACY [10]. BINDIFF (v4.1.0), somehow, is unable to correctly match any of the functions in Table 8 and 9 in `mcvcrt` to their counterparts in `libc`. Its failure roots back to heavy reliance on the program structure and call-graph pattern, which is less likely to be preserved in binaries compiled from completely different source-code bases. For TRACY, only 27 functions are found to have the correct match within the top 50 positions. Among the 27 matches, 5 (8.3%) are ranked one (best-match), 23 (38.3%) are ranked within top-5 match. Notably, when matching binaries compiled for different architectures, TRACY totally fails. Due to the fact that BLEX [20] is not publicly available[6], we cannot compare BINGO-E with dynamic analysis based function matching.

In this experiment, BINGO-E has a significant improvement on BINGO for the cross-OS analysis. In 60 analyzed cases, the false positive rate drops from 16.7% (for BINGO) to 6.7% (for BINGO-E). Among true positive cases, the rate of best-match increases from 22% (for BINGO) to 51.7% (for BINGO-E). The improvement is attributed to the adoption of high-level semantic features (e.g., system-call tags), which eliminate false positive cases due to using low-level semantic features alone.

## 7.5 Answer to RQ4: Applications

### 7.5.1 BINGO's Answer to RQ4: Vulnerability Detection via Matching Open-source Software Bug to Commercial Software

As part of the on-going research, we leveraged on BINGO to perform vulnerability extrapolation [40] — given a known vulnerable code, called *vulnerability signature*, using BINGO, we try to find semantically similar vulnerable code segments in the target binary. This line of work is receiving more attention from the academic research community [40], [9]. However, there is few evidence of using such technique to hunt real-world vulnerabilities. One reason could be these tools cannot handle

5. BinDiff: www.zynamics.com/bindiff.html

6. We also tried to ask for the dataset that is used to evaluate BLEX as a search engine, but we did not get any response from the authors.

TABLE 8: BINGO's returned rankings (top-10 matches) for 60 matched cases in `msvcrt` against `libc`

| Rank | Library functions | # |
|---|---|---|
| 1 | tolower, memset, wcschr, wcsncmp, toupper, memcmp, strspn, wcsrchr, memchr, strrch,r strchr | 11 |
| 2-3 | wcstoul, wcstol, strtoul, fopen, strncpy, strtol, itoa, wcscmp, wcsncat, itow, longjmp, strcspn, wcsncpy, labs, strpbrk, toupper, write, memcpy, memmove, tolower | 20 |
| 4-5 | mbtowc, wcstombs, strcat, remove, mbstowcs, wctomb | 6 |
| 6-10 | rename, strstr, wcspbrk, iswctype, strtok, wcscoll, strcoll, setlocale, qsort, wcsspn, swprintf, wcstod, strerrorr | 13 |

TABLE 9: BINGO-E's returned rankings (top-10 matches) for 60 matched cases in `msvcrt` against `libc`

| Rank | Library functions | # |
|---|---|---|
| 1 | wcscmp, sin, vswprintf, strstr, memcpy, perror, memmove, freopen, strcmp, wcschr, strcat, asin, fsetpos, strcspn, cos, strcpy, acos, qsort, wcslen, wcsncat, memcmp, wcscat, malloc, swprintf, strncat, isleadbyte, strlen, fseek, wcsrchr, longjmp, strpbrk, wcsncmp | 32 |
| 2-5 | getc, wcspbrk, fscanf, fgetwc, strncmp, wcsspn, wcsncpy, wcscspn, swscanf, strspn, tan, calloc | 12 |
| 6-10 | memchr, fgets, strchr, exit, strrchr, ungetwc, isprint, fopen, getchar, strerror, sscanf, wcsstr | 12 |

large complex binaries. Due to program structure agnostic function modeling, BINGO is capable of handling large binaries. Hence, we evaluate the practicality of BINGO in hunting real-world vulnerabilities.

**Zero-day Vulnerability (CVE-2016-0933) Found:** In vulnerability extrapolation, we discovered a zero-day vulnerability in one of 3D libraries used in Adobe PDF Reader. At the high level, we discover a network exploitable heap memory corruption vulnerability in an unspecified component of the latest version of Adobe PDF Reader. The root cause for this vulnerability is the lack of buffer size validation, which subsequently allows an unauthenticated attacker to execute arbitrary code with a low access complexity. We use a previously known vulnerability in an input size handling code segment of the same 3D module as the signature, where the signature function consists of more than 100 basic blocks. We model the known vulnerable function and all other 'unknown' functions in the library using BINGO, then use the known vulnerable function model as a signature, and search for semantically similar functions. BINGO returns a 'potential' vulnerable function (ranked #1) in the suspected 3D library. With some additional manual effort, we are able to confirm this vulnerability. Later, Adobe confirms and subsequently releases a patch for it[7].

