vSIM: Semantics-Aware Value Extraction for Efficient Binary Code Similarity Analysis

Huaijin Wang
The Ohio State University

Zhiqiang Lin
The Ohio State University

***Abstract*—Binary Code Similarity Analysis (BCSA) plays a vital role in many security tasks, including malware analysis, vulnerability detection, and software supply chain security. While numerous BCSA techniques have been proposed over the past decade, few leverage the semantics of register and memory *values* for comparison, despite promising initial results. Existing value-based approaches often focus narrowly on values that remain invariant across compilation settings, thereby overlooking a broader spectrum of semantically rich information. In this paper, we identify three core challenges limiting the effectiveness of value-based BCSA: (1) unscalable value extraction that fails to cover diverse value-producing behaviors, (2) insufficient noise filtering that allows semantically irrelevant artifacts (e.g., global addresses) to dominate, and (3) inefficient comparison that makes value-based matching expensive and brittle. To make value-based BCSA practical at scale, we propose vSIM, a novel framework that systematically captures values computed from register and memory operations, filters out semantically irrelevant values (e.g., global addresses), and normalizes and propagates the remaining values to enable robust and scalable similarity analysis. Extensive evaluation shows that vSIM consistently outperforms state-of-the-art BCSA systems in accuracy, robustness, and scalability, and generalizes across architectures and toolchains, delivering reliable results on diverse real-world datasets.**

I. INTRODUCTION

Binary code similarity analysis (BCSA) aims to detect similarities between binary code segments, underpinning numerous critical security tasks such as malware analysis [1]–[3], vulnerability detection [4]–[9], plagiarism detection [10]–[13], and software supply chain security [14]–[18]. However, BCSA is inherently challenging due to the loss of high-level information (e.g., variable names, function symbols, data types) during compilation. Moreover, compiler optimizations and architectural differences can induce substantial syntactic divergence even among binaries produced from the same source code, further complicating reliable similarity matching [4], [6], [12], [19], [20].

To address these challenges, advanced BCSA techniques have been proposed to capture the underlying semantics of binary code, i.e., its core functionality, since semantics remain stable despite syntactic variations. Existing BCSA approaches can broadly be classified into four categories: (1) learning semantics from raw bytes [21] or assembly code [4], [5], [19], (2) comparing binary code based on input-output (I/O) equivalence [7], [22], [23], (3) analyzing program states post-execution [8], [12], [24], and (4) examining specific invariant values [3], [25]. These methods typically rely on either traditional program analysis techniques or machine learning (ML), each inheriting distinct limitations such as poor scalability [26], limited code coverage [27], [28], and inability to generalize well to out-of-distribution (OOD) samples [29], [30].

Meanwhile, in practice, many security tasks, particularly those related to software supply chains [14], [16], [31], [32], require analyzing massive binary corpora efficiently. Consequently, recent BCSA efforts increasingly adopt ML-based embedding approaches for rapid semantic comparisons. Yet, such ML models frequently encounter robustness issues due to diverse optimizations and prevalent OOD binaries, leading to degraded accuracy [5], [33]. Motivated by these limitations, we propose a non-ML approach that extracts robust, semantics-aware *values* to approximate binary code semantics.

More specifically, we propose vSIM, a novel value-based BCSA framework for accurate, robust, and scalable analysis. Unlike previous approaches that narrowly focus on source-level semantic values (e.g., return results) [3], [23], [25], [34], vSIM systematically captures and utilizes the intermediate register and memory values encountered during under-constrained symbolic execution [26]. These intermediate values offer crucial semantic information often ignored by prior techniques. vSIM further addresses three key challenges inherent to value-based BCSA: unscalable value extraction, semantics-aware value selection, and inefficient value comparison, demonstrating the feasibility and effectiveness of leveraging values for BCSA.

vSIM's workflow consists of four key steps: (1) It employs a customized under-constrained symbolic execution engine to scalably extract register and memory values, capturing essential semantic details at the basic-block level. (2) It applies six heuristics, inspired by common disassembly practices [35]–[37], to retain semantics-aware values and discard irrelevant ones (e.g., global variable addresses). (3) The semantics-aware values are then normalized and concretized to construct function fingerprints, enabling efficient similarity comparison. (4) These fingerprints are further enhanced by propagating callees' fingerprints to callers and incorporating the distinguishability of captured values, significantly boosting robustness and accuracy.

To evaluate vSIM, we perform extensive experiments across three large datasets covering diverse compilation scenarios. When compared against state-of-the-art (SoTA) BCSA tools

Network and Distributed System Security (NDSS) Symposium 2026
23 - 27 February 2026 , San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240213
www.ndss-symposium.org

(e.g., `jTrans` [19]), VSIM demonstrates substantial accuracy improvements exceeding 40% in cross-optimization and cross-compiler scenarios. Additionally, it achieves comparable precision in cross-architecture scenarios with only a 1.5% overhead of alternative approaches such as GMN [6], highlighting VSIM's superior accuracy, robustness, and scalability.

Contributions. We make the following contributions:

- We systematically analyze the challenges of value-based BCSA, clearly identifying the primary obstacles preventing broader adoption of value semantics, motivating our exploration of semantics-aware value-based techniques.
- We present VSIM, a robust, scalable, interpretable, and open source framework for function-level BCSA leveraging semantic values derived directly from under-constrained symbolic execution. VSIM is available at https://github.com/OSUSecLab/vSim.
- We evaluate VSIM across diverse scenarios, demonstrating superior performance over SoTA methods in terms of accuracy, scalability, and vulnerability detection capabilities.

II. Background, Related Work, and Motivation

A. Problem Definition

Binary Code Similarity Analysis (BCSA) quantitatively evaluates the similarity between binary code segments. If two binary segments exhibit a high similarity score, they likely share equivalent or closely related functionalities. This capability is critical for security tasks such as malware analysis and vulnerability detection.

Various BCSA techniques have been proposed to support similarity analysis at multiple granularities, including basic blocks [38], [39], functions [4], [5], [7], [10], [11], [13], [15], [19], [22], [23], [40]–[47], execution traces [3], [48], and entire programs [49], [50]. Among these, *function-level* BCSA has received the most attention and is extensively studied [51]. To facilitate a fair and meaningful comparison with SoTA methods, we also focus on function-level analysis, adhering to a widely accepted definition [15], [19], [52].

Definitions. A *binary function* is defined as a sequence of binary instructions produced by compiling a single source-level function, potentially including compiler optimizations such as inlining. Formally, given a query binary function f_q and a pool of binary functions $\mathcal{P}$, the goal of function-level BCSA is to identify the top-k functions from $\mathcal{P}$ that are most similar to f_q.

B. Related Work

Binary functions are typically compiled from high-level programming languages, a process that inherently strips away semantic details such as variable names and types when translating source code into machine instructions [6], [51]. This loss of human-readable context significantly complicates understanding binary functionality and poses a fundamental challenge to BCSA. Recent work has also explored recovering source-level hints such as function names from stripped binaries using machine learning and large language models [53]–[57], further highlighting the importance of learning robust binary semantics. At the same time, binaries with identical functionality can differ substantially in their syntax due to variations in compilers, optimization strategies, and architectures, which complicates accurate similarity assessment. Consequently, modern BCSA techniques place particular emphasis on extracting semantic, rather than purely syntactic, features for meaningful comparisons.

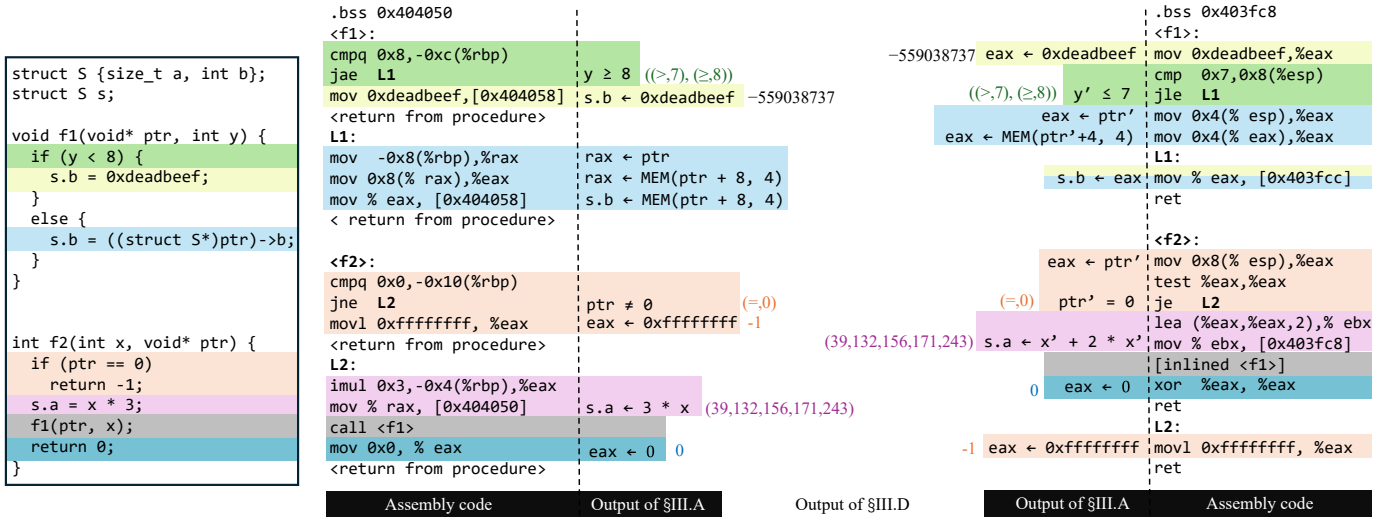
TABLE I: Source of semantic features, comparison methods, and supported characteristics of BCSA solutions for binary functions. The ↓ after the comparison method indicates that the method is insufficiently scalable for large-scale binary code analysis.

Tool	Year	Raw bytes	Assembly code	Program I/Os	Program states	Selected values	Theorem proving ↓	Graph matching ↓	Embedding distance	Comparison model ↓	Other Alg.	Multi-architecture	No need to train	Inlining resilience
DiscovRE [58]	2016	·	●	·	·	·	·	●	·	·	·	●	●	·
Genius [59]	2016	·	●	·	·	·	·	●	·	·	·	●	●	·
BINGO [8]	2016	·	·	●	◗	·	●	·	·	●	·	●	·	❍
BinSim [3]	2017	·	·	·	·	●	●	·	·	·	·	·	●	●
IMF-SIM [7]	2017	·	·	●	·	·	·	·	·	●	·	·	·	●
CoP [12]	2017	·	·	·	●	·	●	●	·	·	·	·	●	·
GEMINI [10]	2017	·	●	·	·	·	·	·	●	·	·	●	·	·
αdiff [21]	2018	●	·	·	·	·	·	·	·	●	·	●	·	·
VulSeeker [9]	2018	·	●	·	·	·	·	·	●	·	·	●	·	·
INNEREYE [11]	2018	·	●	·	·	·	·	●	●	·	·	●	·	·
Asm2vec [4]	2019	·	●	·	·	·	·	·	●	·	·	●	·	❍
DEEPBINDIFF [38]	2020	·	●	·	·	·	·	●	●	·	·	●	·	·
PALMTREE [5]	2021	·	●	·	·	·	·	·	●	·	·	·	·	·
Codee [60]	2021	·	●	·	·	·	·	·	●	·	·	●	·	·
BinUSE [25]	2022	·	·	·	·	●	●	·	·	·	●	●	·	●
BinKit [61]	2022	·	●	·	·	·	·	·	·	·	●	●	·	·
TREX [62]	2022	·	●	◗	·	·	·	·	●	·	·	●	·	·
GMN [6]	2022	·	●	·	·	·	·	·	●	·	·	●	·	·
FIRMSEC [63]	2022	·	●	·	·	·	·	·	●	·	·	●	·	·
XBA [64]	2022	·	●	·	·	·	·	·	●	·	·	·	·	·
jTrans [19]	2022	·	●	·	·	·	·	·	●	·	·	·	·	·
VULHAWK [65]	2023	·	◗	·	·	·	·	·	●	·	●	●	·	·
sem2vec [24]	2023	·	·	·	◗	·	·	·	●	·	·	●	·	·
PEM [23]	2023	·	·	●	·	·	·	·	·	·	●	●	●	●
Hermessim [66]	2024	·	◗	·	·	·	·	·	●	·	·	●	·	·
CI-Detector [33]	2024	·	●	·	·	·	·	·	●	·	·	●	·	●
BinAug [52]	2024	·	●	·	·	·	·	·	●	·	·	●	·	·
δCFG [67]	2024	·	●	·	·	·	·	·	●	·	·	●	·	·
CEBin [41]	2024	·	●	·	·	·	·	·	●	●	·	·	·	·
Clap [68]	2024	·	●	·	·	·	·	·	●	·	·	·	·	·

● Selected semantic feature, comparison approach, or fully supported characteristic.
◗ The feature is not directly used for similarity analysis.
❍ The characteristic is partially supported. XBA supports cross-platform analysis (e.g., Linux ELF and Windows PE). BINGO and Asm2vec leverage selective inlining to improve the function inlining resilience capability.

