SRJ-BinGo: Cross-Architecture Cross-OS Binary Search

ABSTRACT

Code search in binary executables has received considerable attentions in recent years due to its impactful applications, e.g., finding 0day vulnerability in proprietary binary by matching the known vulnerability from open source software. However, developing a generic binary code search tool is a challenging task because of the gigantic syntax difference in binaries due to compiler optimizations, 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 to use static analysis to generate a comprehensive set of features, which captures both the semantic and structural information of the binary code. Feature hashing technique is then adopted to perform binary matching. Furthermore, we introduce partial trace based function models to identify semantically similar functions across architectures and OSs. The experimental results show that BINGO can search for similar functions, at machine-code level, across OS boundaries, even with the presence of program structure distortion, in a scalable manner. Using BINGO, we have found two RCE 0-day vulnerabilities from the off-the-shelf commercial software Adobe PDF Reader.

1. INTRODUCTION

Nowadays, many tasks of software development or security require the code search in binaries. An effective code search technique can greatly help recommend similar code solutions, identify binary semantics, and also reveal 0day vulnerabilities. Especially, among the various vulnerability detection techniques [28], binary code search is a fast yet accurate solution that preludes a further manual check of security experts.

Traditional source code search relies on similarity analysis of some representations of source code, e.g., approaches based on token [19], abstract syntax tree (AST) [17] or program dependency graph (PDG) [12]. All these representations capture the structural information of 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 levels or even obfuscation) and limited availability of high-level program information. These factors have a huge influence on the assembly instructions and their final layout in the compiled binary executable. Despite the huge challenges encountered, various approaches have been proposed to detect the similar binary code, by using static, dynamic or hybrid [any example for hybrid??] analysis.

In general, existing studies mostly make use of the syntax and structural information to identify the similarities in binaries — in a way as we did for the source code search. Jang *et al.* [15] propose to use n-gram models to get small linear snippets of assembly instructions, and normalize them to tolerate the variance in names across different binaries. Matching by linear sequence of assembly instructions is not resistant to structural differences. Thus n-grams are combined with *graphlets* [20] (small non-isomorphic subgraphs of the control-flow graph) to take into account the structural information for comparison. However, performing graphlet matching (subgraphs isomorphism) is computationally costly. As a remedy, the concept of *tracelet* [6], which refers to a partial and continuous execution trace, is proposed to tackle the efficiency issue.

Static analysis is scalable, but not accurate or robust as it relies too much on the structural information of basic blocks. Another line of research adopts dynamic approaches [18, 27, 16, 9], which inspect the invariants of input-output or intermediate values of program at runtime to check the equivalence of applications. These approaches can be effective, but they face challenges from two aspects: the difficulty in setting up the execution environment to dynamically execute, and the scalability issue that prevents large-scale analysis.

To better understand the mainstream binary function matching techniques, Table 1 summarizes the state-of-the-art binary techniques proposed in the literature. TRACY [6] is a pure syntax based function matching technique that uses n-tracelet (basic blocks of n length), which is architecture- and OS-dependent. COP [22] is a plagiarism detection tool that leverages on the theorem prover to search for semantically equivalent code segments, hence, not very scalable and does not support cross-architecture and cross-OS analysis technically they can but ddn't show in the result, so how to decide?. The bug search tool in [23] supports the cross-architecture analysis via the invariants of bug signatures. Since extracting semantic features in quite expensive, [23] is not very scalable and has very limited support for cross-OS analysis due to ignoring the interactions between the OS and the binary. Most importantly, COP and [23] use a pairwise basic-block similarity search as an initial step to identify candidate functions that are similar to the signature. Hence, COP and [23] has an implicit assumption that at least one basic-block is preserved in the signature and target binaries. These assumptions may be too restrictive for real-world binaries especially, when the signature and target binaries (or fucntions) do not share the same code base. Finally, BLEX [10] is the latest dynamic function matching technique using seven semantic features, which captures the precise semantics of the binary code. As a dynamic analysis tool, BLEX is not scalable and, due to implementation limitations, does not support cross-architecture and cross-OS analysis.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

Table 1: Comparison of existing techniques

Tool	Cross-architecture	Cross-OS	Precise semantics	Technique	Similarity matching	Scalable
BLEX [10]	Limited	Limited	Yes	Dynamic	Whole-function matching	No
Tracy [6]	No	No	No	Static	Partial trace (fixed length) matching	Yes
CoP [22]	No	No	No	Static	Pairwise basic-block matching	No
Bug search [23]	Yes	Limited	No	Static	Pairwise basic-block matching	Yes
BinGo	Yes	Yes	Yes	Static	Partial trace (variable length) matching	Yes

In all the techniques listed in Table 1, it is implicitly assumed that both the signature and target binaries share (or forked from) the same code base. However, this assumption is too restrictive and have limited real-world applications [1].

TODO: Need to metion about pre-filtering, where no one has done that except ndss 2016. Also include ndss 2016 to the table and briefly discuss it

beyond this point I need to review - Mahin

A fundamental program in cross-architecture cross-OS binary code search lies in how to mitigate the effects of the architecture, platform and compilation options, which lead to the different structures of basic blocks — different representations used for matching. Thus, a pairwise basic-block similarity search in CoP is not resilient to the change in architecture, OS or compiler options. The capability of [23] in binary code search mainly comes from the domain knowledge of bug signature that is independent from the architecture and OS choice. Tracy [6] is the first study that attempts to address the problem due to the granularity of basic blocks in CFG. The idea is not to use single basic block for matching, but to use the n-length partial trace of CFG for matching[2]. Owing to the fixed length of tracelet and the strict syntax based function matching, it is not able to be architecture- and OS- independent. Last, the dynamic tool BLEX relies on the input-output invariant, not affected by the basic block structures. However, dynamic tools are usually not scalable.

The encountered dilemma in binary search is whether we should use basic block structure to scale the analysis at the cost of losing the accuracy of cross-architecture cross-OS search. To answer this, it is critical to explore how to use the basic block structures without the side-effect due to architecture, OS, compiler difference.

The basic idea of this study is to find the program context for the given and target binaries while attaining the basic block structures. As the CFG of binary code relies much on the compilation options, inlining the functions used by the given binary snippet is helpful for recovering the semantics of the program. In other words, after inlining the commonly-used functions, program with the similar high-level semantics will show more structural similarities in their basic block structures.

Under this motivation, in this paper, we design an approach to address the problems aforementioned with the following merits. First, it should **tolerate** binary code of similar semantics in a relaxed way, rather than only find binaries that share the same source code base. Second, it should be **scalable** to real world binaries, avoiding the pair-wise graph matching or subgraph isomorphism check. Last but not least, it should be architecture- and OS- **neutral**, and be resistant to the variances that arise due to the differences in compiler type and optimization level.

Technically, we propose a binary search framework, named BinGo, which combines a set of key techniques. First, to achieve architecture and OS independency, we lift the low level machine code up to an independent intermediate representation (IR) (i.e., REIL [8]). To recover the semantics of the binary, we do the selective inlining for the callee functions that are highly coupled onto other caller functions. Note that selective inlining is needed to avoid the problem of code size explosion due to inlining all functions called.