**Matching Known Vulnerabilities:** To evaluate whether BINGO is capable of finding vulnerabilities across-platform, we repeat the two experiments reported in [9]. First one is *libpurple* vulnerability (CVE-2013-6484), where this vulnerability appears in one of the Windows application (`Pidgin`) and its counterpart in Mac OS X (`Adium`). In [9], it is reported that without manually crafting the vulnerability signature, matching from Windows to Mac OS X and *vice versa*, achieved the ranks #165 and #33, respectively. In BINGO, we achieve rank #1 for both cases.

The second experiment is SSL/Heartbleed bug (CVE-2014-0160), we compile the `openssl` library for Windows and Linux using `Mingw` and `gcc`, respectively (vulnerable code is shown in Fig. 1). The vulnerability is matched from Windows to Linux, using basic-block centric matching, similar to [9], we achieve the ranks 22 and 24 for `dtls1_process_heartbeat` and `tls1_process_heartbeat` functions, receptively. For Linux to Windows matching, we achieve rank #4 for both functions. This is due to the fact that `Mingw` significantly modifies the program structure through inlining, and hence the BB-structure is not preserved across binaries. Binary compiled by `Mingw` contains 24,923 more basic blocks compared to the Linux binary. This observation confirms that the BB centric matching fails in such conditions, where the program structure is heavily distorted. However, using function model of $k$-tracelet together with selective inlining, we are able to achieve rank #1 for both functions in Windows to Linux matching and *vice versa* with the average semantic similarity score of 0.62 and 0.59, respectively. DISCOVRE [21] fails to identify any of these vulnerable functions, as we observe that in `Mingw` version of `openssl`, library functions (e.g., `memset`, `memcpy`, and `strtol`) are inlined in most cases, leading to program structure distortion that is not handled by DISCOVRE.

7. We received 4,000USD as the bug bounty for this vulnerability **CVE-2016-0933** under *iDefense Vulnerability Contributor Program* run by *VeriSign*.

#### 7.5.2 BINGO-E's Answer to RQ4: New Application of Matching Binaries of Forked Projects

We also apply BINGO-E to detect the code plagiarism for the two projects that may share the same or similar code base. In this experiment, we adopt the same target projects as those used in CoP [4], namely `thttpd-2.25b` and `sthttpd-2.26.4`, where `sthttpd` is forked from `thttpd` for maintenance.

Table 10 shows the results of using `thttpd` as signature functions and `sthttpd` as target functions, while Table 11 shows the results of the matching in the other way around. In Table 10 and 11, the row refers to the compiler (including the level of optimization) used to compile the signature functions, and the column refers to that used to compile the target functions. We observe these facts:

1) The results of these two tables are quite consistent. The reason is twofold: a) functions in `thttpd` and `sthttpd` are quite cloned and most of functions are exactly identical; b) BINGO-E's results are symmetrical as mentioned in §7.2.2, and hence the results are not significantly different when reversing the signature and target functions in matching.
2) With the same optimization level, functions of these two projects are perfectly matched without errors. This perfect result is attributed to the same BB structure in binary due to the same optimization level. However, COP [4], [5] also only reports about 90% accuracy for the matching between `thttpd-2.25b` and `sthttpd-2.26.4` with the same optimization level. As COP relies on low-level semantics (i.e., I/O values), and BINGO-E use low-level/high-level semantic and structural features, all the features are proven to be effective in matching the binary code with the same BB-structure.
3) In general, except using the same optimization level, the matchings between `gcc` G0-G2 produce the best results (the results in **Bold** font), i.e., averagely 93.8% in Table 10 and 93.4% in Table 11. For the same configuration, COP achieves about 85% for the G0-G2 or G2-G0 matching among `thttpd-2.25b` and `sthttpd-2.26.4`. Note that no results are reported for COP on matching binary code compiled by `gcc` G3 in [4], [5].
4) C2-to-C3 or C3-to-C2 in Table 10 and 11 all archive the accuracy of 100%. This indicates the impacts of `clang` C2 and `clang` C3 are similar. However, the impact of `gcc` G3 is different from that of `gcc` G0-G2 — `gcc` G3 usually leads to the accuracies below 80%. The rationale is that G3 would greatly change the BB-structure due to the heavy usage of compiler optimization.

