CovBooster: Coverage Booster for Binary Code Clone Detection by Reduced Signatures

Jeongwoo Lee
Korea University
Seoul, Republic of Korea
jeongwoo@korea.ac.kr

Jeremy D. Seideman
City University of New York
New York, NY, USA
jseideman@gradcenter.cuny.edu

Geonwoo Lee
Korea University
Seoul, Republic of Korea
gnu@korea.ac.kr

Sven Dietrich*
City University of New York
New York, NY, USA
spock@ieee.org

Heejo Lee*
Korea University
Seoul, Republic of Korea
heejo@korea.ac.kr

Abstract

Binary-level code clone detection is a core technique for many applications in security and software engineering, such as malware analysis, software plagiarism detection, and vulnerability discovery. Existing approaches typically rely on function-level signatures, thus the number of distinct signatures grows explosively when the same function is compiled into different forms depending on compilation environments. Such an excessive number of signatures leads to a rapid increase of space and time complexity. In this paper, we propose CovBooster, a novel signature set construction method that significantly enhances binary clone detection by systematically constructing a function coverage graph and reformulating the signatures reduction task as a dominating set problem. This formulation enables CovBooster to identify a small yet representative subset of signatures that collectively cover all variants through structural similarity, eliminating the need for exhaustive enumeration across compilation environments. We evaluated CovBooster on a dataset of 1,440 real-world software binaries, compiled from ten popular packages across 144 distinct compilation environments using BinKit. Our results show that CovBooster covers all evaluated variants effectively using only 25.9% of the compiled variants, equivalent to 37.3 signatures per function on average. This suggests that a small, well-chosen signature set can achieve thorough detection coverage with minimal overhead.

CCS Concepts

• **Security and privacy → Software reverse engineering**.

Keywords

Binary Code Clone, Coverage, Multi-signature

*Corresponding authors.

SAC'26, Thessaloniki, Greece

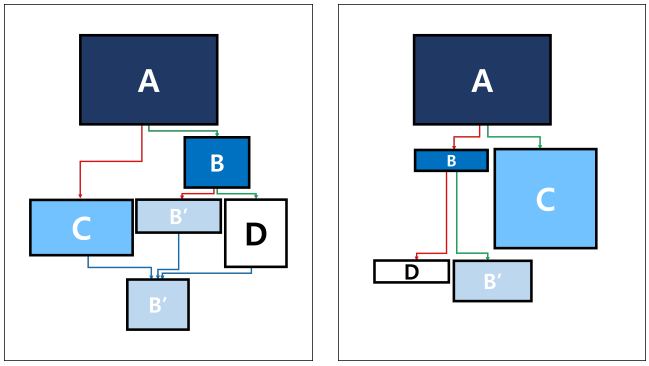
ACM ISBN 979-X-XXXX-XXXX-X/26/03
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Jeongwoo Lee, Jeremy D. Seideman, Geonwoo Lee, Sven Dietrich, and Heejo Lee. 2026. CovBooster: Coverage Booster for Binary Code Clone Detection by Reduced Signatures. In *Proceedings of The 41st ACM/SIGAPP Symposium on Applied Computing (SAC'26).* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The growth of open-source repositories such as GitHub [10] and Stack Overflow [37] has greatly accelerated software development by promoting code reuse. Such reused code, or *code clones*, can enhance development efficiency but also increase the risk of propagating vulnerabilities from the original source [40, 43]. Detecting these vulnerable code clones is therefore essential to prevent large-scale security incidents.

Traditional code clone detection methods typically construct databases of known vulnerable code segments, extracting signatures (such as hash digests or string-based fingerprints), and scanning target codebases [22]. Effective source-level techniques [16, 18, 22, 41] demonstrate robust detection even for non-identical code clones. These methods have been integrated into development workflows for continuous vulnerability tracking [23, 35].



(a) x64, GCC-8.2.0, -O0 **(b) x64, GCC-8.2.0, -O3**

Figure 1: Control Flow Graph (CFG) of `SSL_read` in `OpenSSL_1.0.1f` from different compilation environments. Despite being the same function in the source code, the output binary is structured differently.

However, binary-level code clone detection becomes necessary when source code is unavailable (e.g., proprietary or closed-source

software) or when vulnerabilities emerge post-compilation. Binary detection faces unique challenges due to structural variations caused by diverse compilation environments, such as different compilers, optimization levels, or architectures [11, 21, 32]. These variations complicate reliable identification of clones, as illustrated in Figure 1.

Machine learning (ML)-based methods address these variations using approaches such as natural language processing on assembly code [4], behavioral analysis through execution, or dynamic fuzzing [6, 7, 39]. Despite their accuracy, these ML-driven approaches often require substantial computational resources and specialized hardware, limiting scalability. Therefore, static binary analysis remains actively pursued due to its efficiency and low resource requirements.

Existing static binary code clone detectors typically rely on single signatures extracted from binaries compiled in one environment, limiting detection coverage across significantly different compilation environments. Employing multiple signatures (a *signature set*) to represent each piece of code can significantly enhance detection coverage by capturing structural variations.

In this work, we propose CovBooster, a binary code clone detection framework addressing the challenge of maximizing coverage under compilation diversity. Our approach transforms the overall detection task into a graph-theoretic formulation by constructing a function-variant similarity graph. In this graph, nodes represent compiled function variants, and edges denote successful matches by a base detector. CovBooster casts signature selection as a weighted dominating set problem, extracting minimal subsets of signatures covering all function variants. This approach ensures comprehensive cross-environment detection and scalability.

We implemented a prototype of CovBooster by integrating the detection engine of QuickBCC [15], which we reused as the underlying clone detector for evaluating our signature selection method. QuickBCC offers compact and informative abstractions (strands) suitable for efficient signature generation. Though QuickBCC serves as our evaluation engine due to its open-source availability and efficiency, CovBooster itself remains detector-agnostic, operating solely on extracted signatures and their similarities.

Our evaluation involves 1,440 binaries compiled from ten popular software packages across 144 configurations using BinKit [21]. CovBooster achieves comprehensive coverage across all variants per function with an average of only 37.3 signatures – just 25.9% of available variants. While QuickBCC typically covers only 1–2 variants per function, CovBooster achieves comprehensive variant coverage with compact signature sets, maintaining an average F1-score of 0.98.

