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.

I. 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 [38], 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 [22], abstract syntax tree (AST) [20] or program dependency graph (PDG) [14]. Compared with source code search [26], code search in binary is much more challenging due to many factors (e.g., architecture and OS choice, optimization levels or even obfuscation) that may affect the final instructions and their layout in the compiled binary executable. Despite the huge challenges are encountered, various approaches have been proposed to detect the similar binary code, by using static analysis, dynamic analysis or even both.

Most static approaches make use of syntax and structure information to identify the similarities in binaries. Jang *et al.* [18] 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* [24] (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* [7], 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. Another line of research adopts dynamic approaches [21, 37, 19, 10], which inspect the invariants of input-output or intermediate values of the application 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 detection.

Table I summarizes the state-of-the-art binary matching techniques. BLEX [11] is the latest dynamic function matching technique using seven semantic features, which captures the precise semantics of the binary code. It allows cross-architecture and cross-OS analysis by design. However, BLEX is based on Intel's PIN tool, and hence does not support architectures other than Intel. It is also not scalable to real-world binaries, e.g., it took 57 CPU days to extract features from 1140 binaries. Rest of the techniques in Table I are based on static analysis. TRACY [7] is a pure syntax based function matching technique, which is architecture- and OS-dependent. Similarly, COP [28] 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. Finally, the bug search tool [30] supports the cross-architecture analysis. Since different OSs use different ABI (Application Binary Interface), [30] has very limited support for cross-OS analysis because of ignoring ABI in the analysis. In addition, one major limitation of [30] is that it is built on the assumption that the control flow graph (CFG) structures are preserved in different architectures. This assumption hinders the application of this tool on binaries that stem from the same code base but compiled for different architectures and OSs. Further, COP and [30] use a pairwise basic-block similarity search as an initial step to identify candidate functions that are similar to the signature. COP has the 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 (as discussed in Section II).

A dream tool of code search in binary is desired to address all the problems aforementioned with 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.

TABLE I: Comparison of existing techniques

Tool	Cross-architecture	Cross-OS	Precise semantics	Technique	Similarity matching	Scalable
BLEX [11]	Limited	Limited	Yes	Dynamic	Whole-function matching	No
Tracy [7]	No	No	No	Static	Partial trace (fixed length) matching	Yes
CoP [28]	No	No	No	Static	Pairwise basic-block matching	No
Bug search [30]	Yes	Limited	No	Static	Pairwise basic-block matching	Yes
BinGo	Yes	Yes	Yes	Static	Partial trace (variable length) matching	Yes

With the motivations above, 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 [9]). Based on the IR, we adopt a state-based semantic modeling to capture semantics of the binary code, and also propose an idiom based modeling to capture the high-level behavior of the code. Here, idioms refer to common instruction patterns appearing in binaries regardless of architecture, OS and compiler options. The two modelings are complementary and provide a comprehensive view of the semantics of the binary.

Second, to tolerate the binary code difference, for each function, we generate *partial traces* of various lengths instead of relying on program structures, where a partial trace refers to a sequence of basic-blocks that lie along an execution path in the CFG. 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.

Last, after building a number of partial trace based function models for a binary segment, we extract various features from the semantic model and behavior model mentioned in the first step. By leveraging on *feature hashing* technique [17], 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 [7] and BINDIFF [12] for the same tasks. Further, we also show that recent techniques such as [30], 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 a general solution of cross-architecture cross-OS binary search.

II. BACKGROUND AND PROBLEM STATEMENT

Identifying semantically similar or equivalent binary programs is a challenging problem. In this section, we discuss the problems of the existing binary code search techniques with motivating examples. Then we formally formulate the problem of similarity calculation of binary code in terms of the semantics and structures of the program, and discuss their roles to achieve the scalable binary matching regardless of architecture and OS differences.

A. Binary Preliminary

We start with the basic definitions of the binary analysis. In this work, we consider a general machine with a fixed set of general-purpose registers, condition-code flag registers and a program counter register. The machine supports a set of instructions (see Section IV for details). The execution of the instructions will change the state of the machine, which is modelled by the values of memory, registers, flags and program counter of the machine (called *machine artifacts*) and it is formally defined as follows:

***Definition** 1: (**Machine State**) A machine state is modeled as a tuple* $(mem, reg, flag, pc)$ *denoting the memory* mem*, general-purpose registers* reg*, condition-code flags* flag *and the program counter* pc.

A machine state reflects the status of the machine, while machine state transitions denote the changes in the values of these machine artifacts before and after a program (or an instruction) is executed. The machine state before program execution is called the *pre-state* and the state after execution is called the *post-state*.

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 they are formally defined as follows:

***Definition** 2: (**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.*

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 path.

B. Challenges for Syntax-based Matching

Syntax is the most direct information that can be used for matching. Various approaches have attempted to use instruction patterns, e.g., n-gram [18], graphlet [24] and tracelet [7], to match binaries. However, these works assume 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 [7]), it fails for cross-architecture and cross-OS analysis, where syntax are heavily dependent on the environment in which it runs [30].

```
ADD  EAX, EBX
```
(1)

```
PUSH  EBX
CALL  strlen
CMP   EAX, 0
```
(2)

Fig. 1: Sample code segments

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. 5, one for ARM and one for x86 32bit, both represent the behaviours of stack frame setup in function prologue. However, by relying on syntax representation, it is hard to match them.

C. Challenges for Semantics-based Matching

To make the matching tolerant to the syntax difference in binaries, semantics-based matching has been proposed [28, 30], 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: Semantics of binary code is too low level. Due to the lack of explicit program semantics at binary level, machine state transitions of two totally unrelated code segments may look very similar (or identical), if the number of instructions in those binary segments are too small, e.g., the two code segments in Fig. 1. A binary modeling approach based on machine state transitions has the problem of producing high false positives, especially when the given function (or vulnerability) signature is too small [30].

As shown in Fig. 2, two unrelated segments in Fig. 1 have the same initial machine state (i.e., pre-state) and also the identical post-state after execution. Here, code segment 1 adds the values in `EAX` and `EBX` registers, and move the results to `EAX` register. Since the pre-state values are all set to zero, after execution, `EAX` register will hold the value 0 and the condition-code flag `ZF` (zero flag) will be set to 1, since the result of addition operation is zero. On the other hand, in code segment 2, the value in `EBX` register is pushed to the stack, system API `strlen` is invoked, and finally the return value that is stored in `EAX` register is compared with an immediate value '0'. After execution, similar to code segment 1, `EAX` register will hold the value 1 as `strlen` system API is not interpreted (e.g., [30, 28]), `EAX` register will hold the pre-state value, which is 0. In addition, the comparison operation will set the `ZF` to 1. In both executions, none of the other registers, condition-code flags and memory locations will be modified, hence retaining the pre-state values. From the examples shown in Fig. 1 and 2, it can be seen that machine state transitions alone are not sufficient to model binary code.

C3: Computing the precise semantics is difficult and expensive. Static methods [28, 30] using state-based semantic computation cannot capture the precise semantics due to missing the consideration of system APIs in the analysis. Dynamic analysis based techniques such as [11] might be able to address this problem. However, it is not scalable enough to handle large binaries. For example, in [11], before each function is executed, an environment needs to be set so that the execution does not terminate due to uninitialised memory handling. Unfortunately, considering the fact that a moderate size binary would easily have several thousand functions, this approach is not very scalable.

Pre-state:	**Pre-state:**
$\texttt{Reg} = \{\texttt{EAX} = 0, \texttt{EBX} = 0 \ldots\}$	$\texttt{Reg} = \{\texttt{EAX} = 0, \texttt{EBX} = 0 \ldots\}$
$\texttt{Flag} = \{\texttt{ZF} = 0, \texttt{PF} = 0 \ldots\}$	$\texttt{Flag} = \{\texttt{ZF} = 0, \texttt{PF} = 0 \ldots\}$
$\texttt{Mem} = \{0, 0 \ldots 0\}$	$\texttt{Mem} = \{0, 0 \ldots 0\}$
Post-state:	**Post-state:**
$\texttt{Reg} = \{\texttt{EAX}' = 0, \texttt{EBX} = 0 \ldots\}$	$\texttt{Reg} = \{\texttt{EAX}' = 0, \texttt{EBX} = 0 \ldots\}$
$\texttt{Flag} = \{\texttt{ZF}' = 1, \texttt{PF} = 0 \ldots\}$	$\texttt{Flag} = \{\texttt{ZF}' = 1, \texttt{PF} = 0 \ldots\}$
$\texttt{Mem} = \{0, 0 \ldots 0\}$	$\texttt{Mem} = \{0, 0 \ldots 0\}$

(a) state transitions for Fig. 1(1) (b) state transitions for Fig. 1(2)

Fig. 2: Machine state transitions for code segments in Fig. 1

C4: How to choose the segments of binary for the semantics calculation. Existing approaches [28, 30] assume that basic-block structure is preserved across binaries, thus the matching should be basic-block centric. Based on this, models or features are extracted from the given binary segment at basic-block level and 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 [30] 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."*

Fig. 11(a) shows a real-world case of the infamous heartbleed vulnerability (CVE-2014-0160). Fig. 11(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 structure — with `gcc`, the vulnerable code is represented as a single basic-bloc; 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.