In this experiment, BINGO-E demonstrates the good performance in matching binary functions of two forked projects. Especially for the matchings between gcc G0-G2, BINGO-E achieves the accuracy of 93.6% on average from `thttpd` to `sthttpd` or in the other way around. Under the same configurations, it was reported that COP achieves the accuracy of 88% on average in [4], [5].

### 7.6 Answer to RQ5: Scalability

#### 7.6.1 BINGO's Answer to RQ5

BINGO has demonstrated its scalability through out the experiments. It particular filtering process takes only few milliseconds for `coretutils` binaries to shortlist the candidate target functions. For example, in cross-architecture analysis, it takes 91.8 milliseconds on average to compare entire `coreutils` suite (in total, 103 binaries each containing on average 250 functions) compiled for one architecture against another, while it is reduced to 68.6 milliseconds

TABLE 10: In BINGO-E, percentage of functions ranked #1 in inter-compiler (x86 32bit) comparison of `thttpd` to `sthttpd`. Here, **C** and **G** represent `clang` and `gcc` compilers, respectively and 0-3 represents optimization O0-O3. **Bold** font highlights results of the configuration `gcc` O0-O2, the same as that in COP [4].

| | C0 | C1 | C2 | C3 | G0 | G1 | G2 | G3 |
|---|---|---|---|---|---|---|---|---|
| **C0** | 1.0 | 0.820 | 0.761 | 0.757 | 0.922 | 0.830 | 0.864 | 0.711 |
| **C1** | 0.781 | 1.0 | 0.787 | 0.770 | 0.860 | 0.884 | 0.901 | 0.646 |
| **C2** | 0.676 | 0.760 | 1.0 | 1.0 | 0.761 | 0.824 | 0.829 | 0.789 |
| **C3** | 0.657 | 0.757 | 1.0 | 1.0 | 0.757 | 0.822 | 0.827 | 0.814 |
| **G0** | 0.961 | 0.883 | 0.803 | 0.800 | **1.0** | **0.895** | **0.876** | 0.737 |
| **G1** | 0.819 | 0.853 | 0.824 | 0.822 | **0.853** | **1.0** | **0.968** | 0.797 |
| **G2** | 0.782 | 0.833 | 0.829 | 0.827 | **0.864** | **0.989** | **1.0** | 0.793 |
| **G3** | 0.667 | 0.654 | 0.817 | 0.843 | 0.693 | 0.821 | 0.817 | 1.0 |

TABLE 11: In BINGO-E, percentage of functions ranked #1 in inter-compiler (x86 32bit) comparison of `sthttpd` to `thttpd`. Here, **C** and **G** represent `clang` and `gcc` compilers, respectively and 0-3 represents optimization O0-O3. **Bold** font highlights results of the configuration `gcc` O0-O2, the same as that in COP [4].

| | C0 | C1 | C2 | C3 | G0 | G1 | G2 | G3 |
|---|---|---|---|---|---|---|---|---|
| **C0** | 1.0 | 0.828 | 0.761 | 0.757 | 0.930 | 0.830 | 0.862 | 0.693 |
| **C1** | 0.773 | 1.0 | 0.800 | 0.784 | 0.867 | 0.884 | 0.900 | 0.654 |
| **C2** | 0.676 | 0.747 | 1.0 | 1.0 | 0.774 | 0.824 | 0.829 | 0.789 |
| **C3** | 0.657 | 0.743 | 1.0 | 1.0 | 0.771 | 0.822 | 0.827 | 0.814 |
| **G0** | 0.961 | 0.883 | 0.802 | 0.800 | **1.0** | **0.884** | **0.875** | 0.733 |
| **G1** | 0.819 | 0.853 | 0.824 | 0.822 | **0.853** | **1.0** | **0.967** | 0.769 |
| **G2** | 0.800 | 0.846 | 0.829 | 0.827 | **0.854** | **0.978** | **1.0** | 0.780 |
| **G3** | 0.657 | 0.646 | 0.817 | 0.843 | 0.697 | 0.823 | 0.805 | 1.0 |

for cross-compiler analysis. Besides, function filtering reduces the search space dramatically. After filtering, for each signature function in `coreutils` binary, on average around 21 target functions are shortlisted in cross-architecture analysis and it is further reduced to 13 functions in cross-compiler analysis. For large binaries such as `BusyBox` (around 3,250 functions with 39,179 basic-blocks), it takes on average 6.16 seconds to filter the target functions and for each signature function less than 40 target functions are shortlisted. Similarly, for SSL/Heartbleed vulnerability search in `openssl` (around 5,700 functions with more than 60K basic-blocks), filtering process takes 12.24 seconds, with only 53 functions shortlisted.