The main contributions of this paper are threefold:

- **Graph-Based Signature Selection**: We reformulate the problem of selecting binary code clone detection signatures as a coverage maximization problem over a similarity graph, and solve it using a greedy dominating set heuristic.
- **Signature Set Reduction**: We show that a minimal set of binary signatures (average 37.3 per function) suffices to achieve full clone coverage across diverse compilation environments and binary variants.
- **Framework Applicability**: We demonstrate that our graph-based coverage selection strategy can be integrated into a wide range of clone detection systems, regardless of their underlying signature extraction methodologies, improving their robustness to syntactic and semantic variations.

2 Motivation and Problem Statement

2.1 Motivation

Code clones are similar or duplicate code fragments commonly resulting from library reuse, software forks, or shared templates [22, 40]. Detecting binary code clones is essential for vulnerability discovery, malware analysis, and supply chain integrity [11]. However, binary code clone detection (BCCD) is complicated by the variability introduced by different *compilation environments*. A **compilation environment** includes compiler type/version, target architecture, optimization flags, and OS settings [12, 13, 17]. Even small differences in these factors can yield binaries with divergent syntax and structure (Table 1).

BCCD tools typically rely on syntactic (e.g., hashes), structural (e.g., CFG), or semantic (e.g., embeddings, I/O behavior) features [2, 11, 15, 21]. While semantic approaches are more resilient to environmental changes, they incur high computational cost. An effective solution should balance robustness and efficiency, detecting clones across diverse environments with minimal overhead.

Table 1: Common compilation environment components (C/C++ binaries).

Factor	Examples
Architecture	x64, x86, aarch64, arm32, mips64, mips
Optimization Level	`-O0`, `-O1`, `-O2`, `-O3`, `-Os`
Compiler	GCC, Clang, MSVC

2.2 Problem Statement

We formally define the problem of binary code clone detection, building on prior work [22, 24, 31]. The goal is to identify semantically equivalent binary functions compiled under diverse environments, while minimizing detection errors such as false positives.

Binary Code Clone. Let F be a set of binary functions. Two functions $f, f' \in F$ are considered clones if they perform equivalent computations:

$$f \equiv f' \tag{1}$$

In practice, this means the functions originate from the same or similar source code and perform equivalent computations. While this definition is conceptually simple, deciding semantic equivalence between binaries is known to be undecidable in the general case [24], due to compiler transformations, obfuscation, or instruction reordering. Consequently, clone detectors must rely on heuristics to approximate this relation.

Binary Clone Detector and Signature. A binary clone detector D attempts to approximate the semantic equivalence relation through feature comparison. Formally, it maps a pair of binary functions to a binary label:

$$D : F \times F \rightarrow \{0, 1\} \tag{2}$$

Here, $D(f, f') = 1$ indicates that the detector considers f and f' to be clones. The decision is based on signatures – intermediate

representations that encode properties of the binary. These signatures may be hash-based, graph-based (e.g., CFG, call graphs), or semantic (e.g., symbolic execution traces, embeddings). The choice of signature significantly affects the detector's performance and robustness.

Coverage Graph. To analyze a detector's coverage, we define a coverage graph $G_{D,f} = (V, E)$ for a given target function f. Each vertex in V corresponds to a variant of f compiled under different environments. An edge between two vertices indicates that the detector successfully identifies them as clones:

$$(v_i, v_j) \in E \iff D(v_i, v_j) = 1 \tag{3}$$

In the ideal case, the coverage graph forms a clique, meaning all variants are correctly detected as clones. However, in practice, the graph is often sparse due to false negatives caused by compilation-induced differences. Moreover, the graph may contain incorrect edges connecting semantically unrelated functions, which constitute false positives.

Minimal Signature Set and Dominating Set. To improve detection robustness under compilation diversity, we construct a compact signature set of representative function variants. We model this as a coverage graph where nodes represent compiled variants and edges indicate successful matches, capturing pairwise detectability. The objective is to select a subset of nodes such that every node is either included or adjacent to one, ensuring each variant is covered by at least one signature. Formally, the minimal signature set $V^* \subseteq V$ satisfies:

$$\forall v \in V,\ v \in V^* \vee \exists u \in V^*, (u, v) \in E \tag{4}$$

This corresponds to the classical minimum dominating set problem [5], which balances representativeness and conciseness. Although it is NP-hard, we apply greedy heuristics to select signatures that offer high coverage with minimal redundancy.

False Positives in Cross-function Matching. Another challenge arises when the detector incorrectly matches a binary function f to an unrelated function g from another binary:

$$D(f, g) = 1 \quad \text{but} \quad f \not\equiv g \tag{5}$$

Such false positives degrade the reliability of clone detection systems. In security applications, they can lead to incorrect vulnerability attribution or misclassification of benign binaries as malicious. Minimizing false positives, especially in the presence of environment-induced variability, remains a critical goal in binary similarity analysis.

3 Design of CovBooster

In this section, we describe the design of CovBooster, a scalable binary code clone detection framework that improves coverage across diverse compilation environments through signature sets.

Overview. This work extends existing clone detectors through the introduction of *signature sets*, aimed at enhancing clone detection coverage across diverse compilation environments, as illustrated in Figure 2 [15]. CovBooster systematically constructs a function coverage graph without relying on ad hoc or manually selected signatures and extracts a minimal yet representative set of signatures by solving a variant of the dominating set problem in graph theory.

Our enhancements include: (1) selecting minimal signature sets via dominating set computation, (2) grouping and labeling signatures into coherent sets for database construction, and (3) applying lightweight metadata filters to reduce detection overhead during query time. We find that these techniques contribute to making CovBooster both scalable and robust, supporting effective clone detection across diverse compilation environments.

We will describe these techniques in the following sections: signature set decision in Section 3.1, signature database construction in Section 3.2, and metadata-guided detection in Section 3.3.

3.1 Signature Set Selection

The proposed method constructs a *signature set* to reduce redundancy and improve generalizability in clone detection. Below, we present both the intuition and formal algorithmic process. Signature selection involves: (1) building a coverage graph representing inter-variant similarities, and (2) selecting a minimal subset that guarantees coverage across all function variants. Because a single signature may detect multiple function variants, selecting a minimal and comprehensive set improves both storage and computational efficiency. This method is particularly effective in cross-compilation scenarios where individual signatures fail to represent all syntactic variants.

