BinGo: Cross-Architecture Cross-OS Binary Search

Mahinthan Chandramohan[†], Yinxing Xue[†], Zhengzi Xu[†], Yang Liu[†], Chia Yuan Choo[*], and Tan Hee Beng Kuan[†]
[†]Nanyang Technological University, Singapore
[*]DSO National Laboratories, Singapore

ABSTRACT

Binary code search has received much attention recently due to its impactful applications, e.g., plagiarism detection, malware detection and software vulnerability auditing. However, developing an effective binary code search tool is challenging due to the gigantic syntax and structural differences in binaries resulted from different compilers, architectures and OSs. In this paper, we propose BINGO— a scalable and robust binary search engine supporting various architectures and OSs. The key contribution is a selective inlining technique to capture the complete function semantics by inlining relevant library and user-defined functions. In addition, architecture and OS neutral function filtering is proposed to dramatically reduce the irrelevant target functions. Besides, we introduce length variant partial traces to model binary functions in a program structure agnostic fashion. The experimental results show that BINGO can find semantic similar functions across architecture and OS boundaries, even with the presence of program structure distortion, in a scalable manner. Using BINGO, we have discovered a zero-day vulnerability in Adobe PDF Reader, a COTS binary.

1. INTRODUCTION

Recently, binary code searching has attracted much attention for its important applications in software engineering and security, e.g., software plagiarism detection [26], reverse engineering [7], semantic recovery [23], malware detection [28] and buggy (vulnerable) code identification [29, 10] in various software components where the source code is not available (e.g., legacy applications). We can even search for zero-day vulnerabilities in proprietary binary by matching the known vulnerability from open source software.

Traditional source code search relies on the similarity analysis of some representations of source code, e.g., approaches based on token [22], abstract syntax tree (AST) [20] or program dependency graph (PDG) [16]. All these representations capture the structural information of the program, and yield accurate results for source code search. However, code search in binary is much more challenging due to many factors (e.g., architecture and OS choice, compiler type, optimization level or even obfuscation technique) and limited availability of high-level program information. These factors have a substantial influence on the assembly instructions and their final layout in the compiled binary executable.

Recently, various approaches have been proposed to detect the similar binary code by using static or dynamic analysis. Static analysis [10, 33, 26, 29] relies on the syntactical and structural information of binaries, especially control-flow structures (i.e., organization of basic blocks within a function) to perform the matching. Dynamic approaches [21, 34, 19, 13, 14] inspect the invariants of input-output or intermediate values of program at runtime to check the equivalence of binary programs. Table 1 summarizes the state-of-the-art binary code search techniques proposed in the literature. TRACY [10] is a pure syntax based function matching technique that uses n-tracelet (i.e., basic blocks of length n along the control-flow path), which is architecture- and OS-dependent. COP [26] is a plagiarism detection tool that leverages on the theorem prover to search for semantically equivalent code segments. The bug search tool proposed in [29] supports cross-architecture analysis via the invariants of bug signatures. DISCOVRE tool [35] is proposed to find bugs in binaries across-architectures in a scalable manner, where it uses two filters (numeric and structural) to quickly locate the functions that are similar to the signature function. Finally, BLEX [14] is the latest dynamic function matching tool using seven semantic features generated from running execution.

Table 1: Comparison of existing techniques. Here, ✓✗ denotes *limited* support, while ✓ and ✗ represent *full* and *no* support, respectively.

Tool	Technique	P1 Resilient	P2 Accurate	P3 Scalable
Tracy [10]	Static	✗	✗	✓
CoP [26]	Static	✗	✓✗	✗
Bug search [29]	Static	✗	✗	✗
DISCOVRE [35]	Static	✗	✗	✓
BLEX [14]	Dynamic	✓	✓	✗
BinGo	Static	✓	✓	✓

To better understand the existing approaches, we identify three desired properties for an accurate yet scalable cross-architecture and cross-OS binary search tool.

- **P1.** Resilient to the syntax and structural gaps introduced due to architecture, OS and compiler differences.
- **P2.** Accurate by considering the complete function semantics.
- **P3.** Scalable to large size real world binaries.

Table 1 summarizes the effectiveness of existing works for the three properties. Clearly, none of the existing techniques in Table 1 can achieve all the desired properties above. The structural information used by the static approaches, e.g., basic-block structures used in [26, 29, 35] and fixed length tracelet in [10], fail P1. Most static analysis techniques [10, 29, 35], which ignore the semantics of relevant system libraries and user-defined functions in the matching, cannot pass P2. For COP [26], its ability to capture complete semantics is limited to the execution path only. Scalability is a real challenge in semantic binary search, e.g., COP took half a day to find target functions in Firefox for a few signature functions [26]. The details are further elaborated in §2.2.

In this paper, we propose a binary search engine, named BINGO, which performs *semantic matching* by combining a set of key techniques to address the challenges above. Given a binary function, BINGO first inlines relevant libraries and user-defined functions in order to capture the complete semantics of the function (for P2), then shortlists the candidate target functions using an architecture and OS neutral filtering technique (for P3), and finally extracts the

partial traces of various lengths from the candidate target functions as function models (for P1) and conducts the function similarity matching using machine learning techniques.

Technically, first, to recover the complete semantics from the functions under investigation, we propose a *selective inlining* technique, where the callee (both libraries and user-defined) functions are inlined into the caller such that the complete function semantics are captured [36] (§3). To avoid code size explosion, we selectively inline the callee functions based on the invocation dependency patterns, which differs from the traditional compiler inlining optimization techniques for maximum speed or minimum size [9]. To our best knowledge, this work is the first attempt towards investigating selective inlining in recovering binary semantics.