The major overhead in BINGO is the partial trace generation and semantic feature extraction operation. For example, it takes 4,469s to extract semantic features from 2,611 `libc` functions — on average, taking 1.7s to extract semantic features from a `libc` function, whereas, it takes only 1,123s to extract semantic features from 1,220 `msvcrt` functions (on average 0.9s per function). By virtue of our filtering technique, the time-consuming step of semantic feature extraction is not applied to all the functions in a binary. In practise, semantic feature extraction is a one time job that can be easily parallelized. Finally, in locating the vulnerable function in Adobe PDF Reader, it takes, in total, 88 seconds to filter and extract semantic features from the shortlisted target functions and an additional 2.7 seconds to do the semantic matching. This shows that using BINGO, it allows us to find a zero-day vulnerability in a real-world COTS binary within two minutes, provided a good signature function.

#### 7.6.2 BINGO-E*'s Answer to RQ5: Better Scalability*

For example, in cross-architecture analysis, it takes 485.2 milliseconds on average to compare entire `coreutils` suite (in total, 103 binaries each containing on average 250 functions) compiled for one architecture against another, while it is reduced to 404.6 milliseconds for cross-compiler analysis. For experiments on the cross-OS matching, it takes 123.70s to compare entire `libc` binary against `msvcrt`. For other large binaries such as `BusyBox` (around 3,250 functions with 39,179 basic-blocks), it takes 1307.92 seconds to rank all the functions (402.4 milliseconds per function).

The major overhead in BINGO-E is using emulation to extract low-level semantic features. For example, it takes 2334.4s to extract semantic features from 2,611 `libc` functions. In the experiments we run emulation 16 times to improve the robustness. We found that the results did not change much if we run it only one time. Similarly, we were able to extract semantic features from `msvcrt` with 1,220 functions in 787.96s. For other overheads, to extract 3D-CFG features from `msvcrt`, it takes 334.73s; while it takes around 147.67s to obtain the other high-level semantic features. In practice, the feature extraction process is a one-time job that can be easily parallelized.

In this experiment, BINGO-E has significantly reduced the time for low-level semantic feature extraction. For example, the time for 1,220 `msvcrt` functions is reduced from 1,123s to 787.96s; the time for 2,611 `libc` functions is reduced from 4,469s to 2334.4s. Using the saved time, BINGO-E can extract high-level semantics and structural features to improve the accuracy.

### 7.7 Discussion

#### 7.7.1 Threats to Validity

Threats to internal validity come from these aspects.

- One major internal limitation is about selective inlining — we cannot inline functions that are invoked indirectly (i.e., indirect call). However, this limitation is not particular to BINGO-E but an inherent problem in static analysis technique. Early in 2005, people have identified this problem and proposed solutions, such as VSA (value-set analysis) [41]. However, we did not incorporate VSA into BINGO-E as it involves heavyweight program analysis, which might leads to heavy performance overheads.
- Previously, in BINGO, the constraint solving for low-level semantic features is based on IR language (i.e., REIL), which suffers from the issue of the incomplete support of floating-point (FPU) instructions. Now, IR is not needed, as the CPU emulator, UNICORN, is used. Hence, some part of internal limitation of BINGO-E comes from the limitation of UNICORN. The potential errors in handling different instruction sets may lead to errors for low-level semantic features and affect the final results.
- We adopt IDA Pro to disassemble and generate CFGs from the functions in binaries. As pointed by Meng *et al.* [42], accurate disassembling and binary analysis is not easy. The errors in CFG generation may affect the 3D-CFG features and further affect the results. In future, Dyninst can be used for accurate CFG generation.

Threats to external validity rise from these issues.

- Currently, we mainly use `coreutils` and `BusyBox` for the evaluation of cross-architecture and cross-compiler matching, `msvcrt` and `libc` for cross-OS matching. Besides, `sthttpd` and `thttpd` are used for binary code matching among forked projects. These executables are some common benchmarks that are used in other studies, e.g., `coreutils` in [9], [20], `sthttpd` and `thttpd` in [4], [5]. It is still worth exploring to what extent the existing experimental results can be true when large-size commercial software binaries are used for matching.
- For the tool comparison, we only run the available tools, BINDIFF [24] and TRACY [10], with the by-default configuration. For the comparison with DISCOVRE [21], BugSearch tool [9] and COP [4], [5], we compare the results based on the figures that are reported in the corresponding paper of each tool. In future, when more other tools are available, we would like to run them, tune up their parameters, and conduct detailed comparison with BINGO-E.