Algorithm 1 Constructing a Coverage Graph with TLSH Similarity

Input: Binary clone detector D
Input: Set of function variants F
Input: TLSH similarity threshold θ
Output: Coverage graph $G = (V, E)$

```
V ← F,  E ← ∅          // Initialize nodes and edges
for v, u ∈ V × V do
    s ← D.Similarity(v, u)      // Compute TLSH distance
    if s ≤ θ then
        E ← E ∪ {(v, u)}        // Similar enough → add edge
    end if
end for
return G = (V, E)
```

Building the Coverage Graph. We construct the coverage graph for binary code clone detection as shown in Algorithm 1. Each vertex represents a function compiled under a specific environment. Edges between vertices indicate that the underlying clone detector deems them similar based on a fixed similarity threshold (e.g., TLSH [27]). The resulting graph models the overall landscape of detectable code equivalence across compilation variants. Thus, achieving broad code clone coverage reduces to identifying a dominating set in this similarity graph.

Finding the Signature Set as a Dominating Set. Identifying an optimal dominating set is NP-hard [5]; hence, we employ a practical heuristic (shown in Algorithm 2) inspired by greedy algorithms [20]. Our strategy is guided by two objectives:

- **Maximum Coverage Increase:** Iteratively selecting uncovered nodes with the highest degree to maximize marginal gain.

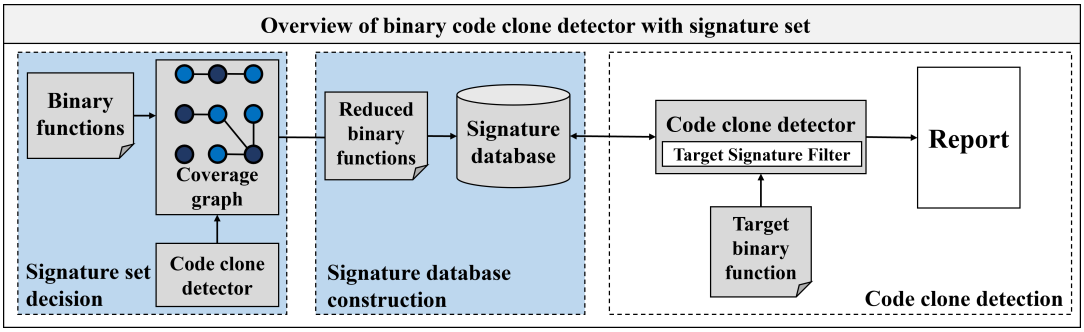


Figure 2: Overview of CovBooster. The highlighted area represents the process for constructing the signature set and generating the signature database.

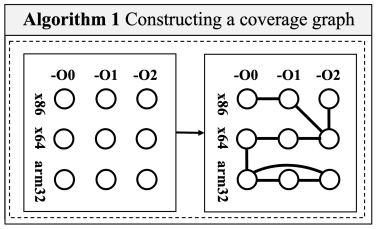


Figure 3: Illustration of Algorithm 1, where a coverage graph is constructed by connecting similar function variants across compilation settings.

Algorithm 2 Selecting Signature Set to Achieve Target Coverage

Input: Graph $G = (V, E)$
Input: Coverage target t
Output: Dominating set $V^* \subseteq V$

$U \leftarrow V, C \leftarrow \emptyset, V^* \leftarrow \emptyset$ // Initialize
while $U \neq \emptyset$ **do**
  Select $u \in U$ with highest degree // Greedy choice
  $V^* \leftarrow V^* \cup \{u\}$ // Add node to dominating set
  for $v \mid (u, v) \in E$ **do**
    $U \leftarrow U \setminus \{v\}$ // Mark neighbor as covered
    $C \leftarrow C \cup \{v\}$
  end for
  $U \leftarrow U \setminus \{u\}$ // Mark selected node as covered
  $C \leftarrow C \cup \{u\}$
  $\Phi_{V^*} = |C|/|V|$ // Compute current coverage ratio
  if $\Phi_{V^*} \geq t$ **then**
    return V^*
  end if
end while

- **Meeting Target Coverage:** Monitoring coverage levels in real-time and terminating once the desired coverage ratio is reached.

The computational complexity for building the coverage graph is $O(n^2)$, and the dominating set selection is $O(n)$ (where $n = |V|$). These computations are performed once during database construction and do not impact runtime performance during clone detection. Additionally, these algorithms are highly parallelizable, making them suitable for preprocessing large-scale datasets.

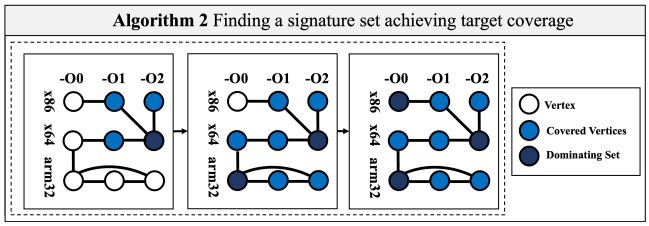


Figure 4: Illustration of Algorithm 2, where a dominating set is constructed by iteratively selecting the highest-degree vertex to form a signature set.

Reducing Coverage Graph Size. The scalability of Algorithm 1 and Algorithm 2 depends on the size of the number of compiled variant set. However, in many real-world scenarios, analysts can limit the compilation space to a handful of commonly used targets. For example, firmware binaries typically target specific architectures. Table 2 shows architecture popularity from Debian Popcon [3], enabling practitioners to focus only on dominant environments (e.g., x86, x64, aarch64). In practice, analysts may restrict the compilation space (e.g., focusing on dominant architectures), and CovBooster naturally operates on the resulting subset of variants when constructing the coverage graph.

Table 2: Submission statistics from Debian Popcon [3], showing architecture-wise distribution of user systems.