Second, to improve the scalability of our approach, we propose an *architecture and OS neutral filtering* technique that narrows down the search space by shortlisting the candidate target functions for binary semantic matching (§4).

Next, to overcome the limitation of basic-block structures (§5), we generate *function models*, which are agnostic to the underlying program structure, via the length variant partial traces[1] (called, partial traces of k lengths). For each function, partial traces of length 1 to k are extracted to form the function model such that it represents the function at various granularity levels. Here, we also take measure to minimize the effects of infeasible paths and compiler specific code in calculating the function similarity scores.

Finally, semantic features are extracted from the function models of candidate target functions, for function similarity scoring, where semantic features capture the machine state transitions in the form of Input/Output pairs (§5).

Experimental results. We evaluate BINGO on a number of real-world binaries containing hundreds of thousands of functions. The experimental results show that BINGO can effectively perform cross-architecture and cross-OS binary code search on these binaries. In terms of P1, P2 and P3, BINGO outperforms the available state-of-the-art tools, i.e., TRACY [10] and BINDIFF [15] for the same tasks. Further, we also show that recent techniques, such as [29] and [35], fail in case of program structure is distortion, while BINGO can handle such cases swiftly. Last but not least, using BINGO, we discovered a zero-day vulnerability (CVE-2016-0933) in the Adobe PDF Reader with the 4000USD bug bounty.

Key contributions. This work makes the following contributions:

- We propose a selective inlining algorithm to capture the complete semantic of the binary functions.
- We introduce an architecture and OS neutral function filtering process that helps narrow down the target function search space.
- We leverage on k-length partial traces to model the function at various granularity levels that is agnostic to underlying program structures.
- We empirically demonstrate that BINGO outperforms the available state-of-the-art tools of binary code search, and report the zero-day vulnerability discovered from Adobe PDF Reader.

2. BACKGROUND AND OVERVIEW

In this section, we provide a motivating example that emphasizes on the need for complete semantics analysis of functions and necessity to have a function model that is agnostic to underlying program structure. Then, we explain the basic idea of our proposed solution and sketch the system overview.

2.1 Motivating Example

[1]A partial trace refers to a sequence of basic-blocks that lie along an execution path in the CFG [10].

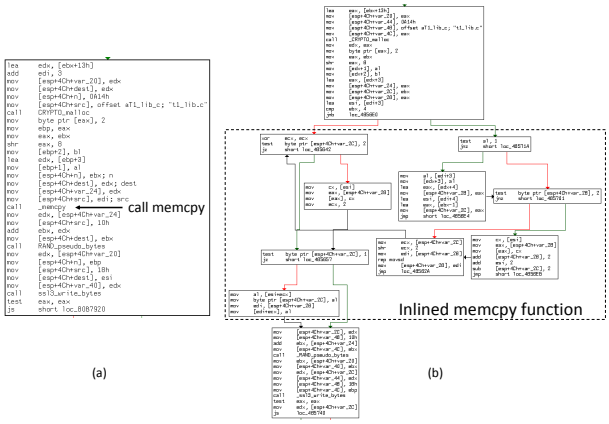

Figure 1: Code segment responsible for Heartbleed vulnerability (CVE-2014-0160) appeared as in the binary (a) compiled with GCC 4.6, and (b) compiled with Mingw32

A binary program consists of a number of functions, where each function is a (directed) graph of basic blocks, i.e., CFG (control-flow graph). The instructions in a function are systematically grouped into several basic blocks, which are considered as the building blocks of binary program and this representation is used by many static binary analysis tools.

Interestingly, the same source code may have different basic-block structures after compilation. Taking the infamous heartbleed vulnerability (CVE-2014-0160) for example, Fig. 1(a) and (b) show the basic-block structures of the same source code compiled with `gcc` and `mingw`, respectively. Apparently, these two binary code segments share no identical basic-block 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 the library function `memcpy` is inlined in `mingw` version; while in `gcc`, it is not. The large syntactic and program structural differences between the two binaries pose the biggest challenge for the existing binary code search tools.

2.2 Challenges for Existing Approaches

Syntax is the most direct information usable for binary matching. Most of existing approaches rely on syntax information have attempted to use instruction patterns [18, 25, 33, 10]. As there is no consistent low-level syntax representation (i.e., assembly instructions) across architectures, these approaches fail for cross-architecture analysis. To make the matching resilient to syntax differences in binaries, semantics-based matching has been proposed [26, 29], in which the machine state transition represents the semantics of a binary. Still, three challenges are faced in semantics-based matching.

C1: The challenge of using program structural properties. Existing approaches [26, 29] assume that basic-block structure is preserved across binaries, thus the matching should be basic-block centric. Based on this, semantic features extracted from the signature function at basic-block level are compared with the counterparts extracted from target functions in a pairwise way. In practise, the assumption is too restrictive to be applied for real-world cases. The following quote from [29] clearly sums up the problem in such assumption: *"Our metric is sensitive to the CFG and the segmentation of the basic block, which we found to be potentially problematic especially for smaller functions."* For example, the basic block in Fig. 1(a) needs to be matched with several basic blocks in Fig. 1(b).