#### 7.7.2 The Focus of Our Study.

In this paper, our goal is to combine the different categories of features and leverage emulation for acceleration. Hence, the focus of our study

is not to systematically evaluate and comprehensively summarize which features are more effective for which matching scenarios or which projects. Currently, throughout the experiments, only some preliminary observations on effectiveness of different features are listed as follow:

- Low-level semantic features are effective, especially when the matching is on different code bases or the binary code obfuscation is applied [4]. However, according to our observation, false positive cases are discovered in case of 1) two binary segments are too short to contain meaningful semantics; 2) they are null check or checksum at the beginning, which leads to the match of two code segments for most give input.
- High-level semantic features capture the semantics of system/library calls, being helpful for cross-OS matching or the matching on different code bases, especially for some specific tasks of binary code search (e.g., malware detection [25]). However, for cross-OS matching, the mapping of system calls on different OSs need to be carefully handled. Besides, like Adobe Reader that implements the `malloc`, some software implements certain system/library calls, which may fail the match if such case is not considered.
- Structural features are most effective if the matching is on the same code base using similar compilers and the optimization levels [10], [22]. Thus, for code plagiarism, structural features can be effective if no binary code obfuscation is applied. The issue of structural features is usually the variance of the BB-structure. Using tracelet or 3D-CFG can be a remedy to the issue.

In future, at least a dozen of configurations (3 choices of architectures, 2 choices of OSs, 2 choices of compilers) can be set up to systematically evaluate under which configuration what features are more effective. Also, it is worth investigation that how binary or function size affects the usefulness of these features.

# 8 Related Work

Our work is relevant to the following three lines of works.

## 8.1 Code Search Using Low-level Program Semantics

Treating the program as a black box, the idea of checking input-output (I/O) values and intermediate values is applied to identify the semantic clones or behavioral clones, even without the usage or the analysis of code. This idea is first applied to detect semantic clones in source code. As early as in 2009, Jiang *et al.* [17] proposed to cluster the code fragments according to the I/O samples of random testing. In 2011, to detect semantic clones, Kim *et al.* [43] proposed the tool MeCC, which models the abstract memory state and matches similar functions by comparing I/O values and guard conditions. Modelling abstract state in MeCC is, to some extent, similar to that we do in BinGo for low-level semantics extraction, but MeCC works on C/C++ code and our approach works on binary code.

The similar idea is applied for binary code search. In recent years, Schkufza *et al.* [18] presented a method to compare x86 loop-free snippets for testing the correctness of binary code transformation (e.g., transforming code compiled by `llvm -O0` to code compiled by `gcc -O3`). Based on I/O values and states of the machine, the equivalence is checked when the execution is complete. In addition to I/O values, intermediate values can be an alternative solution when it is difficult to identify I/O variables in binary code. Zhang *et al.* [44] proposed to use the local variables and local addresses referenced, which are extracted from dynamic execution traces, for comparing the binary code of two versions of the program. In [19], based on the assumption that some specific intermediate values are inevitable in implementing the same algorithm, Jhi *et al.* [19] proposed to use value-sequence to serve as the fingerprint for the binary function and compare value-sequence for measuring similarity. According to this assumption of considering immediate values as the "signature" of an algorithm, Zhang *et al.* [45] presented a method based on value dependency graph to detect algorithm plagiarism. In recent studies, BLEX [20] used low-level semantic features such as the values read from or written to stack/heap and the return value of registers. In [4] and [5], Luo *et al.* adopted symbolic formulas to represent the I/O relations of blocks and applied longest common subsequence (LCS) based fuzzy matching to measure binary code similarity, even when binary code obfuscation is present.

Among the above studies, we use the similar low-level semantic features that are used in [43], [4], [5] to model the memory states and I/O values in BinGo. Different from other studies, we extract these low-level semantics from $k$-traces ($k$-length partial trace). To overcome the drawbacks of low-level semantics in § 4.3.1, in BinGo-E, we incorporate some low-level semantic features proposed in BLEX [20] (§ 4.3.2). Note that all the above studies employ dynamic execution or constraint solving to extract the I/O values, but BinGo-E adopts emulation to speed up the extraction.

## 8.2 Code Search Using High-level Program Semantics

In the domain of malware detection and analysis, system call sequences or system call based behavior graph [46] are widely used as the signatures of malware, as the represent the unique high-level semantics of the program — the functionality the program behaves. Usually, system calls are mapped and categorized into different abstract actions that represent certain behaviors [25]. In such a way, similar malware can be detected regardless of architecture and OS differences.