Architecture	Submissions	Percentage
x64	194,314	93.44%
x86	11,250	5.09%
aarch64	1,033	0.49%
arm32	1,024	0.49%
others	334	0.16%
Total	**207,955**	**100%**

3.2 Signature Database Construction

Since signature sets include multiple variants of a function, the database size will naturally increase. To manage this, CovBooster applies compression and deduplication when storing representative signatures. Each signature is annotated with a signature set ID, indicating its association with a particular dominating set. This set-aware labeling enables compatibility with existing detection engines while supporting generalized detection. During detection, all signatures from the relevant set are considered, enabling tolerant matching even when certain variants are not present. The set membership design also simplifies updates and incremental indexing. For example, consider a function f that has four variants f_1 through f_4; if f_1 and f_4 are selected for the signature set, then only those two are stored. When f_2 appears during detection, it may still be matched if it shares sufficient TLSH similarity with f_1 or f_4, reducing the need to explicitly store all possible variants. This strategy balances storage efficiency with coverage completeness.

Topology of Coverage Graph (Reduced). Figure 5 shows a reduced coverage graph constructed from binaries compiled using different toolchains. The resulting graph contains dense clusters, indicating that despite syntactic diversity, many variants retain structural similarities exploitable for detection. This motivates our choice of a graph-based signature abstraction. Clusters in the graph often correspond to compiler families or optimization configurations, suggesting that some variants are more central or "transferable" across environments.

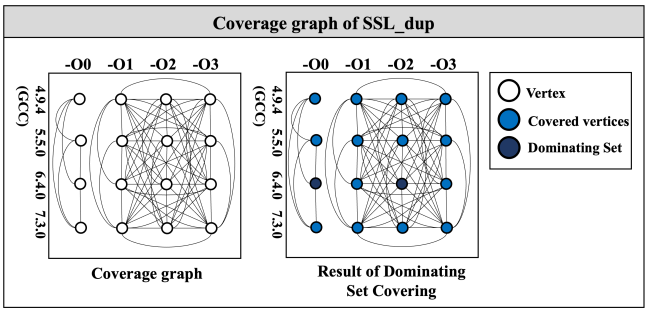


Figure 5: Partial coverage graph showing variation across compilers and optimization levels. Two dominating vertices cover all 16 variants, demonstrating efficient signature selection.

3.3 Metadata-Guided Clone Detection

To reduce detection time, CovBooster employs metadata filtering before comparing against signature sets. Metadata such as target architecture, compiler version, and optimization level is extracted from ELF headers (e.g., `.comment` or `.note`) and used to narrow candidate sets. When reliable metadata is available, comparisons are restricted to a subset of the database to improve scalability. In the absence of metadata, CovBooster performs exhaustive comparisons across all available signatures. Regardless of metadata availability, the system preserves consistent detection accuracy while maintaining optimized performance. The system supports both efficient filtering for standard inputs and exhaustive matching for non-standard or incomplete binaries, thereby enhancing robustness. Since the filtering step does not require disassembly, it is compatible with lightweight or large-scale scanning workflows. As clone detection is increasingly applied to firmware and supply chain analysis, such flexibility becomes essential for practical deployment.

4 Evaluation

In this section, we evaluate CovBooster based on the following research questions:

- **RQ1: Accuracy.** How accurately does CovBooster retrieve semantically equivalent functions across diverse compilation environments?
- **RQ2: Effectiveness.** How effectively does CovBooster outperform static methods like QuickBCC in clone coverage?
- **RQ3: Robustness.** How robust is CovBooster when compared against representative ML-based binary similarity approaches?

4.1 Implementation Overview

CovBooster consists of three main modules: the *Experimental Signature Generator*, *Coverage Booster*, and *Experimental Detector*. It is implemented in Python with about 3K lines of code (LoC), excluding external dependencies. The Experimental Signature Generator performs binary preprocessing, either by lifting the code into an intermediate representation (IR) or by extracting identifiable signatures directly from code segments. In our implementation, binary functions are lifted to VEX IR by identifying strands [2], following the approach used in Firmalice [34]. However, the method is not limited to VEX IR and can support other IRs or static signatures. The Coverage Booster implements the signature set construction algorithm described in Section 3.1. Its goal is to identify a minimal set of representative binaries that maximize clone coverage, thereby reducing redundancy in the signature database. The Experimental Detector is a basic clone detector that evaluates our approach. It uses both signatures and additional fingerprints derived from strand-level attributes and function structure. The detector compares these fingerprints to identify reused code across binaries.

Enhancements in CovBooster. To support the use of signature sets, we modified the structure of the experimental detector as follows:

- Binary functions are decomposed into strands, which are then lifted into VEX IR. The resulting IR serves as the basis for signature generation.
- The detector uses these signatures, combined with fingerprinting, to perform code clone detection across multiple binaries.

Strands, defined as the minimal instruction set required to compute a single variable [2, 15], provide fine-grained abstraction and help preserve semantic equivalence under compilation changes. Our detector follows a standard code clone detection pipeline, which we extended to incorporate signature set support as described in Figure 2.

Signature Set Decision. We implemented the algorithms in Algorithm 1 and Algorithm 2 in approximately 600 lines of Python. These scripts are executed before database construction and return a selected subset of binaries to be used for signature generation. The

output includes file paths and compilation environment metadata for each selected binary.

Signature Database Construction and Detection. All signatures derived from variants of the same function are grouped into a single signature set. We use TLSH (TrendMicro Locality Sensitive Hashing) [27] to store and compare signatures. TLSH offers resilience against small changes in signature content, which is suitable for handling compilation differences.

4.2 Experimental Setup

Experimental Environment. All experiments were conducted on a 12-core AMD Ryzen 9 3900X CPU (4.6GHz) with 32GB RAM running Ubuntu 22.04. All reported times were obtained from repeated executions.

Table 3: CovBooster evaluation dataset details.

Category	Items	Count
Target Binaries	`bool`, `direvent`, `gmp`, `gzip`, `grep`, `libcrypto`, `libssl`, `sed`, `time`, `which`	10
Compilers	GCC (4.9.4–8.2.0), Clang (4.0–7.0)	9
Architectures	x86, x64, aarch64, arm32	4
Opt. Levels	`-O0`, `-O1`, `-O2`, `-O3`	4
Total Variants	*Targets* × *Compilers* × *Arch.* × *Opt.*	**1,440**

Dataset and Compilation Variants. We evaluated CovBooster using real-world binaries compiled from ten popular open-source software packages, as summarized in Table 4. Each package was compiled under diverse environments using multiple compilers, architectures, and optimization levels via the BinKit framework [21]. To ensure fair and consistent evaluation, we selected functions that are present in all compilation variants (referred to as common functions), each with a strand size over 100, so that all detectors are tested on an identical function set. The detailed configuration of the evaluation dataset is summarized in Table 3.