D. Problem Statement and Possible Solution

To compare binaries, we need to define the criteria to compute similarity. In this work, we propose two complementary criteria to capture the semantic and structural information of the binary programs. Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be two partial traces, we formally define the two kinds of similarity as follows:

***Definition** 3: (**Semantic Similarity**) Let s_0 denote a pre-state before executing $\mathcal{I}$, and $s_1 = \langle\langle s_0, \mathcal{I} \rangle\rangle$ denote post-state after executing $\mathcal{I}$ on s_0. Let S be set of all possible pre-states values that both $\mathcal{I}_1$ and $\mathcal{I}_2$ can execute on. The semantic similarity of $\mathcal{I}_1$ and $\mathcal{I}_2$, $SemSim(\mathcal{I}_1, \mathcal{I}_2)$, is defined as $\sum_{s \in S}(\langle\langle s, \mathcal{I}_1 \rangle\rangle -_m \langle\langle s, \mathcal{I}_2 \rangle\rangle)$, where $-_m$ is a function to measure the difference of two machine states.*

Semantic similarity is defined to capture the difference of the effects (i.e., post-state) of the binary program execution from the same input. Note that to calculate the precise effects of the program, we also need to consider the OS relevant information, like system API. Therefore, we consider that semantic similarity is architecture independent and OS dependent.

Semantic similarity is one essential criteria to capture the behavior similarity of programs. However, this definition ignores the difference of program's structures, i.e., the program is treated as a black box. Due to the challenge **C2**, when comparing two binaries, we also want to consider the structures of the binary to make sure they are implementing the same algorithm or computation. This is critical in the tasks like code auditing, plagiarism detection and vulnerability detection.

***Definition** 4: (**Structural Similarity**) The structural similarity of $\mathcal{I}_1$ and $\mathcal{I}_2$, $StrSim(\mathcal{I}_1, \mathcal{I}_2)$, is defined as $f_a^p(\mathcal{I}_1) -_f f_{a'}^{p'}(\mathcal{I}_2)$, where $f_a^p(\mathcal{I}_1)$ (or $f_{a'}^{p'}(\mathcal{I}_2)$) is an abstraction function that maps $\mathcal{I}_1$ (or $\mathcal{I}_2$) to a structural model on architecture a, OS p (or on architecture a', OS p'); and $-_s$ is a function to measure the difference of two structural models.*

Structural similarity is a general definition. The actual design of the abstraction function f (e.g., using instruction patterns: n-gram [18], graphlet [24] and tracelet [7]) reflects different research approaches on how to abstract the structural information of the binaries. Other usable structural information includes AST, PDG, data flow, type information and loop information. It is clear that the abstraction function is usually architecture and OS dependent, e.g., the codes in Fig. 5.

Overall, the similarity of $\mathcal{I}_1$ and $\mathcal{I}_2$ is defined as the combination of their semantic similarity and structural similarity. This similarity definition naturally resolves the challenge **C2**.

To solve **C1**, we need to define a structural model, which is robust to different architectures and OSs. Motivated by the recent work on source code level idiom mining [2], we propose to use the common binary patterns in different architectures and OSs as idioms, and use idioms to capture the program structural information, i.e., to define the abstraction function f. To support cross environments analysis, we propose a mapping of idioms in different architectures and OSs to a common concept such that $-_s$ can be easily computed.

To solve **C3**, we propose a static analysis to capture the state-based semantics [30] so that our approach is scalable. To capture the missing semantics related to system API using the static analysis, we introduce API idioms to complete the semantic information. Although the semantic comparison in this way is not precise as dynamic analysis like [11], it works well with good scalability as shown in experiments. Furthermore, the idiom approach addresses the OS dependency problem in the semantic similarity definition.

To solve **C4**, we propose a partial trace based function modeling, where partial traces of various lengths are extracted from binary code segment and they are organized in a systematic manner to handle program structure distortion. That is, by using partial trace models, no assumptions about the signature and target program structures are made (as in Tracy [7]). Overall, the unified idiom models for different environments make our solution work for all architectures and OSs. The partial trace function modeling and complete semantic information give an accurate solution.

III. 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. 12, 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 IV) 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 V-A1) capture the OS dependent semantics of the binary code. Secondly, structural features and compiler idioms (see Section V-A1) 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 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.

IV. State-based Semantic Modeling

To compute the semantic similarity, we adopt the state-based semantic model [28, 30], which aims to capture the execution effect of the binary code in terms of machine state updates. In this work, we use a RISC-like intermediate representation (IR) to lift the assembly instructions of different architectures to a common representation, which enables us to perform architecture independent analysis. To begin with, we explain the machine state transition rules based on the basic machine

mentioned in Section II-A. We list the RISC-like instruction set in the following.

$$inst ::= \mathsf{Mov}\, r_s\, r_d \mid \mathsf{Binop}_{\otimes}\, r_1\, r_2\, r_d \mid \mathsf{Load}\, r_p\, r_d \mid \mathsf{Store}\, r_p\, r_s \mid \mathsf{Jump}\, r \mid \mathsf{Jal}\, r \mid \mathsf{Nop} \mid \mathsf{Const}\, r\, r_d \mid \mathsf{Halt}$$

where $\mathsf{Mov}\, r_s\, r_d$ copies the content of register r_s to the register r_d; $\mathsf{Binop}_{\otimes}$ refers to binary operations (i.e., $\otimes \in \{\div, \times, \leq, \geq, \ldots\}$); Jump and Jal (Jump-and-link) are unconditional jumps, and $\mathsf{Const}\, r\, r_d$ puts an immediate value (or constant) i into register r_d. Jal is used to implement subroutine calls.

Recall that a machine state is a tuple $(mem, reg, flag, pc)$. In the original formalization [8], *flag* is not considered as part of a machine state. However, our models are generated from partial traces (discussed in Section VI-A). Hence, it is essential to consider the state of condition-code flags (e.g., $cf, pf, \ldots$ in x86 and $zf, vf, \ldots$ in ARM) in semantic similarity computation. For example, at a given program point, across two different binary functions, if the state of condition-code flags is identical then it is an indication[1] that these two functions perform similar operations at the machine-code level. It is noted that the memory is a partial function; trying to access addresses outside of the valid memory halts the machine [8].

The execution of an instruction leads to the transition of the states. As an example, we explain one of the transition rules for Binop instruction. The complete set of machine state transition rules can be found in [8].

$$\frac{\begin{array}{c} mem[pc] = i \quad decode\, i = \mathsf{Binop}_{\otimes}\, r_1\, r_2\, r_d\, f_d \quad reg[r_1] = w_1 \\ reg[r_2] = w_2 \quad reg' = reg[r_d \leftarrow w_1 \otimes w_2] \\ flag' = flag[f_d \leftarrow w_1 \otimes w_2] \end{array}}{(mem, reg, flag, pc) \rightarrow (mem, reg', flag', pc + 1)} \quad (\mathsf{Binop})$$

Looking up the memory at address *pc* yields the instruction *i* (an element of the *inst* set defined above) via a partial function *decode*. In this case, the instruction is $\mathsf{Binop}_{\otimes} r_1\, r_2\, r_d$. Registers r_1 and r_2 contain the operands w_1 and w_2, respectively. The notation $reg[r_d \leftarrow w_1 \otimes w_2]$ denotes the partial function that maps r_d to $w_1 \otimes w_2$ and behaves like reg on all other arguments. Similarly, $flag[f_d \leftarrow w_1 \otimes w_2]$ denotes the partial function that maps the flag f_d to the side-effect (if any) of $w_1 \otimes w_2$ on the conditional flags. The next machine state is calculated by updating the registers and conditional flag files, incrementing the pc by 1, and leaving the memory unchanged. For example, assume zero-flag is set to 0 ($\texttt{zf} = 0$) and the registers `eax` and `ebx` hold the values 1 and -1 respectively, the execution of an instruction '`add eax, ebx`' (binary operation) leads to following state: `eax` now holds the value 0 (i.e., $1 \mapsto 0$); zero-flag is set to 1 (i.e., $0 \mapsto 1$); the program counter is incremented; memory, all other registers and flags are unchanged.