System calls represent program semantics. Similarly, third party library calls can also represent high-level program semantics. BLEX [20] is presented to tolerate the differences in compilation optimization and obfuscation. As mentioned above, BLEX executes functions of the two given binaries with the same inputs and compare the output behaviors. To relax the difference of two binaries, BLEX further uses the high-level semantic features from an execution, i.e., calls to imported library function via the point and system calls made during execution. According to their report, using Library function invocation alone can only gain an accuracy of 17% in matching, while using system calls alone produces an accuracy of 38%.

Hence, for achieving a better matching accuracy, using the names of library function invocation and system calls is insufficient. Hu *et al.* [47] proposed the approach MockingBird, which extracts function signature as a sequence of two dynamic features: Comparison Operand Pairs (COP) and System Call Attributes (SCA). COP refers to the operation result that directly determines the control flow of the execution, and SCA refers to system call attributes that consist of names and argument values of all system calls invoked in an execution. Besides, to support cross-architecture code matching, MockingBird addresses the diversity issue of instruction sets using VEX-IR.

In BinGo-E, we are not only using attributes of system calls and library function calls, but also incorporating opcode and op types. More than names and attributes of system calls, we also consider tags for common system calls to support cross-OS code matching. The idea of system call tags is inspired by the abstract actions that are used for malware detection [25]. Note that BinGo-E adopts no IR.

## 8.3 Code Search Using Structural Information

CFG and its derived representations are the commonly-used structural information for binary code matching. BinDiff [24] builds CFGs of the two binaries and then adopts a heuristic to normalize and match the two CFGs. Essentially, BinDiff resolves the NP-hard graph isomorphism problem (matching CFGs). BinHunt [48], a tool that extends BinDiff, is enhanced for binary diff at the two following aspects: considering matching CFGs as the Maximum Common Induced Subgraph Isomorphism problem, and applying symbolic execution and theorem proving to verify the equivalence of two basic code blocks. To address non-subgraph matching of CFGs, BinSlayer [49] models the CFG matching problem as a bipartite graph matching problem. For these tools, compiler optimization options change the structure of CFGs and fail the graph-based matching. Besides, graph-based matching may produce unsymmetrical results.

Witnessing the weakness of graph-based matching, some researchers proposed new derived representation of binary code (e.g., graphlet [29], tracelet [10]). Saebjornsen *et al.* [16] proposed a binary code clone detection framework that leverage on normalized syntax (i.e., normalised operands) based function modelling technique. Jang

*et al.* [28] propose to use $n$-gram models to get the complex lineage for binaries, and normalize the instruction mnemonics. Based on the $n$-gram features, the code search is done via checking symmetric distance. Krügel *et al.* [29] propose a graphlet-based approach to identify malware, which generates connected $k$-subgraphs of the CFG and apply graph-coloring to detect common subgraphs between a malware sample and a suspicious one. Tracelet [10] is presented to capture syntax similarity of execution sequences and facilitate searching for similar functions. However, these tools relying on structural information may fail during cross-architecture and cross-OS analysis.

In BINGO-E, the structural information of CFG is represented as a new form, i.e., 3D-CFG [26]. To the best knowledge of ours, 3D-CFG is previously used for byte-code clone detection, and our study is the first attempt to apply this idea for binary code matching. Besides, to avoid the noise due to the differences in BB-structure of CFGs, before extracting low-level semantic features, we build function model of $k$-tracelet, and extract low-level semantic features from them. Hence, in BINGO-E, the idea of using $k$-tracelet [10] is adopted to mitigate the impact of differences in BB-structure.

### 8.4 Security Application of Binary Code Search

In cyber security domain, binary code search tool is a powerful weapon for securing binary executables (e.g., malware detection and vulnerability discovery). In many scenarios, static analysis tools are preferred for binary code auditing. Dynamic analysis faces challenges from two aspects: the difficulty in setting up the execution environment, and the scalability issue that prevents large-scale detection.

Pinpointed by Zaddach *et al.* [50], these dynamic approaches are far from practical application onto highly-customized hardware like mobile or embedded devices. Thus it is difficult for these approaches to conduct cross-architecture bug detection. To address this issue, Pewny *et al.* [9] and DISCOVRE [21] propose a static analysis technique with the aim to detect vulnerabilities inside multiple versions of the same program compiled for different architectures. Thus, their approach is good for finding clones of the same program due to architecture or compilation differences, not suitable for finding semantic binary clones among different applications. [51] aims to detect infected virus from executables via a static CFG matching approach. In addition, [52] and [53] also adopt CFG in the recovery of the information of compilers and even authors.