4.3 RQ1: Accuracy of CovBooster

Methodology. We evaluate the ability of CovBooster to retrieve the correct clone—*i.e.*, a semantically equivalent function—from binaries compiled under different settings. For each function, the selected signature set is matched against all variants using TLSH. A match is considered a true positive only when the top-ranked match corresponds to the same function name as the query. To establish a reliable ground truth for function identity, we used binaries compiled with debug symbols and without symbol stripping, as provided by the BinKit framework. To determine a suitable TLSH threshold, we varied the cutoff from 0 to 30 and measured its impact on precision, recall, F1-score, and dominating set size. To avoid incorrectly penalizing precision, we restricted our evaluation to functions that appear consistently across all 144 compilation variants, ensuring that each function has a unique ground-truth identity. Under this setup, matches between different function names genuinely correspond to false positives rather than latent clones or duplicated routines within the original projects. To provide a single composite indicator for threshold selection, we define an Efficiency Score that jointly considers detection accuracy and signature-set compactness. Specifically, the score is computed as a weighted combination of F1-score (40%), precision (30%), and the inverse of the dominating-set size normalized over its maximum value (30%). The resulting metric is used for sensitivity analysis in Figure 6 to illustrate overall trade-offs.

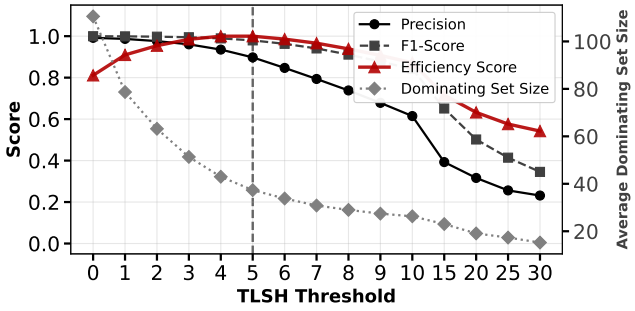


Figure 6: Threshold sensitivity analysis of TLSH cutoff.

Results. Figure 6 illustrates how precision, F1-score, and dominating set size vary with different TLSH cutoffs. As the threshold becomes more permissive, precision and F1-score gradually decrease while the dominating set size shrinks. At threshold 0, the overly strict cutoff retains nearly all raw variants as signatures, leading to unnecessary database expansion. As the threshold increases beyond 6, false positives rise noticeably, reducing precision below 0.94 and steadily degrading F1-score. This occurs because TLSH measures byte-level similarity: a higher cutoff tolerates local n-gram overlaps even when functions differ semantically. Consequently, structurally similar but unrelated helpers or initialization routines are mistaken as clones, explaining the surge of false positives beyond threshold 6. In contrast, threshold 5 provides the most balanced trade-off: it achieves the highest F1-score (0.98) and an average precision of 0.90, with a modest dominating set size of about 37 signatures per function. This configuration minimizes false alarms without sacrificing completeness, demonstrating that a small and representative signature set can fully capture cross-variant equivalence. As summarized in Table 4, recall remains perfect (1.00) across all evaluated binaries at this threshold. Consequently, all subsequent analyses are reported using threshold 5 as the default configuration.

As summarized in Table 4, CovBooster consistently achieves perfect recall (1.00) across all ten target binaries at the selected threshold. Precision ranges from 0.72 to 1.00 depending on the package, with F1-scores between 0.95 and 1.00. Notably, `libcrypto` and `libssl` show slightly lower precision (0.77 and 0.78, respectively), likely due to structurally similar cryptographic helper functions. In contrast, `which` achieves perfect detection, with precision, recall, and F1-score all equal to 1.00. On average, CovBooster attains precision of 0.90 and F1-score of 0.98, confirming that the dominating set approach preserves strong discriminative capability while ensuring exhaustive retrieval. From a security perspective, this guarantee of zero false negatives is crucial: while false positives only increase analyst workload, false negatives correspond to

Table 4: Average clone detection accuracy per binary for CovBooster at threshold ≤ 5. Consistently high recall (1.00) suggests effective variant coverage using compact signature sets.

Target Binary	Precision	Recall	F1-score
bool 0.2.2	0.98	1.00	1.00
direvent 5.3	0.72	1.00	0.95
gmp 6.2.1	0.91	1.00	0.99
grep 3.8	0.94	1.00	0.99
gzip 1.12	0.91	1.00	0.98
libcrypto 1.0.1f	0.77	1.00	0.95
libssl 1.0.1f	0.78	1.00	0.95
sed 4.9	0.99	1.00	1.00
time 1.9	0.98	1.00	1.00
which 2.21	1.00	1.00	1.00
Average	**0.90**	**1.00**	**0.98**

missed vulnerabilities—an unacceptable outcome in vulnerability detection pipelines.

Answer to RQ1. CovBooster achieves consistently high detection accuracy across all ten evaluated binaries, with an average precision of 0.90, recall of 1.00, and F1-score of 0.98 at threshold 5. The dominating-set signatures preserve full cross-variant coverage without introducing false negatives, demonstrating robust clone retrieval across diverse compilation settings.

4.4 RQ2: Clone Coverage Improvement

Methodology. We compare CovBooster against QuickBCC [15], a static strand-based clone detector that selects a single TLSH signature per function. While computationally efficient, this single-signature strategy often fails to generalize across compilers, architectures, and optimization levels, especially when low-level binary transformations introduce structural variations.

To address this, CovBooster builds a per-function *coverage graph*, where nodes represent binary variants and edges denote TLSH similarity. From this graph, it computes a *minimal dominating set*—a compact subset of representative signatures that collectively cover the entire variant space. A variant is considered covered if it is either in the dominating set or similar (within threshold) to one of the selected nodes.