Second, to mitigate the problem of granularity of basic blocks, for each function, we generate *partial traces*[3] of various lengths instead of using single basic block structure or fixed length of partial traces. Once the partial traces are generated, we construct several *function models* for each function, where a function model is a combination of partial traces of various lengths (see Section ?? for details).

Last, for scalability issue, after building a number of partial trace based function models for a binary segment, we extract various features from the function models mentioned in the second step. By leveraging on *feature hashing* technique [14], a fixed length feature vector is generated for each function model. Hence, given a query function, we search for semantically similar or equivalent functions from the set of function models that we have in the database.

We evaluate BinGo on a number of real-world binaries with containing hundreds of thousands of functions. The experimental results show that BinGo can effectively perform cross-architecture and cross-OS matching on these binaries. BinGo further comprehensively outperforms those the state-of-the-art tools, Tracy [6] and BinDiff [11] for the same tasks. Further, we also show that recent techniques such as [23], fails when program structure is distorted or cross OS analysis, while BinGo can handle such cases swiftly. Last but not least, using BinGo, we found two RCE 0-day vulnerabilities from the off-the-shelf commercial software Adobe PDF Reader.

To sum up, we propose an effective binary code search engine which combines static analysis and machine learning. To our best knowledge, our work is the first attempt towards the discussion of the role of selective inlining in recovering binary semantics, and a general solution of cross-architecture cross-OS binary search.

2. BACKGROUND AND PROBLEM STATEMENT

In this section, we give a motivating example of binary matching regardless of architecture and OS differences, and state the challenges of binary code search in finding the semantics and structures for this example. Last, we explain the basic idea of our proposed solution.

2.1 Motivating Example

A binary program consists of a set of functions, where each function is a (directed) graph of basic-blocks, i.e., CFG. The instructions in a function are systematically grouped into several basic-blocks, which are considered as building blocks of binary program and the representation analyzed by most static binary code search tools.

Definition 1. (**Basic-block**) A sequence of assembly instructions without any jumps or jump targets in the middle, where a jump target starts a block, and a jump ends a block.

The same source code may have different basic block structures after compilation. For example, Fig. 1(a) shows the real-world

[1]This assumption is justifiable for only CoP as it is a plagiarism detection technique and for the rest, it is

[2]In particular, 3-tracelet is used in [6].

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

case of the infamous heartbleed vulnerability (CVE-2014-0160) , while Fig. 1(b) and (c) show the vulnerability at binary code level, 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 deeper inspection suggests that in `mingw` version the library function `memcpy` is inlined; while in `gcc`, it is not. Given the `gcc` version as a signature, we may miss the vulnerability in `mingw` version due to basic-block *splitting* (or *merging*). Hence, basic-block centric based function modeling is not robust enough to address real-world problems.

2.2 Challenges for Syntax-based Matching

Syntax is the most direct information that can be used for matching. Existing approaches mostly have attempted to use instruction patterns, e.g., n-gram [15], graphlet [20] and tracelet [6] patterns of instructions in the basic blocks, to match binaries. For matching accuracy, these approaches rely on syntax information (e.g., instruction type) for matching. However, relying on instruction syntax assumes that the systems are running in the same environment, *i.e., the same architecture, same OS, and compiled with same compiler and compiler options*. Though these techniques work well for the given environment (e.g., cases presented in [6]), it fails for cross-architecture and cross-OS analysis, where syntax dramatically changes for the different enviroments that it runs on [23].

Clearly, the key challenge for syntax-based matching is: **C1: There is no consistent binary syntax representation in different environments.** For example, the two binary code segments in Fig. **??**, one for ARM and one for x86 32bit, both represent the behaviours of stack frame setup in function prologue. However, by relying on the direct syntax representation, it is hard to match them.

2.3 Challenges for Semantics-based Matching

To make the matching tolerant to the syntax difference in binaries, semantics-based matching has been proposed [22, 23], which use the machine state transition to represent the semantics of the binary. However, there are three challenges in adopting the semantic information in binary matching.

C2: How to choose the segments of binary for the semantics calculation. Existing approaches [22, 23] assume that basic-block structure is preserved across binaries, thus the matching should be basic-block centric. Based on this, models or features extracted from the given binary segment at basic-block level are compared with the counterparts extracted from target functions in a pairwise way. In practise, the strong assumptions behind are too restrictive to be applied for real-world cases. The following quote from [23] 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."*

C3. How much contextual semantics is needed to complement the basic-block centric (syntactic and structural) matching. Some existing techniques has notice the problem of basic-block, and propose to *blindly* inline all the user-define functions (non-inline the library functions). The blindly inlining strategy would lead to such a problem of code size explosion that the bloated functions are hard to analyze, incurring heavy overhead. **??Mahin, can you give an example or some citation.**

2.4 Proposed Solution

To overcome C1, we propose to convert the instructions into the IR for cross-architecture and cross-OS analysis. To mitigate C2, we borrow the idea of tracelet used in TRACY [6]. Different from the fixed length of 3-tracelet, we use a variant length of partial traces to mitigate the affect of basic-block granularity. Last, to address C3, we propose a selective strategy to strike a balance between the needed contextual semantic and the overheads due to inlining.

DEFINITION 2. **A partial trace** is an instruction sequence obtained from basic-blocks that lie adjacent to each other along a program execution path. Partial traces can be of different length, where partial trace of length k (called, Length-k partial trace) denotes a partial trace that contains the instructions obtained from k adjacent basic-blocks that lie along a program execution pat

DEFINITION 3. **Complete (or true) Semantics** refer to the contextual semantics that the function under analysis and

The existing static binary function matching techniques, in general, fail to take into account the true (or complete) semantics of the functions under investigation. That is, each function in a binary is analyzed in isolation, where the semantics of the callee functions (be it user-defined or library function) are not considered when generating the caller function semantics as they are assumed to be two different functions. However, this assumption is intuitive and valid until we understand the caller-callee relationship (or dependency). For example, assume a user-defined function that manipulates string literals invokes the `strcpy` library function, and in order to capture the true semantics of the caller, it is essential? to inline `strcpy` function at the call site, as the caller function is involved in string manipulation, where it leverages on utility fucntions, such as `strcpy`, provided by the C runtime library to simplyfy its operations, hence, the semantics of the utility functions need to be considered as part of the caller fucntion semantics. Similarly, in another occasion, the programmer might implement her own version of `strcpy` function with additional security properties (called, `strcpy_secure`[4]) and invokes it from all the string manipulation functions that need a secure string copy operation, hence, to capture the true caller semantics, the user-defined `strcpy_secure` function needs to be inlined at the call site.

3. BINGO SYSTEM OVERVIEW

BINGO is a scalable binary search engine by combining static program analysis techniques with feature hashing techniques. Given a binary function, BINGO will return similar functions from the target repository of binary functions, ranked based on their semantic and behavioral similarity. As shown in Fig. **??**, BINGO consists of three major modules, namely, feature extraction module, function model generation module, and machine learning module.