Note that in static analysis, programs are not physically or virtually executed, and hence the machine state transitions can be represented in the symbolic form as symbolic expressions [30, 31], which are 2-vocabulary formulas that specify the symbolic machine state transitions of a partial trace, where

[1]Having identical condition-code flags is the sufficient but not necessary condition for the two binary code segments to be semantically similar.

1) $\langle\!\langle \texttt{push 0} \rangle\!\rangle \equiv \texttt{esp}' = \texttt{esp} - 4 \wedge \texttt{Mem}' = \texttt{Mem}[\texttt{esp} - 4 \mapsto 0]$
2) $\langle\!\langle \texttt{mov ecx, [ebp]; lea eax, [ebp + 4]} \rangle\!\rangle$
   $\equiv \texttt{ecx}' = \texttt{Mem}(\texttt{ebp}) \wedge \texttt{eax}' = \texttt{ebp} + 4$

Fig. 3: Symbolic expression for example IA-32 instructions

unprimed vocabulary (e.g., *eax*) represents the machine artifacts in the *pre-state* and the primed vocabulary (e.g., *eax*′) represents the machine artifacts in the *post-state*.

Therefore, the updates made by partial traces on the machine state are represented by a set of symbolic expressions. Note that in the symbolic expressions used in BINGO, the state transition of the program counter *pc* (i.e., $pc' = pc + 1$) is omitted as it does not contribute to comparing two different programs. As a result, in BINGO, the machine state is essentially characterised by the rest three major machine artifacts and represented as a triple $(mem, reg, flag)$.

The symbolic expressions extracted for two partial traces are given in Fig. 3. The first partial trace consists of only one instruction, where it pushes a 32-bit constant value 0 on the stack. The second partial trace consists of two instructions, which loads the value in the memory location pointed to by the base-pointer (i.e., `ebp`) into the `ecx` register followed by loading `eax` register with the value $\texttt{ebp} + 4$.

Interestingly, partial traces (or even basic-blocks), in general, read values (called *inputs*) from multiple registers, flags and memory locations and write values (called *outputs*) to multiple registers, flags and memory locations during execution. That is, a partial trace, in general, performs the following three tasks: (1) read values from a set of machine artifacts M_a, (2) do some transformations, and (3) write values back to a set of machine artifacts $M_a{}'$. Here, values of machine artifacts $M_a{}'$ need not be same as the initial values of machine artifacts M_a. From the definition of machine state transition in Section II, we can see that the values read by partial trace constitutes the pre-state values and the values written back constitutes the post-state values. However, we can find that symbolic expression models the machine state transitions in symbolic mode. Hence, by using symbolic expression, we extract the relationship between symbolic pre- and post-state values. That is, from the symbolic expression, output vocabularies (symbolic output values or symbolic post-state values) are identified and written in-terms of input vocabularies (symbolic input values or symbolic pre-state values), called *I/O formulae*.

For example, from the symbolic expression generated from the second partial trace, shown in Fig. 3, we can see that there are two output vocabularies ($\texttt{ecx}'$ and $\texttt{eax}'$) and one input vocabulary (`ebx`) and the corresponding I/O formulas are written as follows:

$$\texttt{ecx}' = \texttt{Mem}(\texttt{ebp})$$
$$\texttt{eax}' = \texttt{ebp} + 4$$

The I/O formulae above clearly express the transformation of pre-state values into post-state values in symbolic mode, which are internally used by our semantic similarity function $SemSim$ defined in Section II-D. Later in Section VI-B, we shall discuss how to identify the suitable concrete values for input/output vocabularies such that all the I/O formulas are satisfied.

```
mov  eax, gs : 20          } function prologue
mov  [ebp − 12], eax       }
...
...
mov  eax, [ebp − 12]       } function epilogue
xor  eax, gs : 20          }
```

Fig. 4: Code generated by `gcc` to enable SSP compiler feature

Finally, as discussed in Section II-C, one of the major problems in state-based semantic models is that effects of system APIs are not considered in the semantics. In the IR, it is common to implement subroutine calls (e.g., function calls, system API invocations) using unconditional jumps. For example, in our aforementioned primitive machine, subroutine calls are implemented using Jal instruction [8] as follows:

$$\frac{mem[pc] = i \quad decode\, i = \ \mathsf{Jal}\ r_p\ r_s \quad reg[r] = pc' \quad reg' = reg[r_a \leftarrow pc + 1]}{(mem, reg, flag, pc) \rightarrow (mem, reg', flag, pc')} \ (\mathsf{Jal})$$

Here, the return address is saved to a general-purpose register r_a. From the transition rule, it can be seen that pc' holds the address of the target subroutine. If pc' is statically unresolvable or it falls out side the memory range of the binary under analysis (which is the case for system API), the execution resumes at the memory location r_a without executing the subroutine. Hence, it is evident that the effects of system API are ignored[2] in the state-based semantic models.

V. Structural Modeling

To complement state-based semantic modeling, we introduce an idiom based modeling to capture structure and the high level behaviour of the binary. To this end, we leverage the following analysis — idiom analysis that looks for familiar and recurring machine code patterns that are already observed in the majority of binaries from real world.

A. Idioms

Motivated by source code idioms [2], we find that the same concept applies in binaries, i.e., the recurring patterns of binary instructions reveal some fixed semantics. For example, the idiom given below indicates that the code segment under analysis sets up the stack frame in the function prologue.

```
push ebp;   mov ebp, esp;
```

In addition to revealing high-level meaning, idioms can also explain the compilation process. The presence of code segment (in Fig. 4) in the function prologue and epilogue ensure that the stack integrity is not violated, where it is automatically included by the compiler (in this case, `gcc`) when stack smash protection compiler feature is enabled (using either `-fstack-protector-all` or `-fstack-protector` flag). Generalizing the code segments, shown in Fig. 4, into an idiom helps an analyst to identify that binaries are protected by the stack smash protection (SSP) compiler feature.

Hence, using idioms, we can capture various interesting concepts and properties in a program segment, which in return, help us to understand the high-level behavior. Let $\mathcal{I}$ be a partial trace, consisting of a sequence of *normalized* instructions $\langle i_1, i_2, ..., i_n \rangle$, and n be an n-gram model (n-length sub-sequence in $\mathcal{I}$, e.g., 3-gram model of $\langle i_2, i_3, i_4 \rangle$) . We also define $\mathcal{N} = \{n_1, n_2, ..., n_m\}$ as the set of all possible n-gram models on $\mathcal{I}$ such that $\mathcal{N} \subseteq \mathbb{P}(\mathcal{I})$. Normalized instructions refer to assembly instructions abstracted away from volatile/noisy artifacts such as immediate values and absolute memory locations (see Section V-B2 for details).

***Definition** 5: (**Idiom**)* A n-gram model n_i is considered as an idiom **iff** n_i surpasses the minimum threshold of recurring probability t, which is predefined according to the observation in real-word binaries, and holds a valid high-level meanings.

According to the above definition, we look for idioms observed in real-world binaries and map them to a valid unified high-level meaning (i.e., the tasks achieved by the idiom). Note that the instruction sequences (i.e., idioms) are architecture and OS dependent. However, their high-level meanings are usually agnostic to differences in architectures and OSs.

For example, an instruction sequence that implements the system API invocation in one architecture is totally different from that in another architecture, depending on the calling conventions used (discussed in Section V-A2). However, at high-level they all achieve the same objective by invoking the same or similar system APIs. Therefore extracting idioms, from various architectures and OSes, with mapping to a unified high-level meaning, helps to summarize the behavioral semantics of the program segment (or function) under analysis.

Note that some machine code patterns appear only in certain type of architecture (or OS) and hence, cannot be unified across architectures and OSs. However, we still consider them as idioms as they can be used in the analysis where only the relevant architectures (or OSs) are involved. For example, stack split (or segmentation) capability is available only in `gcc` compiler (using `-fsplit-stack` flag) and for x86 32bit binaries only, and hence the machine code pattern that implements the stack split functionality cannot be unified across architecture (or OS). However, it can be useful for analysis that involve x86 32bit binaries compiled for Linux.