C2. The challenge of covering the complete function semantics. Most of the existing static techniques consider functions in isolation, i.e., semantics of callee functions are not considered as part of

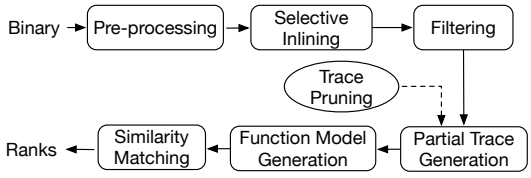

Figure 2: BINGO system architecture

the caller's semantics. This leads to partial semantics problem, especially when the programmer makes the most commonly used code segment as an independent function or implements her own version of the standard C library functions (e.g., Adobe Reader has its own `malloc` implementation). To address this, one can *blindly* inline all the callee functions (e.g., in [36], all the user-defined functions are inlined). Unfortunately, this approach does not work in practise due to two main reasons: (1) heavy inlining may lead to code size explosion [9], which is not scalable to real-world binary analysis, and (2) not all the callee functions are closely related to the caller function in-terms of functionality. For example, `memcpy` is not inlined in Fig. 1(a) but it is in Fig. 1(b). Thus, `memcpy` needs to be inlined for matching with other semantically relevant functions.

C3. The challenge of scalable searching in large real-world binaries. Syntax based techniques, in general, are scalable [10]. However, as discussed earlier, they fail on cross-architecture analysis. To address this problem, semantics based techniques are preferred, however, extracting semantic features incur heavy overhead and not scalable. Therefore, to facilitate scalable semantic matching, an *efficient*, and architecture, OS and compiler *neutral* function filtering step is required. However, exiting approaches focus only on the efficiency aspect of the filter. For example, in [35], only program structural features, such as number of basic blocks, are used as filters to speed up. However, as shown in Fig. 1, such features are not robust enough for cross-architecture, OS and compiler analysis.

2.3 Proposed Solution and System Overview

We propose BINGO as a scalable binary search engine, where given a binary function, BINGO will return the functions from the target binary, ranked based on their semantic similarity.

Fig. 2 shows the work flow of BINGO. First, given a signature function, BINGO will pre-process the binaries, i.e., disassemble and build CFGs from the functions, for further analysis. Next, for each function, in the target binaries, closely related library and other user-defined functions are identified and inlined (§3). Then from these inlined target functions, we shortlist a list of candidate functions that are similar to the signature function by using three filters, which consider different aspects of the binary semantic (§4). Next, from the signature and the shortlisted target functions, length variant partial traces are generated (§5.1), which are then grouped to form the function models (§5.4). During the partial trace generation phase, trace pruning is conducted to remove the irrelevant and infeasible partial traces from the analysis (§5.3). After that, semantic features are extracted from the function models (§5.2), which are later used for semantic similarity matching (§5.4). Finally, based on the similarity scores obtained, BINGO returns the target functions that are semantically similar to the signature function in a ranked order.

To sum up, to address the syntax differences of instructions, we lift the low-level assembly instructions into a corresponding intermediate representation (IR) to facilitate cross-architecture analysis. To mitigate C1 (the single basic block matching to several basic block matching in Fig. 1)), we borrow the idea of tracelet used in TRACY [10]. Different from the original approach that uses a fixed length of tracelet, we use length variant partial traces. Next, to overcome C2 (e.g., whether to inline `memcpy` in Fig. 1(a)), we propose a selective inlining strategy to strike a balance between the needed contextual semantics and the overheads due to inlining. Finally, to address C3, we adopt three filters considering different aspects of the semantics to identify the similar functions.

3. SELECTIVE INLINING

Inlining is a technique in compilers to optimize the binaries for maximum speed or minimum size [9]. This section presents our selective inlining based on function invocation patterns, which has a different goal and strategy compared with compilation process.

3.1 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. 3. Incoming (outgoing) edges in Fig. 3 represent the incoming (outgoing) calls to (from) the function. Here, we elaborate the six patterns as follows.

Case 1: Fig. 3(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. 3(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. 3(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. 3(d) is a variant of 3(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 leads to exception or program termination (e.g., `exit` and `abort`).

Case 5: In Fig. 3(e), The UD callee function f is a variant of 3(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 3(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.

3.2 Inline Decision Algorithm

From the discussions above, in Fig. 3 (a), (b), (d) and (e) there is a clear criteria in deciding whether a callee should be inlined or not. However, for Fig. 3 (c) and (f), the most commonly observed

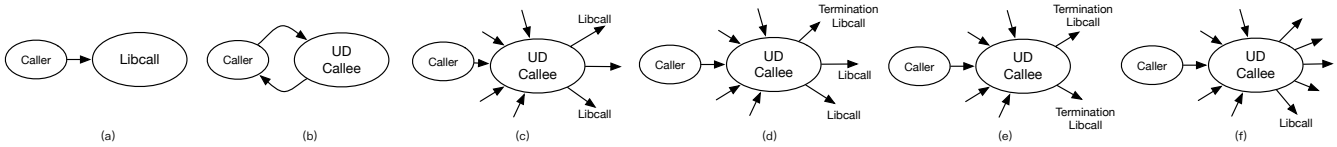

Figure 3: Commonly observed function invocation patterns, where 'UD' denotes user-defined callee function. Here, all the incoming and outgoing edges (or calls) represent user-defined functions, unless specified as *Libcall* or *Termination Libcall* next to them

Algorithm 1: Selective inlining algorithm

Data: caller $\mathcal{F}$, set of callee functions $\mathcal{C}$, set of termination lib. func. $\mathcal{L}_t$, set of inlining lib. func. $\mathcal{L}_s$
Result: inlined function $\mathcal{F}^I$

1 **Algorithm** `SelectiveInline` $(\mathcal{F}, \mathcal{C}, \mathcal{L}_s, \mathcal{L}_t)$
2 **foreach** function f in $\mathcal{C}$ **do**
   // inline selected library functions
3 **if** $f \in \mathcal{L}_s$ **then**
4 $\mathcal{F}^I \longleftarrow \mathcal{F}$.`inline`$(f)$
5 **return** $\mathcal{F}^I$
6 **else if** $f \notin \mathcal{L}_s$ && `isLibCall`(f) **then**
7 **return** null
   // for all other user-defined callee functions
8 $I_u^f \longleftarrow$ `getIncomingCalls`(f)
9 $O_u^f, O_l^f \longleftarrow$ `getOutgoingCalls`(f)
10 $O_u^f \longleftarrow O_u^f \backslash \mathcal{F}$ // remove recursion
11 **if** $|O_u^f| == 0$ && $(|O_l^f \cap \mathcal{L}_s| - |O_l^f \cap \mathcal{L}_t|) \leqslant 0$ **then**
12 **return** null
13 **else**
14 $\alpha = \lambda_e/(\lambda_e + \lambda_a)$ *where* $\lambda_a = |I_u^f|, \lambda_e = |O_u^f|$
   // lower the α, function f is likely to be inlined
15 **if** $\alpha >$ threshold t && `notRecursive`$(\mathcal{F}, f)$ **then**
16 **return** null
17 **else**
18 $\mathcal{F}^I \longleftarrow \mathcal{F}$.`inline`$(f)$
19 **if** $|O_u^f| > 0$ **then**
20 `SelectiveInline`$(f, O_u^f, \mathcal{L}_s, \mathcal{L}_t)$
21 **else**
22 **return** $\mathcal{F}^I$
23 **return** $\mathcal{F}^I$

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 [27]. In this work, similarly, we measure the function coupling by the *function coupling score*.

Definition 1. ***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 λ_a represents the number of UD functions that refers to the callee, and λ_e represents the number of UD functions that is referred to by the callee.

The lower the value of α, 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 λ_e, we only consider the UD functions invoked by the callee, not the library functions. As mentioned in case 3 (Fig. 3(c)) and 6 (Fig. 3(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.

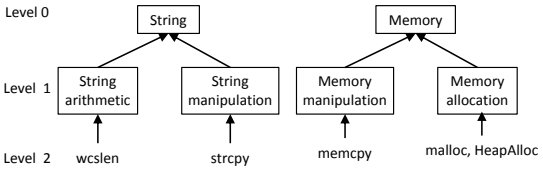

Figure 4: Library function abstraction levels

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 call 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 (3(c)) and Case 6 (3(f)). In this work, we adopt a lightweight static analysis to make the inlining decision for the performance reason (as shown in Section 6). 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.

4. FUNCTION FILTERING

To deal with a huge number of target functions in real-world binaries, an effective filtering process can remove irrelevant target functions before the expensive matching step. BINGO leverages on three types of filters, starting from the specific one to the most general one, to shortlist the candidate target functions.

Filter 1: The first type of filters looks for identical library call invocations in the target functions according to the signature function. If the identical invocations are found, the corresponding targets functions are good candidates for further matching. The reason is that the library call invocations provide an important partial semantics of the function. However, this filter is OS dependent and fails to support library calls that have different names yet with similar functionality (e.g., `memcpy` and `memmove`).

Filter 2: To address the problem in library call name matching, we consider the library call operation types, which help to match the high-level functionality of the library calls. As shown in Fig. 4, there are several ways of abstracting the functionality of a library call. Abstraction level 0 gives the base operation type (e.g., *string* operation), while level 1 gives a more concrete but general abstraction supported cross all OSs. For example, `strcpy` and `strcat` can be mapped to *string manipulation* operation. Therefore, we adopt abstraction level 1 to summarize the library function behavior. In addition, filter 2 is OS neutral, where functions that perform similar task but with different names across OS are mapped to the same operation type,

Algorithm 2: Function Filtering Algorithm

Data: signature function f, set of target functions $\mathcal{T}$
Result: set of candidate target functions $\mathcal{T}_c$

1 **Algorithm** `FunFilter()`
2 $\mathcal{S} \longleftarrow \{\}$ // store similarity score
3 Get the set of library call names $\mathcal{L}_f$, library call operation types $\mathcal{O}_f$ and function instruction type $\mathcal{I}_f$ from the signature function f.
4 **foreach** function t in $\mathcal{T}$ **do**
5 $s^t \longleftarrow \emptyset$
6 $\mathcal{L}_t \longleftarrow$ `getLibFuncNameList`(t) // filter 1
7 $s^t \mathrel{+}= w_1 *$ `getLibFuncNameJaccardSim`$(\mathcal{L}_f, \mathcal{L}_t)$
8 $\mathcal{O}_t \longleftarrow$ `getLibFuncOpType`$(\mathcal{L}_t)$ // filter 2
9 $s^t \mathrel{+}= w_2 *$ `getLibFuncOpTypeJaccardSim`$(\mathcal{O}_f, \mathcal{O}_t)$
10 $\mathcal{I}_t \longleftarrow$ `getFuncInstrType`(t) // filter 3
11 $s^t \mathrel{+}= w_3 *$ `getFuncInstrTypeJaccardSim`$(\mathcal{I}_f, \mathcal{I}_t)$
12 $S[t] = s^t$
13 $\mathcal{S}^s \longleftarrow$ `sortCanditTargetFunc`(S) // sort target functions
14 $\mathcal{T}_c \longleftarrow$ `topNTargetFunc`$(\mathcal{S}^s, \mathcal{N})$
15 **return** $\mathcal{T}_c$

e.g., `malloc` (Windows/Linux) and `HeapAlloc` (Windows) are mapped to *memory allocation* operation type as shown in Fig. 4.

However, library calls can be inlined by the compiler or programmers might implement the same functionality in user-defined functions. Thus, relying on library call names or the operation type will fail. To address this, we propose filter 3.

Filter 3: The third filter is designed look for similarities in the instruction types involved in a binary function as a whole. Here, instruction type refers to a high-level operation carried out by an instruction [24]. In total, instructions are categorized into 14 and 8 instruction types for Intel and ARM architectures, respectively. For example, `mov` instruction is mapped to *data movement* instruction type while `push` is mapped to *stack operation*. In addition, instruction types also support cross-architecture function matching, where assembly instructions from ARM and Intel architectures are mapped to the same instruction type even though they are not identical at the instruction level (e.g., `call` (Intel) and `bl` (ARM) can be mapped to *invoke function* instruction type).

Nevertheless, this filter may suffer from the problem of semantics matching — the instruction types used to implement the functions might look very similar even though they have totally different functionalities at a higher level. Since the filtering process is to shortlist all the target functions that are similar to the signature. For the propose of not missing any potential target functions, some semantically irrelevant functions are tolerated in this step.

Filtering Algorithm Filter 1 is specific and OS dependent. Filter 2 and 3 are general and cross-OS and cross-architecture. Our filtering process is shown in Algorithm 2. At lines 7,9 and 11, we use Jaccard distance [8] to measure the similarity between the signature and target functions in terms of each filter, i.e., their similarity in identical library calls, in library call operation types, and in instruction types. Following the design of applying the filters one by one from the most specific one to the general one, we set the weights ($w_1 > w_2 > w_3 > 0$) to the similarities achieved by Filter 1 to 3. At line 12, we sort the candidate functions according to the overall similarity on three filters (calculated at line 10). Finally, at line 13, we get the top N of the sorting results and use them for function model matching. Note that our filtering process is performed after selective inlining step, hence we keep a mapping from the candidate function to its invoked libraries in order to apply the filters.

5. SCALABLE FUNCTION MATCHING

In BINGO, 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). 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 basic-block centric function modelling [29, 26]. Then, from these partial traces we extract symbolic expressions. Based on I/O samples for the symbolic expressions, we match the partial traces inside two functions. To reduce the noise, we remove the partial traces that are infeasible to reach (via solving the symbolic expressions) or are specific to compilers. Last, we apply Jaccard containment similarity [2] to measure the similarity score of two function models.

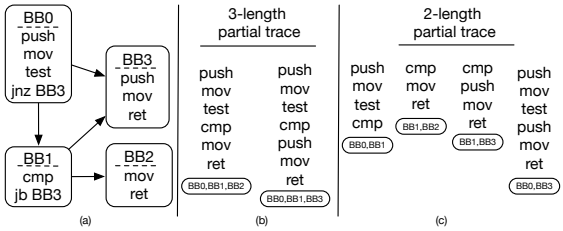

Figure 5: A sample CFG (a) and the extracted 3-length (b) and 2-length (c) partial traces

5.1 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. 5 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. In §5.3.1, we show how the partial traces which are *infeasible* to reach are identified via constraint solving and removed.

5.2 Semantic Feature Extraction

Similar to [29] and [26], 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* [29, 26]. 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 [11], 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 expensive [26] and not scalable for real-world cases, considering the complexity of pair-wise matching between the signature function and a possible huge number of target functions. To tackle the scalability issue, in [29], 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 are ignored [29]. For example, consider the following 2 symbolic expressions:

```
mov  eax, gs : 20
mov  [ebp − 12], eax   } compiler specific code in func. prologue
xor  eax, eax
...
mov  eax, [ebp − 12]
xor  eax, gs : 20      } compiler specific code in func. epilogue
jne  .L5
```

Figure 6: Code generated by `gcc` to enable SSP compiler feature

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

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

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 generated I/O samples (as in [29]), `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. Specifically, we use Z3 constraint solver to generate I/O samples for satisfying all the symbolic expressions in each trace. Generating I/O samples via Z3 is much more scalable than using Z3 to prove the equivalence of two symbolic expressions — I/O sample generation is an one-time job for each partial trace, while equivalence checking (as in [26]) uses the constraint solver every single time a partial trace is compared with another one. In many cases, there are more than one concrete value for any pre- or post-state variable that satisfy all the symbolic expressions. Thus, we set an upper limit N for possible values. According to the study in [14], N is set to 3 in our design.

5.3 Trace Pruning

In BINGO, we adopt two trace pruning methods to reduce the noise in the results and also to speed up the matching process.

5.3.1 *Infeasible Partial Trace Pruning*

BINGO is a static analysis based tool, and therefore it is difficult (or even impossible) to identify all the infeasible partial traces in practice. In BINGO, we prune the obviously infeasible partial traces via the constraint solver. Given the symbolic expressions extracted from a partial trace, we rely on the constraint solver to determine whether all the constraints present in the symbolic expressions can be satisfied. As we use the constraint solver to generate I/O samples in §5.2, we need no additional effort to identify the infeasible partial traces — if the constraint solver is unable to find appropriate concrete values for the pre- and post-state variables, the relevant partial trace is considered infeasible.

We also observe that some partial traces, for which the solver is able to generate models (or I/O samples), might be infeasible during actual execution, as the feasibility of their paths depends on various factors, such as global variables, values in the heap and other dynamic data, that are beyond the scope of our analysis. Further, in general, infeasible path elimination (or detection) by itself is a hard problem [4]. However, compared with the static analysis solutions proposed in the literature [10, 30, 29], this work makes an attempt to reduce the search candidates using a pruning technique.

5.3.2 *Compiler Specific Code Pruning*

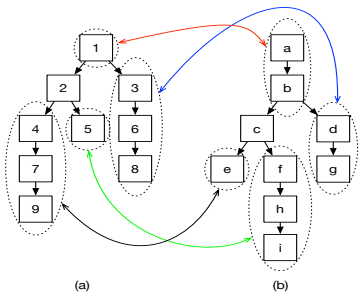

Figure 7: A sample signature (a) and target (b) functions, where the lines indicate the matched partial traces and the nodes refer to the basic-blocks.

Based on the compilation option selected, the compiler could include additional code (i.e., code that is not originally written by the programmer) into the compiled binary. For example, there are more than 150 compiler options available for both `gcc` and `MS Visual Studio`, which a programmer can choose from when compiling her program. Some of these options result in adding extra code into the binary (e.g., `stack-smashing protection` or SSP shown in Fig. 6). The code segments in the function prologue and epilogue in Fig. 6 ensure the stack integrity is not violated. These code segments are automatically included by the `gcc` compiler when the option for stack smash protection is enabled.

In similarity matching between the signature and target functions, the additional compiler specific code can introduce noise by changing the code structures and diluting the similar parts, especially when the functions are small. Thus, it is necessary to identify and remove the compiler specific code from the partial traces. However, directly removing some code from a partial trace might lead to incorrect semantic features, as these features are very sensitive to the underlying code semantics.

To this end, we propose a conservative approach to address this problem by generalizing the compiler specific code into some patterns and systematically pruning the partial traces that contain these patterns. That is, instead of removing the compiler specific code from a partial trace, which is error prone, we just remove the partial trace itself if the compiler specific code pattern accounts for the majority (50% or more) of the code. As an initial step, we only consider three types of compiler specific code patterns (i.e., SSP, function prologue and epilogue) that are very commonly observed in the binaries. This list can be further extended when necessary.

5.4 Function Matching

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 shown in Fig. 7 are listed as follows:

$$\mathcal{M}_{\texttt{sig}} : \{\langle \texttt{1} \rangle, \langle \texttt{2} \rangle, \ldots, \langle \texttt{1}, \texttt{2} \rangle, \langle \texttt{2}, \texttt{5} \rangle, \ldots, \langle \texttt{1}, \texttt{2}, \texttt{4} \rangle, \langle \texttt{2}, \texttt{4}, \texttt{7} \rangle, \ldots\}$$

$$\mathcal{M}_{\texttt{tar}} : \{\langle \texttt{c} \rangle, \langle \texttt{b} \rangle, \ldots, \langle \texttt{a}, \texttt{b} \rangle, \langle \texttt{b}, \texttt{c} \rangle, \ldots, \langle \texttt{a}, \texttt{b}, \texttt{c} \rangle, \langle \texttt{b}, \texttt{c}, \texttt{f} \rangle, \ldots\}$$

In BINGO, 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. 7, 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. 7, 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. 7, 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. 7, 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 [29] and [26], 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, all the above 4 types of function matching are needed to cover all possible basic-block structure variances that arise due to architecture, OS and compiler differences. Finally, considering the function model as a bag of partiral traces, Jaccard containment similarity [2] is used to measure the similarity score between two different function models, and is defined below:

$$sim(\mathcal{M}_{sig}, \mathcal{M}_{tar}) = \frac{\mathcal{M}_{sig} \bigcap \mathcal{M}_{tar}}{\mathcal{M}_{sig}} \quad (3)$$

where $\mathcal{M}_{sig}$ and $\mathcal{M}_{tar}$ refer to function models generated from signature and target functions, respectively.

6. EXPERIMENTAL RESULTS

Implementation. In BINGO, we use IDA Pro to disassemble and generate CFGs from the functions in the binary. Partial traces are generated using the Tracy plugin[2], where they are of three different lengths (i.e., $k = 1, 2$ and 3). Finally, to extract semantic features, similar to [29] and [26], we lift the low-level assembly instructions to an architecture and OS independent intermediate language, REIL [12]. All the experiments were conducted on a machine with Intel Core i7-4702MQ@2.2GHz with 32GB DDR3-RAM. Based on our initial experiments, we set $\alpha = 0.01$ for the function coupling score, $w_1 = 1.0$ for the weight of filter 1, $w_2 = 0.8$ for that of filter 2, and $w_3 = 0.5$ for that of filter 3.

Through our experiments, we aim at answering four research questions (RQs) categorized into three major topics, namely *robustness* (**RQ1-2**), *application* (**RQ3**) and *scalability* (**RQ4**):

RQ1. How robust is BINGO in detecting *semantically equivalent* functions across architectures and compilers with code optimization?

RQ2. How robust is BINGO in identifying *semantically similar* functions, across OS, in *wild* binary executables?

RQ3. What are the key real-world applications of BINGO?

RQ4. How scalable is BINGO?

In RQ1, we evaluate the selective inlining algorithm to show its effectiveness. In RQ2, we compare BINGO with BINDIFF [15] and TRACY [10], indirectly showing the merits of using function model of length variant partial traces. In RQ3, we evaluate BINGO in finding real-world vulnerabilities and compare it against discovRE [35] and [29]. Finally, in RQ4, we demonstrate the performance of BINGO by the virtue of the proposed function filtering.

[2]https://github.com/tech-srl/tracy

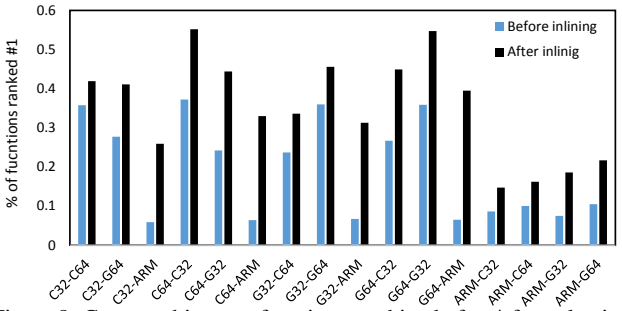

Figure 8: Cross-architecture function matching before/after selective inlining using `coreutils` binaries. Here, **C** and **G** represent compilers `clang` and `gcc`, respectively, and 32 and 64 denote x86 32bit and 64bit architectures.

6.1 Robustness

The robustness of BINGO lies in (1) matching *semantically equivalent* functions, and (2) matching *semantically similar* functions. RQ1 falls under criteria one as we aim to match all functions in binary A to all the functions in binary B, where they both stem from the same source code (i.e., syntactically identical at the source code level) but compiled for different architectures using different compilers and code optimization options. RQ2 focuses more on finding semantically similar functions across binaries that do not stem from the same source code (i.e., syntactically different even at the source-code level with different programming style and conventions). Through answering RQ1-2, we evaluate how good for BINGO to be a search engine that find wild binary executables that share no source-code but perform semantically similar operations.

6.1.1 *Answer to RQ1: Cross-architecture analysis.*

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

Fig. 8 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 matches (after inlining) between binaries compiled for ARM and x86 64bit yield 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 has only 8.

Considering the fact that `coreutils` binaries are relatively small with around 250 functions (on average) per binary, we conduct

Table 2: 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

Table 3: Percentage of functions ranked #1 in intra-compiler (x86 32bit) comparison. Here, **G** and **G'** represent `gcc` v4.8.3 and v4.6 compilers, respectively.

	G0	G1	G2	G3
G'0	0.43	0.36	0.36	0.27
G'1	0.37	0.46	0.48	0.36
G'2	0.4	0.44	0.52	0.41
G'3	0.4	0.44	0.49	0.49

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

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 ranked one in `BusyBox`, while it is 35% for `coreutils` binaries on average across 16 experiments shown in Fig. 8. This is an improvement of 27.5% over the accuracy achieved by [29] for `BusyBox`. However, for top 10 matches (i.e., ranks 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. 8, 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 2 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 2, 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, our selective inlining technique is required to capture the complete semantics in binary analysis, which results in the improved matching accuracy.

6.1.2 *Answer to RQ1: Cross-compiler analysis*

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 very robust, 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 [1]. Further, we also report the individual results obtained by each of these 103 binaries (e.g., `ls`, `true`, and so on) in the `coretuils` suite, for every experiment, in [1].

Table 4: Percentage of functions ranked #1 in inter-compiler (x86 32bit) comparison. 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	-

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 compare to matches between binaries with no code optimization (i.e.,`O0`). That is, high code optimization levels lead to the similar binary code even with diferent compilers, whereas without optimization the influence of the compiler in the binary is very evident. This observation is consistent with [29]. 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.

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 3. 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 3. 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 [1].

Impact of selective inlining. Complying with the results reported in Section 6.1.1, 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. 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. Further, for different types and versions of compiler, in most cases, BINGO attains the best match for around 42% of the functions, which is improved to 72% on average for top 5 matches.

6.1.3 *Answer to RQ2: Cross-platform (OS) analysis*

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.

Table 5: Rankings (top 10 matches) obtained by BINGO for 60 functions 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

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 high-level 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).

In total, we build the ground truth for 60 standard C library functions in `msvcrt` and Table 5 summarizes the results obtained by BINGO. Among these, 11 functions (18.3%) are ranked one, around 51.7% of the functions are within top 5 matches, whiles 83.3% of the functions are ranked within top 10 positions. The results demonstrate the robustness of BINGO in identifying functions that share no code base but with the similar semantics.

Comparison with the state-of-the-art tools. To compare BINGO with the state-of-the-art tools, we repeat the aforementioned experiment using BINDIFF[3], an industry standard binary comparison tool, and TRACY [10], a publicly available academic tool. BINDIFF (v4.1.0), somehow, is unable to correctly match any of the 60 functions in `mcvcrt` to their counterparts in `libc`. Its failure roots back to heavy relying on the program structure and call-graph pattern, which is less likely to be preserved in binaries compiled from completely different source-code bases. Totally, TRACY matches only 27 functions (out of 60) within the top 50 positions. Among the 27 matches, 5 (8.3%) are ranked one (best matches), 23 (38.3%) are ranked within top 5 positions. Notably, when matching binaries compiled for different architectures, TRACY totally fails. Due to the fact that BLEX [13] is not publicly available[4], we cannot compare BINGO with dynamic analysis based function matching.

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.

6.2 Answer to RQ3: Applications

As part of the on-going research, we leveraged on BINGO to perform vulnerability extrapolation [30] — 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 [30, 29]. However, there is few evidence of using such technique to hunt real-world vulnerabilities. One reason could be these tools cannot handle large complex binaries. Due to the program structure agnostic function modeling and effective filtering, BINGO is capable of handling large binaries. Hence, we evaluate the practicality of our tool 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.[5]

Matching Known Vulnerabilities: To evaluate whether BINGO is capable of finding vulnerabilities across-platform, we repeat the two experiments reported in [29]. 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 [29], 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. Thanks to filter 2 (i.e., library call abstraction technique) used in our filtering algorithm, we identify four library calls (i.e., 'string manipulation' : `strlcpy`, 'time' : `time`, 'Input/output' : `ioctl` and 'internet address manipulation' : `inet_ntoa`) that match the vulnerable functions across OS. 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 [29], 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 basic-block 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 basic-block centric matching fails in such conditions, where the program structure is heavily distorted. However, using function model of variant-length partial traces 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 tool [35] fails to identify any of the aforementioned vulnerable functions, as we observe that in `Mingw` version of `openssl`, library functions (such as `memset`, `memcpy`, and `strtol`) are inlined in most cases, leading to the program structure distortion that is not handled by DISCOVRE.

6.3 Answer to RQ4: Scalability

BINGO has demonstrated its scalability through out the experiments. It particular filtering process takes only few milliseconds for

[3] www.zynamics.com/bindiff.html

[4] 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.

[5] We received 4,000USD as the bug bounty for this vulnerability under *iDefense Vulnerability Contributor Program* run by *VeriSign*.

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 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 on all the functions in a binary. In practise, semantic feature extraction is an 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.

6.4 Threats to Validity

The major limitation in selective inlining is that we cannot inline functions that are invoked indirectly (i.e., indirect call). However, this limitation is not particular to BINGO 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) [3]. However, we did not incorporate VSA into BINGO as it involves heavyweight program analysis, which might leads to heavy performance cost. Further, not all floating-point (FPU) instructions are currently handled by our IR language, REIL. However, in the future, we will be adding support for more FPU instructions. Currently, we use the following parameter setup to achieve the results, ($\alpha = 0.01$, $w_1 = 1.0$, $w_2 = 0.8$, $w_3 = 0.5$). In future, we will investigate the impact of different values in other case studies.

7. RELATED WORK

Binary similarity analysis Saebjornsen *et al.* [33] is one of the pioneers in binary code search, where they propose a binary code clone detection framework that leverage on normalized syntax (i.e., normalised operands) based function modelling technique. Jang *et al.* [18] 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. Binary control flow graphs are considered in similarity check. [6] aims to detect infected virus from executables via a CFG matching approach. [25] proposes 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. Besides, [32] and [31] also adopt CFG in recovery of the information of compilers and even authors. Tracelet [10] is presented to capture syntax similarity of execution sequences and facilitate searching for similar functions. The aforementioned syntax based binary similarity matching techniques fail during cross-architecture analysis.

Value based equivalence check Input-output (I/O) and intermediate values can be leveraged for identification of semantic clones, regardless of the availability of source code. Jiang *et al.* [21] regard the problem of detecting semantic clones as a testing problem. They use random testing and then cluster the code fragments according to the I/O samples. [34] presents a method to compare x86 loop-free snippets for testing transformation correctness. The equivalence check is based on the selected inputs, outputs and states of the machine when the execution is complete. Note that intermediate values are not considered in [21][34]. Nevertheless, intermediate values are used to mitigate the problem of identifying I/O variables in binary code. In [19], Jhi *et al.* state the importance of some specific intermediate values that are unavoidable in various implementations of the same algorithm and thus qualify to be good candidates for fingerprinting. According to this assumption, the studies on plagiarism detection [38] and matching execution histories of program [39] are proposed.

Diff based equivalence check BINDIFF [15] 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 [17], 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 [5] models the CFG matching problem as a bipartite graph matching problem. For these tools, the compiler optimization options may change the structure of CFGs and fail the graph-based matching. Recently, BLEX [13] is presented to tolerate such optimization and obfuscation differences. Basically, BLEX borrows the idea of [21] to execute functions of the two given binaries with the same inputs and compare the output behaviors. To relax the difference of two binaries, BLEX uses seven semantic features from an execution (e.g., calling imported library functions).

Bug Detection based on Binary Analysis 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.* [37], 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.* [29] and DISCOVRE [35] 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.

8. CONCLUSION

In this work, we propose the scalable solution of binary code search framework, BINGO, which aims to search similar binary code regardless of the differences in architectures and OS. At modelling aspect, BINGO leverages on the complete function semantics and length variant partial trace based function models to perform the similarity search. The promising experimental results deliver the claim of cross-architecture and cross-OS binary search. Further BINGO has outperformed the state-of-the-art tools like TRACY and BINDIFF. In security application, we also discovered a zero-day vulnerability in Adobe PDF Reader.

References

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

[2] 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*, pages 927–938. ACM, 2010.

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

[4] R. Bodik, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. In *Foundation of Software Engineering ESEC/FSE'97*, pages 361–377. Springer, 1997.

[5] 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*, pages 4:1–4:10, 2013.

[6] 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*, pages 129–143, 2006.

[7] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. Technical report, DTIC Document, 2009.

[8] S.-H. Cha. Comprehensive survey on distance/similarity measures between probability density functions. *City*, 1(2):1, 2007.

[9] 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*, 22(5):349–369, 1992.

[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*, page 37, 2014.

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

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

[13] M. Egele, M. Woo, P. Chapman, and D. Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 303–317, 2014.

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

[15] 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*, pages 161–173, 2004.

[16] 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*, pages 321–330, 2008.

[17] 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*, pages 238–255, 2008.

[18] 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*, pages 81–96, 2013.

[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*, pages 756–765, 2011.

[20] 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*, pages 96–105, 2007.

[21] 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*, pages 81–92, 2009.

[22] 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.*, 28(7):654–670, 2002.

[23] 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*, pages 1128–1139. ACM, 2014.

[24] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Recent Advances in Intrusion Detection*, pages 207–226. Springer, 2005.

[25] 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*, pages 207–226, 2005.

[26] 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*, pages 389–400. ACM, 2014.

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

[28] 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*, pages 416–430. Springer, 2015.

[29] 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*, pages 709–724, 2015.

[30] 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*, pages 406–415. ACM, 2014.

[31] 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*, pages 21–28, 2010.

[32] 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*, pages 172–189, 2011.

[33] 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*, pages 117–128. ACM, 2009.

[34] 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*, pages 305–316, 2013.

[35] 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.

[36] 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*, pages 331–340. ACM, 2015.

[37] 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.

[38] 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*, pages 111–121, 2012.

[39] 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*, pages 197–206, 2005.