In the feature extraction module, two types of semantic features are extracted from partial execution traces: semantic features and structural features. State-base semantic features (see Section **??**) represent the low-level effects of executing the binary code in terms of machine state (i.e., characterised by register, condition-code flag and memory values) at various program points. Semantic features of API idioms (see Section **??**) capture the OS dependent semantics of the binary code. Secondly, structural features and compiler idioms (see Section **??**) can help to quickly find the binary code with similar program structural or representations. These features complementarily summarize the behaviour of binary code at various granularity levels, providing a comprehensive view to overcome the differences at syntax and structural levels. This is the one of

[4]It is worth noting that `strcpy` is one of the banned function calls by Microsoft and hence, its better to use more secure variants such as `strncpy_s` and `strcpy_s` [1]

the key contributions of this work for achieving accurate yet robust matching results.

In the function model generation module, for each function, a number of *function models* are generated based on different combinations of partial traces. Specifically, partial execution traces of various lengths are combined in different ways to represent the function models that account for structural changes in the compiled code, such as function inlining and outlining introduced by the compilers. For a given function, if partial traces of n different lengths are generated, the function can be represented by $2^n - 1$ different function models.

Finally, the machine learning module applies feature hashing techniques on functions models, generated from each function, where they are put into 'bins' based on their proximity to each other. That is, function models that are similar, in terms of syntax and semantic features, will likely to get into the same bin. Hence, for a given search query, using feature hashing, the appropriate bin is located and the matching functions are obtained from there. This improves searching time.

4. SELECTIVE INLINING

In this section, we explain the approach of selective inlining, which helps to take into account the true (or complete) semantics of the functions under investigation.

Therefore, one might assume that inlining (or inline expand?) all the callee functions at their respective call sites would solve the partial (or incomplete) semantics problem present in existing binary function matching techniques. However, this is simply not going to work due the several reasons such as: (1) heavy inlining may lead to code size explosion [5], where performing similarity matching on bloated functions incur heavy overhead and hence, not scalable to real-world binary analysis, and (2) not all the callee functions (both user-defined and library functions) are closely related, in-terms of functionality, to the caller function, hence, inlining such functions might dilute the core functionality of the caller function, which in return, leads to poor matching results.

Hence, there needs to be a systematic way in deciding the target functions to be inlined. Interestingly, this is a typical challenge faced by the compilers, where they have to decide what functions need to be inlined, during the compilation process, in order to optimize the binaries for maximum speed or minimum size [5]. However, we both have different set of objectives and challenges, where compilers have access to high-level programming constructs and their inlining decision is influenced by various factors such as user-defined `INLINE` pragmas, target function body-size, call-size (i.e., overhead required to invoke the function), saving estimation (i.e., estimated shrink in binary after inline), effects on caching, paging and register pressure, whereas we need to make the inline decision with the minimum information available from the stripped binaries and are more concerned about capturing the true semantics of the functions while maintaining the scalability.

4.1 Caller-callee relationship patterns

In contrast to compilers, we rely on the caller-callee relationship to arrive at the inline decision. To this end, we have identified seven commonly observed caller-callee relationship patterns and they are summarised in Fig. 2. In the figure, incoming and outgoing edges represents the incoming and outgoing calls to adf from the function, respectively.Fig 2(a) depicts the scenario where callee is a user-defined (UD) function that is referred by (i.e., incoming calls) many other UD functions and this function itself refers to many other UD and library functions. This pattern indicates that callee is likely to be a dispatcher function (e.g.,) and hence, not suitable to be inlined within the caller.

Fig 2(b) depicts the scenario where the called UD function is referred by many other UD functions and on the other hand, this function refers to several library functions and a very few (or zero) UD functions. This pattern suggests that the callee is behaving as a utility function and hence, most likely to be inlined depending on the number of other UD functions 'referred to' by this callee (lower the reference to other UD functions, better chance of inlining). Fig 2(c) and (d) are variants of (b), where they have zero reference to other UD functions (an ideal candidate for inlining), however, in Fig 2(d), all the invoked library functions are of termination type and hence, not inlined, whereas, Fig 2(c) is inlined as the majority of the invoked library functions are not of termination type. Here, termination type refers to the library functions that leads to exception or program termination (e.g., `exit`, `abort` and `error`)

Fig 2(e) depicts the recursive relationship between the caller and callee and hence, inlined. It is important to note that the recursive functions are, unlike compilers [5], inlined only once. Finally, Fig 2(f) depicts the direct invocation of library call by the function under investigation, where only the most common library functions (e.g., `memcpy` and `strlen`) are inlined. To this end, we have identified over 60 library functions, from both Linux (`libc`) and Windows (`msvcrt`), to be inlined. However, we don't see any limitation in adding more library functions to this list.

4.2 Inline decision algorithm

From the discussions above, it can be seen that Figures 2(c), (d), (e) and (f) have a clear criteria in deciding whether a callee (both user-defined and library function) is inlined or not. However, for Figures 2(a) and (b), there needs to be a systematic decision making procedure and hence, we borrow the *coupling* concept from the software quality and architecture recovery community. That is, we view the caller-callee dependency as coupling between two software packages and leverage on the software package instability metrics (we refer to it as *inline metric* in this work) to decide whether a callee in inlined or not. It is given by the following formula:

$$\alpha = \frac{\lambda_e}{\lambda_e + \lambda_a} \tag{1}$$

Where λ_a represents the number of UD functions refer to the callee and similarly, λ_e represents the number of UD functions referred to by the callee. The lower the value of α, more likely the callee is inlined. For example, when the callee doesn't refer to any other UD functions (i.e., $\lambda_e = 0$), it is assumed to be behaving as a utility function (where, $\alpha = 0$) and hence, inlined. It is important to note that when calculating λ_e, we only consider the UD functions invoked by the callee as the invocation of library functions doesn't influence a UD function behaving as a dispatcher or a utility function.

DEFINITION 4. (**Software coupling**) *we need to formally define software coupling and its analogous to caller-callee dependency relationship*

The proposed selective inlining algorithm is presented in Algorith 1. As first, if the callee is one of the selected library functions, 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 (I_u^f) and the library and UD functions that are referred to by the callee (O_l^f and O_u^f, respectively) are identified (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 inline metric is calculated and if its below some threshold value t, the callee is inlined else not (lines 13-24). This recursive procedure is continued until all the related functions to analysed.

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**
  // 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** $|O_u^f| == 0$ && $O_l^f \backslash \mathcal{L}_t == 0$ **then**
12 **return**
13 **else**
14 $\lambda_a = |I_u^f|$
15 $\lambda_e = |O_u^f|$
16 $\alpha = \lambda_e/(\lambda_e + \lambda_a)$
  // lower the α, f is likely to be inlined into $\mathcal{F}$
17 **if** $\alpha >$ threshold t && `notRecursive`$(\mathcal{F}, f)$ **then**
18 **return**
19 **else**
20 $\mathcal{F}^I \longleftarrow \mathcal{F}$.`inline`$(f)$
21 **if** $|O_u^f| > 0$ **then**
22 `SelectiveInline`$(f, O_u^f, \mathcal{L}_s, \mathcal{L}_t)$
23 **else**
24 **return** $\mathcal{F}^I$
25 **return** $\mathcal{F}^I$

5. EFFECTIVE PRE-FILTERING

In the real-world scenario, to effectively search for a semantically similar function from a pool of several hundred thousand (or more) target functions compiled for various architectures and operating systems using different types of compilers with varying optimization levels, there has to be a robust and scalable pre-filtering process in place such that the candidate target functions are identified within no time.