1) Idiom Classification: The structural model of a binary program can be characterised by various properties such as data structures used (e.g., array, structure, etc.), variable types involved (e.g., local and global variables), and other structural properties. In addition, sometimes, compilers influence a computation through the various compiler features.

To reflect different aspects of the structural information, we classify idioms into three major types based on the nature of information related. These idioms together can be considered as the concrete implementation of function $f_a^p(\mathcal{I})$ defined in Definition 4. Note that this classification is not the only way or complete way to model different concepts in behavior models. The different applications of the binary searching may require different idioms to be used. Further more, the idioms can be extended based on demands.

1: Structural idioms capture the code properties, e.g., data structures, variable types and other structural aspects like

[2]One can use function summary to captures the semantics of system API. However, it is not practical and summarizing all the functions in system libraries can be an error prone task.

functions, boundary details and register access patterns. These idioms help to identify the structural details and code properties of the program segment. For example, the presence of an instruction pattern `sub/add esp,Imm` in a code segment compiled for x86 32bit architecture indicates stack frame size manipulation operation. In addition, structural idioms capture other common structural properties such as save/restore of callee/caller-save registers[3], the presence of function prologue/epilogue and the presence of loops. Structural idioms can also capture the code properties such as local/global variable access, function parameter and array manipulation.

Structural idioms are architecture dependent, whereas their high-level meanings are unified across architectures. Further, one cannot precisely extract code properties using structural idioms. For example, by only using light-weight static analysis, it is impossible to distinguish between a C-style string and an array[4] [13]. However, in our analysis, if there is a consecutive memory access pattern, we speculate the data structure based on the co-occurrence of other idioms — we map consecutive memory access pattern to a string given the presence of string related system APIs in the code segment. This approach may not be precise, but works reasonably well and efficiently, compared to other more heavy-weight program analysis techniques [3].

2: Semantic idioms capture the system API related information. This type of idioms provide the high-level semantic meaning of the code segment under analysis, which captures the system API invocation patterns. For example, the presence of `strlen` system API indicates that the program handles strings, more specifically, string arithmetic. In addition to extracting the system API, it is observed that identifying the co-occurrence of two or more system APIs gives more insight into the high-level behaviour of the program segment. For example, the presence of `strlen` and `memcpy` system APIs in a code segment indicates buffer manipulation operation involving string arithmetic.

For semantic idioms, we cannot simply rely on `call` instruction mnemonic (in x86) to identify the call invocations. Unfortunately, there are alternative ways to invoke a function without using `call` instruction [25]. For example, `call` instruction can be replaced by `push/jmp` or `push/ret` instruction sequences.

Learning the possible system API invocation patterns from existing binaries and generalizing them into meaningful semantic idioms help the analysts to understand the high-level behavior of binary programs under analysis. Further, to accommodate cross-OS analysis, we also apply two levels of abstraction for system API — different APIs can be used to accomplish the same objective in different OSs (e.g., Windows vs. Linux) or in different versions of the same OS (e.g., Windows XP vs. Windows 7). Hence, an abstraction process is used to unify the system APIs across OSs and versions (see details in Section V-B2).

[3]Callee-saved registers are used to hold long-lived values that should be preserved across calls. Similarly, caller-saved registers are used to hold temporary quantities that need not be preserved across calls.

[4]Arrays are stored as consecutive objects in memory with no information about the size of the array. Similarly, C-style strings are too stored as arrays with a terminating element of 0.

```
mov ip, sp
stm sp!, fp,ip,lr,pc
sub fp, ip, Imm
```

(a) ARM version

```
push ebp
mov ebp,esp
sub esp,Imm
```

(b) x86 32bit version

Fig. 5: Function prologue (i.e., stack frame set-up) for ARM and x86 32bit architectures.

3: 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.

2) Cross-Architecture and Cross-OS Mapping: Idioms that are identical in terms of high-level behavior can be implemented using completely different ISAs (Instruction Set Architectures) in different architectures, where each ISA has its own instruction set, register structure and addressing modes. For example, consider the following function prologues for ARM and x86 32bit architectures (We omitted the register normalization for the ease of understanding) shown in Fig. 5a and 5b, ARM and x86 use completely different sets of instruction sequences to set up the stack frame in the function prologue. Moreover, even within the same architecture family the implementation conventions can drastically vary, which makes matching more difficult. For example, in x86-32 (IA-32), the function parameters are passed through the stack, however, in x86-64 (IA-64 or AMD64), the first several function parameters are passed using the registers and the rest are passed through the stack. Unfortunately, this discrepancy does not stop there. It gets even more complicated across OSs, where for the same architecture different OSs have different ABIs (Application Binary Interface). For example, for non-float/double parameter passing, in x86-64, Microsoft ABI uses the registers `rcx`, `rdx`, `r8` and `r9`, whereas, system V ABI (the ABI used in Linux) uses the registers `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9`.

Hence, in BINGO, we work with high-level meanings of idioms which are unified across different OSs and architectures. Note that the high-level meaning behind idioms can be unified across architecture and OS through *a manual vetting process*.

B. Idiom Extraction

To extract idioms, the assembly instructions in the binary need to be modeled in a suitable manner such that the recurring

Algorithm 1: Idiom extraction from binary programs

Data: binary corpus β
Result: set of idioms $\Re$, idioms co-occurrence matrix $\Re_M$

```
1  ℜ ⟵ ∅
2  ℜ_M ⟵ ∅
3  dict[·] = {}                                   // n-gram dictionary
4  foreach binary b in β do
5      F := getFunctions(b)
6      F_n := normalizeFunctions(F)               // algorithm 2
7      ℘ := getPartialTraces(F_n)
8      foreach partial trace p in ℘ do
9          foreach n = 1 To MAX do
10             S_n := getNgramModel(p, n)
11             updateDict(dict_n, S_n)
   // extract idioms based on a threshold value t
12 ℜ := getIdioms(dict, t)                        // manual vetting involved
13 ℜ_M := getIdiomCoOccurrenceMatrix(ℜ, F_n)
14 return ℜ, ℜ_M
```

patterns can be easily identified. To this end, we leverage on statistical language models [16] to extract the idioms (Section V-B1). The most successful class of language models are n-gram models, which are widely used for source-code level applications such as code suggestions [23], cross-language porting [29], coding standards [1] and code de-obfuscation [32]. Similarly, n-gram models are applied for binary programs in the domains of malware detection [17], clone detection [36] and program lineage analysis [18].

The n-gram model is based on Markov assumption that given an instruction sequence $\langle i_1, i_2, ...i_n \rangle$ the occurrence of instruction i_n depends only on the $(n-1)$ most recent instructions. After we extract the idioms based on n-gram models, we identify the high-level meanings of idioms which are unified across different OSs and architectures (Section V-A2).

1) Extraction Process: The idiom extraction process is presented in Algorithm 1, which takes the binary corpus β and returns the set of idioms extracted (or learned) $\Re$ and the idiom co-occurrence matrix $\Re_M$. Idiom co-occurrence matrix reflects how frequent two idioms appear together. That is, in the matrix, if the cell corresponding to idioms n_a and n_b holds the value 10, then it means idioms n_a and n_b appear together in 10 unique functions[5]. One of the key steps in idiom extraction is the *instruction normalization* process, which helps to remove noise in the data. For example, the instruction `mov eax,0x10` is normalized to `mov eax,imm`, replacing the concrete value (i.e., `0x10`) with the a symbolic variable `imm`. The normalization process is discussed in Section V-B2.

Once functions are normalized, similar to state-based models, partial traces are extracted (line 7). Then from each partial trace, n-gram models of different sizes are generated (line 10) and the n-gram dictionary is updated accordingly (line 11), where it keeps track of the frequency of each n-gram. Once all n-gram models are extracted, n-gram with higher term frequency (i.e., term frequency $>$ threshold t) are considered for idioms. However, in the manual vetting process, as part of `getIdioms` function, we generalize the instruction sequence patterns, possibly with wildcard tokens ($*$) [33], into meaningful idioms. For example, Fig. 6a and 6b present the two variants of stack frame set-up instruction sequence, we generalize them into an idiom using a wildcard token as follows:

`push Reg_BP; mov Reg_BP,Reg_SP; *; sub Reg_SP,Imm;`

In addition, during vetting, we also remove idioms that are not 'useful'. For example, the instruction `xchg reg,reg` in x86 32bit architecture (e.g., `xchg eax,eax`) generally refers to `Nop` instruction[6], hence, not very useful for our analysis. Finally, vetted idioms along with normalized functions Fn are passed to function `getIdiomCoOccurrenceMatrix` to build the co-occurrence matrix.

In total, we have extracted 11 compiler idioms, 7 structural idioms and for semantic idioms, we have abstracted systems API that belongs to following categories: string operation, memory operation, network, inpit/output, file access, synchronization, and system information.

2) Instruction Normalization: Algorithm 2 (due space limitation, shown in Appendix B) presents the steps involved in normalizing the assembly instructions. Normalizing the instructions, especially the operands, considerably reduces the noise and helps to identify the recurring code patterns (or idioms). In general, operands can be categorized into three types: register, memory address, and immediate value. To this end, given an instruction, we first determine the operand type and based on the type the instruction is normalized.

The immediate values are replaced with a symbolic variable `Imm` (line 18), where immediate values are, in general, considered very volatile and hence need to be normalized. For registers, we map the register name (e.g., `eax`, `ebx`, ...) with the type, where register type refers to the task for which the registers are conventionally used for. For example, in x86 architecture, the registers `esp`, `ebp`, `rbp` and `rsp` are conventionally used to deal with stack, where `esp`, `rsp` registers are called stack pointers and `ebp`, `rbp` registers are called frame (or base) pointers. Hence, in our analysis, stack pointers and frame pointers are replaced with the symbolic registers `Reg_SP` and `Reg_BP`, respectively (line 20 to 23). However, not all the registers are used for the purpose they are conventionally meant for all the time. For example, `esi`, `edi`, `rsi` and `rdi` registers are conventionally used for strings manipulation, but at times, they are used in computations that have no string involved. Hence, apart from stack related registers, all other registers are mapped to a more generic symbolic register `Reg`.

For memory accesses, we first determine whether the memory address (if accessed directly by specifying an absolute address) falls within the `.data` or `.bss` segments of the binary, and map these addresses to a symbolic memory address `Mem_Global`. In C/C++, the initialized (or explicitly initialized) global, static and constant variables are stored in `.data` segment and uninitialized (or implicitly zero initialized)

[5]Unique function refers to functions that have unique MD5 hash values, where if two functions have the same hash value, we remove the second function from our dataset F considering it as a duplicate of the first function.

[6]NOP (short for No Operation) is an assembly instruction that does nothing.

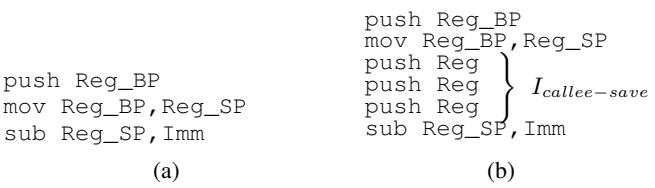


Fig. 6: Two variants of stack frame set-up instruction sequence. Here $I_{callee-save}$ shows the callee-save registers.

variables are stored in `.bss` segment of the binary. In our analysis, by symbolic memory access `Mem_Global`, we refer to both initialized and uninitialized global, static and constant variables. Indirect memory accesses[7] that are related to the stack (e.g., base register is either `esp` or `ebp` in x86 32bit architecture) are mapped to the symbolic address `Mem_Stack`. Finally, all other direct and indirect memory accesses are mapped to a more generic symbolic address `Mem`.

To make BINGO capable of analyzing cross-OS binaries, we use a multi-level abstraction function `abstractSystemAPI` at line 12 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. 7.

Based on the requirement of the analyst, there may be several ways to abstract a system API. As shown in Fig. 7(b), abstraction level 1 can be more expressive in providing more meaningful information about the API or it can be limited as in Fig. 7(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.[8]), which are less likely to contain an exploitable vulnerability.

Note that the function normalization process above is x86 architecture specific, however, a similar approach with some modifications can be successfully applied to normalize ARM

[7]Indirect memory access refers to accessing memory using an address expression (i.e., without specifying an absolute address) of the form '[*base* + *index* × *scale* + *offset*]', where *base* and *index* are registers, and *scale* and *offset* are integer constants

[8]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.

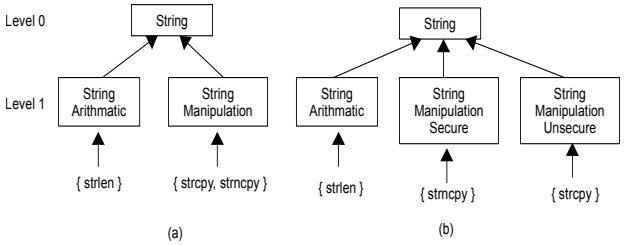


Fig. 7: System API Abstraction Levels

instructions. For example, ARM has different addressing modes compared to x86, which need to be properly handled in normalization process.

VI. BINGO FRAMEWORK

This section, we present the four modules in BINGO in details. As the first step, BINGO disassembles the binary and splits it into basic functional units or functions. Then, it constructs the control-flow graph (CFG) for each function, where each CFG is consists of basic-blocks. Later, the constructed CFGs are used to generate the partial traces.

A. Partial Trace Extraction

In BINGO, we propose a partial trace based function modeling, which is more flexible in terms of granularity, compared with basic-block centric function modeling techniques [30, 28]. That is, for each function, we generate partial traces of various lengths, and by varying the length the granularity of single building block is adjusted[9].

Our partial trace extraction is based on the technique proposed by David *et. al.* [7]. We omit the algorithm and explain the results using one example. Fig. 8 depicts a sample CFG of a function and Fig. 9 presents the extracted partial traces of length 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 already determined. Note that the feasibility of the flow of execution is not considered when generating the partial traces. However, later in Section VI-C1, 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.

B. Semantic Feature Extraction

In this section, we elaborate the feature extraction process from state-based semantic models and behavior models.

Feature extraction from state-based semantic models As discussed in Section IV, a set of I/O formulas is extracted from each partial trace, and these formulas capture the relationship between symbolic pre- and post-state values. However, the extracted I/O formulas cannot be directly fed into our machine learning module. Hence, we leverage on the constraint solver to extract features by solving the formulas, where solving refers to

[9]Length of one yields a basic-block centric models, where the basic-block is considered as a single building block.

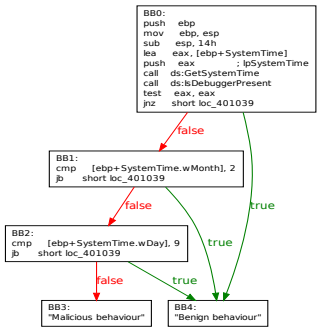


Fig. 8: Sample CFG of a function

finding appropriate values for the input and output vocabularies such that all the constraints present in the I/O formulas are satisfied. The obtained concrete values for input/output vocabularies constitute the features of state-based semantic model. For example, consider the following two I/O formulas extracted from a simple example of partial trace:

$$\texttt{zf}' \equiv (\texttt{edx} < 2) \wedge (\texttt{edx} > 0) \quad (3)$$

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

For the above I/O formulas, it can be seen that there are two output vocabularies (`zf'` and `ecx'`) and one input vocabulary (`edx`), where the input vocabulary is common for both I/O formulas. Hence, it is essential to ensure that the concrete value obtained for input vocabulary `edx` satisfies the both I/O formulas. That is, for `zf'` to be `true`, `edx` can only take the concrete value `1`, hence, `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 input/output vocabularies, as it takes all the I/O formulas, extracted from a partial trace, into consideration when arriving at the solution. Note that if there are more than one concrete value for the input/output vocabularies, we set a limit to $\mathcal{N}$ possible values, where $\mathcal{N}$ is set to 3 in our experiment based on the empirical results reported in [11].

In contrast, in the sampling-based feature extraction technique, proposed in the literature [30], each I/O formula is considered independently, where the relationship between the two I/O formulas is ignored (both have the same input vocabulary `edx`). This approach may lead to inaccurate semantics, especially in case of real-world execution instructions that are executed sequentially. Hence, the relationship between formulas needs to be maintained. However, it is observed that using a constraint solver is costly, compared to sampling-based feature extraction technique. As feature extraction is an one time task and can be paralleled for binaries that are totally independent of each other, we adopt the constraint solver based approach for its accuracy.

Feature extraction from behavior models Feature extraction from behavior models is straightforward, where the high-level meanings of the extracted idioms are used as features. For example, if the machine code pattern '`pop reg; pop reg; pop reg; pop reg;`' is observed in the binary code under analysis,

```
push ebp
mov ebp, esp
sub esp, 14h
lea eax, [ebp+SysT]
push eax
call ds:LibCall1
call ds:LibCall2
test eax, eax
cmp [ebp+SysT.wMonth],2
cmp [ebp+SysT.wDay],9
```

(a) Tracelet $\langle BB0, BB1, BB2 \rangle$

```
push ebp
mov ebp, esp
sub esp, 14h
lea eax, [ebp+SysT]
push eax
call ds:LibCall1
call ds:LibCall2
test eax, eax
cmp [ebp+SysT.wMonth],2
benign behaviour
```

(b) Tracelet $\langle BB0, BB1, BB4 \rangle$

```
cmp [ebp+SysT.wMonth],2
cmp [ebp+SysT.wDay],9
benign behaviour
```

(c) Tracelet $\langle BB1, BB2, BB4 \rangle$

```
cmp [ebp+SysT.wMonth],2
cmp [ebp+SysT.wDay],9
malicious behaviour
```

(d) Tracelet $\langle BB1, BB2, BB3 \rangle$

Fig. 9: Tracelets extracted from the CFG shown in Fig. 8

using learned idiom patterns and their corresponding high-level meanings (as discussed in Section V-A2), the code pattern is identified as '`restore callee-save registers`' and added to features of the behavior model.

C. Function Model Generation

Compiler optimizations and/or differences in environments often lead to changes in the basic-block structure. Thus, basic-block centric function modeling is not preferred [30, 28]. For example, given a search query that consists of a single basic block, we may miss the similar function in the target binary pool due to the structural difference, where the semantically similar target function can be of several basic-blocks and the similarity between the query function and each of the basic-blocks in the candidate target function can be below the pre-defined threshold value. This is a common observation that arises due to differences in compiler optimization and the compiler types that lead to basic-block splitting (or merging) and function inclining (or outlining). Hence, in BINGO, as discussed in Section VI-A, we use k-length partial traces to model the functions. The key advantage of using partial trace, over other techniques [30, 28], lies in its resilience to the structural changes incurred by the aforementioned compiler optimization techniques, which otherwise might fail the similarity search. However, in the step of generating partial traces, the generated partial traces need to be valid or feasible in real-world executions. To this end, we propose a *pruning* technique to remove partial traces that are *infeasible* in practise.

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 practise. In BINGO, we prune the obvious infeasible partial traces with the help of a constraint solver. That is, given a partial trace, we extract the set of I/O formulas, as explained in Section IV, and try to generate a model that satisfied all the constraints present in the I/O formulas (i.e., able to generate appropriate concrete values for input/out vocabularies), if such model is not available then that partial trace is considered infeasible. Some partial traces, for which the solver is able to generate models, 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 static analysis. Infeasible path elimination is a

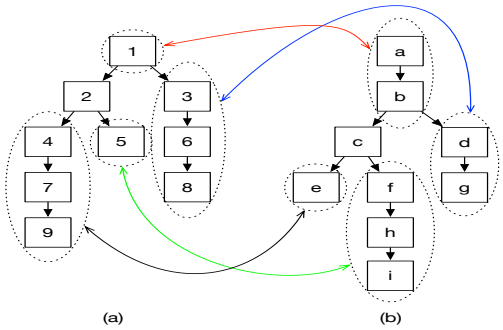


Fig. 10: This depicts the signature and target functions, where the lines indicate the matched partial traces.

difficult task even in dynamic analysis. Comparing to the static analysis solutions proposed in the literature [7, 31, 28, 30], this work makes an attempt to reduce the search candidates using pruning technique.

Even with the infeasible partial traces pruned, it is still difficult to compare the signature function against those functions in the target binary pool. Thus, a systematic way of performing the similarity matching among partial traces is desired. Specifically, the question "what length of partial traces in the search query needs to be matched with what length of partial traces in the target function?" still persists.

2) Function Modeling: It is understood that partial traces need to be organized in a systematic way to effectively compare two different functions. As the signature and the target functions are shown in Fig. 10 (a) and (b) respectively, it can be seen that these two functions are matched as follows: the partial traces $\langle 4,7,9\rangle$, $\langle 3,6,8\rangle$, $\langle 1\rangle$, $\langle 5\rangle$, $\langle 2\rangle$ in the signature function are matched with the partial traces $\langle \mathtt{d}\rangle$, $\langle \mathtt{c}\rangle$, $\langle \mathtt{z},\mathtt{a}\rangle$, $\langle \mathtt{e},\mathtt{f},\mathtt{g}\rangle$, $\langle \mathtt{b}\rangle$ in the target function, respectively. Hence, we can say those combination of partial traces *precisely model* the underlying functions. Here, precise model of a function (called *precise function model*) refers to a combination of partial traces that can cover every single basic-block in the CFG without any overlap. That is, a basic-block should not be covered by more than one partial trace. Precise models for the functions shown in Fig. 10 is as follows:

$$\mathtt{Signature\ Function} : \{\langle 1\rangle, \langle 2\rangle, \langle 5\rangle, \langle 4,7,9\rangle, \langle 3,6,8\rangle\}$$
$$\mathtt{Target\ Function} : \{\langle \mathtt{c}\rangle, \langle \mathtt{b}\rangle, \langle \mathtt{d}\rangle, \langle \mathtt{z},\mathtt{a}\rangle, \langle \mathtt{e},\mathtt{f},\mathtt{g}\rangle\}$$

However, it is challenging to identify all possible combinations of partial traces that can precisely model a function. Further, it is also not very scalable for larger functions, which are very common in real-world binaries. Hence, in BINGO, to improve the scalability, we limit the function models to a fixed number by relaxing the precision. That is, instead of ensuring that the partial traces do not overlap, we construct function models (called *relaxed function models*), where none of the basic-blocks are missed, however, there can be overlapping of partial traces. In the rest of the sections, for simplicity, relaxed function model is also refereed to as function model and is formally defined as follows:

Definition** 6: **(Function model) It is a unique combination of partial traces. Given $\mathtt{N}$ different lengths of partial traces, generated from a function, in total, $2^{\mathtt{N}}-1$ function models are constructed for that function.

D. Function Model 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[10] and target functions[11], 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. 10, 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. 10, 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. 10, 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. 10, 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 [30, 28].

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 [7], 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 [30] and [28], 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

[10]Signature function refers to the search query function in hand.

[11]Similarly, target functions refer to functions in the target binary pool against which the signature function is matched.

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.

E. 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 [27] based on the assumption that, if two items are similar, the hashed value of the two items will remain similar. In BINGO, we use MinHashing [5] 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:

$$sim(mh_a, mh_b) = \frac{|mh_a[i] = mh_b[i]|}{n} \quad (5)$$

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

VII. IMPLEMENTATION AND EXPERIMENTATION

The semantic feature extraction engine in BINGO uses REIL [9] 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?

A. 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.

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 V 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 III, 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 V. 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.

2) Answer to RQ2: Table II 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.

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 VII 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 III), 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 [7], 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 [10] 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.

B. Accuracy

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 II, 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 IV, 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 IV, 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.

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 VI. From the table, it can be seen that compare to the results obtained with abstract structural features, summarised in Table V, 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.

C. Applications

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 [31] — 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 [31, 30]. 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*[12]. 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 [30]. 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 [30], 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. 11). The vulnerability is matched from Windows to Linux, using basic-block centric matching, similar to [30, 28], 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 (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*.

D. Scalability

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

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

`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 II). These results demonstrate the scalability of BINGO.

VIII. RELATED WORK

A. Code Search based on Similarity

Source code similarity analysis CCFINDER [22] was proposed to detect source code clones based on a token-by-token comparison. DECKARD [20] builds the AST and applies local sensitive hashes to search for the similar subtrees of source code. Besides, PDG based clone detection [14] can tolerate some syntactic modifications or small gaps of additional statements, but the detection might be slow.

Binary similarity analysis 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. [24] 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, [35] and [34] also adopt CFG in recovery of the information of compilers and even authors. Tracelet [7] is presented to capture semantic similarity of execution sequences and facilitate searching for similar functions.

B. 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.* [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 input and output. [37] 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][37]. Nevertheless, intermediate values are used to mitigate the problem of identifying input-output 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 [40] and matching execution histories of program [41] are proposed.

Diff based equivalence check BINDIFF [12] 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 [15], 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 [4] 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 [10] 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).

C. 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.* [39], 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.* [30] 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.

IX. 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] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 281–293. ACM, 2014.

[2] M. Allamanis and C. Sutton. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 472–483. ACM, 2014.

[3] G. Balakrishnan and T. Reps. Divine: Discovering variables in executables. In *Verification, Model Checking, and Abstract Interpretation*, pages 1–28. Springer, 2007.

[4] 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.

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

[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] 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.

[8] A. A. de Amorim, M. D. N. Giannarakis, C. Hritcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Micro-policies: A framework for verified, tag-based security monitors. In *IEEE Symposium on Security and Privacy*, 2015.

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

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

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

[12] 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.

[13] A. Fog. Calling conventions for different c++ compilers and operating systems. *Copenhagen University College of Engineering*, 2009.

[14] 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.

[15] 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.

[16] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 837–847. IEEE, 2012.

[17] 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.

[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] S. Karaivanov, V. Raychev, and M. Vechev. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 173–184. ACM, 2014.

[24] 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.

[25] A. Lakhotia and E. U. Kumar. Abstracting stack to detect obfuscated calls in binaries. In *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*, pages 17–26. IEEE, 2004.

[26] M. Lee, J. Roh, S. Hwang, and S. Kim. Instant code clone search. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pages 167–176, 2010.

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

[28] 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.

[29] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 651–654. ACM, 2013.

[30] 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.

[31] 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.

[32] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from big code. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 111–124. ACM, 2015.

[33] N. Rosenblum, B. P. Miller, and X. Zhu. Recovering the toolchain provenance of binary code. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 100–110. ACM, 2011.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.

APPENDIX A
SUPPORTING DATA FOR EXPERIMENTS

TABLE II: Results of BINGO, across compilers (`gcc` and `clang`) with different code optimization levels

.	Rank 1	Top 5	Top 10	Tot. Sig. Func	Tot. Tar. func	Time (s)
clang_O0-clang_O2	0.193	0.525	0.911	15740	10141	262.4
clang_O0-clang_O3	0.194	0.517	0.91	15740	10100	208.9
clang_O0-gcc_O0	0.149	0.462	0.897	15740	14807	308.9
clang_O0-gcc_O2	0.241	0.509	0.872	15740	10579	206.3
clang_O0-gcc_O3	0.222	0.599	0.909	15740	10337	252.2
clang_O2-clang_O0	0.299	0.716	0.934	10141	15740	198.9
clang_O2-clang_O3	0.997	0.999	1	10141	10100	118.7
clang_O2-gcc_O0	0.2	0.651	0.925	10141	14807	212.9
clang_O2-gcc_O2	0.417	0.642	0.931	10141	10579	218.4
clang_O2-gcc_O3	0.511	0.92	0.983	10141	10337	232.6
clang_O3-clang_O0	0.298	0.71	0.929	10100	15740	244.2
clang_O3-clang_O2	0.99	0.997	0.998	10100	10141	136.2
clang_O3-gcc_O0	0.2	0.646	0.92	10100	14807	263.7
clang_O3-gcc_O2	0.414	0.638	0.928	10100	10579	183.3
clang_O3-gcc_O3	0.512	0.92	0.983	10100	10337	232.8
gcc_O0-clang_O0	0.419	0.897	0.981	14807	15740	423.5
gcc_O0-clang_O2	0.234	0.701	0.965	14807	10141	198.5
gcc_O0-clang_O3	0.235	0.697	0.963	14807	10100	300.0
gcc_O0-gcc_O2	0.307	0.64	0.959	14807	10579	351.0
gcc_O0-gcc_O3	0.246	0.703	0.947	14807	10337	362.7
gcc_O2-clang_O0	0.412	0.863	0.965	10579	15740	244.6
gcc_O2-clang_O2	0.434	0.891	0.99	10579	10141	170.2
gcc_O2-clang_O3	0.431	0.89	0.99	10579	10100	200.2
gcc_O2-gcc_O0	0.339	0.809	0.96	10579	14807	221.4
gcc_O2-gcc_O3	0.69	0.971	1	10579	10337	185.1
gcc_O3-clang_O0	0.291	0.573	0.705	10337	15740	226.6
gcc_O3-clang_O2	0.436	0.685	0.742	10337	10141	141.0
gcc_O3-clang_O3	0.439	0.688	0.743	10337	10100	187.7
gcc_O3-gcc_O0	0.204	0.506	0.684	10337	14807	231.42
gcc_O3-gcc_O2	0.614	0.664	0.721	10337	10579	142.9

TABLE III: Rank distribution of matched functions, in `BusyBox` binaries, across architectures

Rank 1	Top 3	Top 5	Top 10	Top 20	Top 50	Top 100
0.312	0.555	0.672	0.799	0.907	0.999	1.0

TABLE IV: For the 10 largest `coreutils` binaries, the best and the worst function model selected by BINGO. The % improvement denotes the improvement in ranking (top 10 positions) due to best model over worst.

	Best case		Worst case		
Binary	Model #	within top 10	Model #	within top 10	% imp.
stat	6	0.825	11	0.754	9.42
factor	4	0.700	13	0.583	20.07
sort	7	0.674	3	0.573	17.63
df	1	0.681	11	0.582	17.01
ls	4	0.473	15	0.438	7.99
dir	4	0.473	15	0.438	7.99
vdir	4	0.473	15	0.438	7.99
du	1	0.653	11	0.584	11.82
mv	1	0.699	11	0.592	18.07
cp	1	0.702	11	0.587	19.59

APPENDIX B
ALGORITHM FOR FUNCTION NORMALIZATION

Algorithm 2: `normalizeFunctions`$(\cdot)$ - Normalization process of assembly instruction. Here the notation $\bigoplus$ denotes the concatanation operation

Data: Set of control-flow graphs, CFG
Result: Set of normalized control-flow graphs, CFG_n

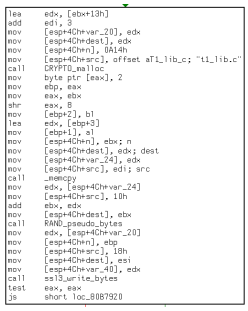
$CFG_n \longleftarrow \langle\rangle$
foreach control-flow graph *cfg* in CFG **do**
  $cfg_n \longleftarrow \langle\rangle$
  foreach basic-block *bb* in cfg **do**
    $bb_n \longleftarrow \langle\rangle$
    foreach instruction *i* in *bb* **do**
      $m :=$ `getMnemonic`(i)
      $O_n \longleftarrow \langle\rangle$ // normalized operand sequence
      if $m ==$ `call` **then**
        $o :=$ `getOperands`(`i`)
        if `operandType`$(o) ==$ `SYS_API` **then**
          $O_n \longleftarrow$ `abstractSystemAPI`(o)
        else
          // all other calls are considered function calls
          $O_n \longleftarrow$ `FuncCall`
      else
        foreach operand o in `getOperands`(`i`) **do**
          if `operandType`$(o) ==$ `Immediate` **then**
            $O_n \longleftarrow O_n \bigoplus$ `Imm`
          else if `operandType`$(o) ==$ `Register` **then**
            if `getRegType`$(o) ==$ `SP` **then**
              $O_n \longleftarrow O_n \bigoplus$ `Reg_SP` // stack pointer
            else if `getRegType`$(o) ==$ `BP` **then**
              $O_n \longleftarrow O_n \bigoplus$ `Reg_BP` // frame pointer
            else
              $O_n \longleftarrow O_n \bigoplus$ `Reg`
          else if `operandType`$(o) ==$ `Memory` **then**
            if `isGlobal`(o) **then**
              $O_n \longleftarrow O_n \bigoplus$ `Mem_Global`
            // stack related memory addressing
            else if `getBaseReg`$(o) ==$ `Stack` **then**
              $O_n \longleftarrow O_n \bigoplus$ `Mem_Stack`
            else
              $O_n \longleftarrow O_n \bigoplus$ `Mem`
      $i_n := \langle m, O_n \rangle$
      // normalized instructions are added to basic-blocks
      $bb_n \longleftarrow bb_n \bigoplus i_n$
    // normalized basic-blocks are added to cfg
    $cfg_n \longleftarrow cfg_n \bigoplus bb_n$
  $CFG_n \longleftarrow CFG_n \bigoplus cfg_n$
return CFG_n

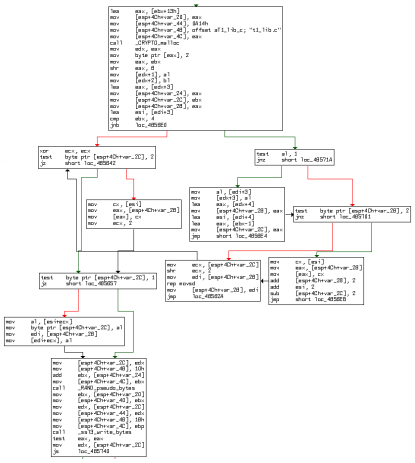
APPENDIX C
THE EXAMPLE OF HEARTBLEED VULNERABILITY (CVE-2014-0160)

```
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)

Fig. 11: 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

APPENDIX D
THE SYSTEM ARCHITECTURE OF BINGO

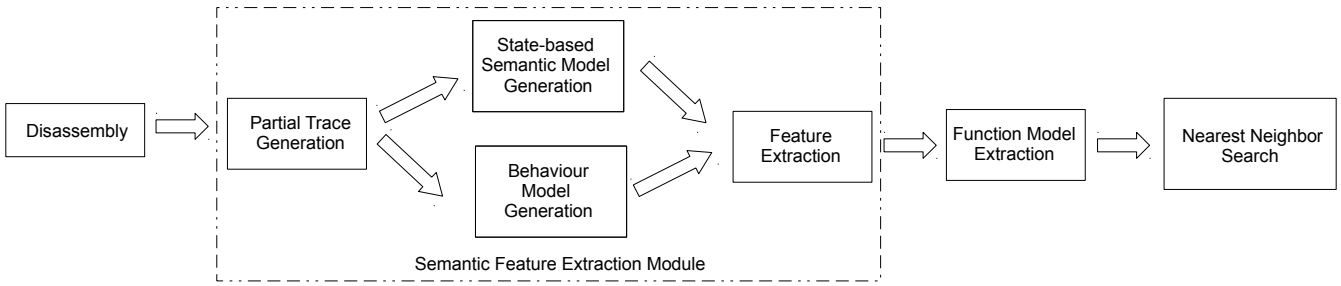


Fig. 12: BINGO: Overview of system architecture

APPENDIX E
MORE SUPPORTING EXPERIMENTAL DATA

TABLE V: Rank distribution of matched functions **with** abstract structural, in `coreutils` binaries, across architectures (ARM, x86-32 and x86-64). We present the individual ranks for largest 14 binaries and at the end present the average of all other binaries

Binary	arm-gcc_32		arm-gcc_64		gcc_32-arm		gcc_32-gcc_64		gcc_64-arm		gcc_64-gcc_32	
	Rank 1	Top 5	Rank 1	Top 5	Rank 1	Top 5	Rank 1	Top 5	Rank 1	Top 5	Rank 1	Top 5
cp	0.165	0.737	0.108	0.849	0.067	0.428	0.092	0.602	0.253	0.758	0.233	0.791
df	0.176	0.696	0.155	0.831	0.041	0.405	0.126	0.599	0.268	0.739	0.287	0.749
dir	0.114	0.653	0.064	0.771	0.052	0.342	0.103	0.532	0.181	0.739	0.246	0.837
du	0.163	0.705	0.173	0.855	0.047	0.384	0.101	0.576	0.246	0.676	0.247	0.697
factor	0.208	0.733	0.316	0.816	0.099	0.446	0.216	0.631	0.388	0.908	0.387	0.892
ls	0.114	0.653	0.064	0.771	0.052	0.342	0.103	0.532	0.181	0.739	0.246	0.837
mv	0.165	0.7	0.094	0.817	0.055	0.415	0.079	0.544	0.262	0.749	0.233	0.726
od	0.168	0.674	0.198	0.78	0.074	0.4	0.157	0.639	0.264	0.901	0.389	0.88
sort	0.153	0.729	0.092	0.815	0.085	0.475	0.105	0.665	0.225	0.763	0.205	0.755
split	0.198	0.813	0.283	0.967	0.063	0.448	0.157	0.735	0.261	0.826	0.373	0.922
stat	0.182	0.768	0.245	0.936	0.071	0.495	0.183	0.67	0.277	0.872	0.339	0.862
stdbuf	0.184	0.816	0.386	0.976	0.103	0.494	0.242	0.737	0.349	0.952	0.394	0.909
tail	0.179	0.813	0.13	0.917	0.045	0.5	0.115	0.68	0.25	0.815	0.287	0.836
vdir	0.114	0.653	0.064	0.771	0.052	0.342	0.103	0.532	0.181	0.739	0.246	0.837
Avg. large	0.163	0.725	0.169	0.848	0.065	0.423	0.134	0.620	0.256	0.798	0.294	0.824
Avg. all other	0.273	0.889	0.3622	0.973	0.099	0.608	0.260	0.762	0.367	0.762	0.466	0.919

TABLE VI: Rank distribution of matched functions **without** abstract structural similarity, in `coreutils` binaries, across architectures (ARM, x86-32 and x86-64). We present the individual ranks for largest 14 binaries and at the end present the average of all other binaries

Binary	arm-gcc_32		arm-gcc_64		gcc_32-arm		gcc_32-gcc_64		gcc_64-arm		gcc_64-gcc_32	
	Rank 1	Top 5	Rank 1	Top 5	Rank 1	Top 5	Rank 1	Top 5	Rank 1	Top 5	Rank 1	Top 5
cp	0.121	0.618	0.072	0.758	0.052	0.335	0.085	0.539	0.144	0.595	0.149	0.546
df	0.096	0.622	0.034	0.746	0.03	0.333	0.086	0.586	0.136	0.568	0.164	0.569
dir	0.075	0.532	0.039	0.732	0.023	0.266	0.066	0.636	0.087	0.52	0.149	0.554
du	0.111	0.614	0.069	0.694	0.018	0.298	0.065	0.435	0.111	0.5	0.109	0.471
factor	0.09	0.652	0.08	0.773	0.045	0.303	0.125	0.639	0.133	0.653	0.153	0.611
ls	0.075	0.532	0.039	0.732	0.023	0.266	0.066	0.636	0.087	0.52	0.149	0.554
mv	0.134	0.603	0.096	0.692	0.045	0.346	0.054	0.497	0.16	0.577	0.15	0.51
od	0.083	0.583	0.066	0.658	0.048	0.298	0.067	0.627	0.105	0.763	0.133	0.707
sort	0.117	0.58	0.05	0.693	0.056	0.352	0.055	0.579	0.107	0.521	0.097	0.503
split	0.094	0.647	0.066	0.842	0.035	0.306	0.116	0.725	0.118	0.632	0.145	0.609
stat	0.124	0.64	0.101	0.886	0.045	0.337	0.113	0.803	0.114	0.671	0.169	0.761
stdbuf	0.104	0.701	0.045	0.881	0.065	0.39	0.143	0.857	0.134	0.761	0.206	0.714
tail	0.131	0.697	0.081	0.791	0.03	0.364	0.073	0.756	0.116	0.593	0.122	0.646
vdir	0.075	0.532	0.039	0.732	0.023	0.266	0.066	0.636	0.087	0.52	0.149	0.554
Avg. large	0.102	0.611	0.0626	0.757	0.0384	0.318	0.0842	0.639	0.117	0.599	0.146	0.593
Avg. all other	0.161	0.784	0.204	0.912	0.0589	0.460	0.160	0.899	0.157	0.791	0.194	0.811

TABLE VII: Ranks obtained by BINGO for functions in `msvcrt.dll` (closed-source) against `libc.so` (open-source).

Ranks	Matched functions	No. of Functions	%	Cum. %
1	`tolower, memset, wcschr, wcsncmp, toupper, memcmp, strspn, wcsrchr, memchr, log`	10	15.6	15.6
2-3	`wcstoul, wcstol, strtoul, fopen, strncpy, strtol, itoa, wcscmp, wcsncat, itow, strrchr, strchr, longjmp, strcspn, wcsncpy, labs, strpbrk, toupper, write, memcpy, memmove, tolower, modf`	23	35.9	51.6
4-5	`mbtowc, wcstombs, strcat, remove, mbstowcs, wctomb`	6	9.3	60.9
6-10	`rename, strstr, wcspbrk, iswctype, strtok, wcscoll, strcoll, setlocale, qsort, wcsspn, swprintf, wcstod, strerrorr`	13	2.3	81.3
11-20	`strncat, raise, strtod, wcscspn, wcstok, atol`	6	9.3	90.6
21-50	`ldexp, perror, system, wcsxfrm`	4	6.3	96.9
51-100	`frexp, strxfrm`	2	3.1	100.0