In this work, we extend the previous comprehensive survey by Haq and Caballero [51], which covered BCSA techniques up to 2020, by incorporating the recent developments until 2025. Specifically, we analyze 30 influential studies published over the past decade that leverage semantic features for binary similarity evaluation (see Table I). While this review does not claim to be exhaustive, the selected works sufficiently illustrate prevailing trends in BCSA methodologies.

Within these reviewed studies, we identify five primary semantic feature sources: raw bytes, assembly code, program input-output pairs (I/Os), program execution states, and specifically selected values. Accordingly, we categorize existing methods into four representative classes based on their core methodologies: (1) learning semantics directly from raw bytes or assembly code, (2) verifying equivalence via program I/Os, (3) assessing similarity through program execution states, and (4) matching selected representative values. To facilitate our discussion of the technical challenges in existing approaches, we use an illustrative example in Figure 1. Specifically, the source code shown in Figure 1(a) is compiled



(a) Source code (b) Compiled by clang –O0 (c) Compiled by gcc –m32 –O3

Fig. 1: A contrived motivating example. Matching highlighted code snippets in (b) and (c) correspond to the source code in (a). From left to right, (b) shows the assembly code, collected store operations after §III-A Value Extraction, and the elements of function fingerprints (after §III-C Normalization and §III-D Concretization); (c) shows them from right to left. The fingerprints of `f1` and `f2` of (b) and (c) are presented in Figure 8.

into two binary functions for AMD64 and X86 architectures, depicted in Figure 1(b) and Figure 1(c), respectively.

(1) Learning Semantics from Raw Bytes or Assembly Code. As summarized in Table I, assembly code stands out as the predominant semantic feature for BCSA, leveraged in 22 out of the 30 surveyed studies from the past decade. Notably, 19 of these studies generate embeddings for binary functions or basic blocks, facilitating efficient similarity computations via metrics such as cosine or Euclidean distance. Earlier approaches like DiscovRE [58] and GENIUS [59] employed graph matching techniques on Control Flow Graphs (CFGs), while recent studies pursued interpretable, resilient features robust against binary variations [24], [61], [62], [66]. Besides embedding-based methods, GMN [6] directly utilizes a comparison model that computes similarity scores without generating embedding representations, which requires more computational resources for retrieval but may offer improved accuracy [6], [41]. Raw bytes are generally considered less informative, as their semantics are only revealed through disassembly; consequently, `αdiff` [21] is the only work relying on raw bytes for BCSA.

Despite impressive benchmark results, ML-based BCSA methods continue to face significant robustness challenges. Observation-❶: *ML models are commonly trained on binaries generated from a restricted set of compilers. When encountering binaries compiled with unseen or newly-released compilers, their performance can deteriorate significantly [5], [52]* (see also §IV-B1). Observation-❷: *Only a limited number of studies try to mitigate the impact of function inlining, a major practical challenge for BCSA*. Most methods analyze assembly instructions of binary functions in isolation, ignoring callees, thus potentially biasing similarity assessments under varying inlining strategies. Notably, GMN mitigates this impact by explicitly disabling function inlining optimization when constructing its dataset. Although selective inlining approaches [4], [8] have been proposed, recent analyses suggest these techniques inadequately capture real-world compiler inlining behaviors, which continuously evolve [69].

(2) Verifying Equivalence via Program I/Os. Previous works [7], [22], [23] execute binaries with concrete inputs and track their corresponding outputs. Two binaries are deemed functionally equivalent if they yield identical outputs for the same inputs. However, since these inputs are typically generated randomly, given that BCSA methods usually lack prior knowledge about the binary, many code blocks may remain unexecuted, resulting in incomplete semantic extraction due to *low code coverage*. For example, the branch condition `if (ptr == 0)` in Figure 1(a) is satisfied only when the input `ptr` equals zero, a condition rarely met by random input generation.

To mitigate the low coverage issue, IMF-SIM [7] and PEM [23] manipulate binary execution and increase code coverage. *These dynamic approaches sample lots of inputs for each binary function; thus, many binary code blocks are executed multiple times, which is unscalable and inefficient* (Observation-❸). Moreover, BLEX [22] and PEM manipulate the instruction pointer to execute uncovered code, which may lead to incorrect results since many execution traces are infeasible. Alternative solutions include TREX [62] and BINGO [8], employing ML-based and program state-based methods, respectively. Specifically, TREX utilizes a lightweight technique to generate dummy inputs for isolated code fragments, leveraging program I/O behaviors to guide its model training. BINGO will be detailed in the subsequent discussion.

(3) Assessing Similarity through Program States. Program states are often captured from execution traces of binary code. Before execution, an initial state (*pre-state*) is established, and the resulting state after execution (*post-state*) is recorded [8], [12]. The differences between these two states reflect the behavior and effect of the executed code segment. Typically, program states include the contents of general-purpose registers, memory, and processor flags. To extract these transitions, symbolic execution is employed to collect symbolic expressions for all updated components.

BINGO [8] and COP [12] operate by slicing partial traces from a binary function's control flow graph (CFG) and collecting sets of program states from these slices. To assess similarity between two binary functions, BINGO applies a comparison model to evaluate the similarity of their respective program states, while COP employs a graph matching algorithm to align program states along the CFGs of the functions.

However, *capturing entire program states may incorporate irrelevant information, compromising analysis accuracy* (Observation-❹). For instance, accessing `((struct S*)ptr)->b` in Figure 1(b) and Figure 1(c) corresponds to offsets `ptr + 8` and `ptr + 4`, respectively, due to architecture-specific differences in the size of preceding attributes such as `size_t a`. Differences in structure alignment rules can further alter memory layouts and the resulting symbolic expressions, thereby degrading accuracy. A promising mitigation is to selectively extract only meaningful symbolic expressions and literals, i.e., those involving selected values of registers, memory, and flags, for similarity analysis, which will be discussed subsequently.

(4) Matching Selected Values. Similar to using program states, selected values are derived from the binary code via symbolic execution [3], [25]. However, instead of capturing entire pre- and post-states, these approaches examine how specific values are used within binary code. By identifying values employed in meaningful contexts, such as external function call arguments and return values, these methods mitigate the impact of irrelevant expressions. Existing works [3], [25] indicate selected values achieve promising BCSA results. For example, `BinSim` detects similarities among obfuscated malware programs, and BinUSE accurately identifies software clones across compilers, optimization levels, and architectures.

C. Motivation

Having reviewed the recent BCSA literature, we have noticed that Observation-❶, ❷, and ❹ associated with capturing precise and robust semantics from assembly and program states. Additionally, dynamic approaches relying on program I/Os are constrained by low code coverage and scalability (Observation-❸). These difficulties primarily stem from inherent limitations of underlying techniques.

To overcome these observed difficulties, this work is positioned to address a critical, underexplored area in value-based BCSA (Figure 2). Existing approaches are bifurcated. Methods that select values often restrict themselves to easily identifiable markers like return and external function calls,

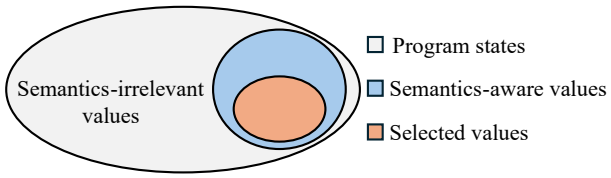


Fig. 2: Position of our motivation in value-based BCSA.

thereby overlooking a wealth of semantic information present elsewhere. In contrast, approaches like BINGO and COP capture extensive program states, a process that is inefficient and susceptible to noise from semantically irrelevant data. Our position is that a more effective strategy involves the systematic extraction of semantics-aware values directly from the binary. Realizing this vision, however, necessitates overcoming three principal challenges: scalable value extraction, semantics-aware value selection, and efficient value comparison.

C1: State explosion prohibits scalable value extraction. State explosion is an internal challenge of symbolic execution [70]. For example, to analyze the binary code of function `f1` in Figure 1(b), a symbolic engine symbolizes the variable `y` stored at `-0x10(%rbp)` and simulates the instruction `cmpq 0x8,-0x10(%rbp)`, which sets the flags with symbolic expressions. Next, it simulates the branch instruction `jae L1`, doubling the program states with constraints $y \geq 8$ and $y < 8$. With the increasing number of branches, the number of states grows exponentially, leading to the *state explosion problem*. Thus, lengthy execution traces, which is often needed to reach external function calls or returns, exacerbate this problem, hindering scalable value extraction.

C2: Semantics-aware value selection is non-trivial. While using external calls and returns as selected values is practical, determining semantic relevance for other binary code values remains challenging since linking binary-level values to their source code semantics is still difficult. Consequently, existing methods overlook potentially meaningful values, limiting useful signature extraction. For example, function `f1` (Figure 1(a)) neither returns values nor invokes external functions, thus presenting few obvious semantic values. Therefore, a generalized approach to value selection, agnostic to direct source-level correspondences, is needed for comprehensive binary analysis. Different from existing approaches that explicitly define semantic values to be used [3], [25], we aim to identify semantically-irrelevant values, which can then be filtered out to focus on the most relevant information.

C3: Inefficient and difficult value comparison. Even when robust and informative values are extracted, rapidly comparing them remains challenging. For instance, semantically equivalent expressions such as `3 * x` and `x' + 2 * x'` (Figure 1(b) and (c), respectively) differ syntactically. Consequently, time-consuming theorem provers are typically used for equivalence checking [3], [25]. Moreover, values with equivalent semantics may differ structurally. Consider the code highlighted in orange in Figure 1(b) and Figure 1(c): if

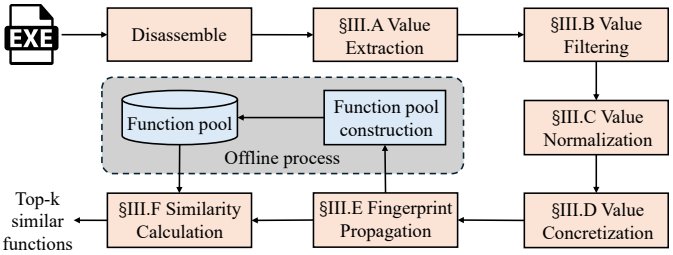


Fig. 3: The workflow of VSIM.

```
mov -0x8(%rbp), %rax     t1 ← LReg(rbp) - 8      (L, rbp, 0x7fff0000)
                         ptr ← LMem(t1, 8)       (L, 0x7fefff8, ptr)
                         rax ← ptr               (S, rax, ptr)
mov 0x8(%rax), %eax      t2 ← LMem(ptr+8, 4)     (L, 0x2000000, t2)
                         rax ← 32Uto64(t2)       (S, rax, 32Uto64(t2))
mov %eax, [0x404058]     SMem(0x404058, t2)      (S, 0x404058, t2)
```

Assembly | **Intermediate representation (IR)** | **Collected operations**

Fig. 4: Value extraction for basic block **L1** of Figure 1(b). In the lifted IR, the ← operator assigns a value to a variable; `LReg(`r`)` loads a value from register r; `LMem(`a`, `n`)` loads n bytes from memory address a; `32Uto64(`v`)` zero-extends a 32-bit value v to 64 bits; `SMem(`a`, `v`)` stores value v to memory address a. In collected operations, `L` and `S` are load and store operations, respectively.

`-0x10(%rbp)` and `0x8(%esp)` store symbolic variables `ptr` (8 bytes) and `ptr'` (4 bytes), the resulting path constraints `ptr` $\neq$ `0` and `ptr'` = `0` cannot be directly compared due to differences in pointer sizes between architectures.

III. Design

This section illustrates how we design our framework, VSIM, to address the challenges identified in §II. Figure 3 shows the workflow of VSIM. After disassembling the input binary executable, the workflow of VSIM can be summarized into six main steps: value extraction, value filtering, value normalization, value concretization, fingerprint propagation, and similarity calculation. Since we aim at function-level similarity analysis, given a query function f_q, VSIM's output is the pool $\mathcal{P}$'s functions ordered by their similarity score to f_q.

To efficiently and comprehensively extract values from binary code for similarity analysis (**C1**), we customize a symbolic execution engine to perform basic block-level symbolic execution (it is also referred as under-constrained symbolic execution [26]), recording all changes in registers and memory. To identify meaningful semantic values from the extracted data (**C2**), we apply heuristic rules commonly used in binary disassembly and rewriting, filtering out addresses such as global variable pointers. The remaining values are used for function similarity analysis. For fast and reliable similarity comparison (**C3**), VSIM avoids heavyweight theorem proving but carefully normalizes and concretizes the semantic values for direct, efficient similarity comparison. Moreover, VSIM propagates the callees' fingerprints to the caller's fingerprint and considers distinguishabilities for different values. Lastly, the similarity scores are computed with the fingerprints of the query function and the functions in the pool, and the top-k most similar functions are returned.

A. Value Extraction

Before extracting values from binary code, VSIM disassembles the binary into assembly instructions and recovers its function entries, a common practice in function-level BCSA techniques [10], [13], [15], [19], [40]. For a query binary function f_q, VSIM uses a customized symbolic execution engine to extract operations by comprehensively recording register and memory loads and stores during the execution of f_q.

Basic Block Granularity Selection. Although analyzing an entire function captures complete semantics, it often leads to state explosion [8], [25]. Existing methods either analyze sequences of basic blocks on a CFG (also called "tracelet") [4], [8], [53], [71] or a basic block [38], [72]. VSIM collects values at the basic block level, which offers two benefits for efficiency:

- Each basic block is simulated only once, avoiding redundant computations. Other larger granularities (e.g., execution paths) require simulating a sequence of basic blocks, and the overlapping basic blocks are simulated multiple times.
- A basic block has a single entry and exit point; thus, there is no need for SMT solvers to check the satisfiability of path constraints since all instructions are executed in order.

We acknowledge that analyzing each basic block independently overlooks the inter-block relations and loses some contexts. Nevertheless, our work deliberately focuses on harnessing values for BCSA, while incorporating basic block interactions needs a comprehensive analysis of the control flow structure, adding considerable complexity. Notably, leveraging value information alone achieves SoTA performance at the function level (see §IV-B1). Moreover, our ablation study (Appendix A) shows that simply considering 2- or 3-length basic block sequence introduces significant overheads but cannot achieve considerable accuracy improvement.

Basic Blocks-Level Symbolic Execution. VSIM depends on under-constrained symbolic execution [26], which allows the symbolic engine to start the simulation from an arbitrary instruction of a program and symbolizes inputs on the fly. Figure 4 shows an example of simulating the first three instructions of basic block **L1** from Figure 1(b). Existing symbolic execution engines often lift the binary code into intermediate representations (IRs) to facilitate the simulation and multi-architecture support [73]–[75]. During simulating IRs, the engine collects all "Load" and "Store" operations on registers and memory.

Initially, registers and memory are uninitialized except for read-only sections (e.g., `.rodata` and `.text`) and special registers (e.g., `rbp` and `rsp`)[1]. For example, the engine first

[1]Concretizing the base and stack pointer before starting under-constrained symbolic execution is a common practice since it alleviates the need for symbolic reasoning the layout of the stack [26], [74].

TABLE II: Values extracted by VSIM of each operation. "C" and "S" denote the concrete and symbolic values, respectively.

Operation		Available values
Register	Load	Register name (C) and the loaded value (S&C)
	Store	Register name (C) and the stored value (S&C)
Memory	Load	Address (S&C) and the loaded value (S&C)
	Store	Address (S&C) and the stored value (S&C)

loads a concrete value from `rbp` (the stack frame base pointer), assigning it to temporary variable `t1`. Next, it loads eight bytes from the memory address in `t1`. Since that memory is uninitialized, the engine symbolizes these bytes as a symbolic value `ptr`[2]. In the next IR, `ptr` is stored into register `rax`.

For the second instruction, the engine has to load four bytes from the memory address pointed to by `ptr + 8`. However, the symbolic nature of `ptr` prevents direct address resolution. To proceed, VSIM records `ptr + 8`, assigns it a concrete value v (e.g., `0x2000000`), and maintains a mapping between v and `ptr + 8`. If another instruction uses the value of `ptr + 8`, VSIM replaces `ptr + 8` with v to continue the simulation. With this concretized address, the engine loads from memory, symbolizes the loaded bytes as a 32-bit value $t2$, extends $t2$ to 64 bits, and stores it in `rax`. Finally, $t2$ is stored at memory address `0x404058` (a pointer to the `.bss` section).

It is evident that this strategy for concretizing `ptr + 8` cannot guarantee the feasibility of the memory address and hurts the soundness of our symbolic engine. However, this limitation is acceptable for basic block analysis. The sequential execution of a basic block ensures that constraint satisfiability need not be checked. Moreover, the subsequent value filtering mitigates the impact of the infeasible memory addresses since the values being used as memory addresses are irrelevant to the functionality and can be ignored (see §III-B).

Comprehensively Record the Values. To achieve this goal, VSIM tracks all "Load" and "Store" operations on registers and memory during the symbolic execution. Table II summarizes the extracted values: for registers, both the register name (concrete) and the loaded or stored value (concrete or symbolic) are recorded; for memory, both the address and the value are captured. Formally, VSIM records a set of register values V_{reg} and a set of memory values V_{mem}, where each register tuple is $(op, name, v)$ and each memory tuple is $(op, addr, v)$, with $op \in$ {Load, Store}. Additionally, branch conditions—typically stored in flag registers—are recorded immediately after simulating branch instructions.

Figure 4 displays the operations collected after simulating the basic block, while Figure 1(b) presents the store operations in a more readable format, with the left and right sides of $\leftarrow$ representing the target and the stored value, respectively. Particularly, the value v of any operation is a bit vector with a fixed size, which can be either concrete or symbolic. A concrete value denotes that all bits are known (e.g., `0` in Figure 1(b)), while a symbolic value denotes that some bits are unknown and represented by symbolic expressions (e.g., `3 * x` in Figure 1(b)).

[2]We name the symbolic value with "ptr" for readability, while the engine does not assume it is a pointer and does not know it is the "ptr" in the source code.

B. Value Filtering

The value extraction process generates numerous register and memory operations within a basic block, capturing the values used during symbolic execution. However, not all these values are essential to the binary function's semantics. In other words, many can vary across different compilations of the same source code. Existing works neglect the impact of these differences (e.g., COP [12]) while using the program states, which may lead to inaccurate similarity results. Others manually select return results or arguments of external calls but miss abundant semantically significant values (e.g., `BinSim` and BinUSE).

Defining semantically significant values—those that capture a function's core computational logic—is difficult across all contexts. Therefore, instead of trying to identify these values directly, VSIM adopts a pratical, subtractive approach. It filters out semantically irrelevant information, such as pointers and memory addresses, which are common in binary code but add little value for similarity analysis. The remaining values are thus considered semantics-aware, effectively isolating the computational essence of the function.

For example, as shown in Figure 1(b) and Figure 1(c), the global variable "`struct S s`" is stored in different memory addresses, and the symbolic expression pointing to `((struct S*)ptr)->b` can be `ptr + 8` and `ptr + 4` in different binaries. Although they may be useful for detecting structural similarities on the memory layout aspect, they do not contribute to the core logic of a program and can vary across different compilation settings. Consequently, VSIM removes these address values and pays attention to the remaining values, which are more likely to be semantically significant.

Algorithm 1 Value filtering

Require: $V_{reg} = \{(op_i, name_i, v_i)\}$, register value set.
Require: $V_{mem} = \{(op_j, addr_j, v_j)\}$, memory value set.
Ensure: V_S: the set of semantic values
1: $V_S \leftarrow \emptyset$
2: $A \leftarrow \{addr | (op, addr, v) \in V_{mem}\}$ ▷ Address values
3: // Update A according to external calls
4: **for** (op, _, v) in $V_{reg} \cup V_{mem}$ **do**
5: **if** $v \notin A \wedge op =$ Store **then**
6: **if** concrete(v) $\wedge$ $\neg$concrete_address(v) **then**
7: $V_S \leftarrow V_S \cup \{v\}$
8: **else if** symbolic(v) $\wedge$ $\neg$symbolic_address(v) **then**
9: $V_S \leftarrow V_S \cup \{v\}$
10: **end if**
11: **end if**
12: **end for**
13: **return** V_S ▷ Identified addresses are removed

Algorithm. To determine if a value is an address, VSIM relies on heuristics common in reverse engineering tasks [35], [76]–[78]. Algorithm 1 outlines this process for removing semantically irrelevant values. The algorithm begines with two sets of extracted values: register values (V_{reg}) and memory values (V_{mem}). Since each element in V_{mem} is a tuple

A is the set of address values (line 2-3 of Algorithm 1)
$D \leftarrow \{[S.start, S.end] \mid S \in \text{data sections}\}$
$E \leftarrow \{[S.start, S.end] \mid S \in \text{executable sections}\}$
$bp \leftarrow$ Initial concrete value stored in base pointer
$\epsilon \leftarrow$ Configurable stack size threshold
$\mathbf{N}(e)$: Return the set of subtrees in e's expression tree

$\mathbf{HC1}(v) = \exists (start, end) \in D : v \in [start, end]$
$\mathbf{HC2}(v) = \exists (start, end) \in E : v \in [start, end]$
$\mathbf{HC3}(v) = |v - bp| \leq \epsilon$

$\mathbf{HS1}(e) = \exists v \in \mathbf{N}(e) : \mathbf{HC1}(v) \vee \mathbf{HC2}(v) \vee \mathbf{HC3}(v)$
$\mathbf{HS2}(e) = \exists n \in \mathbf{N}(e) : n \in A$
$\mathbf{HS3}(e) = \exists \text{addr} \in A : e \in \mathbf{N}(\text{addr})$

$$\frac{\mathbf{HC1}(v) \vee \mathbf{HC2}(v) \vee \mathbf{HC3}(v)}{A \leftarrow A \cup \{v\} \qquad \text{concrete\_address}(v) = \text{True}}$$

$$\frac{\neg(\mathbf{HC1}(v) \vee \mathbf{HC2}(v) \vee \mathbf{HC3}(v))}{\text{concrete\_address}(v) = \text{False}}$$

$$\frac{\mathbf{HS1}(e) \vee \mathbf{HS2}(e) \vee \mathbf{HS3}(e)}{A \leftarrow A \cup \{e\} \qquad \text{symbolic\_address}(e) = \text{True}}$$

$$\frac{\neg(\mathbf{HS1}(e) \vee \mathbf{HS2}(e) \vee \mathbf{HS3}(e))}{\text{symbolic\_address}(e) = \text{False}}$$

Fig. 5: Heuristic rules for value filtering.

$(op, addr, v)$, vSIM first iterates through V_{mem} to create an initial set of known addresses, A (line 2). A is then expanded with addresses from external calls, such as the first argument of `realloc` or the return value of `malloc` (line 3). Next, to collect semantic values that change the program's state, vSIM iterates through both V_{reg} and V_{mem}, focusing on non-address values and "Store" operations (line 5). For each of these values, vSIM applies additional heuristic rules to assess semantic relevance, handling concrete values (lines 6) and symbolic values (lines 8) separately.

Heuristic Rules. Our heuristic rules for identifying addresses adapt established symbolization practices from disassembly [78]–[80]. While not perfectly sound or complete, these heuristics are effective because our approach generates fingerprints from sets of values, making it resilient to minor inaccuracies. Values identified as addresses by these rules are considered semantically irrelevant and are filtered out. Figure 5 provides a formal definition of these rules.

The rules rely on the A constructed in Algorithm 1, the ranges of data sections D (e.g., ".rodata", ".data", ".bss"), and the ranges of executable sections E (e.g., ".text", ".plt", and ".init"). bp is the initial concrete value of the base pointer, which is commonly used in under-constrained symbolic execution [26]. ϵ is a configurable stack size threshold (e.g., 1GB for 64-bit binaries and 128 MB for 32-bit binaries).

The **H**euristic rules for **C**oncrete values (**HC**) rely on the value itself and its position in the program's memory layout. Specifically, if a value falls within the range of a data section (**HC1**) or an executable section (**HC2**), it is likely used as an address. Similarly, a value near the stack pointer typically references to a local variable (**HC3**). A value satisfying any of these conditions is considered an address and is filtered out.

The **H**euristic rules for **S**ymbolic values (**HS**) apply similar logic, basing on the expression that form the value. An expression has a tree structure whose nodes are operators and leaves are concrete values or symbolic variables (e.g., left-most tree in Figure 6). If a concrete leaf node of an expression tree is an address, the entire expression is likely an address (**HS1**). Similarly, if the subtree of an expression is an identified address, or the expression itself is a subtree of a pointer arithmetic expression, it is likely an address (**HS2** and **HS3**).

Example. As illustrated in Figure 1(b), simulating the instructions yields several store operations. For instance, consider the instruction `mov 0xdeadbeef, [0x404058]`. The value `0xdeadbeef` is neither in the data or executable sections nor near the stack pointer, so it is identified as a semantic value. Similarly, the values `-1` and `3 * x` remain for similarity analysis. In contrast, the value `ptr` is associated with the pointer `ptr + 8`. Since 8 is not an address (**HC1** and **HC2**) and `ptr + 8` is an address being used by a memory read operation, `ptr` must be an address; and it is therefore not considered a semantic value according to **HS3**.

After the value filtering, we have a set of values collected from registers, memories, and the branch conditions of basic blocks. All basic block values of a binary function are aggregated for similarity comparison. However, they cannot be directly compared since values with the same semantics can have different structures, as **C3** elaborates in §II-B. To address this issue, vSIM generates fingerprints for efficient similarity comparison via normalization and concretization.

C. Value Normalization

Before generating fingerprints, we normalize the collected semantic values to ensure they are comparable, focusing on essential computational semantics while ignoring irrelevant details. We normalize concrete values, symbolic values, and branch conditions separately.

Normalizing Concrete Values. A concrete value is a bit vector $\texttt{BV}(value, size)$, where $value$ is an unsigned integer and $size$ is the bit width. For instance, a 32-bit integer -1 is `BV(0xffffffff, 32)`. To normalize the concrete values, vSIM discards the $size$ field and treats the $value$ as signed by default, unless it is used with floating-point operations. Hence, the size of a variable, which can differ across architectures (e.g., 8 bytes on AMD64 vs. 4 bytes on x86), becomes irrelevant. As a result, `0xdeadbeef`, `0xffffffff`, and `0` are normalized to -559038737, -1, and 0, respectively, in Figure 1(b). Similar normalization is applied to Figure 1(c), resulting in the same normalized values.

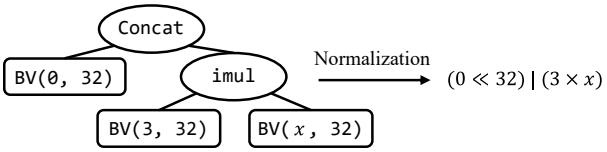


Fig. 6: An example of expression tree normalization. $\mathrm{BV}(a, 32)$ denotes a 32-bit vector with unsigned value a. `Concat` concatenates two bit vectors to form a larger one with the sum of their bit widths. `imul` multiplies two bit vectors.

Normalizing Symbolic Values. A symbolic value is a bit-vector of known width together with an expression tree whose internal nodes are operators (e.g., `imul` in Figure 6) and whose leaves are concrete constants or symbolic variables. For example, simulating the instructions highlighted in purple in Figure 1(b) yields, for register `rax`, the expression tree shown on the left of Figure 6. vSIM then normalizes expressions by dropping bit-width annotations, interpreting all values as signed, and rewriting `Concat` by OR-combining the high and low halves. Because the upper 32 bits are zero, `Concat(0, e)` simplifies to `e`; thus the normalized expression is `3 * x`. In our implementation, bit-vectors are represented using Python's built-in `int`. Other operators (including `imul`) are normalized using analogous, operator-specific rules. Meanwhile, the corresponding code in Figure 1(c) produces `x' + 2 * x'`, so although both snippets originate from the same source, the collected symbolic values remain different.

Normalizing Branch Conditions. Branch conditions are `bool`-typed symbolic values (i.e., a single bit stored in flag registers). They are always generated in pairs, and one is the negation of the other (e.g., `y` $\geq$ `8` and `y < 8`). Therefore, we only consider one condition from each pair. For instance, from the instructions highlighted in green in Figure 1(b), `y` $\geq$ `8` and `y < 8` are the pair of branch conditions. We keep the former and discard the latter. Similarly, the branch conditions of Figure 1(c) are normalized to `y'` $\leq$ `7` and `y' > 7`, and we only consider the latter. Although `y` $\geq$ `8` and `y' > 7` differ syntactically, they have the same effect when `y` and `y'` are integers. The concretization step will mitigate this issue.

D. Value Concretization

To enable efficient similarity comparisons, vSIM concretizes symbolic expressions so their semantic features can be compared more directly. vSIM collects three categories of values, i.e., concrete values, symbolic values, and branch conditions, and only needs to concretize the latter two since concrete values are already in a comparable form.

Concretizing Symbolic Values. A straightforward way to concretize a symbolic value is to substitute its symbolic variables with concrete test inputs and then evaluate the normalized expression. For example, given `3 * x` in Figure 1(b), vSIM replaces `x` with elements of a randomly generated array $A = [57, 44, 13, 81, 52]$, yielding $\{171, 132, 39, 243, 156\}$; sorting gives the tuple $(39, 132, 156, 171, 243)$, which we record as an element of `f2`'s fingerprint. The expression `x' + 2 * x'` in Figure 1(c) produces the same tuple when `x'` is instantiated with the same A. Matching tuples suggest (but do not prove) semantic equivalence since this check is a necessary but not sufficient condition. This concretization-based matching is far more efficient than invoking a theorem prover; due to limited space, we provide formal analysis and in Appendix C. We also evaluate robustness to the choice of the random array A and observe negligible impact on vSIM's performance; details appear in Appendix B.

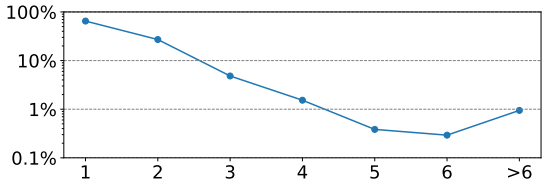


Fig. 7: Expressions distribution according to α.

Avoiding Permutations of Many Variables. If an expression has α symbolic variables, evaluating it with permutations of these variables (i.e., $\alpha!$ times) ensures that two equivalent expressions yield the same sets. This quickly becomes infeasible for large α. To ensure the practicality of vSIM, we set a threshold β, i.e., vSIM discards an expression when its $\alpha > \beta$. Based on empirical evidence, we set $\beta = 6$ because expressions with more than six variables account for less than 1% of all cases, making the impact of discarding them is minor.

Our empirical study with our evaluation datasets investigates how expressions distribute according to the numbers of variables (α). Figure 7 presents the distribution, where the x-axis is α, and the y-axis is the ratio of expressions in log scale. The ratio of expressions with more than 6 variables is less than 1%. Considering that concretizing such an expression requires computing it $7! = 5040$ times, setting the threshold $\beta = 6$ is a cost-effective decision. Moreover, our quantitative analysis of various β in Appendix D demonstrates that setting $\beta = 6$ effectively balances semantic preservation and efficiency.

Concretizing Branch Conditions. Since the branch conditions are also symbolic expression trees, vSIM can concretize them in the same way as the symbolic values. However, the branch conditions are boolean expressions, so the results are always 0 or 1, yielding trivial information for similarity analysis.

Our approach is splitting the branch conditions into two parts: the symbolic expression itself and the comparison operator with the concrete operand. For instance, the branch condition `y` $\geq$ `8` is split into `y` and $(\geq, 8)$. vSIM ignores the symbolic part `y` since it must be assigned to a register or memory before the comparison operation, and therefore, the symbolic part is already included in the symbolic values if it is computed in the function; otherwise, it is an unchanged parameter with trivial semantics. Thus, the remaining $(\geq, 8)$ contributes to the values of the branch conditions.

Because compilers can optimize a condition in multiple ways (e.g., `y` $\geq$ `8` and `y' > 7` in Figure 1(b) and 1(c)), vSIM extends each tuple to include equivalent forms, storing them

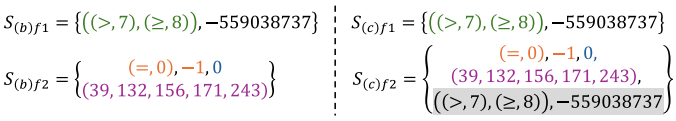


Fig. 8: Fingerprints of `f1` and `f2` in Figure 1(b) and (c).

in a set (e.g., $\{(\geq,8),(>,7)\}$). To collect the patterns of optimized branch conditions, we compile the binaries using different optimization levels with the debugging information. Then, we gather pairs of branch conditions that are compiled from the same source code but optimized differently. In this way, we collect a set of potential optimization patterns of branch conditions, which are used to extend the tuples of branch conditions. Specifically, we use the binaries of project `libmicrohttpd` built by BinKit (introduced in §IV-A) to collect the patterns, and those binaries are not used in our evaluation.

E. Fingerprint Propagation

Addressing function inlining is critical for BCSA [69], yet few techniques consider this challenge. Some approaches predict inlining relations [4], [8], [33], manually expanding callers before generating embeddings for similarity comparison. However, these predictions are often uninterpretable, error-prone, and sensitive to compiler variations [69].

Unlike previous works that rely on structural features such as CFGs, this study focuses on register and memory values for similarity analysis. Therefore, vSIM can adopt a straightforward yet effective method by merging the values of callee functions with those of the caller. For instance, from Figure 1(b) and (c), vSIM can collect `f1`'s and `f2`'s sets of values presented in Figure 8. Because `f1` is inlined in `f2` in Figure 1(c), the set of values $S_{(c)f2}$ includes the values of $S_{(c)f1}$ (i.e., the elements with gray background). Thus, propagating the callee's values of $S_{(b)f1}$ to $S_{(b)f2}$ results in $S_{(b)f1} \cup S_{(b)f2}$, which can help accurate similarity analysis.

Selection of Propagation Step. Although the union strategy is simple and effective, it may produce large fingerprints for binary functions, thereby slowing down similarity analysis. To mitigate this, we set a propagation threshold γ. A previous study [69] showed that an inlining depth of 3 covers over 90% of inlined functions. Consequently, we set the default γ to 3 and evaluate its impact on accuracy and time cost in §IV-B2.

F. Similarity Calculation

A binary function's fingerprint is a set of values. Thus, a straightforward approach to similarity analysis computes a similarity score, such as the Jaccard similarity, by comparing two sets. While this pairwise comparison is efficient and reasonably effective, we found that the human-readable elements within the fingerprints allow us to further enhance the performance.

Our fingerprints include concrete values, symbolic values, and branch conditions. Some values uniquely indicate the semantics of a binary function, whereas common values tend to be less informative. For example, compared to the value `0`, the expression `3 * x` is likely more distinguishable due to its informativeness. Thus, the corresponding element $(39,132,156,171,243)$ in the function fingerprint should gain a higher weight in the similarity comparison.

Function Pool Construction. The function pool stores the fingerprints of all binary functions in the dataset. Besides, we record the frequency of each value to assign weights to the values that compose the fingerprints. Values that appear more frequently are deemed less informative and are thus assigned lower weights. Let the occurrence of a value v be $Occ(v)$, then, its weight is calculated by Equation 1. We add 1 to the occurrence to avoid division by zero, and we apply the natural logarithm to scale the weight logarithmically with respect to its occurrence, which grows slowly. It smooths out the differences in occurrence frequency, prevents extremely common values from being overly penalized, and ensures that the weight decreases gradually as occurrence increases.

$$W(v) = \frac{1}{\ln(Occ(v)+1)} \tag{1}$$

Similarity Analysis. To compute the similarity between two binary functions, we use the Jaccard similarity with weights defined in Equation 2, where A and B are the fingerprints of two binary functions.

$$Jaccard_w(A,B) = \frac{\sum_{v\in A\cap B} W(v)}{\sum_{v\in A\cup B} W(v)} \tag{2}$$

IV. Evaluation

We have implemented vSIM atop `angr` [74] with over 4.4K lines of our own Python code. This section describes our evaluation results. We first present our experimental setup, including baselines, evaluation metrics, and datasets in §IV-A, then present our detailed results in §IV-B.

A. Experiment Setup

Baselines. To comprehensively compare vSIM with advanced BCSA tools with various technical pipelines, we select `jTrans` [19], `Clap` [68], GMN [6], and PEM [23] as our evaluation baselines, due to their availability and strong performance in recent evaluations. Among these, `jTrans`, `Clap`, and GMN are ML-based solutions. The former two employs embedding distance, and GMN uses a comparison model for similarity measurement, whereas PEM represents a recent non-ML-based approach. These methods have been evaluated across multiple datasets and have consistently outperformed various other BCSA tools [4], [6], [7], [10], [13], [15], [22], [40], [62], [81]. The following paragraphs briefly introduce the selected baselines.

- **`jTrans`** employs a customized transformer to learn semantic embeddings of binary functions. Notably, it replaces the BERT [82] natural language position encoding with a design tailored to capture jump targets based on intra-procedure CFG structures. Evaluation demonstrates that `jTrans` outperforms other machine learning-based BCSA tools [4], [10], [13], [40], [81].

- **Clap** employs a cross-modality training paradigm to align embeddings between binary code and natural language descriptions. The authors of Clap designed a dataset engine to generate a large-scale dataset of binary functions and their corresponding natural language descriptions, with 195 million pairs of aligned data, for training the model. As reported in their paper, Clap significantly outperforms jTrans and other BCSA tools in terms of accuracy.
- **GMN** leverages graph matching networks [83] to learn similarities between binary functions. Extensive evaluations demonstrate that GMN's accuracy outperforms other BCSA tools [4], [13], [15], [62]. However, it is less scalable than embedding-based methods due to the overhead of using a comparison model for each pair of functions.
- **PEM** leverages QEMU to manipulate program execution and extract predicates and string literals from execution traces for BCSA. However, most existing BCSA solutions exclude string literals [6], [19], [24], [25], [62], as these typically pertain to logging or informational purposes rather than core program behavior. Therefore, we derive PEM-s from PEM by omitting string literals, allowing us to focus on semantic comparisons. Specifically, we observe that PEM generates an intermediate file to record string lengths before performing similarity comparison. We thus set these lengths to 0, then run the similarity comparison process with the modified intermediate files. This slight tweak ensures that PEM-s strictly follows the pipeline of original PEM and produces reproducible results.

Evaluation Metrics. For a direct and fair comparison, we use the commonly used mean reciprocal rank (MRR) and Recall@1 metrics [4], [6], [19], [66], which are common information retrieval metrics. Recall@1 measures the proportion of the true positive among the top-1 predictions, and MRR calculates the average reciprocal rank of the true positive in the top-k predictions. Assume that the testing function set is $\mathcal{Q} = \{f_1, f_2, \ldots, f_N\}$, and $rank(f_i)$ is the rank of the true positive function f in the function pool $\mathcal{P}$, the Recall@1 and MRR@k are defined as follows:

$$\text{Recall@1} = \frac{1}{N}\sum_{f\in\mathcal{Q}} 1(rank(f) = 1), \tag{3}$$

$$\text{MRR@}k = \frac{1}{N}\sum_{f\in\mathcal{Q}} \frac{1(rank(f) \le k)}{rank(f)}, \tag{4}$$

where $1(\cdot)$ is the indicator function; it returns 1 if the condition is true, otherwise 0. Particularly, when k is not given, k is $+\inf$ by default, and $1(rank(f) \le k)$ is always 1. The higher values of Recall@1 and MRR@k denote better accuracy.

Datasets. To fairly evaluate vSIM against selected baselines, we reuse the datasets evaluated by previous BCSA solutions. PEM dataset is used to evaluate vSIM's performance in cross-optimization, BinKit for cross-compiler scenarios, GMN Dataset-2 for cross-architecture scenarios, while the last one is used to assess vSIM's applicability in real-world security tasks. Their statistics are summarized in Table III. Overall, our evaluation includes over five thousand binaries compiled by two toolchains (GCC and Clang), over six compiler versions, and across five architectures (x86, AMD64, ARM32, ARM64, and MIPS32). The following provides more details.

TABLE III: The statistics of used datasets.

Dataset[1]	# Binaries	# Functions[2]
PEM [23]	18	40,271
BinKit [61]	1,872	437,049
GMN Dataset-2 [83]	4,077	174,355
GMN Dataset-Vulnerability [83]	6	9849

[1] Binaries are available at https://doi.org/10.5281/zenodo.17751555.
[2] The number of collected binary functions for evaluation. A portion of binary functions of the binaries in GMN datasets are used since we reuse the sampled binary function pairs of GMN for fair comparision. The total number of binary functions of GMN datasets is far greater than our listed values.

- **PEM Dataset.** Released by Xu et al. [23], it contains six projects: Curl, GMP, ImageMagick, Sqlite3, Zlib, and OpenSSL. The binaries are compiled with GCC (ver. 9.4) using -O0, -O2, and -O3, targeting on the cross-optimization scenario. The whole dataset includes over 20,000 different source functions.
- **BinKit Dataset.** Released by Kim et al. [61], this dataset comprises 213,400 binaries from 51 projects, compiled with multiple versions of GCC and Clang. We use it to evaluate vSIM in cross-compiler scenarios. A noted limitation of this dataset, also highlighted by jTrans [19], is the lack of diversity; for instance, GNU Coreutils contributes 89,856 binaries (42.1%). Consequently, unlike previous works [61], [62] that deduplicate using (binary, symbol) pairs, we deduplicate using (project, symbol) pairs, except for the "main" functions (different binaries within a project have distinct "main" functions). Moreover, to mitigate bias caused by repeated binary functions compiled from closely related compilers (e.g., GCC-10.3 vs. GCC-11.2), we select binaries compiled with the oldest and latest versions of GCC and Clang (i.e., GCC-4.9.4, GCC-11.2, Clang-4, and Clang-13). Thus, we leverage 1,872 binaries from BinKit dataset in our cross-compiler scenario experiment.
- **GMN Dataset-2.** Marcelli et al. [6] introduce three datasets: Dataset-1, Dataset-2, and Dataset-Vulnerability. We use Dataset-2 to evaluate cross-architecture scenarios. Specifically, we reuse the GMN-sampled binary function pairs, so each comparison setting (e.g., XA+XO) mixes optimization levels and architectures; this contrasts with datasets that compare binaries compiled under fixed settings (e.g., O0 vs. O3). Dataset-1 is excluded since its binaries are compiled without function inlining, a less common option in real-world scenarios.
- **GMN Dataset-Vulnerability.** We evaluate real-world vulnerable-function detection using the Dataset-Vulnerability corpus of Marcelli et al. [6]. The corpus comprises six OpenSSL binaries: two extracted from production firmware (NetGear R7000 and TP-Link Deco-M4) and four obtained by compiling OpenSSL for x86, AMD64, ARM, and MIPS.

TABLE IV: Cross-optimization evaluation with PEM dataset.

Tool	MRR[1]		Recall@1		Retrieval time (s)[2]
	O0,O3	O0,O2	O0,O3	O0,O2	
jTrans	0.467	0.511	0.374	0.420	**6.1**
Clap	0.603	0.647	**0.548**	0.596	**6.1**
PEM-s	0.605	0.701	0.534	0.629	1853
vSim	**0.621**	**0.709**	0.545	**0.642**	87
(No concrete)	0.489	0.584	0.405	0.495	53
(No symbolic)	0.580	0.666	0.506	0.598	58
(No constraint)	0.603	0.687	0.527	0.618	74
(No filtering)	0.527	0.614	0.453	0.548	329
(No propagation)	0.508	0.540	0.464	0.498	22
(No weights)	0.596	0.700	0.520	0.633	54

[1] k is +inf by default when it is not given.
[2] The comparison processes of three tools are highly parallelized with 64 threads. The reported time is the average of 5 runs.

TABLE V: Cross-compiler evaluation with BinKit dataset.

Recall@1 of jTrans (Std Dev 0.107)				
O0 / O3	**GCC-11.2**	**GCC-4.9**	**Clang-13**	**Clang-4**
GCC-11.2	0.441	0.395	0.243	0.199
GCC-4.9	0.196	0.217	0.148	0.119
Clang-13	0.128	0.123	0.114	0.087
Clang-4	0.113	0.118	0.089	0.078
Recall@1 of Clap (Std Dev 0.014)				
GCC-11.2	0.596	0.582	0.583	0.579
GCC-4.9	0.575	0.580	0.573	0.548
Clang-13	0.562	0.561	0.601	0.598
Clang-4	0.567	0.579	0.588	0.586
Recall@1 of vSim (Std Dev 0.026)				
GCC-11.2	0.711	0.689	0.667	0.678
GCC-4.9	0.637	0.701	0.649	0.611
Clang-13	0.638	0.662	0.666	0.623
Clang-4	0.672	0.656	0.660	0.664

The best results are highlighted in light blue and the worst results are highlighted in gray.

Across these binaries, covering eight CVEs. Following the setup of [6], we fingerprint each vulnerable function in the four compiled binaries and use these fingerprints as queries to locate their counterparts in the two firmware binaries.

B. Evaluation Results

We conduct extensive experiments to assess the accuracy and efficiency of SoTA BCSA approaches. Specifically, we compare vSim against jTrans, GMN, and PEM-s using the BinKit, GMN Dataset, and PEM datasets across cross-optimization, cross-compiler, and cross-architecture scenarios. To further analyze vSim's advantages over these SoTA approaches, we evaluate different variants of vSim under various configurations. Additionally, we assess vSim's applicability in real-world security tasks, specifically vulnerable function detection, using GMN Dataset-Vulnerability. We also measure the overhead of vSim in terms of feature extraction and similarity comparison to demonstrate its scalability. Our experiments are conducted on a server equipped with two Intel Xeon Silver 4314 CPUs @ 2.40GHz (32 cores) and 512GB of memory. Our evaluation is centered around the following research questions (RQs):

- **RQ1:** *How accurate is* vSim *compared with baselines?*
- **RQ2:** *How does each design improve* vSim*'s accuracy?*
- **RQ3:** *How effective is* vSim *in security applications?*
- **RQ4:** *What is the overhead of* vSim*?*

*1) **RQ1: Accuracy of** **vSim**:* To evaluate the accuracy of vSim, we compare it against baselines under challenging scenarios. As described in §II-A, BCSA retrieves the top-k most similar functions for a query function f_q from a function pool $\mathcal{P}$. For each f_q, one binary function $p_i \in \mathcal{P}$ is compiled from the same source code, while all other functions in $\mathcal{P}$ are compiled from different sources. Since binaries compiled from the same source code share identical functionality, they should exhibit the highest similarity scores. As the size of $\mathcal{P}$ (i.e., Poolsize) increases, the retrieval task becomes more challenging, resulting in lower MRR and Recall@1 scores for BCSA tools [19]. In our evaluation, we adopt the largest Poolsize used in jTrans and Clap experiments: 10,000 for the PEM and BinKit datasets and 101 for GMN Dataset.

Cross-Optimization. This evaluation measures the ability of BCSA tools to retrieve similar functions across different optimization strategies, using the dataset released by PEM [23]. Table IV presents the results of jTrans, Clap, PEM, and vSim with a Poolsize of 10,000. Query binaries are compiled with -O0 while the pools consist of binaries compiled with different optimization levels (-O3 or -O2).

Our evaluation shows that vSim achieves the best MRR in different comparison scenarios. Although Clap has the highest Recall@1 in the comparison between binaries compiled with -O0 and -O3, vSim gets the second highest Recall@1, which is merely 0.5% lower than that of Clap. In addition, vSim also achieves the highest Recall@1 on average across two settings. This evaluation demonstrates that vSim can accurately retrieve similar functions across different optimization levels, achieving even better performance than the SoTA BCSA tools.

Cross-Compiler. This evaluation examines the performance when query and pool binaries are compiled with different compilers. Since the released PEM dataset uses binaries compiled with GCC only, we employ the BinKit dataset to evaluate cross-compiler scenarios, using the Poolsize of 10,000. However, we skip PEM in this evaluation since the binaries of BinKit dataset include unsupported operations of PEM's customized QEMU engine. Table V shows the results for jTrans, Clap, and vSim on BinKit with a Poolsize of 10,000. We report results under the most challenging setting [19], [68], where query functions are compiled with -O0 and pool functions with -O3. In this scenario, vSim outperforms jTrans and Clap across all comparisons and exhibits robust performance. Specifically, the Recall@1 of jTrans drops from 0.441 to 0.078 (an 82.3% reduction) when comparing binaries compiled with GCC-11 and Clang-4, with a standard deviation of 0.107 across settings. In contrast, vSim's Recall@1 ranges from 0.611 to 0.711 (a 14.1% drop) with 0.026 standard deviation.

TABLE VI: Cross-architecture evaluation with GMN dataset. XO and XA denotes cross-optimization and cross-architecture, respectively. XA+XO indicates the function pairs are compiled with different architectures and optimizations.

Tool	MRR@10		Recall@1		Retrieval time (s)[2]
	XO	XA+XO	XO	XA+XO	
GMN[1]	0.75	0.71	0.66	0.61	1005
vSIM	0.86	0.70	0.79	0.63	15

[1] Results of GMN refer to GMN (CFG + BoW opc 200), the top-performing model [6] compared with many other BCSA solutions [4], [13], [62], [81], [83], [84].
[2] The retrieval process was unparallelized using one thread.

We attribute the significant performance drop in `jTrans` to the out-of-distribution (OOD) problem common in ML-based techniques [5], [52]. Since embedding models are trained on binaries from a limited set of compilers, binaries from unseen compilers become OOD samples, degrading the embedding qualities. In contrast, vSIM computes similarity directly from the semantic values of binary functions without relying on a fixed training dataset, making it more robust to the compiler evolution. This robustness, along with higher accuracy, suggests that value-based BCSA is a promising approach for long-term binary code analysis.

`Clap` also exhibits a stable performance across different compilers. We hypothesize that the cross-modality training paradigm employed in `Clap` contributes to its stability, as it learns to align embeddings between binary code and natural language descriptions, making it less sensitive to compiler variations. However, it is expensive to generate natural language descriptions for functions in scale. `Clap`'s authors design a dataset engine that extracts the descriptions from the source code, combining manual efforts and automated LLMs, while neither the dataset nor the engine are publicly available. Moreover, generating such descriptions for binaries without source code is even more challenging. Therefore, by avoiding this dependency entirely, vSIM emerges as a significantly more practical and economical solution.

Cross-Architecture. This evaluation assesses the capability of BCSA tools to retrieve similar functions across different architectures. Table VI shows the results for vSIM and GMN on GMN Dataset. Since `jTrans` and `Clap` support only AMD64 architecture and PEM's artifact merely provides AMD64 binaries, we do not compare them in this scenario. For each query function, GMN samples 100 negative and 1 positive function pair, making the evaluation equivalent to a Poolsize of 101. Although the Poolsize is relatively small due to GMN's limited scalability, this evaluation is still challenging as the positive functions are overwhelmed by 100 times of negative samples. Comparing the cross-optimization (XO) and cross-architecture plus cross-optimization (XA+XO) scenarios, vSIM achieves higher accuracy than GMN in the XO scenario and comparable performance in the XA+XO scenario, with much faster similarity comparison—vSIM's comparison time is merely 1.5% of GMN's cost.

While vSIM performs well in XA+XO scenarios, it does not outperform ML-based methods as significantly as it does in other settings. Our investigation reveals that the differences in literal and macro definitions across architectures pose an obstacle when using values directly. For example, the system call number for `getpid` is 39 on AMD64 and 172 on ARM64, leading to variations in collected values. To mitigate this issue, integrating decompiler knowledge to recover the higher-level abstractions of these values, such as recovering the system call symbols, can benefit. Nonetheless, vSIM is a prototype that explores the potential of directly using value for BCSA, and we leave exploring this mitigation to future work.

Answer to RQ1: vSIM outperforms SoTA BCSA solutions in cross-optimization and cross-compiler scenarios and achieves comparable accuracy in cross-architecture scenarios. These results demonstrate that using semantics-aware values enhances accuracy and robustness, particularly when analyzing binaries compiled with unseen compilers.

*2) **RQ2: Contribution of Design Choices**:* This section examines how different semantic components contribute to the performance of vSIM. In particular, we assess the impact of disabling specific design choices on vSIM's accuracy. First, we evaluate the effects of removing concrete values, symbolic values, and constraints from the binary function fingerprints. Next, we analyze the roles of value filtering, fingerprint propagation, and value weighting in the similarity analysis.

Impact of Different Kinds of Values. As shown in Table IV, we generate variants of vSIM by removing individual components from the fingerprints, namely vSIM (No concrete), vSIM (No symbolic), and vSIM (No constraint). The results indicate that concrete values contribute the most to accuracy: disabling them leads to significant drops in MRR and Recall@1 (19.2% and 24.2%, respectively). In addition, both symbolic values and constraints play important roles, improving MRR and Recall@1 across different settings consistently. Overall, every component in the binary function fingerprints substantially enhances vSIM's accuracy, demonstrating the effectiveness of comprehensive value extraction.

Impact of the Value Filtering. We next assess the effect of the value filtering process in vSIM. When this process is disabled (denoted as vSIM (No filtering) in Table IV), the average Recall@1 drops by 15.7% compared to the full version. These significant decreases emphasize the importance of filtering out semantically irrelevant values in achieving accurate value-based BCSA. Moreover, without filtering out semantically irrelevant values, the comparison time increases by 2.8 times, indicating that using semantics-aware values not only significantly improves accuracy but also enhances efficiency.

Impact of the Fingerprint Propagation. As described in §III-E, vSIM employs a threshold γ to determine the step of callee fingerprints to be propagated. We vary this γ from 0 to 9 and beyond 9 (i.e., we set it to be 100) to evaluate its impact. Specifically, $\gamma = 0$ indicates that the propagation is disabled, while $\gamma > 9$ means all callee fingerprints are propagated.

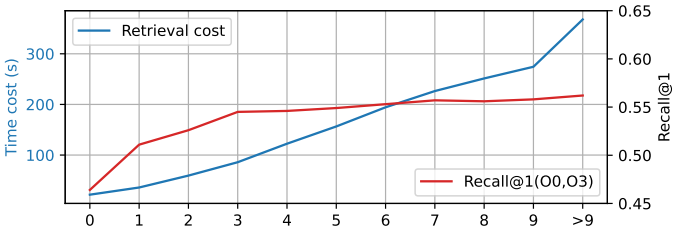
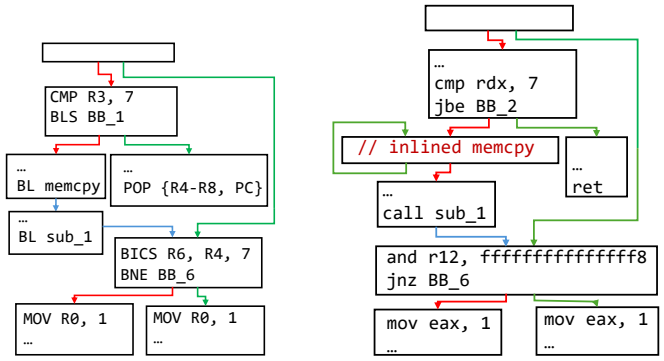


Fig. 9: Time cost and Recall@1 according to γ.

TABLE VII: MRR@10 of GMN Dataset-Vulnerability.

Tool	Netgear R7000				
	x86	AMD64	ARM32	MIPS32	Average
GMN[1]	**0.88**	0.54	**1.00**	0.79	0.80
vSim	**0.88**	**0.81**	0.88	**0.88**	**0.86**
	TP-Link Deco-M4				
GMN[1]	0.67	**0.73**	0.70	0.78	0.72
vSim	**0.8**	0.69	**0.79**	**0.81**	**0.77**

[1] Results of GMN refer to GMN (CFG + BoW opc 200) [6].



(a) Netgear R7000 (ARM32) (b) AMD64

Fig. 10: Simplified CFG of `MDC2_Update`.

Extraction:	`R6←R4 & ∼(0x7)`	`r12 ← r12 & 0xfffffffffffffff8`
Normalization:	`R4 & ∼7`	`r12 & (-8)`
Concretization:	(8, 40, 48, 56, 80)	(8, 40, 48, 56, 80)

Fig. 11: Normalization and concretization example.

Figure 9 illustrates that both the retrieval time and Recall@1 increase with the γ on the PEM dataset. When the propagation is disabled, the time cost is minimal, but the Recall@1 is also the lowest. Compared with the default $\gamma = 3$, the Recall@1 reduces by nearly 15% when the propagation is disabled.

While the time cost grows linearly with γ, Recall@1 saturates after $\gamma > 3$. For $\gamma > 9$, the retrieval time triples, yet Recall@1 (O0, O3) improves by only 3%, with negligible gains in average Recall@1. This experiment indicates that propagating callee fingerprints to callers can enhance accuracy significantly, with a propagation step of 3 offering a good trade-off between performance and efficiency.

Impact of the Weights. Weights in vSim represent the distinguishability of different values in BCSA. Since not all values are equally significant—some appear frequently across binary functions while others uniquely indicate specific functionality—disabling weights results in about 2pt decrease in Recall@1 on average. This experiment confirms that weights contribute to the accuracy of vSim by emphasizing the importance of specific values in the similarity analysis.

Answer to RQ2: Both value filtering and fingerprint propagation are critical process in vSim's accuracy. Among the fingerprint components, concrete values have the greatest impact, while symbolic values, constraints contribute similarly. Our design choices aggregatedly improve vSim's accuracy.

*3) **RQ3: Vulnerability Detection with** vSim:* This experiment demonstrates vSim's effectiveness in real-world security applications. Given a known vulnerable function, the goal is to retrieve its vulnerable counterparts from a pool of binary functions. We evaluate vSim on the GMN Dataset-Vulnerability, which covers eight CVEs extracted from real-world firmware images and binaries compiled for four architectures. As shown in Table VII, compared with GMN (CFG + BoW opc 200), the top-performing model reported in [6], vSim achieves a higher MRR@10 in five out of eight evaluation settings and matches GMN in one setting, achieving a higher average MRR@10. We also provide the detailed ranks of vulnerable functions in Table XI of Appendix E.

Case Study. Figure 10 presents the simplified CFGs of the function `MDC2_Update` for both AMD64 and ARM32 architectures. Notably, the ARM32 binary (from Netgear R7000 firmware) and its AMD64 counterpart exhibit similar CFG structures, with the key difference being that the ARM32 version calls `memcpy` in one branch, whereas the AMD64 version inlines the `memcpy`, introducing a back edge. This slight difference causes GMN to falsely rank these functions (e.g., ranking Figure 10(a) instance at 30 of Figure 10(b) rather than top-1).

In contrast, vSim successfully identifies the similarity between these functions and ranks the ARM32 version at top-1. For example, despite differences in assembly syntax—`CMP R3, 7; BLS BB_1` in ARM32 versus `cmp rdx, 7; jbe BB_2` in AMD64—vSim extracts equivalent constraint values $((>, 7), (\geq 8))$. Similarly, it identifies that `BICS R6, R4, 7` is semantically equivalent to `and r12, 0xfffffffffffffff8` (see Figure 11), even though the expressions and register sizes are distinct. These stable and interpretable values enable vSim to perform robust and accurate vulnerable binary function detection.

Answer to RQ3: vSim outperforms GMN in the detection of vulnerable functions on GMN's Dataset-Vulnerability. The case study illustrates the framework's ability to extract and leverage interpretable, stable values from binary code for real-world security applications.

*4) **RQ4: Overhead of** vSim:* The majority of the execution time is spent on value extraction and fingerprint generation, which are one-time efforts. Our evaluation indicates that on average, vSim requires approximately 0.4 seconds to extract

values from a binary function per core. Fingerprint generation takes roughly 1.66 seconds, yielding a total average time of about 2.07 seconds per binary function per CPU core for these one-time processes. In comparison, BinUSE [25], a recent value-based work extracting symbolic expressions from a subgraph of a binary function's CFG, requires approximately 5.88 seconds per function for an incomplete analysis. Contrarily, vSIM extracts values from all basic blocks comprehensively.

Regarding the retrieval process, vSIM requires around 87 seconds to obtain the top-k predictions, which involves 100 million function pair comparisons within a pool of 10,000 functions. With our server's 32 CPU cores, the average time per function pair comparison is about 2.7×10^{-6} seconds per core. Compared with BinUSE [25], which takes approximately 19.13 seconds per pair using the well-developed SMT solver Z3 [85], vSIM is several orders of magnitude faster. As shown in Table IV and Table VI, vSIM achieves significantly faster retrieval compared to existing BCSA solutions, such as GMN (which uses a comparison model) and PEM (which employs a customized algorithm). Although some ML-based tools, like `jTrans` and `Clap`, can offer even faster retrieval by leveraging embedding representations, they are expensive to train, less accurate than vSIM in various scenarios, and may encounter the out-of-distribution (OOD) problem.

Answer to RQ4: The primary cost lies in the one-time processes of value extraction and fingerprint generation. However, vSIM significantly outperforms many existing BCSA tools in the retrieval process, demonstrating its scalability for analyzing large-scale binary codebases.

V. Discussions and Future Work

Neglecting String Literals. String literals can be strong signatures for code reuse, prior work (e.g., PEM) reports high accuracy when leveraging them. We intentionally exclude string literals from vSIM for two reasons. First, they mostly encode logging or informational messages rather than behavior; modifying them seldom affects functionality. Accordingly, many BCSA solutions omit string literals [6], [13], [19], [24], [25], [62]; Ye et al. [39] further offer an instruction alignment solution as semantic anchor points, replacing the string-based anchors of DeepBinDiff. Second, binary software composition analysis (BSCA), a downstream application of binary code similarity analysis (BCSA) for clone detection, often explicitly exploits string literals to boost performance [14], [16]–[18], [86]. We therefore defer string-literal support in vSIM to such application-specific scenarios and instead focus on semantic values that directly capture program behavior.

Inlining Callees' Semantics. vSIM relies on a straightforward method to inline callees' semantics, while prior works (e.g., `Asm2vec`) often use algorithms or trained models to perform selective inlining. A straightforward inlining approach can easily results in overly-lengthy instructions, causing performance degradation of embedding models [4] and difficulty in collecting features [8]. However, Figure 9 demonstrates that vSIM is resilient to including the rich semantics of callees.

Missing High-Level Abstraction. Some semantic values are associated with specific usage patterns, such as the system call numbers, which can have the same semantic meaning but different values across different architectures and platforms, leading to different fingerprints. As described in §IV-B1, recovering high-level abstractions can potentially mitigate this issue.

Analyzing Obfuscated Binaries. The continuously evolving obfuscation techniques bring extra challenges to BCSA, such as unpacking binary code, decrypting data, resolving mixed boolean-algorithmic, and identifying virtual machine dispatchers and functional codes [87]–[93]. Thus, we leave it as challenging future work. However, analyzing normally compiled binaries remains crucial for many applications, such as license violation detection [86], vulnerability detection, and software composition analysis [14], [17].

VI. Conclusion

This paper explores the challenges and potential of using semantics-aware values for binary code similarity analysis (BCSA). Our study identifies three key challenges: unscalable value extraction, difficulty in identifying semantically representative values, and time-consuming comparison. By efficiently extracting, filtering, normalizing, concretizing, and propagating values, vSIM enables fast and precise analysis. Our evaluations on large-scale datasets and a real-world task demonstrate vSIM's superior performance, confirming the feasibility and effectiveness of value-based methods.

Acknowledgments

We would like to thank the anonymous reviewers including our shepherd for their constructive feedback. This research was supported in part by DARPA award N6600120C4020, and NSF awards 2112471 and 2326882. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the DARPA and NSF.

References

[1] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, 2008.

[2] J. Jang, M. Woo, and D. Brumley, "Towards automatic software lineage inference," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 81–96.

[3] J. Ming, D. Xu, Y. Jiang, and D. Wu, "Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking," in *Proceedings of the 26th USENIX Security Symposium*, 2017.

[4] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 ieee symposium on security and privacy*. IEEE, 2019, pp. 472–489.

[5] X. Li, Y. Qu, and H. Yin, "Palmtree: Learning an assembly language model for instruction embedding," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3236–3251.

[6] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, "How machine learning is solving the binary function similarity problem," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2099–2116.

[7] S. Wang and D. Wu, "In-memory fuzzing for binary code similarity analysis," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 319–330.

[8] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: Cross-architecture cross-os binary search," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 678–689.

[9] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 896–899.

[10] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 363–376.

[11] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *Network and Distributed Systems Security (NDSS) Symposium*, 2019.

[12] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection," *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1157–1177, 2017.

[13] L. Massarelli, G. A. D. Luna, F. Petroni, R. Baldoni, and L. Querzoni, "SAFE: Self-attentive function embeddings for binary similarity," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2019, pp. 309–329.

[14] L. Jiang, J. An, H. Huang, Q. Tang, S. Nie, S. Wu, and Y. Zhang, "BinaryAI: Binary software composition analysis via intelligent binary source code matching," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[15] Z. Yu, W. Zheng, J. Wang, Q. Tang, S. Nie, and S. Wu, "CodeCMR: Cross-modal retrieval for function-level binary source code matching," in *Advances in Neural Information Processing Systems*, vol. 33. Curran Associates, Inc., 2020, pp. 3872–3883.

[16] W. Tang, Y. Wang, H. Zhang, S. Han, P. Luo, and D. Zhang, "LibDB: An effective and efficient framework for detecting third-party libraries in binaries," *19th International Conference on Mining Software Repositories*, 2022.

[17] H. Wang, Z. Liu, S. Wang, Y. Wang, Q. Tang, S. Nie, and S. Wu, "Are we there yet? filling the gap between binary similarity analysis and binary software composition analysis," in *2024 IEEE 9th European Symposium on Security and Privacy*, 2024, pp. 506–523.

[18] H. Wang, Z. Liu, Y. Dai, S. Wang, Q. Tang, S. Nie, and S. Wu, "Preserving privacy in software composition analysis: A study of technical solutions and enhancements," in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 2329–2341.

[19] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, "jTrans: jump-aware transformer for binary code similarity detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA. Association for Computing Machinery, 2022, pp. 1–13.

[20] W. K. Wong, D. Wu, H. Wang, Z. Li, Z. Liu, S. Wang, Q. Tang, S. Nie, and S. Wu, "DecLLM: Llm-augmented recompilable decompilation for enabling programmatic use of decompiled code," *Proc. ACM Softw. Eng.*, vol. 2, no. ISSTA, Jun. 2025.

[21] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αdiff: Cross-version binary code similarity detection with DNN," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 667–678.

[22] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *Proceedings of the 23rd USENIX Security Symposium*. USENIX Association, Aug. 2014, pp. 303–317.

[23] X. Xu, Z. Xuan, S. Feng, S. Cheng, Y. Ye, Q. Shi, G. Tao, L. Yu, Z. Zhang, and X. Zhang, "PEM: Representing binary program semantics for similarity analysis via a probabilistic execution model," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 401–412.

[24] H. Wang, P. Ma, S. Wang, Q. Tang, S. Nie, and S. Wu, "sem2vec: Semantics-aware assembly tracelet embedding," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, May 2023.

[25] H. Wang, P. Ma, Y. Yuan, Z. Liu, S. Wang, Q. Tang, S. Nie, and S. Wu, "Enhancing DNN-based binary code function search with low-cost equivalence checking," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 226–250, 2022.

[26] D. A. Ramos and D. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, 2015, pp. 49–64.

[27] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2021.

[28] W. Chen, H. Wang, W. Gu, and S. Wang, "Rltrace: Synthesizing high-quality system call traces for os fuzz testing," in *International Conference on Information Security*. Springer, 2023, pp. 99–118.

[29] K. Lee, H. Lee, K. Lee, and J. Shin, "Training confidence-calibrated classifiers for detecting out-of-distribution samples," *arXiv preprint arXiv:1711.09325*, 2017.

[30] Y.-C. Hsu, Y. Shen, H. Jin, and Z. Kira, "Generalized ODIN: Detecting out-of-distribution image without learning from out-of-distribution data," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[31] X. Zhan, L. Fan, S. Chen, F. Wu, T. Liu, X. Luo, and Y. Liu, "ATVHunter: Reliable version detection of third-party libraries for vulnerability identification in android applications," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1695–1707.

[32] Z. Yuan, M. Feng, F. Li, G. Ban, Y. Xiao, S. Wang, Q. Tang, H. Su, C. Yu, J. Xu *et al.*, "B2SFinder: detecting open-source software reuse in cots software," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2019, pp. 1038–1049.

[33] A. Jia, M. Fan, X. Xu, W. Jin, H. Wang, and T. Liu, "Cross-inlining binary function similarity detection," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[34] Q. Feng, M. Wang, M. Zhang, R. Zhou, A. Henderson, and H. Yin, "Extracting conditional formulas for cross-platform bug search," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 346–359.

[35] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 627–642.

[36] H. Eom, D. Kim, S. Lim, H. Koo, and S. Hwang, "R2I: A relative readability metric for decompiled code," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024.

[37] E. Bauman, Z. Lin, and K. Hamlen, "Superset disassembly: Statically rewriting x86 binaries without heuristics," in *Network and Distributed Systems Security (NDSS) Symposium*, 2018.

[38] Y. Duan, X. Li, J. Wang, and H. Yin, "DeepBinDiff: Learning program-wide code representations for binary diffing," in *Network and Distributed Systems Security (NDSS) Symposium*, 2020.

[39] C. Ye, A. Zhou, and C. Zhang, "Enhancing semantic-aware binary diffing with high-confidence dynamic instruction alignment," in *Network and Distributed Systems Security (NDSS) Symposium*, 2026.

[40] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 01, 2020, pp. 1145–1152.

[41] H. Wang, Z. Gao, C. Zhang, M. Sun, Y. Zhou, H. Qiu, and X. Xiao, "CEBin: A cost-effective framework for large-scale binary code similarity detection," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA. Association for Computing Machinery, 2024, p. 149–161.

[42] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Cross-architecture binary semantics understanding via similar code comparison," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 57–67.

[43] A. Zhou, Y. Hu, X. Xu, and C. Zhang, "ARCTURUS: Full coverage binary similarity analysis with reachability-guided emulation," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 4, Apr. 2024.

[44] Y. Hu, H. Wang, Y. Zhang, B. Li, and D. Gu, "A semantics-based hybrid approach on binary code similarity comparison," *IEEE Transactions on Software Engineering*, vol. 47, no. 6, pp. 1241–1258, 2021.

[45] Y. Hu, Y. He, W. He, H. Li, Y. Zhao, S. Wang, and D. Gu, "Binary cryptographic function identification via similarity analysis with path-insensitive emulation," *Proc. ACM Program. Lang.*, vol. 9, no. OOPSLA1, Apr. 2025.

[46] Z. Li, P. Ma, H. Wang, S. Wang, Q. Tang, S. Nie, and S. Wu, "Unleashing the power of compiler intermediate representation to enhance neural program embeddings," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2253–2265.

[47] W. K. Wong, H. Wang, P. Ma, S. Wang, M. Jiang, T. Y. Chen, Q. Tang, S. Nie, and S. Wu, "Deceiving deep neural networks-based binary code matching with adversarial programs," in *2022 IEEE International Conference on Software Maintenance and Evolution*, 2022, pp. 117–128.

[48] J. Zeng, Y. Fu, K. A. Miller, Z. Lin, X. Zhang, and D. Xu, "Obfuscation resilient binary code reuse through trace-oriented programming," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, 2013, pp. 487–498.

[49] T. Benoit, J.-Y. Marion, and S. Bardin, "Scalable program clone search through spectral analysis," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE, 2023, pp. 808–820.

[50] H. Flake, "Structural comparison of executable objects," in *Detection of intrusions and malware & vulnerability assessment, GI SIG SIDAR workshop (DIMVA)*. Bonn: Gesellschaft für Informatik e.V., 2004, pp. 161–173.

[51] I. U. Haq and J. Caballero, "A survey of binary code similarity," *ACM Comput. Surv.*, vol. 54, no. 3, Apr. 2021. [Online]. Available: https://doi.org/10.1145/3446371

[52] W. K. Wong, H. Wang, Z. Li, and S. Wang, "BinAug: Enhancing binary similarity analysis with low-cost input repairing," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024.

[53] X. Jin, K. Pei, J. Y. Won, and Z. Lin, "SymLM: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1631–1645.

[54] L. Jiang, X. Jin, and Z. Lin, "Beyond classification: Inferring function names in stripped binaries via domain adapted llms," in *Network and Distributed System Security (NDSS) Symposium*, 2025.

[55] W. K. Wong, H. Wang, Z. Li, Z. Liu, S. Wang, Q. Tang, S. Nie, and S. Wu, "Refining decompiled c code with large language models," *arXiv preprint arXiv:2310.06530*, 2023.

[56] Y. Du, C. Wang, and H. Wang, "Programming language techniques for bridging llm code generation semantic gaps," in *Proceedings of the 1st ACM SIGPLAN International Workshop on Language Models and Programming Languages*, 2025, pp. 40–45.

[57] Z. Liu, Y. Yuan, S. Wang, and Y. Bao, "Sok: Demystifying binary lifters through the lens of downstream applications," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 1100–1119.

[58] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient cross-architecture identification of bugs in binary code," in *Network and Distributed Systems Security (NDSS) Symposium*, 2016.

[59] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 480–491.

[60] J. Yang, C. Fu, X.-Y. Liu, H. Yin, and P. Zhou, "Codee: A tensor embedding scheme for binary code search," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2224–2244, 2021.

[61] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim, "Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1661–1682, 2023.

[62] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, "Learning approximate execution semantics from traces for binary function similarity," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2776–2790, 2023.

[63] B. Zhao, S. Ji, J. Xu, Y. Tian, Q. Wei, Q. Wang, C. Lyu, X. Zhang, C. Lin, J. Wu, and R. Beyah, "A large-scale empirical analysis of the vulnerabilities introduced by third-party components in iot firmware," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA, 2022, pp. 442–454.

[64] G. Kim, S. Hong, M. Franz, and D. Song, "Improving cross-platform binary analysis using representation learning via graph alignment," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA, 2022, pp. 151–163.

[65] Z. Luo, P. Wang, B. Wang, Y. Tang, W. Xie, X. Zhou, D. Liu, and K. Lu, "VulHawk: Cross-architecture vulnerability detection with entropy-based binary code search." in *Network and Distributed Systems Security (NDSS) Symposium*, 2023.

[66] H. He, X. Lin, Z. Weng, R. Zhao, S. Gan, L. Chen, Y. Ji, J. Wang, and Z. Xue, "Code is not natural language: Unlock the power of semantics-oriented graph representation for binary code similarity detection," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 1759–1776.

[67] J. Wang, C. Zhang, L. Chen, Y. Rong, Y. Wu, H. Wang, W. Tan, Q. Li, and Z. Li, "Improving ML-based binary function similarity detection by assessing and deprioritizing control flow graph features," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4265–4282.

[68] H. Wang, Z. Gao, C. Zhang, Z. Sha, M. Sun, Y. Zhou, W. Zhu, W. Sun, H. Qiu, and X. Xiao, "Clap: Learning transferable binary code representations with natural language supervision," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 503–515.

[69] A. Jia, M. Fan, W. Jin, X. Xu, Z. Zhou, Q. Tang, S. Nie, S. Wu, and T. Liu, "1-to-1 or 1-to-n? investigating the effect of function inlining on binary similarity analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–26, 2023.

[70] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[71] Y. David and E. Yahav, "Tracelet-based code search in executables," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. ACM, 2014, pp. 349–360.

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

[73] S. Poeplau and A. Francillon, "Systematic comparison of symbolic execution systems: intermediate representation and its generation," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 163–176.

[74] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna, "SOK: (state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 2016, pp. 138–157.

[75] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," *Acm Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[76] K. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin, "Probabilistic disassembly," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 1187–1198.

[77] A. Zhou, C. Ye, H. Huang, Y. Cai, and C. Zhang, "Plankton: Reconciling binary code and debug information," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 912–928.

[78] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," in *2021 IEEE symposium on security and privacy (SP)*. IEEE, 2021, pp. 833–851.

[79] C. Ye, Y. Cai, A. Zhou, H. Huang, H. Ling, and C. Zhang, "Manta: Hybrid-sensitive type inference toward type-assisted bug detection for stripped binaries," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '24, vol. 4, 2025, pp. 170–187.

[80] S. Wang, P. Wang, and D. Wu, "UROBOROS: Instrumenting stripped binaries with static reassembling," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 236–247.

[81] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, R. Baldoni *et al.*, "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis," in *Proceedings of the 2nd Workshop on Binary Analysis Research (BAR)*, 2019.

[82] J. Devlin, "BERT: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[83] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in *Proceedings of the 36th International conference on machine learning*. PMLR, 2019, pp. 3835–3845.

[84] N. Shalev and N. Partush, "Binary similarity detection using machine learning," in *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*, ser. PLAS '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 42–47.

[85] L. D. Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008, pp. 337–340.
[86] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, "Finding software license violations through binary code clone detection," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 63–72.
[87] H. Wang, S. Wang, D. Xu, X. Zhang, and X. Liu, "Generating effective software obfuscation sequences with reinforcement learning," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1900–1917, 2020.
[88] B. Liu, J. Shen, J. Ming, Q. Zheng, J. Li, and D. Xu, "MBA-Blast: Unveiling and simplifying mixed Boolean-Arithmetic obfuscation," in *30th USENIX Security Symposium (USENIX Security 21)*, Aug. 2021, pp. 1701–1718.
[89] M. Schloegel, T. Blazytko, M. Contag, C. Aschermann, J. Basler, T. Holz, and A. Abbasi, "Loki: Hardening code obfuscation against automated attacks," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, Aug. 2022, pp. 3055–3073.
[90] N. Zhang, D. Alden, D. Xu, S. Wang, T. Jaeger, and W. Ruml, "No free lunch: On the increased code reuse attack surface of obfuscated programs," in *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2023, pp. 313–326.
[91] R. Little and D. Xu, "Inspecting compiler optimizations on mixed boolean arithmetic obfuscation," in *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, 2023.
[92] N. Zhang, D. Xu, J. Ming, J. Xu, and Q. Yu, "Inspecting virtual machine diversification inside virtualization obfuscation," in *2025 IEEE Symposium on Security and Privacy (SP)*, 2025, pp. 3051–3069.
[93] Z. Feng and D. Xu, "Debra: A real-world benchmark for evaluating deobfuscation methods," in *Proceedings of the 2025 Workshop on Software Understanding and Reverse Engineering*, ser. SURE '25, 2025, pp. 76–88.

TABLE VIII: The impact of basic block sequences.

BB Seq.	Recall@1		Offline time(s)[1]		Retrieval
	O0,O3	O0,O2	Extraction	Fingerprint	time(s)
1	0.545	0.642	1,247	3,764	87
2	0.550	0.640	2,083	24,637	90
3	0.534	0.634	4,497	797,696[2]	94

[1] The offline time is measured on the binary with the largest number of functions in the PEM dataset, i.e., OpenSSL compiled with `-O0`.
[2] We use 32 threads to parallelize the offline fingerprint generation since it can take more than a week. The other two settings use 1 thread.

APPENDIX A
SIMULATING BASIC BLOCK SEQUENCE

In developing VSIM, we deliberately negelect structural features, such as control-flow graphs (CFGs) of binary functions, despite their potential to improve accuracy [10], [19], [40]. On the other hand, symbolically executing each basic block avoids collecting features from infeasible paths, which may compromise VSIM's performance, although dead code can still affect VSIM. In this section, we perform an ablation study by simulating 2- and 3-length sequences of basic blocks to quantitatively measure how different granularities influence VSIM's efficiency and accuracy.

Table VIII presents the effect of block-sequence length on VSIM's recall and runtime. Using 3-length sequences, offline fingerprint generation did not finish within one week, rendering this setting impractical. Moving from 1-length (the default VSIM) to 2-length sequences yields slight accuracy change, i.e., Recall@1 shifts by -0.011 (O0 vs. O3) and -0.008 (O0 vs. O2), but substantially increases cost: fingerprinting rises from around 1 hour to more than a week, and retrieval time grows from 87s to 94s. Although longer sequences may capture the structural knowledge and richer semantics [8], [71], their overhead undermines practicality for VSIM. Thus, developing a scalable approach that integrates both value-based and structural information is a promising yet challenging direction for future work.

TABLE IX: The impact of different random arrays.

Seed	MRR		Recall@1	
	O0,O3	O0,O2	O0,O3	O0,O2
42	0.621	0.709	0.545	0.642
3	0.621	0.707	0.546	0.640
5	0.622	0.708	0.546	0.640

APPENDIX B
DIFFERENT RANDOM ARRAYS

VSIM relies on a random array to get numbers for concretizing symbolic expressions. Obviously, different values can change the concretized feature of a symbolic expression; we thus evaluate the impact of different random arrays in this section. In short, the impact is nearly negligible. We clarify that all experiments in §IV use a fixed random array generated with random seed 42, this section further evaluates the accuracy using random seed 3 and 5. As presented in Table IX, the values of MRR and Recall@1 change by less than 0.002.

APPENDIX C
EFFICIENCY BENEFITS OF VALUE CONCRETIZATION

We formalize why our concretization strategy is fundamentally more scalable than invoking SMT solvers to compare sets of symbolic expressions. Consider two sets of single-variable expressions, $F = \{f_1(a), f_2(a), \ldots, f_n(a)\}$ and $G = \{g_1(a), g_2(a), \ldots, g_m(a)\}$.

- **SMT Baseline.** An SMT solver must be invoked for each pair (f_i, g_j) to decide equivalence. Thus, the online comparison of (F, G) requires $O(nm)$ pairwise checks (ignoring the solver's per-call cost).
- **VSIM's Approach.** VSIM separates the work into one-time offline preprocessing and a lightweight online phase, yielding $O(n + m)$ online time:
  1) *Offline concretization.* VSIM independently concretizes (normalizes) every expression in F and G, in total $O(n + m)$ time.
  2) *Offline ordering.* The concretized elements are stored in ordered sets, which takes $O(n \log n + m \log m)$ time.
  3) *Online comparison.* Comparing (F, G) reduces to intersecting two ordered sets, which runs in $O(n+m)$ time.

For expressions over multiple symbolic variables, SMT-based comparison typically must generate extra constraints for variable correspondences, which can require considering all permutations in the worst case, i.e., a factorial blow-up [12], [25]. For example, comparing $f(a, b)$ and $g(x, y)$ requires checking the equivalence of $f(a, b)$ with both $g(x, y)$ (mapping $a \to x, b \to y$) and $g(y, x)$ (mapping $a \to y, b \to x$). When

both $2! = 2$ mappings fail, $f(a, b)$ and $g(x, y)$ are deemed non-equivalent. In contrast, vSIM absorbs this permutation into the offline concretization step, so the online comparison still runs in $O(n + m)$ time.

TABLE X: vSIM's performance with different β values.

Threshold (β)	MRR[1]		Recall@1		Retrieval time (s)[2]
	O0,O3	O0,O2	O0,O3	O0,O2	
7	0.621	0.708	0.546	0.641	88
6 (vSIM)	0.621	0.709	0.545	0.642	87
5	0.622	0.708	0.547	0.641	86
4	0.620	0.708	0.545	0.640	86
3	0.623	0.709	0.547	0.641	86
2	0.619	0.708	0.543	0.641	82
1	0.607	0.697	0.531	0.630	77
0 (No symbolic)	0.580	0.666	0.506	0.598	58

[1] k is `+inf` by default when it is not given.
[2] The comparison processes of three tools are highly parallelized with 64 threads. The reported time is the average of 5 runs.

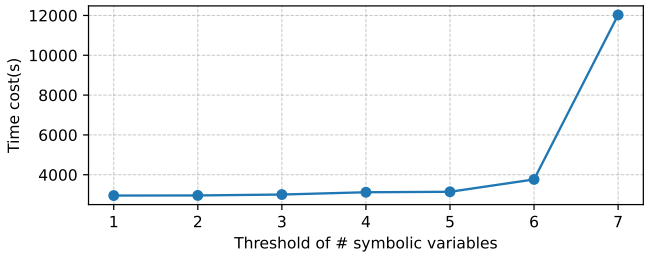


Fig. 12: Fingerprint generation overhead according to β.

APPENDIX D
THE IMPACT OF DIFFERENT β

When concretizing symbolic values (§III-D), vSIM encounters efficiency issues for expressions with more than six variables, so it ignores them. This section presents an ablation study on how different thresholds β affect vSIM's performance. In particular, the "(No symbolic)" row of Table IV corresponds to $\beta = 0$, i.e., no symbolic expressions are considered. Table X reports results when $\beta \in [0, 7]$. When the threshold is below 3, vSIM's MRR and Recall@1 increase steadily, as does retrieval time. However, MRR, Recall@1, and retrieval overhead are similar when $\beta > 3$, indicating comparable online performance.

In contrast, the offline fingerprint-generation overhead follows a different trend from accuracy. Because our datasets contain diverse binaries, we use an OpenSSL binary from the PEM dataset, which was built by GCC with `-O0`, as an illustrative example. Figure 12 shows how the generation time scales with β: from 1 to 6, it grows modestly from 49 minutes to about an hour. This overhead is acceptable given the concurrent performance gains. However, increasing the β from 6 to 7 causes a more than $3\times$ jump in generation time due to factorial growth in complexity, even though such expressions constitute a small fraction of the overall dataset. Thus, setting $\beta = 6$ balances time cost with the preserved semantics.

Although vSIM still achieves SoTA performance due to the rarity of such complicated expressions, discarding them leads to the loss of semantics. A more sophisticated mechanism capable of handling these expressions could enhance vSIM's semantic preservation. For example, using a model that generates identical embeddings for semantically equivalent expressions would allow vSIM to concretize them without losing semantics.

APPENDIX E
RANKS OF VULNERABLE FUNCTIONS

TABLE XI: Vulnerability test ranks. The ranks with the best MRR@10 of Table VII for each architecture are in **bold**.

Tool	NetGear R7000 (ARM32)			
	x86	AMD64	ARM32	MIPS32
vSIM	**2;1;1;1**	**4;1;1;1**	2;1;1;1	**2;1;1;1**
GMN[1]	**1;1;1;2**	1;1;30;7	**1;1;1;1**	1;1;1;7
	TP-Link Deco M4 (MIPS32)			
	x86	AMD64	ARM32	MIPS32
vSIM	**2;1;1;1;1**	19;1;45;1;1	**2;1;1;1;3**	**24;1;1;1;1**
	2;1;5;1	1;1;4;1	**1;1;4;1**	**1;1;4;1**
GMN[1]	22;1;1;24;2	**9;1;1;79;2**	3;1;1;60;33	1;1;3;3;1
	1;2;1;1	**1;1;1;1**	1;1;1;1	1;3;1;1

[1] Results of GMN refer to GMN (CFG + BoW opc 200) [6].

APPENDIX F
ARTIFACT AVAILABILITY

The source code of vSIM, along with scripts, datasets used in our experiments, and detailed instructions for reproducing the results, are publicly available at https://doi.org/10.5281/zenodo.17751555. We will also actively maintain the vSIM repository at https://github.com/OSUSecLab/vSim to provide future updates and improvements.