Results. Across all ten binary families, to cover all 144 compilation variants, QuickBCC typically requires 111 unique reference variants per function, due to its inability to generalize a single signature across different build settings. Although QuickBCC generates one signature per compilation setting in principle, some variants that compile to nearly identical strand sequences produce the same hash. As a result, certain compiler–optimization combinations yield overlapping signatures, slightly reducing the total count below 144 while still lacking full cross-variant generalization. This high number stems from its single-signature model, which cannot adapt to syntactic variability caused by compiler backend changes, instruction reordering, or differing calling conventions. In a few exceptional cases, several variants of the same function compile to identical strand patterns, producing fewer distinct signatures, but such instances are rare and do not significantly affect the overall trend.

Table 5: Per-family statistics: QuickBCC vs. CovBooster (required average variants and percentage relative to 144 variants). CovBooster achieves an average 66% reduction in the number of required variants compared to QuickBCC.

Target Binary	QuickBCC Required Variants	CovBooster Required Variants
bool 0.2.2	115.4	42.4
direvent 5.3	104.0	35.0
gmp 6.2.1	94.1	32.6
grep 3.8	126.8	45.9
gzip 1.12	130.8	35.7
libcrypto 1.0.1f	107.3	32.9
libssl 1.0.1f	99.9	29.6
sed 4.9	118.0	47.0
time 1.9	109.8	36.0
which 2.21	105.2	36.3
Overall	111.1 (77.2%)	**37.3 (25.9%)**

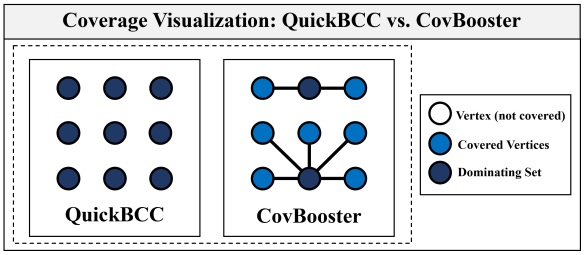


Figure 7: Signature selection in QuickBCC vs. CovBooster.

In contrast, CovBooster achieves *full variant coverage* across all evaluated functions while using only 37.3 signatures per function on average, corresponding to roughly 25.9% of all available variants. Although the average dominating set size slightly increases compared to the previous configuration (threshold 10), this adjustment yields markedly higher precision (0.90) and F1-score (0.98) as shown in Table 4. These results demonstrate that CovBooster can maintain complete coverage while significantly reducing redundancy—achieving the same detection completeness as QuickBCC with just one-third of the required signatures.

The effectiveness of CovBooster is further illustrated in Figure 7, which contrasts the coverage structures of the two systems. QuickBCC's approach corresponds to isolated nodes, each covering only itself, whereas CovBooster selects a small set of central nodes that cover surrounding variants through TLSH similarity. This graph-based strategy enables robust and generalized detection without overfitting to specific compilation environments.

Although the absolute matching time is small in our experimental setting (1,440 variants across ten projects), the reduction in

signature-set size has significant implications in large-scale deployments. In practical software composition analysis (SCA), firmware auditing, or supply-chain scanning systems, signature databases often contain hundreds of thousands of functions. Because clone detection typically performs pairwise comparisons between a query and all stored signatures, reducing the signature set from N to $|V^*|$ directly lowers the matching cost from $O(N)$ to $O(|V^*|)$. Thus, while the per-function time savings may appear modest in small datasets, the same 66% reduction becomes substantially more impactful in large-scale, n-to-n matching workflows.

In summary, CovBooster achieves exhaustive clone coverage with approximately 37 representative signatures per function—a 66% reduction compared to QuickBCC—while maintaining near-perfect precision and recall. This confirms that the proposed coverage-driven signature selection substantially enhances both efficiency and reliability in binary clone detection.

Answer to RQ2. CovBooster achieves full cross-compiler and cross-architecture clone coverage using only 37.3 representative signatures per function—a 66% reduction compared to the 111 signatures required by QuickBCC. Despite this reduction, CovBooster preserves complete variant coverage and maintains high accuracy, demonstrating that coverage-driven signature selection substantially improves efficiency and scalability without sacrificing detection completeness.

4.5 RQ3: Comparison with ML Approaches

To evaluate CovBooster in the broader context of code clone detection tools, we qualitatively compare it with representative ML-based approaches in Table 6. Tools like SAFE [26], Asm2Vec [4], and Trex [28] use deep learning or embeddings for semantic understanding, but vary in generalization and deployment complexity. From the table, CovBooster matches or exceeds ML-based tools in architectural and compiler robustness while maintaining a fully interpretable static analysis pipeline. CovBooster uses graph-based signature selection and metadata filtering, enabling immediate deployment without the need for large-scale training or retraining. Each match in CovBooster is traceable to TLSH similarity and coverage graph decisions, which benefits high-assurance use cases such as firmware auditing and supply chain analysis. Although ML-based tools offer abstraction benefits, many still face issues with generalization and transparency. More recent models like Gemini [42] and CEBin [38] show progress, but challenges like training data dependency and black-box reasoning remain. CovBooster provides a lightweight and deterministic alternative in scenarios demanding reliability and explainability.

Dataset bias and generalization. ML-based clone detectors often struggle when applied beyond their training scope. Most embedding models are trained on limited datasets with fixed compilers or optimization flags, causing them to learn compiler-specific traits rather than true semantic invariants. When evaluated on unseen variants, their similarity may reflect layout artifacts instead of functionality, leading to unstable results.

Table 6: Qualitative comparison of clone detection tools. CovBooster uniquely supports all compilation variations across architecture and compiler.

Tool	Type	Public	Arch.	Compiler
SAFE	Embedding	△	△	×
Asm2Vec	Embedding	△	×	△
Trex	ML + Static	◯	◯	△
QuickBCC	Static	◯	×	△
CovBooster	Static	◯	◯	◯

Legend: ◯ Supported △ Partially supported or limited × Not supported

Explainability and traceability. Embedding-based similarity provides limited interpretability, making it difficult to explain or justify individual matching results. In contrast, CovBooster presents explicit and reproducible evidence through TLSH distances and coverage-graph relationships, allowing analysts to independently verify each detection, which is essential for reliability and assurance-oriented analysis.

Answer to RQ3. CovBooster shows strong robustness across diverse compilation environments, consistently preserving stable similarity judgments regardless of architecture, compiler, or optimization settings. Its use of deterministic TLSH scores and coverage-graph reasoning ensures that matching results remain explainable, reproducible, and dependable—qualities essential for high-assurance applications such as firmware auditing, large-scale SCA, and supply-chain security analysis.