In this study, we make as signature the vulnerable binary functions from open source projects, and search them in the closed-source commercial executable (e.g., Adobe PDF Reader). The same or similar vulnerability can be discovered, if an open source software bug is forked into the commercial solution.

## 9 CONCLUSION

In this work, we present the scalable solution of binary code search framework, BINGO-E, which aims to search similar binary code regardless of the differences in architecture, OS and compilation options. Rather than only relying on low-level semantic features in BINGO, we further incorporate high-level semantic features and structural features in BINGO-E. To speed up low-level feature extraction, we adopt emulation in BINGO-E. The promising experimental results live up to the expectation towards effective and efficient cross-architecture, cross-OS and cross-compiler binary code search. Further, BINGO-E has outperformed the state-of-the-art tools like TRACY and BINDIFF, and also significantly improved the previous version of our tool (i.e., BINGO). In security application, we also discovered a zero-day vulnerability in Adobe PDF Reader. In future, we are planning to apply BINGO-E on matching open-source software bugs to the commercial software's binaries, in order to reveal more unknown vulnerabilities due to the forking of common open-source software.

## 10 ACKNOWLEDGEMENTS

This research is supported in part by the National Research Foundation, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-30).

## REFERENCES

[1] "The gpl-violations.org project," http://gpl-violations.org/, accessed: 2016-09-22.

[2] "Vmware lawsuit - software freedom conservancy," https://sfconservancy.org/news/2015/mar/05/vmware-lawsuit/.

[3] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, "Finding software license violations through binary code clone detection," in *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings*, 2011, pp. 63–72. [Online]. Available: http://doi.acm.org/10.1145/1985441.1985453

[4] 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*. ACM, 2014, pp. 389–400.

[5] ——, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017.

[6] J. Caballero, N. M. Johnson, S. McCamant, and D. Song, "Binary code extraction and interface identification for security applications," DTIC Document, Tech. Rep., 2009.

[7] D. Kim, W. N. Sumner, X. Zhang, D. Xu, and H. Agrawal, "Reuse-oriented reverse engineering of functional components from x86 binaries," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 1128–1139.

[8] J. Ming, D. Xu, and D. Wu, "Memoized semantics-based binary diffing with application to malware lineage inference," in *ICT Systems Security and Privacy Protection*. Springer, 2015, pp. 416–430.

[9] 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, SP 2015, San Jose, CA, USA, May 17-21, 2015*, 2015, pp. 709–724. [Online]. Available: http://dx.doi.org/10.1109/SP.2015.49

[10] Y. David and E. Yahav, "Tracelet-based code search in executables," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, 2014, p. 37. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594343

[11] S. K. an Seunghoon Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *Proceedings of the 38th IEEE Symposium on Security and Privacy*, 2017, p. to appear.

[12] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Software Eng.*, vol. 28, no. 7, pp. 654–670, 2002. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TSE.2002.1019480

[13] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: scalable and accurate tree-based detection of code clones," in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, 2007, pp. 96–105. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.30

[14] P. P. F. Chan and C. S. Collberg, "A method to evaluate CFG comparison algorithms," in *2014 14th International Conference on Quality Software, Allen, TX, USA, October 2-3, 2014*, 2014, pp. 95–104. [Online]. Available: http://dx.doi.org/10.1109/QSIC.2014.28

[15] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, 2008, pp. 321–330. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368132

[16] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 117–128.

[17] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, 2009, pp. 81–92. [Online]. Available: http://doi.acm.org/10.1145/1572272.1572283

[18] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, 2013, pp. 305–316. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451150

[19] Y. 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, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, 2011, pp. 756–765. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985899

[20] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *USENIX Security Symposium*, 2014.

[21] E. Sebastian, Y. Khaled, and G.-P. Elmar, "discovre: Efficient cross-architecture identification of bugs in binary code," in *In Proceedings of the 23nd Network and Distributed System Security Symposium*. NDSS, 2016.

[22] 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, Santa Barbara, CA, USA, June 13-17, 2016*, 2016, pp. 266–280. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908126

[23] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Choo, and H. B. K. Tan, "Bingo: Cross-architecture cross-os binary search," in *Proceedings of 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'16)*, 2016, p. to appear.

[24] H. Flake, "Structural comparison of executable objects," in *Detection of Intrusions and Malware & Vulnerability Assessment, GI SIG SIDAR Workshop, DIMVA 2004, Dortmund, Germany, July 6.7, 2004, Proceedings*, 2004, pp. 161–173. [Online]. Available: http://subs.emis.de/LNI/Proceedings/Proceedings46/article2970.html

[25] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "A quantitative study of accuracy in system call-based malware detection," in *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, 2012, pp. 122–132. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336768

[26] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, 2014, pp. 175–186. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568286

[27] "Unicorn: the ultimate cpu emulator," http://www.unicorn-engine.org/, accessed: 2016-09-22.

[28] J. Jang, M. Woo, and D. Brumley, "Towards automatic software lineage inference," in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, 2013, pp. 81–96. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/papers/jang

[29] C. Krügel, E. Kirda, D. Mutz, W. K. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," in *Recent Advances in Intrusion Detection, 8th International Symposium, RAID 2005, Seattle, WA, USA, September 7-9, 2005, Revised Papers*, 2005, pp. 207–226. [Online]. Available: http://dx.doi.org/10.1007/11663812_11

[30] M. Wang, H. Yin, A. V. Bhaskar, P. Su, and D. Feng, "Binary code continent: Finer-grained control flow integrity for stripped binaries," in *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 2015, pp. 331–340.

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

[32] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, 2014, pp. 175–186.

[33] G. Meng, Y. Xue, Z. Xu, Y. Liu, J. Zhang, and A. Narayanan, "Semantic modelling of android malware for effective malware comprehension, detection, and classification," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, 2016, pp. 306–317.

[34] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: mining and recommending API usage patterns," in *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, 2009, pp. 318–343.

[35] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[36] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[37] P. Agrawal, A. Arasu, and R. Kaushik, "On indexing error-tolerant set containment," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 927–938.

[38] T. Dullien and S. Porst, "Reil: A platform-independent intermediate representation of disassembled code for static code analysis," *Proceeding of CanSecWest*, 2009.

[39] "BinGo: Cross-Architecture Cross-OS Binary Searche," https://sites.google.com/site/bingofse2016/, 2016, [Online; accessed 11-March-2016].

[40] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014, pp. 406–415.

[41] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum, "Wysinwyx: What you see is not what you execute," in *Verified software: theories, tools, experiments*. Springer, 2005, pp. 202–213.

[42] X. Meng and B. P. Miller, "Binary code is not easy," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, 2016, pp. 24–35. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931047

[43] H. Kim, Y. Jung, S. Kim, and K. Yi, "Mecc: memory comparison-based clone detector," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, 2011, pp. 301–310. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985835

[44] X. Zhang and R. Gupta, "Matching execution histories of program versions," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, 2005, pp. 197–206. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081738

[45] F. Zhang, Y. Jhi, D. Wu, P. Liu, and S. Zhu, "A first step towards algorithm plagiarism detection," in *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, 2012, pp. 111–121. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336767

[46] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and efficient malware detection at the end host," in *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, 2009, pp. 351–366. [Online]. Available: http://www.usenix.org/events/sec09/tech/full_papers/kolbitsch.pdf

[47] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Cross-architecture binary semantics understanding via similar code comparison," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, 2016, pp. 57–67. [Online]. Available: http://dx.doi.org/10.1109/SANER.2016.50

[48] D. Gao, M. K. Reiter, and D. X. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *Information and Communications Security, 10th International Conference, ICICS 2008, Birmingham, UK, October 20-22, 2008, Proceedings*, 2008, pp. 238–255. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-88625-9_16

[49] M. Bourquin, A. King, and E. Robbins, "Binslayer: accurate comparison of binary executables," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop 2013, PPREW@POPL 2013, January 26, 2013, Rome, Italy*, 2013, pp. 4:1–4:10. [Online]. Available: http://doi.acm.org/10.1145/2430553.2430557

[50] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014. [Online]. Available: http://www.internetsociety.org/doc/avatar-framework-support-dynamic-security-analysis-embedded-systems%E2%80%99-firmwares

[51] D. Bruschi, L. Martignoni, and M. Monga, "Detecting self-mutating malware using control-flow graph matching," in *Detection of Intrusions and Malware & Vulnerability Assessment, Third International Conference, DIMVA 2006, Berlin, Germany, July 13-14, 2006, Proceedings*, 2006, pp. 129–143. [Online]. Available: http://dx.doi.org/10.1007/11790754_8

[52] N. E. Rosenblum, X. Zhu, and B. P. Miller, "Who wrote this code? identifying the authors of program binaries," in *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*, 2011, pp. 172–189. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23822-2_10

[53] N. E. Rosenblum, B. P. Miller, and X. Zhu, "Extracting compiler provenance from program binaries," in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'10, Toronto, Ontario, Canada, June 5-6, 2010*, 2010, pp. 21–28. [Online]. Available: http://doi.acm.org/10.1145/1806672.1806678