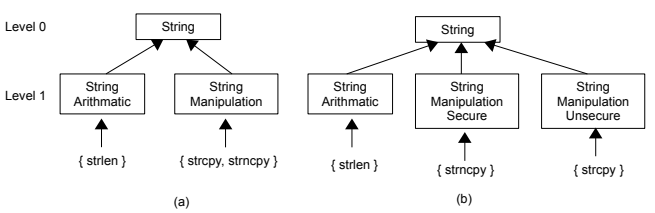


Figure 3: System API Abstraction Levels

To make BINGO capable of analyzing cross-OS binaries, we use a multi-level abstraction function `abstractSystemAPI` at line 22 to abstract system APIs based on their type. For example, the system API (or library call) `strlen` deals with string objects, and hence the API can be naturally abstracted to 'string' type. To this end, we use two levels of granularity (i.e., level 0 and 1) to abstract the system APIs, where each abstraction level play a different role in BINGO — abstraction level 0 abstracts the system API to their basic type, whereas level 1 provides more meaningful information about the API. For example, using our abstraction function, the system APIs `strlen`, `strcpy` and `strncpy` can be abstracted into two levels shown in Fig. 3.

Based on the requirement of the analyst, there may be several ways to abstract a system API. As shown in Fig. 3(b), abstraction level 1 can be more expressive in providing more meaningful information about the API or it can be limited as in Fig. 3(a). One of the key applications of abstraction level 0 is that it is mostly used in pre-filtering process, where if a signature involves string manipulation operations then it is wise to quickly retrieve the target programs that also involve string manipulations. Similarly, abstraction level 1 is used to precisely match the signature with the target programs, where it can be used to specify additional constraints for the matching process. Level 1 abstraction is quite useful for vulnerability signature matching, e.g., the analyst can remove functions, from the filtered target programs, that use secure system API (e.g., `strncpy`, `__strncpy_chk`, etc.[5]), which are less likely to contain an exploitable vulnerability.

5.1 Cross-architecture, OS and compiler pre-filtering algorithm

To this end, in BINGO, we have proposed a across-architecture, OS and compiler friendly pre-filtering algorithm that can filer the candidate target functions in a scalable fashion.

Algorithm 2: Pre-filtering function

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

Algorithm `Prefilter()`
 $\mathcal{S} \longleftarrow \{\}$
 foreach function t in $\mathcal{T}$ **do**
  $\mathcal{L}_f \longleftarrow$ `getLibFuncList`(f)
  $t_{in} \longleftarrow$ `getInlinedFunc`(t)
  foreach libcall l in $\mathcal{L}_f$ **do**
   s^t += w_1 * `getLibFuncNameSim`(l, t)
   $l^O \longleftarrow$ `getOpType`(l)
   s^t += w_2 * `getLibFuncOpTypeSim`(l^O, t)
   $l^I \longleftarrow$ `getInstType`(l)
   s^t += w_3 * `getLibFuncInstrTypeSim`(l^I, t_{in})
  s^t += w_4 * `getFuncInstrTypeSim`(f, t_{in})
  $S[t] = s^t$
 $\mathcal{S}^s \longleftarrow$ `sortCanditTargetFunc`(S)
 $\mathcal{T}_c \longleftarrow$ `topNTargetFunc`$(\mathcal{S}^s, \mathcal{N})$
 return $\mathcal{T}_c$

6. SCALABLE FUNCTION MATCHING

In BINGO, we leverage on a partial trace based function modeling, which is more flexible in terms of granularity, compared with basic-block centric function modeling techniques [23, 22]. That is, for each function, we generate partial traces of various lengths (called, *k-length partial traces*), where by varying the length, the semantics of the function is captured at various granularity levels. Our partial trace extraction is based on the technique proposed by David *et. al.* [6]. We omit the algorithm and explain the results using one example. Fig. 4 depicts a sample CFG of a function and the extracted 2-and 3-length partial traces (i.e., for $k = 2, 3$). From the partial traces it can be see that the original control-flow instructions (`jnx xxx`, `jb xxx` and `jb xxx`) are omitted as the flow of execution is

[5]With `FORTIFY_SOURCE` compiler feature, whenever possible, `gcc` tries to uses buffer-length aware replacements for functions like `strcpy`, `memcpy`, `memset`, `gets`, etc., which are more secure.

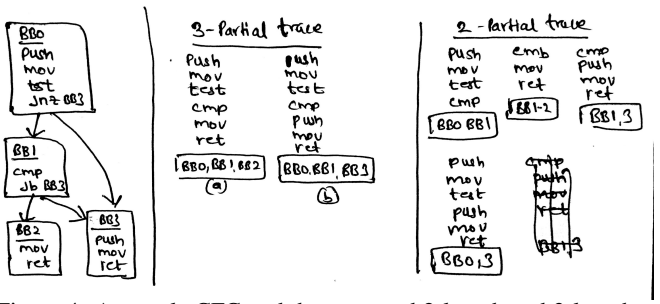


Figure 4: A sample CFG and the extracted 2-length and 3-length partial traces

already determined. Note that the feasibility of the flow of execution is not considered when generating the partial traces. However, later in Section 6.2.1, we show how the partial traces that are *infeasible* are identified with the help of a constraint solver and removed from our analysis. Here, infeasible partial traces refer to program execution flows that are not possible during any concrete execution.

6.1 Semantic Feature Extraction

Similar to [23] and [22], a set of symbolic formulas (called, *symbolic expressions*) are extracted from each partial trace, where they capture the effects of executing the partial 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* [23, 22]. That is, if a machine state, before executing a partial trace, is given by $\mathcal{X} = \langle mem, reg, flag \rangle$ then the machine state, just immediately after executing the partial trace, is given by $\mathcal{X}' = \langle mem', reg', flag' \rangle$, where $\mathcal{X}$ 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 signature and target functions, one can leverage on a constraint solver, such as Z3 [7], to calculate the semantic equivalence between the symbolic expressions extracted from the signature and target functions. However, using a constraint solver to measure semantic equivalance is very expensive [22] and not scalable for real-world scenarios, where each signature function might have to be matched against several hundred thousand (or more) target functions and hence, not practical. However, to tackle the scalability issue, in [23], a machine learning based function similarity matching technique is proposed. In which, Input/Output (or I/O) samples are generated from the symbolic expressions and are fed into the machine learning module to find semantically similar functions. Here, I/O samples are generated by randomly assigning concrete values to the pre-state variables in the symbolic expressions and the corresponding values for the post-state variables are obtained such that the concrete input and output values constitute to the I/O samples.

Unfortunately, one of the major drawbacks in randomly generating I/O samples is that the dependency among the symbolic expressions are ignored. For example, consider the following 2 symbolic expressions extracted from a single partial trace:

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

$$\texttt{ecx}' \equiv \texttt{edx} + 4 \quad (3)$$

where symbolic expression 1 (Eq. 2) has one pre-state (i.e., `edx`) and a post-state variable (i.e., $\texttt{zf}'$), similarly, the symbolic expression 2 (Eq. 3) also got 1 pre- (i.e., `edx`) and a post (i.e., $\texttt{ecx}'$) state variable. Most importantly, it is observed that these two symbolic expressions are dependent on each other through a common pre-state variable `edx`. However, in randomly generated I/O samples (as in [23]), `edx` can be assigned two different values in each symbolic expression, which is undesirable and may lead to inaccurate semantics.

Hence, it is essential to ensure that the concrete value obtained for pre-state variable `edx` satisfies the both symbolic expressions. That is, for $\texttt{zf}'$ to be `true`, `edx` can only take the concrete value `1`, hence, $\texttt{ecx}'$ is forced to take the concrete value `5`. From this example, it is evident that constraint solver is required to find appropriate values for pre- and post-state variables, as it takes all the symbolic expressions, extracted from the same partial trace, into consideration when arriving at the solution. To this end, we leverage on Z3 constraint solver to generate I/O samples from each partial trace such that all the symbolic expressions are satisfied. However, it is important to note that using constraint solver to generate I/O samples are much more scalable than using it to prove the equivalence of two symbolic expressions. That is, I/O sample generation is a one-time process (and can be easily parallalized), where it needs to be generated only once for each partial trace; on the other hand, for equivalence checking (as in [22]), constraint solver is used every single time a partial trace is compared with another one and hence, not scalable. Note that if there are more than one concrete value for any pre- or post-state variable such that all the symbolic expressions are satisfied, we set a limit to $\mathcal{N}$ such possible values, where following the observations reported in [10], $\mathcal{N}$ is set to 3 in our design.

6.2 Noise Reduction

In this section, we explain the two key measures taken, by BINGO, to reduce the impressions that might arise due to using less precise I/O samples based function *similarity* matching in contrast to more precise, but unscalable, function *equivalence* matching.

6.2.1 Pruning of partial traces

BINGO is a static analysis based tool and hence, it is relatively difficult (or impossible) to identify all the partial traces that are infeasible in practice. In BINGO, we prune the obvious infeasible partial traces with the help of a constraint solver. That is given the symbolic expressions extracted from a partial trace, we leverage on the constraint solver to determine whether all the constraints present in the symbolic expressions can be satisfied, if not, that partial trace is considered infeasible. Fortunately, as we are already using constraint solver to extract I/O samples from the symbolic expressions, we don't need an additional effort to identify the infeasible partial traces. That is, if the constraint solver is unable to find appropriate concrete values for the pre- and post-state variables , the partial trace from which the symbolic expression are extracted is consider infeasible.

Unfortunately, some partial traces, for which the solver is able to generate models (or I/O samples), might be infeasible during real world 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, infeasible path elimination (or detection), in general, by itself is a hard problem [**?**]. However, comparing to the static analysis solutions proposed in the literature [6, 24, 23], this work makes an attempt to reduce the search candidates using pruning technique.

6.2.2 Removal of additional code included by the compiler

Compiler idioms capture the compilation information, via code patterns commonly included for a specific compilation option during compilation process. These idioms help to identify the additional code included by the compiler during compilation process. For example, there are more than 40 compiler options available for `gcc` that a programmer can choose from when compiling the code,

and most of the features insert additional code into the compiled binary. One such example is SSP. Interestingly, some compiler idioms help to understand more about the program. For example, the presence of SSP in selected functions (i.e., `-fstack-protector` compiler feature) in a program indicates that these functions have buffers larger than 8 bytes. Compiler idioms are architecture- and OS-specific, however, their high-level meanings are unified across architectures and OSs.

6.3 Function Matching

One of the key advantages of function model is that it enables *n-to-m*, *1-to-n*, *n-to-1* and *1-to-1* matches across the signature[6] and target functions[7], eliminating the impact of optimizations and differences of compilers.

n-to-m Partial traces of the length $n(\in \mathbb{Z}_{>1})$ generated from the signature function are matched against the partial traces of the length $m(\in \mathbb{Z}_{>1})$ generated from target function. For example, from Fig. **??**, partial trace matches: $\langle 3, 6, 8\rangle$ and $\langle d, g\rangle$.

1-to-n Each basic-block (i.e., partial trace of the length 1) in the signature function is matched against the partial traces of $n(\in \mathbb{Z}_{>1})$ generated from target function. From Fig. **??**, partial trace matches: $\langle 1\rangle$ and $\langle a, b\rangle$.

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

1-to-1 Each basic-block in the signature function is matched against each basic-block in the target function. It is also known as pairwise comparison of basic-blocks. From Fig. **??**, partial trace matches: $\langle 2\rangle$ and $\langle c\rangle$.

Among the four types of matching, *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. On the other hand, *n-to-m* mapping is generally preferred when the signature (target) function is part of a huge target (signature) function and appropriately selected values for n and m will maximize the function similarity. Finally, if none of the aforementioned matching techniques work, we resort to *1-to-1* matching to compare the basic-block similarity in a pairwise way, which is similar to those basic-block centric comparison approaches [23, 22].

In BINGO, the function model matching is not fixed, where, for a given signature function, the searching algorithm will try all possible function models and pick the best one that maximizes the signature-target function similarity. In contrast, the techniques proposed in the literature do not have the flexibility to try out all possible matchings. For example, in tracelet-based modeling [6], the authors recommended that the tracelet size should be larger (e.g., $k > 2$), for both signature and target functions, hence, only *n-to-m* matching is performed, in fact, it is *n-to-n* matching as they consider the same tracelet size for both signature and target functions. Further, in [23] and [22], pairwise comparison of basic-blocks (i.e., *1-to-1* matching) is performed as an initial step to identify potential target functions, which inherently assumes that the program structure is maintained across signature and target functions and at least one basic-block in the target function resembles a basic-block in the signature function.

[6]Signature function refers to the search query function in hand.
[7]Similarly, target functions refer to functions in the target binary pool against which the signature function is matched.

6.4 Locality Sensitive Hashing

In BINGO, we leverage on Locality-sensitive Hashing (LSH) technique to perform the function matching more efficiently and in a scalable manner. Locality-sensitive hashing is an algorithm for searching similar items in large and high dimensional dataset [21] based on the assumption that, if two items are similar, the hashed value of the two items will remain similar. In BINGO, we use Min-Hashing [3] technique to hash the extracted features and generate hash signature for each function model. To this end, we use 1000 (i.e., $n = 1000$) hash functions to generate the signature, which leads to an error of 3.16%. The similarity between two function model is given by this equation:

$$sim(mh_a, mh_b) = \frac{|mh_a[i] = mh_b[i]|}{n} \quad (4)$$

where the jaccard similarity between two function models is approximated by MinHash similarity between the two models.

7. IMPLEMENTATION AND EXPERIMENTATION

The semantic feature extraction engine in BINGO uses REIL [8] intermediate language to lift the low-level assembly instruction to an architecture and OS independent form that enables us to do cross-architecture and cross-OS analysis. Further, we leverage on IDA Pro to disassemble and generate CFG from functions in the binary. In addition, partial traces are generated using the Tracy plugin. In BINGO, we limit partial trace length to 4 (i.e., $k = 4$), that is, from each function, we extract partial traces at lengths 1 to 4. Hence, for each function, BINGO is able to construct 15 ($2^4 - 1$) functions models. All the experiments were conducted on a machine with Intel Core i7-4702MQ @ 2.2GHz with 32GB DDR3-RAM.

Through our experiments, we aim to answer seven research questions (RQs) categorized into four major topics below.

Under **robustness**, we answer these RQs:

RQ1. How robust is BINGO in detecting *semantically equivalent* functions across architectures?

RQ2. How robust is BINGO in detecting *semantically equivalent* functions across compilers with code optimizations?

RQ3. How robust is BINGO in identifying *semantically similar* functions, as a search engine for *wild* binary executables?

Under **accuracy**, we assess the design choices made in BINGO and how they affect the overall accuracy of the tool by way of answering the following two RQs:

RQ4. Do function models affect the accuracy of BINGO?

RQ5. Do behavior similarity features affect the accuracy of BINGO?

Under **application**, we discuss the potential applications of BINGO and briefly explain its application in detecting semantically similar vulnerabilities in closed-source application.

RQ6. What are the key real-world applications of BINGO?

Finally, under **scalability**, we answer the last RQ.

RQ7. How scalable is BINGO?

7.1 Robustness

In this experiment, we aim to evaluate the robustness of BINGO under two criteria: (1) matching semantically equivalent functions, and (2) matching semantically similar functions. The RQ1 and

RQ2 fall 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. RQ3 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). This is a first step towards building a search engine for wild binary executables that share no source-code but perform semantically similar operations.

7.1.1 Answer to RQ1

We conducted two experiments. First experiment is to match all functions in `coreutils` binaries compiled for one architecture (i.e., x86 32bit, x86 64bit and ARM (32bit)) to the semantically equivalent functions in binaries compiled for another architecture. For example, binaries compiled for ARM architecture is matched against the binaries compiled for x86 32bit and x86 64bit architectures, and *vice versa*. In all three architectures, the binaries are compiled using `gcc` (v4.8.2) and `clang` (v3.0) with the optimization level `O2` (used as the default settings in many Linux distributions). Table **??** summarizes the results for 14 largest `coreutils` binaries (e.g., `cp`, `df`, ...) and the last row shows the averages for rest of the 89 binaries . It can be seen that on average 18% of the functions in the largest 14 binaries are ranked 1, this increases to 30.4% for rest of the binaries. Further, around 71% of the large binaries are ranked within top 5 positions, where it is 82% for other small binaries. Interestingly, it can be seen that matching between binaries compiled for ARM and x86 64bit yields higher ranking (within top 5 positions) compared to ARM and x86 32bit binaries. One reason could be, ARM and x86 64bit binaries are register intensive compared to x86 32bit binaries, where in ARM and x86 64bit architectures, there are 16 general-purpose registers and hence, registers are used freely compared to x86 32bit binaries which has only 8.

However, the `coreutils` binaries are relatively small with around 100-150 functions (on avg.) per binary. Hence, to compare the robustness of BinGo in larger binaries with several thousand functions, we repeated the experiment with `BusyBox` (v1.21.1), where functions in x86 are matched with functions in x64. From Table **??**, it can be seen that the results are still comparable with the results of `coreutils` binaries, where around 31% of the functions are ranked 1 in `BusyBox`, where it is around 29% for 103 `coreutils` binaries averaged across six experiments shown in Table **??**. For top 5 positions, `coreutils` binaries achieve better results compared to `BusyBox` (81% vs. 67%), however, this might be due to size of the binary, where in `BusyBox` a function needs to be compared against 2410 functions and in `coreutils` binaries it is limited to several hundreds functions. This illustrates the robustness of BinGo across architectures.

7.1.2 Answer to RQ2

Table **??** summarizes the results obtained by BinGo for different compilers (`gcc` and `clang`) with different code optimizations levels (O0, O2 and O3). Here, '`gcc_O3-clang_O0`' refers to, functions in `coreutils` suite compiled using `gcc` with optimization level 'O3' are matched against functions in `coreutils` suite compiled using `clang` with optimization level 'O0'. From the table, it can be seen that BinGo's performance is very robust in most cases (except when `gcc_O3` is used as a signature), i.e., roughly 90% of the functions are ranked within top 10 positions, which is very promising.

We also observe that, within same compilers, better ranks are achieved when the functions are matched from optimization level O3 to O0 (i.e., highest to no optimization) and the worst ranks are achieved when the matching is inverted, i.e., from O0 to O3. This could be attributed to the fact that in O3 the compiler merges as many functions as possible, using function inlining, and makes the functions very compact. Hence, for one function in O3 there may be many semantically similar functions in O0, where there is no optimization at all (i.e., *1-to-n* matching). However, on the other hand, for several functions in O0 there will be only one function in O3 which may be semantically less similar to all of them individually (i.e., *n-to-1* matching). This behavior is not particular to certain compiler as its observed both in `gcc` and `clang` compilers.

7.1.3 Answer to RQ3

To evaluate BinGo as a search engine for binary programs, we carried out an experiment with open-source and close-source binaries. In real-world, it is too restrictive to assume that the function being searched (or the binary that contains the function) is always open-source. Thus, in this experiment, we choose Windows `mscvrt.dll` as the binary (closed-source) from which the search queries are obtained and Linux `libc.so` as the target binary (open-source) on which the search is carried out. In addition, we also wanted to assess the real-world applicability of BinGo in handling large real-world binaries.

Building Ground Truth. In conducting this experiment, we faced a challenge in obtaining a ground truth. Ideally, for each function in `msvcrt.dll`, we want to identify the corresponding semantically similar function in `libc`, so that we can faithfully evaluate BinGo. Hence, to reduce the biases in obtaining the ground truth through manual analysis or expert advice, we relied on the function names and matched functions based on their names. Further, in building the ground truth, several variants of the same functions are grouped into one family. For example, in `libc`, there are 15 variants of `memcpy` functions are there such as `memcpy_ia32`, `memcpy_ssse3`, ... For example, according to our ground truth, `strcpy` function in `msvcrt.dll` is matched with `strcpy` (or its variants) from `libc`. Owing to the same naming system, matching based on function name is reliable.

Using such approach, we found the ground truth for 64 basic functions in `msvcrt.dll` and Table **??** summarises the ranks obtained by BinGo. Among 64 functions, 10 functions (15.6%) obtained rank 1, around 36% of them are ranked within top 10 positions, whiles 90% of the functions are ranked within top 20 positions. This results is comparable with `BusyBox` results (presented in Table **??**), where 80% of the functions are ranked within top 10 positions and around 91% of all are ranked within top 20 positions. These results demonstrate the robustness of BinGo.

Comparison with state-of-the-art tools. To evaluate the effectiveness of BinGo compared to the state-of-the-art tools, we carried out the aforementioned experiment on BinDiff (www.zynamics.com/bindiff.html), an industry standard binary comparison tool, and on Tracy [6], recently proposed academic solution. Bindiff (v4.1.0), however, is unable to correctly match any of the 64 functions in `mcvcrt.dll` to their counterparts in `libc`. This is due to the fact that BinDiff heavily relies on the program structure and call-graph pattern, which is less likely to be preserved in binaries compiled from completely different source-code base. Tracy managed to match only 27 functions (out of 64) out of which 5 are ranked 1, 18 functions ranked within top 5 positions, 26 functions ranked within top 20 positions and finally, all the 27 functions are ranked within 50 positions. Note that, Tracy completely failed when comparing binaries compiled for different architectures. Unfortunately, we are unable to compare our results with dynamic analysis based function matching too BLEX [9] as the tool is not openly available. Further, we also tried

to ask for the dataset that is used to evaluate BLEX as a search engine (in the paper, it is mentioned that 1000 functions are randomly picked from coreutils suite, but the details of the functions are missing). Unfortunately, we did not get any response from the authors.

7.2 Accuracy

7.2.1 Answer to RQ4

To assess the influence of function modeling module on correctly identifying the target function, we conducted a set of experiments to test various function models. We considered the largest 10 binaries in the `coreutils` suite and repeated the cross compiler experiment, with different code optimization levels. That is, given the functions in a binary compiled using `gcc` (O3), search for semantically equivalent functions in the binary compiled using `clang` (O0). It is noted that the compiler and optimization level choices are made based on the data in Table **??**, where `gcc_O3-clang_O0` yields the lowest rank for cross-compiler analysis. For each binary, we repeated the experiment 15 times trying out all possible function models. In total, 150 separate experiments are conducted and the results are summarized in Table **??**, where for each binary the best and the worst function models along with their ability to rank functions within top 10 positions. From the results, it is evident that function models cannot be fixed. That is, there is no generalized mechanism to fix the number of used function models and work well for all kinds scenarios.

From Table **??**, out of 10 binaries, pairwise basic-block similarity matching (i.e., fucntion model 1) yields optimal results for only 4 binaries (i.e., `df`, `du`, `mv` and `cp`). In addition, correctly identifying the optimal function model considerably improves the outcome. For example, choosing function model 4 for `factor` can improve the results by nearly 20% with respect to worst case scenario of choosing function model 13. These findings indicate that having a proper function model is vital to achieve better searching accuracy.

7.2.2 Answer to RQ5

We also evaluated the influence of structural similarity features on improving the overall ranking. To this end, using the same `coreutils` suite, we repeated the cross-architecture experiment (RQ1) without abstract structural features. The results are summarized in Table **??**. From the table, it can be seen that compare to the results obtained with abstract structural features, summarised in Table **??**, these results are relatively poor. That is, for 10 large binaries, % of functions ranked ranked 1 improved by 96.3% (averaged across all 6 experiments), whereas, for the rest of the binaries, it is improved by 91.2%. These outcomes clearly prove the influence of structural similarity features on searching semantically similar (equivalent) functions.

7.3 Applications

7.3.1 Answer to RQ6

This subsection discusses the possible applications of BINGO and presents the results for one of the applications that we carried out for this paper. BINGO as a search engine, we can leverage on it to improve the efficiency of black-box fuzzing (or fuzz testing). Using BINGO, we can try to match the functions in the fuzzing target with the functions in open-source binaries. Given a match, the analyst will be able to understand the functionality of the fuzzing target and generate fuzzing inputs accordingly as in guided fuzzing. Additionally, BINGO can be useful in reversing engineering proprietary code, where the analyst can find semantically similar functions in the open-source binaries and effectively reverse engineer the binary by leveraging on the knowledge gained through understanding the semantically similar open-source binaries.

Vulnerability Extrapolation: As part of the on-going research, we leveraged on BINGO to perform vulnerability extrapolation [24] — given a known vulnerability code, called *vulnerability signature*, using BINGO, we tried to find semantically similar vulnerable code segments in the target binary. This line of work is receiving more attention from the academic research community [24, 23]. However, there is few evidence of using such technique to hunt real-world vulnerabilities. One possible reason could be these tools cannot handle large complex binaries. By virtue of the complete semantic and structural models, BINGO is capable of handling large binaries, hence, we evaluated the practicality of our tool in hunting real-world vulnerabilities.

Two RCE 0-day Vulnerabilities Found: In vulnerability extrapolation, We found two RCE 0-day vulnerabilities in one 3D library used in Adobe Reader. At the high level, we discovered a network exploitable heap memory corruption vulnerability in an unspecified component of the latest version of Adobe Reader, the root cause is due to lack of buffer size validation and this subsequently allows an unauthenticated attacker to execute arbitrary code with a low access complexity. We used a previously known vulnerability in an input size handling code segment of the same 3D module as a signature, where the signature function consists of more than 100 basic-block. We modeled the known vulnerable function and all other 'unknown' functions in the library using function modeling module in BINGO. Later, using the known vulnerable function model as a signature, we searched for semantically similar functions in the library. BINGO returned two 'potential' vulnerable functions in the library (ranked #1) , where we consider them as variants of the same 'known' vulnerability. These two 0-day vulnerabilities are confirmed by Adobe and currently in the process of coordinated disclosure with the *iDefense Vulnerability Contributor Program* run by *VeriSign, Inc*[8]. We will be able to put relevant technical details after these vulnerabilities are official patched and assigned a CVE identifier number by Adobe.

Matching Known Vulnerabilities: To evaluate whether BINGO is capable of finding vulnerabilities across-platform. We repeated the two experiments reported in [23]. First one is *libpurple* vulnerability (CVE-2013-6484), where this vulnerability appear on one of the Windows application (`Pidgin`) and the counterpart in Mac OS X (`Adium`). In [23], 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 managed to achieve the rank #1 for both cases. Thanks to our semantic idioms, using which we identified the four system APIs (i.e., 'string manipulation' : `strlcpy`, 'time' : `time`, 'Input/output' : `ioctl` and 'internet address manipulation' : `inet_ntoa`) that matched the vulnerable functions across OS. The second experiment is SSL/Heartbleed bug (CVE-2014-0160), we complied the `openssl` library to 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 [23, 22], we achieved the ranks 22 and 24 for `dtls1_process_heartbeat` and `tls1_process_heartbeat` functions, receptively. For Linux to Windows matching, we achieved the rank 4 for both functions. This is due to the fact that `Mingw` significantly modified the program structure and hence, the basic-block structure is not preserved across binaries. That is, the binary compiled using `Mingw` contained 216 more functions

[8] We received, in total, $8,000 as a bug bounty for these two vulnerabilities.

(also 17,114 more basic-blocks) compared to Linux binary. Hence, we can see that basic-block centric matching fails in such conditions. However, using partial trace based function modeling with abstract structural features, we are able to achieve rank 1 for both functions in Windows to Linux matching and *vice versa*.

7.4 Scalability

7.4.1 Answer to RQ7

BINGO is quite scalable, on average, taking only 1.9s to search for a `msvcrt` function, using a function model, from a pool of more than 100,000 `libc` function models. The most costliest operation in BINGO is the semantic feature extraction from functions. For example, it took 4469s to extract semantic features from 2611 `libc` functions — on average, taking 1.7s to extract semantic features from a `libc` function. On the other hand, it took only 1123s to extract semantic features from 1220 `msvcrt.dll` functions (on average 0.9s per function). Hence, depending on the complexity of the function, time for extracting semantic features varies. The major bottleneck in the process is the constraint solving part, where constraint solver are slow in general.

Note that semantic feature extraction can be parallelized as extracting features from one function A is independent from function B. Further, semantic feature extraction is a one time cost. Finally, in hunting real-world vulnerabilities, in our case study, it took roughly 30 mins to extract semantic features from over 1200 3D library functions in Adobe Reader and it took only 2.7s to search for vulnerable signature, which resulted in two zero-day vulnerabilities. Fur the for small binaries such as coretutils, it took on average 229s to match 11,950 functions across binaries (see Table **??**). These results demonstrate the scalability of BINGO.

8. RELATED WORK

8.1 Code Search based on Similarity

Source code similarity analysis CCFINDER [19] was proposed to detect source code clones based on a token-by-token comparison. DECKARD [17] builds the AST and applies local sensitive hashes to search for the similar subtrees of source code. Besides, PDG based clone detection [12] can tolerate some syntactic modifications or small gaps of additional statements, but the detection might be slow.

Binary similarity analysis Jang *et al.* [15] 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. [4] aims to detect infected virus from executables via a CFG matching approach. [20] 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, [26] and [25] also adopt CFG in recovery of the information of compilers and even authors. Tracelet [6] is presented to capture semantic similarity of execution sequences and facilitate searching for similar functions.

8.2 Binary Code Differencing

Value based equivalence check Input-output and intermediate values can be leveraged for identification of semantic clones, regardless of the availability of source code. Jiang *et al.* [18] regard the problem of detecting semantic clones as a testing problem. They use random testing and then cluster the code fragments according to the input and output. [27] 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 [18][27]. Nevertheless, intermediate values are used to mitigate the problem of identifying input-output variables in binary code. In [16], 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 [30] and matching execution histories of program [31] are proposed.

Diff based equivalence check BINDIFF [11] 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 [13], 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 [2] 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 [9] is presented to tolerate such optimization and obfuscation differences. Basically, BLEX borrows the idea of [18] 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).

8.3 Bug Detection based on Binary Analysis

Dynamic analysis like fuzzing or MAYHEM face challenges from two aspects: the difficulty in setting up the execution environment to trigger the vulnerability, and the scalability issue that prevents large-scale detection. Pinpointed by Zaddach *et al.* [29], these dynamic approaches are far from practical application onto highly-customized hardware like mobile or embedded devices. Thus it is difficult for dynamic approaches to conduct cross-architecture bug detection. To address this issue, Pewny *et al.* [23] propose a purely static analysis that does not have to handle the peculiarities of the actual hardware platform except its CPU architecture. The goal of their work aims to detect the vulnerability inside the multiple versions of the same program for the different architectures. Thus, their approach is good for finding clones of the same program due to architecture or compilation differences, not suitable for find more relaxed binary clones among different programs.

9. 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 modeling aspect, BINGO leverages on both the state-based semantic model as well as the structural model 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 find two RCE 0-day vulnerabilities from Adobe PDF Reader.

References

[1] Security development lifecycle (sdl) banned function calls. https://msdn.microsoft.com/en-us/library/bb288454.aspx. Accessed: 2016-02-29.

[2] 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.

[3] A. Z. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29. IEEE, 1997.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 309–320. ACM, 2011.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of massive datasets*. Cambridge University Press, 2014.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] H. Shahriar and M. Zulkernine. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Comput. Surv.*, 44(3):11, 2012.

[29] 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.

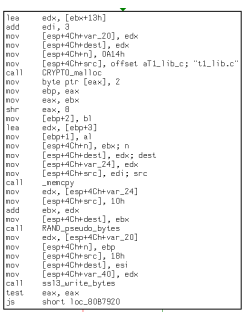
[30] F. Zhang, Y. Jhi, D. Wu, P. Liu, and S. Zhu. A first step towards algorithm plagiarism detection. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pages 111–121, 2012.

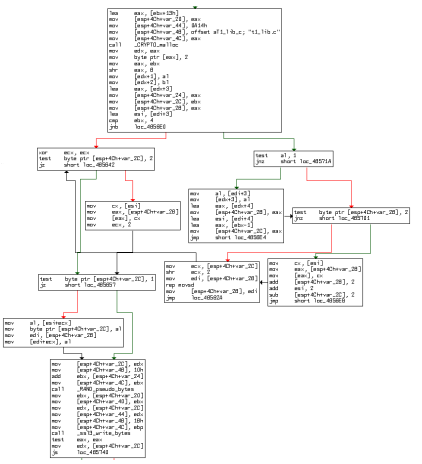
[31] X. Zhang and R. Gupta. Matching execution histories of program versions. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 197–206, 2005.

```
1  buffer = OPENSSL_malloc(1 + 2 +
       payload + padding);
2  bp = buffer;
3  /* Enter response type, length and
       copy payload */
4  *bp++ = TLS1_HB_RESPONSE;
5  s2n(payload, bp);
6  memcpy(bp, pl, payload);
7  bp += payload;
8  /* Random padding */
9  RAND_pseudo_bytes(bp, padding);
10 r = ssl3_write_bytes(s,
       TLS1_RT_HEARTBEAT, buffer, 3 +
       payload + padding);
```

(a)

(b)





(c)

Figure 1: SSL/Heartbleed vulnerability (CVE-2014-0160) appeared as in the (a) actual source code, (b) binary compiled with GCC 4.6 for Linux OS, and (c) binary compiled with Mingw32 for Windows OS

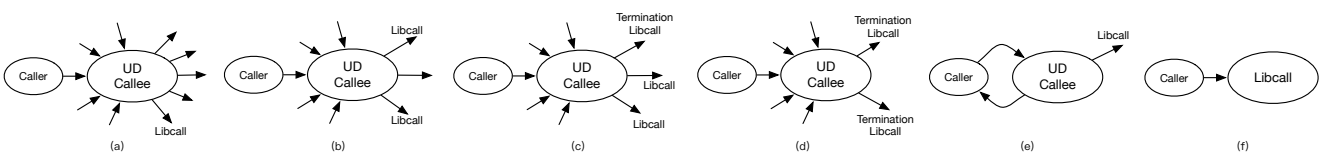


Figure 2: Commonly observed callar-callee relationship patterns, where ‘UD callee’ denotes user-defined called fucntion.