5 Discussion

5.1 Applicability of CovBooster

The core idea of CovBooster—selecting a compact yet representative signature set—is applicable beyond QuickBCC. This general principle of maximizing coverage through compact selection can extend to various signature extraction methods. Code clone detectors based on control-flow graphs (CFGs), call graphs (CGs), or learned embeddings may benefit from a similar coverage-based selection strategy. While our implementation targets strand-level binary signatures, the method generalizes to other granularities such as functions, basic blocks, or even control-flow graphs [1, 19]. It may also extend to source-level clone detection [22] by applying coverage graphs over abstract syntax trees or intermediate representations.

5.2 Limitations and Optimization Opportunities

Although CovBooster uses more signatures than single-signature baselines, it remains efficient due to metadata-based filtering. This filtering narrows the search space by selecting only relevant signature candidates based on architecture, compiler, and optimization-level metadata. Further improvements may be achieved by incorporating richer structural metadata, such as control-flow summaries or instruction histograms, to enhance precision in scenarios with high variance. Our use of a simple max-degree heuristic yields efficient

results—only 25.9% of variants are selected as signatures while still achieving full coverage. However, this greedy strategy does not always produce the most compact set. More advanced heuristics, such as semantic scoring or multi-step lookahead, may identify smaller sets without sacrificing coverage or accuracy. Lastly, CovBooster assumes reliable similarity scores across compilation variants, such as TLSH-based distances. In environments with heavy optimization, obfuscation, or noise, these scores may be unstable or inconsistent. Future work may address this by combining multiple detectors or using ensemble similarity models to improve robustness across edge cases and adversarial inputs. In practical deployments, aggressive optimization or obfuscation may weaken the similarity relationships used to build coverage graphs. Such transformations can increase the number of signatures required for full coverage. Enhancing CovBooster 's robustness under these real-world conditions remains an important direction for future work.

5.3 Threats to Validity

This study has several threats to validity. First, CovBooster was implemented by the authors, which may introduce bias despite extensive testing and cross-checking with QuickBCC. Second, our dataset—ten open-source packages compiled under 144 environments—is diverse but may not fully represent real-world binaries with obfuscation or proprietary toolchains, limiting generalizability. Third, the reliance on TLSH as the similarity metric provides efficiency but may be unstable under aggressive optimizations or adversarial transformations. Fourth, the greedy heuristic for signature selection, while effective in practice, does not guarantee optimality, and alternative strategies might yield different trade-offs. Finally, our evaluation assumes ground truth based on debug-preserved function identities; although we restricted analysis to functions consistently present across all variants, residual biases in function naming or latent intra-project clones may still influence measured precision.

6 Related Work

6.1 Code Clone Detection

Code clone refers to code segments reused across different projects or binaries. Code clones can result from straightforward copy-paste actions or more subtle reuse patterns. Researchers leverage clone detection for diverse applications, including vulnerability discovery, license violation detection, plagiarism identification, and malware analysis [11, 22]. Code clones can be identified at various abstraction levels: source code, binary code, and semantics [31].

Source Code Clone Detection. Source code clone detection identifies repeated or reused code segments within source code [31]. Approaches typically focus on accuracy and scalability to handle large codebases efficiently. Existing techniques tokenize source code or extract abstract representations, followed by heuristic filtering or hashing methods for efficient comparisons [16, 22, 33, 41]. NICAD [30] also employs exemplar-based source-level clone selection, but such techniques do not generalize to binary code. However, these methods assume the availability of source code and thus cannot be directly applied to binaries.

Binary Code Clone Detection. Binary code clone detection is critical for vulnerability identification, plagiarism detection, malware analysis, and software forensics when source code is unavailable [11]. Techniques typically compare binary functions, basic blocks, or execution paths. For instance, methods like strands or instruction sequences identify clones by comparing structural or syntactic features [2, 14, 15]. Other approaches leverage higher-level abstractions, such as control flow graphs (CFG) or semantic embeddings, to achieve robustness against changes in compilation environments [1, 44]. However, accurately identifying function boundaries remains a key challenge in binary-level analysis [8].

6.2 Impact of Compilation Environments

Different compilation environments—varying architectures, compilers, or optimization levels—affect binary code significantly. Even semantically identical functions can differ syntactically or structurally depending on compilation settings [17]. Thus, many approaches emphasize resilience against compilation environment variations to enhance robustness. For example, some techniques use semantic-aware signatures derived from control-flow graphs, embeddings, or input-output behaviors to minimize sensitivity to compilation changes [9, 21, 29].

6.3 Machine Learning Approaches

Machine learning (ML)-based approaches offer powerful solutions by automatically extracting code embeddings or semantic features [25]. These methods, including deep learning-based NLP techniques, typically achieve high accuracy but require extensive training data and computational resources [36, 38]. Recent research aims to balance ML approaches with traditional heuristics to improve detection accuracy and efficiency [38, 41]. However, ML models' effectiveness often depends on extensive training datasets, careful feature engineering, and tuning, which can be costly in practice.

7 Conclusion

We proposed a method to improve the coverage of binary code clone detectors by using a signature set instead of single signatures, formulating the problem as an instance of the dominating set problem. Our algorithm efficiently selects a minimal set of signatures that maximize coverage across diverse compilation environments, enabling detection without requiring per-environment signatures. Through implementation and evaluation, we showed that even a small subset of signatures can cover most code clones in our corpus, outperforming single-signature baselines. We believe this approach can enhance various clone detectors and be extended to different levels of granularity.

Data Availability

The source code and dataset for CovBooster are available at https://github.com/UNILESS/CovBooster-public.

Acknowledgment

We thank the anonymous reviewers for their insightful comments to improve the quality of the paper. We are also grateful to Seonwoong Yoon for his valuable contribution to improving the algorithm. This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP), grant funded by

the Korea government (MSIT) (No. RS-2024-00440780 Development of Automated SBOM and VEX Verification Technologies for Securing Software Supply Chains), ICT Creative Consilience Program (IITP-2025-RS-2020-II201819, 10%), and the Research Foundation of the City University of New York (RFCUNY).

References

[1] Saed Alrabaee, Lingyu Wang, and Mourad Debbabi. 2016. BinGold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs. Digital Investigation 18 (2016), S11–S22.

[2] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical similarity of binaries. ACM SIGPLAN Notices 51, 6 (2016), 266–280.

[3] Debian 2025. Debian Popularity Contest. https://popcon.debian.org/, Accessed October 2025.

[4] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In 2019 IEEE Symposium on Security and Privacy (SP). 472–489.

[5] Irit Dinur and Samuel Safra. 2005. On the hardness of approximating minimum vertex cover. Annals of mathematics (2005), 439–485.

[6] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. DeepBinDiff: Learning program-wide code representations for binary diffing. In NDSS.

[7] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket execution: Dynamic similarity testing for program binaries and components. In 23rd USENIX Security Symposium (USENIX Security). 303–317.

[8] Sebastian Eschweiler, Khaled Yakdan, Elmar Gerhards-Padilla, et al. 2016. Discovre: Efficient cross-architecture identification of bugs in binary code.. In NDSS. 58–79.

[9] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: A semantic learning based vulnerability seeker for cross-platform binary. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE).

[10] GitHub 2025. GitHub. http://github.com/, Accessed October 2025.

[11] Irfan Ul Haq and Juan Caballero. 2021. A survey of binary code similarity. ACM Computing Surveys (CSUR) 54, 3 (2021), 1–38.

[12] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2017. Binary code clone detection across architectures and compiling configurations. In 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC). 88–98.

[13] Yikun Hu, Yuanyuan Zhang, Juanru Li, Hui Wang, Bodong Li, and Dawu Gu. 2018. BinMatch: A semantics-based hybrid approach on binary code clone analysis. In 2018 IEEE international conference on software maintenance and evolution (ICSME). 104–114.

[14] He Huang, Amr M Youssef, and Mourad Debbabi. 2017. BinSequence: Fast, accurate and scalable binary code reuse detection. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIACCS). 155–166.

[15] Hajin Jang, Kyeongseok Yang, Geonwoo Lee, Yoonjong Na, Jeremy D Seideman, Shoufu Luo, Heejo Lee, and Sven Dietrich. 2021. QuickBCC: Quick and Scalable Binary Vulnerable Code Clone Detection. In IFIP SEC. 66–82.

[16] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: finding unpatched code clones in entire os distributions. In 2012 IEEE Symposium on Security and Privacy (SP). 48–62.

[17] Yuede Ji, Lei Cui, and H Howie Huang. 2021. BugGraph: Differentiating source-binary code similarity with graph triplet-loss network. In Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (ASIACCS). 702–715.

[18] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering 28, 7 (2002), 654–670.

[19] Wooseok Kang, Byoungho Son, and Kihong Heo. 2022. Tracer: Signature-based static analysis for detecting recurring vulnerabilities. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS).

[20] Omar Kettani, Faycal Ramdani, and Benaissa Tadili. 2013. A Heuristic Approach for the Vertex Cover Problem. International Journal of Computer Applications 82, 4 (2013).

[21] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. 2022. Revisiting Binary Code Similarity Analysis using Interpretable Feature Engineering and Lessons Learned. IEEE Transactions on Software Engineering (2022), 1–23.

[22] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A scalable approach for vulnerable code clone discovery. In 2017 IEEE Symposium on Security and Privacy (SP). 595–614.

[23] Labrador Labs 2025. Labradorlabs. https://labradorlabs.ai/, Accessed October 2025.

[24] Arun Lakhotia, Mila Dalla Preda, and Roberto Giacobazzi. 2013. Fast location of similar code fragments using semantic 'juice'. In Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop. 1–6.

[25] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. 2022. How machine learning is solving the binary function similarity problem. In 31st USENIX Security Symposium (USENIX Security). 2099–2116.

[26] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. 2019. SAFE: Self-attentive function embeddings for binary similarity. In Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference (DIMVA).

[27] Jonathan Oliver, Chun Cheng, and Yanggui Chen. 2013. TLSH–a locality sensitive hash. In 2013 Fourth Cybercrime and Trustworthy Computing Workshop. IEEE, 7–13.

[28] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2020. Trex: Learning execution semantics from micro-traces for binary similarity. arXiv preprint arXiv:2012.08680 (2020).

[29] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In 2015 IEEE Symposium on Security and Privacy (SP). 709–724.

[30] Chanchal K Roy and James R Cordy. 2008. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In 2008 16th iEEE international conference on program comprehension. IEEE, 172–181.

[31] Chanchal K Roy, James R Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Science of computer programming 74, 7 (2009), 470–495.

[32] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. 2009. Detecting code clones in binary executables. In Proceedings of the 18th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA). 117–128.

[33] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. SourcererCC: Scaling code clone detection to big-code. In Proceedings of the 38th International Conference on Software Engineering (ICSE). 1157–1168.

[34] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In NDSS.

[35] Snyk 2025. Snyk. https://snyk.io/, Accessed October 2025.

[36] Zirui Song, Jiongyi Chen, and Kehuan Zhang. 2024. Bin2Summary: Beyond Function Name Prediction in Stripped Binaries with Functionality-Specific Code Embeddings. Proceedings of the ACM on Software Engineering 1 (2024), 47–69.

[37] Stack Overflow 2025. Stack Overflow. https://stackoverflow.com/, Accessed October 2025.

[38] Hao Wang, Zeyu Gao, Chao Zhang, Mingyang Sun, Yuchen Zhou, Han Qiu, and Xi Xiao. 2024. 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 (ISSTA).

[39] Shuai Wang and Dinghao Wu. 2017. In-memory fuzzing for binary code similarity analysis. In 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). 319–330.

[40] Seunghoon Woo, Dongwook Lee, Sunghan Park, Heejo Lee, and Sven Dietrich. 2021. V0Finder: Discovering the Correct Origin of Publicly Reported Software Vulnerabilities. In 30th USENIX Security Symposium (USENIX Security).

[41] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, et al. 2020. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In 29th USENIX Security Symposium (USENIX Security).

[42] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. 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 (CCS).

[43] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. 2020. Patch based vulnerability matching for binary programs. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA). 12 pages.

[44] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. 2017. Spain: security patch analysis for binaries towards understanding the pain and pills. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE).