Beyond Similarity Scores: Evidence-Based Third-Party Library Detection for C/C++ Binaries

CHENGYUE LIU, Nanyang Technological University, Singapore
ZHENGZI XU✉, Imperial Global Singapore, Singapore
LYUYE ZHANG, Nanyang Technological University, Singapore
JIAHUI WU, Nanyang Technological University, Singapore
KAIXUAN LI, Nanyang Technological University, Singapore
YANG LIU, Nanyang Technological University, Singapore

Detecting third-party libraries (TPLs) in C/C++ binaries is essential for software supply chain security, enabling vulnerability identification and license compliance. Existing methods predominantly rely on similarity matching: extracting features from binaries and comparing them against library databases. However, similarity scores alone cannot reliably determine library presence. Low similarity causes false negatives when matchable features are limited. More critically, high similarity does not guarantee accuracy: libraries often share features through shared dependencies, forks, or similar functionality, causing multiple candidates to match even when only one is present. These issues suggest that similarity matching is effective for narrowing candidates but insufficient as the final decision mechanism. Rather than relying solely on similarity scores, reliable detection requires multi-source evidence to verify each candidate.

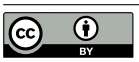
To this end, we propose *BLADE*, which reframes TPL detection as evidence-based candidate verification. Instead of relying on similarity scores to make final decisions, *BLADE* retrieves candidates broadly to mitigate false negatives, and then collects evidence from multiple sources, which an LLM analyzes through structured verification workflows to filter false positives: first confirming candidates with clear identity markers, then systematically checking remaining candidates against common false positive patterns. To evaluate *BLADE*, we build the largest C/C++ binary TPL benchmark to date, comprising 3,403 binaries and 1,016 libraries. Results show that *BLADE* achieves 97.60% precision and 93.74% recall (F1: 95.63%), improving F1-score by 41.83 percentage points over the best baseline. The average cost is $0.0378 per binary. *BLADE* has been deployed in a commercial software composition analysis product, demonstrating practical feasibility at scale.

CCS Concepts: • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: Binary Analysis, Third-Party Library Detection, SCA, LLM

ACM Reference Format:
Chengyue Liu, Zhengzi Xu, Lyuye Zhang, Jiahui Wu, Kaixuan Li, and Yang Liu. 2026. Beyond Similarity Scores: Evidence-Based Third-Party Library Detection for C/C++ Binaries. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA034 (October 2026), 22 pages. https://doi.org/10.1145/3832125

Authors' Contact Information: Chengyue Liu, Nanyang Technological University, Singapore, Singapore, CHENGYUE001@e.ntu.edu.sg; Zhengzi Xu, Imperial Global Singapore, Singapore, Singapore, z.xu@imperial.ac.uk; Lyuye Zhang, Nanyang Technological University, Singapore, Singapore, zh0004ye@e.ntu.edu.sg; Jiahui Wu, Nanyang Technological University, Singapore, Singapore, jiahui004@e.ntu.edu.sg; Kaixuan Li, Nanyang Technological University, Singapore, Singapore, kaixuan.li@ntu.edu.sg; Yang Liu, Nanyang Technological University, Singapore, Singapore, yangliu@ntu.edu.sg.



ACM 2994-970X/2026/10-ARTISSTA034
https://doi.org/10.1145/3832125

1 Introduction

Modern software development relies heavily on open-source third-party libraries (TPLs), which accelerate development but also introduce supply chain security risks. High-profile incidents such as the 2020 SolarWinds attack, which compromised over 18,000 organizations [2], have demonstrated the potential impact of TPL vulnerabilities. In response, both the United States and European governments now require software vendors to disclose the TPLs used in their products [7, 19]. Accurately identifying TPLs in software has therefore become essential for supply chain security.

Among all language ecosystems, C/C++ presents unique challenges for this task. C/C++ underpin critical systems such as operating systems and infrastructure software, yet they lack standardized package management, making library usage far less transparent than in modern languages. Developers integrate libraries through various mechanisms (e.g., static linking, dynamic linking, or direct source inclusion), which complicates identification. Moreover, most real-world software is distributed only in binary form without source code access [30]. These factors make binary-level TPL detection for C/C++ both essential and difficult.

Existing methods for binary TPL detection follow a common paradigm: they extract features from known libraries to build a database, compare target binaries against this database, compute how similar each binary is to each library, and report libraries whose similarity scores exceed a threshold. Features have evolved from simple string literals and function names [12, 43], to structural features such as control-flow graphs and function-call graphs [28, 29], and more recently to neural embeddings that compare decompiled pseudo-code with source functions [14]. Throughout this evolution, the underlying approach remains the same: all these methods determine library presence by measuring how similar a binary is to known libraries.

While similarity-based approaches can retrieve potential matches, similarity scores cannot serve as reliable evidence for determining whether a library is actually present. On one hand, low similarity may cause false negatives: similarity scores depend on the number of matchable features between a binary and a library, but many factors limit feature availability, such as libraries with limited code, partial usage of library functionality, or symbol stripping during compilation. When too few features match, similarity falls below the preset detection threshold (the minimum score required to report a match), causing present libraries to be missed. More critically, high similarity does not guarantee accuracy: high similarity means many features matched, but these features may not be unique to a single library. Libraries often share features through common dependencies (e.g., both `tar` and `cpio` include `gnulib`), fork relationships (e.g., OpenSSL and BoringSSL), or similar functionality. When multiple libraries share the same features, a binary shows high similarity with all of them, and if all exceed the threshold, false positives result. The fundamental issue is that similarity-based methods aggregate all matches into a single score, treating every match as an equal vote while discarding the information each match carries: *what* matched (an identity marker or a generic string), *where* it matched (core code or a vendored subdirectory), and *how* candidates relate to each other (independent libraries or forks sharing code). Therefore, reliable detection requires deeper analysis beyond similarity scores alone, such as collecting evidence from multiple sources and verifying each candidate against this evidence.

This analysis requires collecting comprehensive evidence, including identity markers in binaries (e.g., copyright notices), repository structures of libraries, detailed matching patterns between libraries and binaries, and comparisons among all candidate libraries, then making judgments based on understanding this evidence. However, this type of analysis is difficult to cover comprehensively with handcrafted rules, and manual analysis cannot scale due to cost. What we need is an approach that can both understand diverse evidence and execute automatically. Large language models (LLMs) have the capabilities to perform such analysis, making them a promising solution. However,

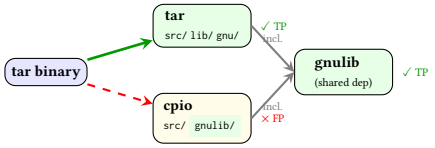


Fig. 1. Shared dependency causes false positive.

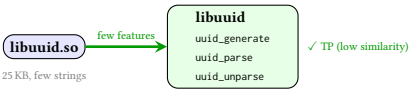


Fig. 2. Sparse features cause false negative.

LLMs cannot reliably solve this problem end-to-end; they require careful problem decomposition, structured evidence, and rule-guided workflows to work effectively.

In this paper, we propose *BLADE* (**B**inary **L**ibrary **A**nalysis and **D**etection **E**ngine), which reframes binary TPL detection as evidence-based candidate verification. Instead of relying on similarity scores for final decisions, *BLADE* uses feature matching as a means to narrow the analysis scope: the first stage retrieves candidates broadly to mitigate false negatives, and the second stage analyzes multi-source evidence to filter false positives. Specifically, the first stage extracts strings and symbols from the binary as matching features, compares them against a pre-built library feature database, and selects the top-N matching libraries as candidates. The second stage collects evidence from four sources: binary-side information, library repository metadata, binary-library matching analysis, and cross-candidate comparison. An LLM then serves as the analysis executor, following structured verification workflows. The output includes detected libraries along with the evidence and analysis process supporting each decision.

To evaluate *BLADE*, we constructed the largest C/C++ binary TPL detection benchmark to date, comprising 3,403 binaries generated under diverse compilation configurations (gcc/clang, x86_64/arm, stripped symbols), covering 1,016 TPLs and 3,771 reuse relations, which is more than 10× larger than datasets used in prior work [12, 14, 15, 29, 43]. Experimental results show that *BLADE* achieves 97.60% precision, 93.74% recall, and an F1-score of 95.63%, improving F1-score from 53.80% to 95.63% over the best baseline. The average processing cost is only 0.64 seconds and $0.0378 per binary, demonstrating practical feasibility for large-scale deployment. *BLADE* has been integrated into a commercial software composition analysis (SCA) product for compliance checking in industrial settings. In summary, this paper makes the following contributions:

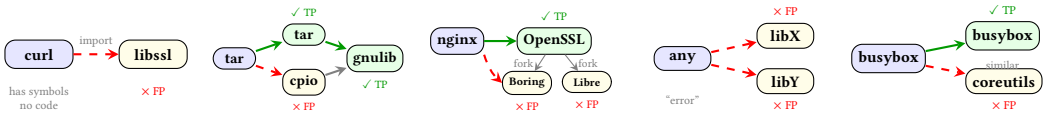
- **Problem analysis and approach.** We analyze why similarity-based detection produces false positives, identifying systematic causes rooted in how libraries share, reuse, and evolve code. Based on this analysis, we reframe TPL detection as evidence-based candidate verification.
- **Framework and implementation.** We design and implement *BLADE*, an evidence-based detection framework. The retrieval stage prioritizes recall through broad candidate selection. The verification stage collects targeted evidence and uses an LLM guided by structured analysis to filter false positives. *BLADE* is released as open source to support reproducibility.
- **Comprehensive evaluation.** We construct the largest C/C++ binary TPL detection dataset to date (3,403 binaries, 1,016 libraries). *BLADE* achieves 95.63% F1-score, outperforming the best baseline by 41.83 points. The dataset is also publicly available to facilitate future research.

2 Motivation

2.1 Motivation Examples

We present two examples that illustrate the limitations of similarity-based matching approaches.

Example 1: False positives from shared dependencies. As shown in Figure 1, `tar` and `cpio` are both GNU archiving tools that depend on `gnulib`, a collection of common utility modules. When analyzing a `tar` binary, similarity-based tools [14, 43] report `cpio` as a false positive because both projects share gnulib code, leading to overlapping string features. However, by analyzing the



(a) C1: Dynamic Linking (b) C2: Shared Deps (c) C3: Evolutionary (d) C4: Generic Features (e) C5: Similar Func.

Fig. 3. Five categories of false positives in similarity-based detection.

distribution of matched features, we can observe that matches for `tar` cover the entire repository and include identity-revealing strings such as "GNU tar", while matches for `cpio` are limited to the gnulib subdirectory. This pattern indicates shared dependencies rather than actual cpio code, allowing us to confirm tar while rejecting cpio.

Example 2: False negatives from sparse features. As shown in Figure 2, `libuuid.so.1.0.0` is a binary from the `util-linux` package, responsible for generating and parsing UUIDs. The file is small (approximately 25 KB) and contains limited string literals (fewer than 100), causing similarity-based tools to miss it due to low similarity scores. However, by examining the exported symbols, we find that `uuid_generate`, `uuid_parse`, and `uuid_unparse` form a consistent pattern that corresponds exactly to the libuuid API. This evidence confirms the library's presence despite the limited feature count.

These examples illustrate a key insight: similarity scores alone cannot reliably determine library presence. In Example 1, high similarity leads to a false positive; in Example 2, low similarity leads to a false negative. Reliable detection requires understanding the root causes of such errors, collecting targeted evidence, and applying appropriate analysis strategies to reach sound conclusions.

2.2 Practical Observations

The examples above hint at recurring patterns. To systematically understand the root causes of detection failures, we analyzed results from deploying TPL detection in industrial settings. We applied three mature detection tools (two mainstream commercial SCA products, anonymized per vendor request, and BinaryAI [14]) to analyze 10 software systems from automotive and mobile domains (totaling 5.97 GB across 32,162 binary files), including in-vehicle infotainment platforms and mobile operating systems. Working with domain engineers, we verified detection results by focusing on cases with clear ground truth (e.g., binaries containing explicit license information of reused libraries) that were still missed or incorrectly reported, as well as cases where different tools produced inconsistent results. We summarize two key observations below.

Observation 1: Missed libraries are often matchable. We observed that missed libraries are often retrievable during the matching phase but filtered out due to insufficient similarity scores. To quantify this, we examined around 200 false negative cases: using string and symbol features against the feature database of one commercial tool, when candidates were ranked by string match count alone, the correct library appeared in top-1 for 55%, top-5 for 84%, and top-10 for 90%. This suggests that feature matching itself has sufficient recall. The challenge is not finding matches, but filtering true positives from the candidates that emerge when thresholds are relaxed.

Observation 2: False positives stem from identifiable causes. We observed that false positives, despite their varied manifestations, arise from a limited set of root causes. To characterize these patterns, we analyzed around 200 false positive cases from the three tools. They fall into two broad groups: *relationship-based* false positives, where the binary and candidate are genuinely related but the library is not embedded, and *coincidental overlap*, where features match by chance. Within each group, we identified distinct mechanisms, yielding five categories in total. Figure 3 illustrates this taxonomy, and we describe each category below.

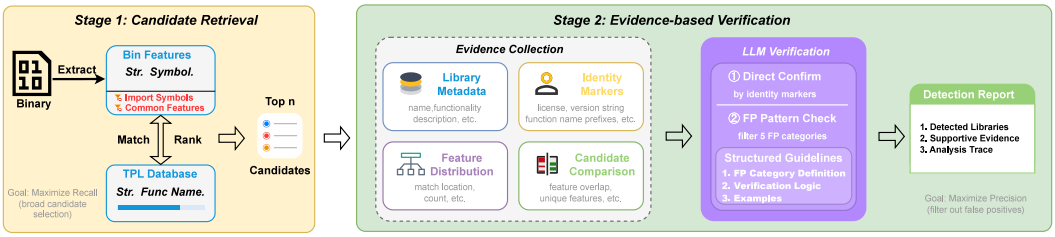


Fig. 4. Overview of *BLADE*: a two-stage architecture with candidate retrieval and evidence-based verification.

Group 1: Relationship-based false positives involve a genuine relationship between the binary and the reported library, but the library is not actually embedded. We identify three such patterns:

- **C1 (Dynamic Linking):** The candidate is linked as an external dynamic library. Its symbols appear in the binary's import table, but the actual code is in a separate `.so` file. For example, a binary dynamically linking `libssl.so` contains imported symbols like `SSL_connect` but no OpenSSL code. Imported symbols should not be used as matching features.
- **C2 (Shared Dependencies):** The binary and the candidate both statically embed the same third-party code. For instance, `tar` and `cpio` both include `gnulib`, so analyzing a `tar` binary incorrectly matches `cpio`, but these matches come from gnulib rather than cpio's own code.
- **C3 (Evolutionary Relationships):** Candidates share code ancestry through forks, platform variants, or project families. For example, OpenSSL has forked into BoringSSL and LibreSSL; the COIN-OR suite contains related modules (`coin-cgl`, `coin-cbc`) that share infrastructure code.

Group 2: Coincidental overlap involves no actual relationship; features happen to match by coincidence. We identify two such patterns:

- **C4 (Generic Features):** Matches consist only of ubiquitous elements such as ELF section names (`.text`, `.data`), libc functions (`malloc`, `memcpy`), or common strings ("error", "default").
- **C5 (Similar Functionality):** Independent libraries implementing the same standard or protocol have overlapping features without shared code. For example, OpenSSL and mbedTLS both implement TLS, leading to similar API names and shared protocol constants.

These categories suggest that false positives are not random noise but follow predictable patterns, making them amenable to targeted identification and filtering. Combined with Observation 1, which showed that correct libraries are often retrievable but lost to threshold filtering, these findings motivated our two-stage approach: ❶ retrieve candidates broadly to address false negatives, then ❷ apply targeted verification strategies to filter false positives from each category.

3 Approach

3.1 Overview

Based on the observations in Section 2.2, we design *BLADE* with a two-stage architecture: broad candidate retrieval followed by evidence-based verification. Figure 4 presents the overall workflow. Given a target binary, *BLADE* outputs the detected libraries along with supporting evidence. **The first stage** (Section 3.2) retrieves candidate libraries through feature matching, prioritizing recall. We extract string literals and function symbols from the binary and match them against a pre-built library feature database, selecting the top-n most similar libraries as candidates. During matching, we exclude imported symbols (mitigating C1: dynamic linking) and overly common features (mitigating C4: generic features) to reduce false positives at this early stage. **The second stage** (Section 3.3) verifies candidates and filters false positives across all five categories. We first

collect evidence from four sources: identity markers, library metadata, feature distribution patterns, and candidate overlaps. An LLM then analyzes this evidence through a two-step verification process: (1) directly confirming candidates with clear identity markers, and (2) checking each candidate against all five false positive patterns and filtering those that match. Candidates passing all checks are confirmed as final results.

3.2 Candidate Retrieval

This stage retrieves candidate libraries by matching features extracted from the target binary against a pre-built library database. Prior approaches set empirical thresholds to balance precision and recall, but this trade-off inevitably misses true positives while still admitting false positives. We instead prioritize recall at this stage and delegate precision to the verification stage, which can analyze relationships between libraries (e.g., forks, shared dependencies) that similarity scores cannot capture. Meanwhile, we mitigate false positives from C1 (imported symbols) and C4 (generic features) at this stage by filtering features before matching.

3.2.1 Feature Selection and Extraction. We select string literals and function names as matching features. These features are stable across different compilation configurations, remain available in stripped binaries, and have been widely adopted by prior tools [10, 12, 28, 43].

Library-side. We download open-source library repositories and use `tree-sitter` [31], a code parsing tool, to parse C/C++ source files and extract string literals and function names. For each repository, we iterate through version tags to extract features from multiple versions. The extracted data are stored in a PostgreSQL database with inverted index tables for efficient querying. Additionally, we compute feature frequency statistics across the database, recording how many libraries contain each feature. This frequency information enables filtering of overly common features during matching (mitigating C4) and provides evidence for identifying generic matches during verification. This extraction is performed offline as a preprocessing step.

Binary-side. For a target binary, we use the `strings` utility to extract printable string literals. We use `lief` [22], a binary parsing library, to parse the binary and extract symbol tables, distinguishing exported symbols (functions defined in the binary) from imported symbols (external references). For C++ binaries, symbol names are mangled by the compiler; we apply demangling using `c++filt` to recover the original function names for matching.

3.2.2 Feature Matching. The matching consists of three steps: (1) filter features before matching, (2) perform exact matching against the library database, and (3) rank and select top-n candidates.

Feature filtering. We exclude two types of features before matching. First, we exclude imported symbols from symbol matching. Imported symbols are references to external functions loaded at runtime from shared libraries, not functions compiled into the binary. Matching on imported symbols would incorrectly suggest that the referenced library is compiled into the binary when it is actually dynamically linked (C1). Second, we exclude common features that appear in more than N libraries (e.g., N=100), such as the string "error" or the function name `init()`. Matches on such features reflect coincidental overlap rather than systematic code sharing (C4).

Matching. We perform exact matching between the filtered binary features and the library feature database. We run string matching and symbol matching separately: binary strings are matched against library strings, and exported symbols are matched against library function names. Each candidate must match at least K features (e.g., K=4) to be considered; this threshold ensures that matches reflect meaningful code sharing rather than coincidental overlap.

Ranking and selection. For each matching type (string and symbol), candidates are ranked by the number of matched features. If two candidates have the same count, we prefer the one where matched features make up a larger portion of its total features. After ranking both lists, we merge

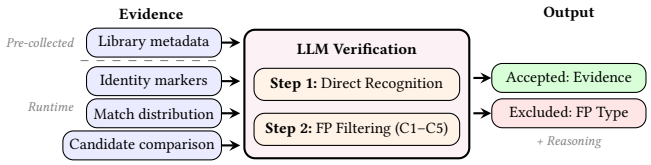


Fig. 5. LLM-based verification with evidence collection and two-step verification.

them using round-robin (alternately picking from each list and skipping duplicates). The threshold n is set high (e.g., $n = 10$) to prioritize recall: including extra candidates incurs only modest cost in verification, while missing a true positive is unrecoverable.

3.3 Candidate Verification

This stage identifies and excludes false positives among the candidates selected in the retrieval stage, particularly the five patterns identified in Section 2. Feature matching alone cannot distinguish the remaining candidates: all have matched features, and similarity scores cannot tell us whether two libraries are forks or whether matches concentrated in a subdirectory indicate vendored code. While LLMs possess such analysis capabilities, directly asking an LLM whether a binary uses a specific library would force it to rely on pre-trained knowledge and guesswork. To make verification reliable, we collect concrete evidence from multiple sources and guide the LLM through structured checks against the five false positive categories.

As shown in Figure 5, the verification stage consists of three parts. First, we collect evidence from four sources (Section 3.3.1): identity markers, library metadata, match distribution, and candidate comparison, providing the foundation for subsequent analysis. Second, we design a two-step verification logic (Section 3.3.2): (1) directly confirming candidates via identity markers, and (2) checking each candidate against all five false positive categories and excluding those that match. Finally, we encode the verification logic into a structured LLM prompt for execution (Section 3.3.3).

3.3.1 Evidence Collection. While the retrieval stage uses features (strings and symbols) simply for matching and counting, the verification stage requires evidence, which is interpretable information that supports decision-making. To provide the concrete evidence needed for verification, we collect evidence from four complementary sources: ❶ *Library metadata* describes each candidate's identity, functionality, and relationships such as known forks, helping distinguish evolutionary variants (C3). ❷ *Identity markers* extracted from the binary reveal library presence through version strings, copyright notices, or characteristic function names, enabling direct confirmation. ❸ *Match distribution* measures how matched features distribute across each library's source tree, detecting concentrated matches that indicate shared dependencies (C2). ❹ *Candidate comparison* computes feature overlap and unique markers between candidates, distinguishing functionally similar libraries (C3, C5). Library metadata is pre-collected offline during database construction; the other three are computed at runtime for each binary.

Pre-collected evidence. **(1) Library metadata** describes each candidate library's identity and functionality (e.g., distinguishing OpenSSL from its fork BoringSSL). It includes project information (name, vendor, description, license), source directory structure, and functional summaries (known forks, related components, declared dependencies), as summarized in Table 1 (left). We collect this metadata from multiple sources: the GitHub API provides project information; `tree-sitter` parses directory structures; and an LLM extracts functional descriptions from READMEs and build configurations. The LLM summaries are grounded in actual repository content. This metadata is pre-collected offline during database construction.

Table 1. Library metadata by extraction tool (left) and identity markers by compilation stage (right).

Tool	Info	Example
GitHub API	Name, vendor	`openssl/openssl`
	Description	"TLS/SSL toolkit..."
	License	`Apache-2.0`
`tree-sitter`	Directories	`src/`, `crypto/`
LLM	Summary	"cryptography library for SSL/TLS..."
	Dependencies	`zlib`, `perl`
	Forks	BoringSSL, LibreSSL
	Components	libssl, libcrypto

Stage	Type	Example
Source	Log messages	`"TLS handshake failed"`
	Function names	`SSL_CTX_new`
Build	Copyright	`"Copyright OpenSSL"`
	Version strings	`"OpenSSL 1.1.1k"`
Compiler	Debug paths	`/ssl/ssl_lib.c`
	Namespaces	`google::protobuf`
Linker	Exported symbols	`EVP_EncryptInit`
Package	File metadata	`libssl.so.1.1`

```
tar (True Positive)
Distributed: 692 matches
|-src/       280 (40%)
|-lib/       195 (28%)
|-gnu/       120 (17%)
|-rmt/        62 (9%)
...
```

```
cpio (False Positive)
Concentrated: 156 matches
|-gnulib/    148 (95%)
|-src/         4 (3%)
|-lib/         2 (1%)
|-headers/     1 (1%)
...
```

Fig. 6. Match distribution for a `tar` binary across two candidates.

Runtime evidence. **(2) Identity markers** are human-readable text embedded in binaries that reveal library identity. This evidence can directly confirm a candidate's presence through: ❶ *explicit markers* such as version strings, copyright notices, and license text, which developers preserve for legal compliance or version tracking; and ❷ *implicit markers* where library names appear in function prefixes or log messages, which developers embed for namespace isolation or debugging. We extract raw strings using the `strings` utility, then classify them into semantic categories using regular expressions (e.g., `Copyright.*\d{4}` → copyright notices), apply deduplication and noise filtering, and aggregate similar items into summaries (e.g., 300 functions with `SSL_*` prefix → "300 `SSL_*` functions"). Table 1 (right) lists evidence types by their origin in the compilation pipeline. Aggregation preserves semantic content while reducing token count for efficient LLM processing. Additionally, we extract the `DT_NEEDED` entries (the ELF field that records dynamic library dependencies) from the binary; this enables detection of dynamically linked libraries (C1).

(3) Match distribution measures how matched features are spatially distributed across a candidate library's source tree. This evidence reveals whether matches come from core library code or vendored dependencies: a skewed distribution (e.g., >90% in one directory) signals shared dependency artifacts rather than genuine library inclusion. For each candidate L with matched features F_L, we retrieve each feature's file location from the database and compute the proportion in each top-level directory d:

$$\text{dist}(L, d) = \frac{|\{f \in F_L : d(f) = d\}|}{|F_L|} \tag{1}$$

Figure 6 illustrates this with a `tar` binary: matches for the true positive `tar` distribute across directories (e.g., `src/`: 40%, `lib/`: 28%), while matches for the false positive `cpio` concentrate in one vendored directory (e.g., `gnulib/`: 95%).

(4) Candidate comparison analyzes relationships between candidate libraries to identify competing alternatives. We compute two metrics: ❶ *Feature overlap* quantifies shared features between candidates. For candidates L_i and L_j with matched feature sets F_i and F_j, we compute:

$$\text{overlap}(L_i, L_j) = \frac{|F_i \cap F_j|}{\min(|F_i|, |F_j|)} \tag{2}$$

High overlap (e.g., >50%) signals that only one is likely present. ❷ *Unique features* extracts distinguishing markers for each candidate: $U_i = F_i \setminus \bigcup_{j \neq i} F_j$. For example, OpenSSL and BoringSSL may share 70% of features due to their fork relationship, but each retains unique markers (e.g., "OpenSSL 1.1.1k" vs `BORINGSSL_*` symbols) that enable disambiguation.

3.3.2 Category-Guided Verification. After collecting evidence, we verify each candidate through two steps: Step 1 confirms candidates via identity markers; Step 2 checks the remaining candidates against all five false positive categories, excluding those that match and accepting the rest. **Step 1: Direct recognition.** This step confirms candidates that have clear identity markers in the binary. Specifically: ❶ we use regular expressions to search for each candidate's name in the identity markers extracted in Section 3.3.1; ❷ for each match, the LLM determines whether it constitutes a genuine identity marker (e.g., version string "OpenSSL 1.1.1k", copyright notice, or function prefix `ikcp_*` for kcp) or a coincidental occurrence. Confirmed candidates are accepted immediately; others proceed to Step 2.

Step 2: False positive filtering. Based on the four types of evidence collected, we apply different strategies for the five false positive categories. C1, C2, and C4 can be judged independently for each candidate; C3 and C5 require comparative analysis among candidates with overlapping features. Below we describe each category.

• **C1 (Dynamic Linking).** Binary files contain dynamic linking information (ELF's `DT_NEEDED`, PE's Import Table, Mach-O's `LC_LOAD_DYLIB`) that records external dynamic libraries required at runtime, providing reliable evidence for identifying dynamic linking dependencies. When building our library database, we also extract the shared library filenames that each library may produce (e.g., OpenSSL corresponds to `libssl.so` and `libcrypto.so`). Therefore, when a candidate's corresponding dynamic library file appears in this list, it indicates an external dependency rather than embedded code, and should be excluded.

• **C2 (Shared Dependencies).** The C/C++ ecosystem lacks a unified package manager, so developers typically introduce third-party dependencies by copying source code. However, for engineering management purposes, such external code is often placed in designated directories with recognizable naming patterns, commonly `third_party/`, `vendor/`, `external/`, or directories named after the dependency library (e.g., `gnulib/`, `zlib/`). Based on this pattern, we analyze the directory distribution of matched features within each candidate's source code: if matches concentrate in such directories, it indicates the binary contains the candidate's third-party dependency code rather than the candidate itself, and the candidate should be excluded.

• **C3 (Evolutionary Relationships).** Forks and project families are common in the open-source ecosystem. For example, OpenSSL has spawned BoringSSL and LibreSSL, and the COIN-OR project contains multiple subprojects sharing infrastructure code (e.g., coin-cbc, coin-cgl). Some relationships are documented in project READMEs, descriptions, or repository metadata, which we extract when building the database. When multiple candidates with evolutionary relationships appear together, we compare the occurrence count of each candidate's unique features in the binary. Here, unique features refer to features that are exclusive to a candidate and not shared with related libraries. For example, OpenSSL's version string "OpenSSL 1.1.1k" does not appear in BoringSSL, while BoringSSL's distinctive `BORINGSSL_*` symbols do not appear in OpenSSL. We keep the candidate with significantly more matching unique features; if a candidate was already confirmed in Step 1 via identity markers, its related candidates are excluded directly; if unique feature counts are similar, we conservatively keep both to avoid incorrect exclusion.

• **C4 (Generic Features).** Features vary greatly in their discriminative power. Some features appear in nearly all libraries, such as common strings ("error", "default"), ELF section names (`.text`, `.data`), and common libc function names (`malloc`, `free`); matching such features provides no

System Prompt

❶ Task Description
"Given candidates and evidence, determine true positives..."

❷ Background Knowledge ← *§2.2*
FP categories C1–C5: causes, description, examples. etc.

❸ Evidence Definitions ← *§3.3.1*
• Identity markers • Library metadata • Match distribution • Candidate comparison

❹ Verification Guidelines ← *§3.3.2*
• Step 1: Direct recognition • Step 2: FP filtering (C1–C5)

❺ Output Schema
`{decision, evidence_used, reasoning}` for each candidate

❻ Few-shot Examples
Representative cases covering C1-C5, and unknown programs

Fig. 7. System prompt structure for LLM-based verification.

meaningful signal. Other features have clear library attribution, such as `uuid_generate` for libuuid or `SSL_connect` for OpenSSL; matching such features reliably indicates the library's presence. We record the occurrence frequency of each feature in our database (how many libraries contain it). The retrieval stage already filters out extremely high-frequency features (appearing in more than 100 libraries), but moderately frequent generic features may still pass through. During verification, the LLM combines frequency statistics with the actual content of features to make judgments: if a candidate's matched features are all generic and lack other supporting evidence (such as identity markers), it is excluded; otherwise, the candidate is kept.

• **C5 (Similar Functionality).** Some libraries, though completely independent in code, have similar features because they implement the same standard or protocol. For example, OpenSSL and mbedTLS both implement the TLS protocol, thus sharing protocol-related API names (e.g., `SSL_read`, `SSL_write`) and constants; multiple TTML subtitle libraries contain URLs defined in W3C standards. Such candidates lack the queryable evolutionary relationships of C3; their feature overlap stems from the standard definitions themselves. After the LLM identifies this pattern based on feature content, it applies the same analysis logic as C3: comparing unique features of each candidate to determine which library's implementation is actually present in the binary.

3.3.3 LLM-Based Verification. The verification logic described in the previous section is executed by an LLM. All evidence from Section 3.3.1 is collected programmatically before LLM invocation; the LLM's role is purely analytical, applying the two-step verification process to the provided evidence. This separation ensures the LLM operates on grounded, verifiable information rather than generating facts from pre-trained knowledge. We process all candidates for a binary in a single invocation, allowing the LLM to reason about relationships between candidates (e.g., once OpenSSL is confirmed, the LLM can immediately exclude its fork BoringSSL).

Prompt design. Our prompt design follows a principle: systematically encoding the methodology presented in this paper into structured instructions. The prompt consists of a *system prompt* containing fixed instructions and a *user prompt* containing per-binary evidence.

The *system prompt* (Figure 7) directly maps to the paper's sections: ❶ *Task Description* specifies the verification objective: given candidate libraries and evidence, determine which are true positives. ❷ *Background Knowledge* explains the five false positive categories C1–C5 (Section 2.2), enabling the LLM to understand the problem structure. For each category, the prompt provides detailed guidance: definition (what it is), cause (why it produces matches), detection method (how to identify

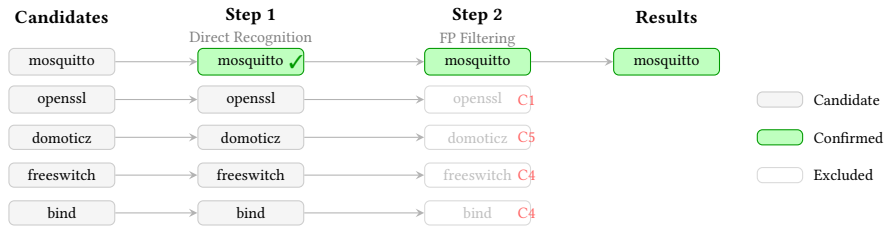


Fig. 8. Verification example for a binary containing `mosquitto`.

***BLADE* Output**

Library: eclipse/mosquitto *(MQTT broker and client library)*

Evidence:
- Feature matches: 169 strings, 107 symbols matched with `mosquitto`
- Identity marker: "Mosquitto is free software"
- Identity symbols: 107 exported `mosquitto_*` functions

Analysis:
This binary contains the mosquitto library. The retrieval stage matched 169 strings and 107 symbols with mosquitto, indicating feature similarity. The identity marker "Mosquitto is free software" and 107 exported `mosquitto_*` *symbols confirm the library's presence.*

Fig. 9. Example of *BLADE*'s structured output with evidence and analysis trace.

it), positive and negative examples, and key decision rules. ❸ *Evidence Definitions* describe the four evidence types and their formats (Section 3.3.1), explaining what information each provides and how to interpret it. ❹ *Verification Guidelines* detail the two-step process (Section 3.3.2): direct recognition via identity markers, followed by systematic checks against all five false positive categories. ❺ *Output Schema* defines the expected JSON format for results. ❻ *Few-shot Examples* demonstrate the expected analysis process with representative cases covering shared dependencies (C2), evolutionary relationships (C3), generic features (C4), and unknown programs.

The *user prompt* contains only the current binary's four types of evidence in structured format. Placing fixed content in the system prompt and variable content in the user prompt not only ensures instruction consistency but also enables LLM prompt caching to reduce inference costs.

Output format. For each candidate, the LLM outputs: a decision (accept or exclude), supporting evidence or the applicable false positive category, and an analysis summary explaining the decision.

Verification example. Figure 8 illustrates the complete verification process. Consider analyzing a binary that contains the `mosquitto` library, where the retrieval stage returns five candidates: `mosquitto`, `openssl`, `domoticz`, `freeswitch`, and `bind`. In Step 1, we find exported symbols with the `mosquitto_` prefix (e.g., `mosquitto_connect`, `mosquitto_publish`) and identity markers such as "Mosquitto is free software", which directly confirm `mosquitto`. In Step 2, we check each remaining candidate: `openssl` is excluded because it appears in the binary's `DT_NEEDED` section, indicating dynamic linking (C1); `domoticz` is excluded because its matches consist of generic MQTT protocol terms shared with the confirmed `mosquitto` library (C5); `freeswitch` and `bind` are excluded because their matches consist only of generic features like libc function names (C4).

Output example. Figure 9 shows example output for the `mosquitto` detection above. The output includes the identified library, supporting evidence, and analysis trace. We also applied additional engineering optimizations such as extracting version numbers from identified version strings and optimizing output format for readability, which we omit here for brevity.

Table 2. Dataset distribution across compilation configurations. TPLs column shows unique libraries per configuration; Total shows unique count after deduplication.

Type	Configuration	Binaries	TPLs	Reuse Rel.
Main	gcc+x86_64	1,210	986	1,338
Variation	clang+x86_64	1,182	971	1,312
	gcc+arm	1,011	866	1,121
Total		**3,403**	**1,016**	**3,771**

4 Evaluation

We evaluate *BLADE* by addressing the following research questions:

- **RQ1 (Effectiveness):** How effective is *BLADE* compared to existing methods?
- **RQ2 (Ablation):** What is the contribution of each component?
- **RQ3 (Generalization):** Does *BLADE* generalize to unseen, proprietary binaries?
- **RQ4 (Efficiency):** How efficient and cost-effective is *BLADE*?

4.1 Setup

4.1.1 Baselines. To evaluate *BLADE*'s effectiveness, we select baselines from both academic literature and commercial practice, considering state-of-the-art methods, methodological diversity, and industry adoption. We include four academic tools: **BAT** [12] (string-based, reimplemented based on the paper), **OSSPolice** [10] (feature weighting, reproduced from public code), **B2SFinder** [43] (multi-feature fusion, evaluated by the original authors on our dataset), and **BinaryAI** [14] (SOTA, neural embedding-based, evaluated via official API). We also obtained evaluation access to two commercial tools **CT1** and **CT2** (anonymized per vendor request). We also attempted to include LibAM [17] and ModX [40]; however, LibAM exhausted memory on our largest available server (256GB RAM) during database construction, and according to [17], it requires over 1TB RAM. ModX's implementation is not publicly released.

4.1.2 Dataset. Evaluating binary TPL detection requires a benchmark dataset with ground-truth labels. However, existing baselines either do not release their datasets [10, 12, 14, 43] or lack publicly available ground-truth labels [17]. Therefore, we constructed a new benchmark dataset. Our dataset contains **3,403 binary files** compiled from 1,085 C/C++ projects collected from Conan and vcpkg, covering **1,016 TPLs** and **3,771 reuse relationships** (binary-library pairs). We chose Conan and vcpkg because they are the most widely-used package managers in the C/C++ ecosystem, containing popular open-source libraries with complete dependency metadata and build configurations. We prioritized projects with more than three TPL dependencies to ensure complexity and representativeness. According to package manager metadata, these projects span 1,931 topic labels including image processing, audio processing, and document compression, ensuring diversity.

Since real-world binaries may originate from different compilers and architectures, we compiled each project under three settings to evaluate robustness: `gcc+x86_64` (baseline), `clang+x86_64` (compiler variation), and `gcc+arm` (architecture variation). All binaries were compiled with Release-level optimization and stripped to simulate real-world deployment. As some projects failed to compile under certain configurations, the number of binaries varies slightly across settings (Table 2). After compilation, we extracted binaries from the `bin` and `lib` directories and removed duplicates.

Ground truth was derived primarily from package manager dependency metadata, following established methodologies [10, 14, 43]. Conan and vcpkg provide explicit dependency declarations for each project, which we used as the primary source. We cross-validated this metadata against build scripts, SBOM files (e.g., `CMakeLists.txt`), and license information. A reuse relationship is a (binary, library) pair indicating that the binary contains code compiled from that library; since a single

binary may include multiple TPLs, the number of reuse relationships exceeds the number of binaries. Of the total 3,771 reuse relationships, 93.53% (3,527) were unambiguous based on package manager metadata and build scripts, requiring no manual review. Two authors independently reviewed the remaining 244 ambiguous cases (6.47%), which primarily involve binaries containing multiple libraries. The initial agreement rate on these cases was 78.69% (192/244). The 52 disagreements primarily involve two patterns: (1) a main library with embedded header-only dependencies, and (2) sub-projects from the same ecosystem merged during the build process. All disagreements were resolved through cross-validation against source code and build scripts.

4.1.3 TPL Corpus. Since baseline tools and *BLADE* both require pre-built TPL databases, we constructed a TPL corpus for evaluation. We curated this corpus to reflect practical SCA scenarios, where the primary goal is identifying libraries with potential security vulnerabilities. Following prior work [14, 34], we started with popular GitHub projects (over 1,000 stars); however, many high-star repositories are end products (e.g., games, operating systems) or learning projects rather than reusable libraries, so we filtered these out. We then supplemented with libraries having known CVEs from NVD, as these are critical targets for security-focused detection, and ensured coverage of all TPLs appearing in our test dataset. After deduplication, our corpus contains 3,927 libraries, covering widely-used libraries and those with security relevance. We built the database for *BLADE* following Section 3, and for baselines following their respective methods.

4.1.4 Implementation. We implemented *BLADE* in Python. Experiments were conducted on a workstation with Intel Core i9-13900K CPU and 32GB RAM. For the candidate retrieval stage (Section 3.2), we set the common feature threshold N=100, minimum distinctive matches K=4, and number of candidates n=10. We set N=100 to filter obviously common features while avoiding over-filtering, as widely-reused libraries like zlib and OpenSSL have features appearing in many projects. We set K=4 based on empirical testing to balance reducing noise while maintaining low matching requirements. Top-10 candidates achieve near-maximum recall while keeping verification cost manageable. For library metadata extraction (Section 3.3.1), we used Claude Code [3] with Claude Opus 4.5 [4] and web search enabled to extract and summarize the codebase to generate library-side metadata; this was performed offline during database construction. For candidate verification, we evaluate four LLMs: GPT-5 (primary), GPT-5-mini (cost-reduced), GPT-4.1 (prior generation), and gpt-oss-120b [20] (open-weight), covering different capability levels, model generations, open-source availability, and reasoning support. We access gpt-oss-120b via OpenRouter [21] and the others via OpenAI's official API. API temperature is fixed at 0 and random seed at 42.

4.1.5 Evaluation Metrics. We evaluate at the *reuse relationship* level: each (binary, library) pair in the detection output is compared against the ground truth. A detected library is a True Positive (TP) if it matches a library in the ground truth, a False Positive (FP) if it does not appear in the ground truth, and a False Negative (FN) if a ground-truth library is missing from the output. We compare based on library names only, ignoring version differences, and treat minor name variations (e.g., "OpenSSL" vs "openssl") as equivalent. We report Precision ($P = \mathrm{TP}/(\mathrm{TP} + \mathrm{FP})$), Recall ($R = \mathrm{TP}/(\mathrm{TP} + \mathrm{FN})$), and F1-score ($F1 = 2PR/(P + R)$) computed from aggregated counts across all test cases.

4.2 RQ1: Effectiveness

We evaluate all tools using the setup described above; Table 3 presents the results. In terms of *overall effectiveness*, *BLADE* with GPT-5 achieves **93.74% recall**, **97.60% precision**, and **95.63% F1-score**, improving F1 by 41.83 points over BAT (the best baseline at 53.80%). Specifically, *BLADE* improves recall by 36.09 points over BinaryAI (the highest-recall baseline at 57.65%) and precision

Table 3. Detection performance comparison across compilation configurations.

	Overall			gcc+x86_64			gcc+arm			clang+x86_64		
Tool	R	P	F1	R	P	F1	R	P	F1	R	P	F1
CT1	32.46	49.20	39.11	32.63	48.28	38.94	31.70	50.14	38.84	32.90	49.31	39.47
CT2	31.40	79.04	44.94	31.51	79.73	45.17	30.45	79.53	44.04	32.14	77.96	45.52
BAT	42.67	72.77	53.80	42.74	72.01	53.64	41.50	71.36	52.48	43.66	74.87	55.16
OSSPolice	22.89	28.54	25.40	21.78	26.58	23.94	23.69	31.52	27.05	23.36	28.20	25.55
B2SFinder	19.92	47.90	28.14	20.21	47.70	28.39	19.77	47.74	27.96	19.77	48.23	28.04
BinaryAI	57.65	46.19	51.29	57.41	46.40	51.32	56.81	45.22	50.36	58.63	46.80	52.05
***BLADE*-G5**	**93.74**	**97.60**	**95.63**	**93.65**	**97.66**	**95.61**	**93.67**	**97.58**	**95.59**	**93.90**	**97.55**	**95.69**
BLADE-G5m	91.62	92.65	92.13	91.41	91.75	91.58	91.88	92.38	92.13	91.62	93.83	92.71
BLADE-G4.1	90.77	96.34	93.47	91.26	96.83	93.96	90.37	96.94	93.54	90.62	95.35	92.92
BLADE-OSS	91.00	91.94	91.47	91.27	91.75	91.51	90.75	91.66	91.20	90.95	92.39	91.66

R = Recall (%), P = Precision (%), F1 = F1-score (%). G5 = GPT-5, G5m = GPT-5-mini, G4.1 = GPT-4.1, OSS = gpt-oss-120b.

by 18.56 points over CT2 (the highest-precision baseline at 79.04%). In terms of *robustness*, the results are consistent across all three compilation configurations (recall: 93.65%–93.90%, precision: 97.55%–97.66%), demonstrating stability under compiler and architecture variations. In terms of *LLM generalization*, GPT-5-mini achieves 91.62% recall and 92.65% precision, GPT-4.1 (a previous-generation model) achieves 90.77% recall and 96.34% precision, and gpt-oss-120b (an open-weight model) achieves 91.00% recall and 91.94% precision. These results demonstrate that *BLADE*'s effectiveness is not dependent on a specific high-end model, remains stable across model generations, and supports open-source models for local deployment. Notably, both reasoning models (GPT-5, GPT-5-mini, gpt-oss-120b) and the instruct model (GPT-4.1) achieve strong performance, indicating that *BLADE* is not restricted to reasoning models. Beyond overall accuracy, we also examined whether the LLM genuinely follows our verification logic rather than making arbitrary decisions. We analyzed the exclusion reasons across all 24,153 candidates: the LLM excluded 20,531 (85%) using all five categories, with C4 (generic features) accounting for 46.80%, C2 (shared dependencies) 23.66%, C5 (similar functionality) 21.15%, C3 (evolutionary relationships) 5.84%, and C1 (dynamic linking) 2.55%. This distribution shows that the LLM actively utilizes all five categories, with C4 being the most frequent, which aligns with the prevalence of generic features in binaries.

Comparison with baselines. We analyzed error cases from each tool to understand the root causes of the performance gap. For false negatives, threshold-based methods (BAT, OSSPolice, B2SFinder) tend to miss libraries when matched features are limited, reflecting the inherent precision-recall trade-off in similarity-based approaches. BinaryAI's false negatives may partly stem from database coverage differences, although we selected popular libraries to minimize this factor. For false positives, even though BinaryAI uses function semantic vectors and computes distances in vector space, it fundamentally still relies on similarity comparison and exhibits systematic errors across all five categories. For example, when analyzing the binary of `libaom` (an AV1 codec library), BinaryAI reported `ffmpeg` (an upstream framework that depends on libaom, C2), `libvpx` (the VP9 codec with shared architectural heritage, C3), and `dav1d` (a competing implementation of the AV1 standard, C5) as false positives. *BLADE* correctly identifies only `libaom`: identity markers such as version strings ("AOMedia Project AV1 Decoder v3.8.0") and 72 exported `aom_*` symbols confirm its presence, while the other candidates are excluded in the false positive filtering step. These cases demonstrate that similarity-based methods fundamentally cannot distinguish the five false positive categories. *BLADE* addresses these limitations: relaxed retrieval (requiring only 4 matched features and selecting top-10 candidates) reduces false negatives, while evidence-based verification identifies and excludes C1–C5 false positives.

False negative analysis. *BLADE* produces 236 false negatives. We identified three primary sources, accounting for 222 cases (94%). First, 182 cases are not retrieved because these libraries leave

few matchable features after compilation (e.g., small utilities with limited code). This is a common limitation for feature-based detection approaches. Second, 25 cases enter the candidate set but are incorrectly excluded: these libraries match some features but lack identity markers (e.g., version strings, copyright notices), and their matched features overlap with other candidates without unique distinguishing features, causing the verifier to classify them as coincidental matches. Third, 15 cases involve interface-compatible or aliased libraries (C3/C5 categories): when multiple libraries provide alternative functionality, *BLADE* selects one but chooses incorrectly. This ambiguity also causes some false positives.

False positive analysis. *BLADE* produces 87 false positives. We identified two primary sources, accounting for 81 cases (93%). First, 46 cases involve C3/C5 category library ambiguity: some result from incorrect selection among functionally equivalent or naming-variant libraries (corresponding to the third category of false negatives above); others stem from our design choice to report all candidates when their features cannot be clearly distinguished, preferring false positives over false negatives to avoid missing potentially vulnerable libraries (see Section 3.3). This is reasonable for downstream applications targeting vulnerability detection. Second, 35 cases involve C2 transitive dependency misjudgment: a library does not enter the candidate set, but another library that depends on it does, and the LLM cannot determine whether the matched features originate from the main library itself or from its dependencies, incorrectly reporting the dependency. This commonly occurs when developers copy dependency code into their projects rather than using package managers. For example, `fmtlog` embeds the `tscns` implementation in its header file with only a single-line comment [23], causing *BLADE* to incorrectly report `tscns` when detecting `fmtlog`.

Answer to RQ1: (1) *BLADE* achieves 93.74% recall and 97.60% precision, outperforming all baselines across all metrics. (2) Performance remains stable across compilers and architectures. (3) The approach generalizes across models with varying capability, reasoning type, and open-source availability (all achieving >91% F1).

4.3 RQ2: Ablation Study

BLADE introduces three key design choices over traditional similarity-based methods: ❶ a *two-stage architecture* that separates candidate retrieval from verification; ❷ *structured evidence* that collects identity markers, library metadata, and feature relationships; and ❸ *category-guided verification* that systematically analyzes candidates against the five false positive categories (C1–C5).

To evaluate each contribution, we design three corresponding ablation experiments: ❶ **w/o Verification** removes the verification stage entirely and outputs top-10 retrieval results directly, testing whether verification improves over retrieval alone; ❷ **w/o Structured Evidence** provides the LLM with only candidate names, matched feature counts, and examples (no identity markers, metadata, or feature relationships), testing whether structured evidence grounds LLM judgments; ❸ **w/o Category-Guided Verification** provides full evidence but uses a simplified prompt that retains only the task description and output format, excluding the background knowledge and verification guidelines from Figure 7, testing whether systematic reasoning improves accuracy.

Table 4 presents the results. ❶ Removing the verification stage makes the system rely entirely on feature-matching retrieval. As shown in Figure 10, as n increases from 1 to 100, recall improves from 67.36% to 96.66% and plateaus, while precision drops sharply from 75.08% to 1.82%. We choose n=10 as the balance point, achieving 94.14% recall (Table 4). However, precision is only 12.73%. After verification, precision improves to 92.65%, a gain of 79.92 percentage points. This demonstrates that verification is essential for filtering false positives. ❷ Removing structured evidence forces the LLM to rely only on candidate names and match counts. As shown in Table 4,

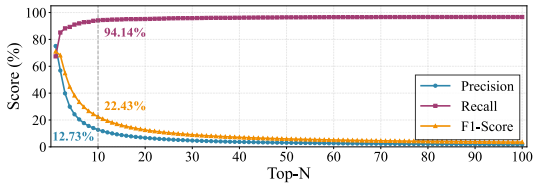


Fig. 10. Retrieval-only performance with varying n (number of candidates).

Table 4. Ablation study results (GPT-5-mini).

Configuration	R	P	F1
***BLADE*-G5m**	**91.62**	**92.65**	**92.13**
w/o Verification	94.14↑	12.73↓	22.43↓
w/o Structured Evidence	83.45↓	73.44↓	78.13↓
w/o Category-Guided Verification	92.12↑	81.63↓	86.56↓

Table 5. Generalization Results (GPT-5).

Tool	R	P	F1
BAT	34.93	42.15	38.20
OSSPolice	19.86	37.66	26.01
B2SFinder	19.86	54.72	29.14
BinaryAI	51.37	66.96	58.14
BLADE	**90.81**	**96.98**	**93.79**

both recall and precision drop significantly (R: 83.45%, P: 73.44%, F1: 78.13%), with F1 decreasing by 14.00 points. This demonstrates that structured evidence is essential for accurate identification. ❸ Removing category-guided verification causes recall to slightly increase (92.12%) but precision to drop by 11 points (81.63%), resulting in F1 decreasing by 5.57 points to 86.56%. The recall increase occurs because without systematic C1–C5 exclusion guidance, the LLM becomes more conservative and excludes fewer candidates, but this also retains more false positives. This demonstrates that category-guided verification primarily helps the LLM correctly exclude false positives.

Answer to RQ2: All three components contribute significantly: ❶ verification improves precision by 79.92 points over retrieval alone; ❷ structured evidence improves F1 by 14.00 points by grounding LLM judgments; ❸ category-guided verification improves F1 by 5.57 points through systematic analysis.

4.4 RQ3: Generalization

RQ1 and RQ2 are evaluated on open-source projects. To verify whether *BLADE* generalizes to unseen proprietary software, we obtained a real-world test case from a commercial in-vehicle platform's OTA (Over-The-Air) update package through our industry partner; after extraction, we obtained 277 binary files (average size: 1.8MB). This test case has never been publicly released; therefore, neither the binaries nor their TPL usage could appear in any LLM's training data. We established ground truth with the vendor's engineers based on their source code, identifying 283 TPL reuse relationships. Table 5 presents the results of *BLADE* and baseline tools. Due to data confidentiality requirements from our industry partner, commercial tools (CT1, CT2) were not permitted to analyze these proprietary binaries and are excluded from this comparison.

BLADE achieves 93.79% F1-score on this proprietary dataset, with 90.81% recall and 96.98% precision. Compared to the open-source benchmark, both metrics remain stable (F1 differs by only 1.84 percentage points), indicating that *BLADE* generalizes to unseen software ecosystems, with effectiveness stemming from evidence-based verification rather than memorization.

Table 6. Efficiency comparison (seconds per binary). (1) = single-thread; (100) = 100-thread.

CT1	CT2	BAT	OSSPolice	B2SFinder	BinaryAI	*BLADE* (1)	*BLADE* (100)
1.71	0.03	0.80	32.70	288.16	16.84	58.78	0.64

Answer to RQ3: *BLADE* achieves consistent performance (93.79% F1) on a proprietary system unseen during training, demonstrating cross-dataset stability and genuine reasoning rather than memorization.

4.5 RQ4: Efficiency

To evaluate *BLADE*'s efficiency, scalability, and cost, we recorded the analysis time for each binary and calculated average costs; Table 6 presents the results. In terms of *time*, *BLADE* averages 58.78 seconds per binary in single-threaded mode, with candidate retrieval taking only 0.18 seconds and the remaining time spent on verification. While this is slower than pure feature-matching methods such as BAT, it remains significantly faster than approaches that require decompilation (e.g., B2SFinder) or rely on reverse engineering tools such as IDA Pro. In terms of *scalability*, since the analysis bottleneck lies in waiting for API responses rather than local computation, *BLADE* can fully leverage parallelism: with 100-thread concurrency, wall-clock time drops to 0.64 seconds per binary (92× speedup), processing all 3,403 files in just 36 minutes. In terms of *cost*, using GPT-5 averages $0.0378 per binary; using GPT-5-mini reduces this to $0.0074 (approximately one-fifth of GPT-5), while still achieving 92.13% F1. Additionally, approximately 76% of input tokens benefit from prompt caching, further reducing actual costs. This cost is far lower than manual analysis, making automated evidence-based verification economically viable at scale.

In addition to the per-binary analysis cost above, *BLADE* requires a one-time offline preprocessing step for library metadata extraction (Section 3.3.1). For our corpus of 3,927 libraries, the average extraction time per library is approximately 83 seconds, and the average cost is approximately $0.34 per library. Repository size has minimal impact on cost; for example, zlib (6.3MB) costs $0.27 and OpenSSL (198MB) costs $0.31. The total preprocessing cost for the entire corpus is approximately $1,335. Using 5× concurrency, the total extraction time is approximately 18 hours. Since this is a one-time cost during database construction and the resulting metadata can be reused across all subsequent analyses, it does not affect per-binary analysis efficiency.

Answer to RQ4: *BLADE* is efficient and cost-effective. With 100-thread concurrency, it processes each binary in 0.64 seconds. Using GPT-5 costs $0.0378 per binary (95.63% F1); using GPT-5-mini reduces cost to $0.0074 while maintaining 92.13% F1.

5 Threats to Validity

Ground truth labeling. Our ground truth is based on manual annotation, which may contain omissions or errors. To mitigate this, we follow established methodologies from prior work [10, 14, 43] by cross-referencing multiple sources: package manager dependency metadata, build scripts, SBOM files, and license files. Two authors independently labeled the dataset, with disagreements resolved through discussion until consensus was reached.

LLM hallucination. LLMs may generate incorrect conclusions unrelated to the input evidence. Our design mitigates this risk through several mechanisms: First, the LLM does not directly identify TPLs; instead, it verifies candidates produced by the retrieval stage (based on string and symbol matching). Second, all reasoning must be grounded on the four types of structured evidence we collect programmatically (identity markers from binaries, library metadata, match distribution, and

candidate comparison), rather than relying on the LLM's pre-trained knowledge. Third, each result includes evidence and reasoning traces, ensuring traceability.

Data leakage. The LLM's pre-training data may contain information about open-source libraries. We examine this risk from both architectural design and empirical evaluation. Architecturally, the LLM does not access raw binary content and cannot propose candidate libraries on its own; it verifies candidates produced by the retrieval stage based on programmatically collected structured evidence. The ablation experiment (w/o Structured Evidence) shows that when evidence is removed and the LLM judges solely from candidate names and match counts, F1 drops by 14.00 points (Table 4), indicating that evidence is the primary basis for its judgments. Additionally, RQ3 evaluates on 277 proprietary binaries that have never been publicly released, and the results consistent with the open-source benchmark (F1 differs by only 1.84 percentage points) indicate that performance does not depend on memorization of specific software.

Reproducibility and LLM dependency. *BLADE* relies on LLM APIs for candidate verification, raising concerns about long-term reproducibility as commercial models may be updated or deprecated. We address this from three perspectives. First, *BLADE* is architecturally LLM-agnostic: it constrains the LLM to verify candidates over programmatically collected structured evidence, with no model-specific fine-tuning or prompt engineering. As shown in Table 3, four LLMs spanning different capability levels, model generations, and open-source availability all achieve >91% F1, confirming that performance is not tied to any single model. Second, we evaluate gpt-oss-120b [20], an open-weight model deployable locally without commercial API dependency, which achieves 91.47% F1. This ensures independent replication remains feasible even if specific commercial APIs change. Third, we open-source our replication package to enable researchers to reproduce and extend results with any compatible LLM.

Dataset representativeness. Our evaluation dataset may not represent all scenarios. To mitigate this, we collected 1,085 projects from the most widely-used C/C++ package managers (Conan and vcpkg), yielding 3,403 binaries covering 1,016 TPLs across 1,931 topic labels (e.g., image processing, audio processing, document compression) to ensure diversity. Additionally, we compiled binaries with different compilers (gcc/clang) and architectures (x86/ARM) in release mode with stripped symbols to simulate real-world deployment.

6 Related Work

Binary TPL Detection for C/C++. Existing approaches can be broadly divided into two categories based on their feature extraction sources: Binary-to-Binary (B2B) and Binary-to-Source (B2S) methods. B2B techniques, represented by works such as LibDX [28], LibDB [29], LibAM [17], and ModX [40], directly extract features from TPL binaries but suffer from poor scalability, as maintaining binary collections across versions, architectures, and compilation configurations is infeasible. To overcome these issues, research has shifted toward B2S approaches, which construct databases from source code and thus offer better scalability, but face the fundamental challenge that compilation and optimization significantly alter features. Within B2S approaches, there has been a clear evolution in feature sophistication: early B2S works such as BAT [12], OSSPolice [10], and B2SFinder [43] relied on strings and function names, which were often too sparse and led to missed detections; later studies incorporated structural features such as Control Flow Graphs, which enriched representations but remained limited by the effectiveness of graph similarity techniques; most recently, BinaryAI [14] has advanced toward semantic-based detection by comparing decompiled pseudo-code embeddings with source embeddings, reflecting a trend from string-based to structural to semantic feature matching. Other works have explored learning-based matching, including neural networks for order-aware instruction matching [41], cross-modal retrieval between binary and source [42], and fingerprint-based detection [34, 46]. Separately, version

identification [9, 13, 45] assumes TPL detection is complete and focuses on distinguishing library versions, which is a downstream task of our work. Despite these advances, all existing approaches remain within the feature matching paradigm. By contrast, our work reformulates TPL detection as evidence-based candidate verification, retrieving candidates broadly to ensure recall, then analyzing multi-source evidence to filter false positives through targeted verification.

Source Code TPL Detection for C/C++. Code clone detection is the foundational technique underlying source code TPL detection, with extensive research establishing various methods for identifying duplicated or similar code fragments [1, 5, 24, 25]. Building on these techniques, several approaches have been proposed to detect third-party library reuse at the source code level. CENTRIS [33] employs code clone mapping to identify modified open-source software reuse, matching code segments between a target project and known libraries to pinpoint reused components even when they have been partially modified. OSSFP [34] improves detection accuracy by selecting fingerprinting functions, which are functions distinctive to a particular library, to serve as reliable indicators of library presence. Tang et al. [30] take a lighter-weight approach, proposing a simple name-mapping strategy to identify library dependencies in the C/C++ ecosystem and conducting a large-scale empirical study on TPL usage patterns. However, these source-level techniques are inherently incompatible with binary TPL detection scenarios, where source code is unavailable and compilation fundamentally transforms code structure and control flow. The features they rely on, such as source-level code clones and syntactic patterns, are largely lost or obfuscated during compilation, making them inapplicable to binary analysis settings.

LLMs in Program Analysis. LLM applications in program analysis fall into two categories [37]: 1) LLMs as analysis components, integrating LLMs into frameworks for vulnerability detection [27], penetration testing [8], and malware analysis [11, 16, 18]; 2) domain adaptation, fine-tuning or prompting models for tasks like fault localization [38], program repair [36], binary authorship analysis [26], vulnerability detection [6, 32, 44], and fuzzing [35, 39]. These approaches predominantly use LLMs for pattern recognition (embedding generation, classification, similarity matching). In contrast, our work leverages LLMs for multi-step reasoning over structured evidence to verify candidate libraries, combining automated evidence collection with LLM analysis capabilities.

7 Conclusion

We presented *BLADE*, which reframes binary TPL detection as evidence-based candidate verification. Rather than relying on similarity scores, *BLADE* retrieves candidates broadly and verifies them through a two-stage process: confirming with identity markers, then filtering against five false positive patterns (C1–C5). Evaluation on the largest C/C++ binary TPL benchmark (3,403 binaries, 1,016 libraries) shows *BLADE* achieves 95.63% F1-score, improving over the best baseline by 41.83 points, with consistent results across compilers, architectures, and LLM backends. With 100-thread concurrency, *BLADE* processes each binary in 0.64 seconds at $0.0378.

8 Data Availability

Source code of *BLADE* and evaluation data is available at: https://github.com/OpenBinarySCA/blade.

Acknowledgments

This research is supported by the National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme, and by the National Research Foundation Singapore and the Cyber Security Agency under the National Cybersecurity R&D Programme (NCRP25-P04-TAICeN).

References

[1] Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. 2019. A systematic review on code clone detection. *IEEE access* 7 (2019), 86121–86144. doi:10.1109/access.2019.2918202

[2] Rahaf Alkhadra, Joud Abuzaid, Mariam AlShammari, and Nazeeruddin Mohammad. 2021. Solar winds hack: In-depth analysis and countermeasures. In *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. IEEE, 1–7. doi:10.1109/icccnt51525.2021.9579611

[3] Anthropic. 2025. Claude Code: An Agentic Coding Tool. https://claude.com/product/claude-code. Accessed: 2026-01-24.

[4] Anthropic. 2025. Claude Opus 4.5. https://www.anthropic.com/news/claude-opus-4-5. Accessed: 2026-01-24.

[5] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering* 33, 9 (2007), 577–591. doi:10.1109/tse.2007.70725

[6] Yiu Wai Chow, Max Schäfer, and Michael Pradel. 2023. Beware of the Unexpected: Bimodal Taint Analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 211–222. doi:10.1145/3597926.3598050

[7] European Commission. 2022. Cyber Resilience Act. https://ec.europa.eu/commission/presscorner/detail/en/ip_22_5373.

[8] Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. 2024. PentestGPT: Evaluating and Harnessing Large Language Models for Automated Penetration Testing. In *33rd USENIX Security Symposium (USENIX Security 24)*. 847–864. https://www.usenix.org/conference/usenixsecurity24/presentation/deng

[9] Chaopeng Dong, Siyuan Li, Shouguo Yang, Yang Xiao, Yongpan Wang, Hong Li, Zhi Li, and Limin Sun. 2024. Libvdiff: Library version difference guided oss version identification in binaries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12. doi:10.1145/3597503.3623336

[10] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. 2017. Identifying open-source license violation and 1-day security risk at large scale. In *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*. 2169–2185. doi:10.1145/3133956.3134048

[11] Mohamed Amine Ferrag, Ammar Battah, Norbert Tihanyi, Ridhi Jain, Diana Maimuț, Fatima Alwahedi, Thierry Lestable, Narinderjit Singh Thandi, Abdechakour Mechri, Merouane Debbah, and Lucas C. Cordeiro. 2025. SecureFalcon: Are We There Yet in Automated Software Vulnerability Detection With LLMs? *IEEE Transactions on Software Engineering* 51, 4 (2025), 1248–1265. doi:10.1109/TSE.2025.3548168

[12] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. 2011. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. 63–72. doi:10.1145/1985441.1985453

[13] Xulun Hu, Weidong Zhang, Hong Li, Yan Hu, Zhaoteng Yan, Xiyue Wang, and Limin Sun. 2020. VES: A Component Version Extracting System for Large-Scale IoT Firmwares. In *International Conference on Wireless Algorithms, Systems, and Applications*. Springer, 39–48. doi:10.1007/978-3-030-59019-2_5

[14] Ling Jiang, Junwen An, Huihui Huang, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2024. BinaryAI: Binary Software Composition Analysis via Intelligent Binary Source Code Matching. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639100

[15] Ling Jiang, Hengchen Yuan, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2023. Third-party library dependency for large-scale SCA in the C/C++ ecosystem: How far are we?. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1383–1395. doi:10.1145/3597926.3598143

[16] Robert J. Joyce, Tirth Patel, Charles Nicholas, and Edward Raff. 2023. AVScan2Vec: Feature Learning on Antivirus Scan Data for Production-Scale Malware Corpora. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec)*. doi:10.1145/3605764.3623907

[17] Siyuan Li, Yongpan Wang, Chaopeng Dong, Shouguo Yang, Hong Li, Hao Sun, Zhe Lang, Zuxin Chen, Weijie Wang, Hongsong Zhu, et al. 2023. Libam: An area matching framework for detecting third-party libraries in binaries. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–35. doi:10.1145/3625294

[18] Ruitong Liu, Yanbin Wang, Haitao Xu, Zhan Qin, Fan Zhang, Yiwei Liu, and Zheng Cao. 2025. PMANet: Malicious URL detection via post-trained language model guided multi-level feature attention network. *Information Fusion* 113 (2025), 102638. doi:10.1016/j.inffus.2024.102638

[19] Office of Management and Budget. 2022. Enhancing the Security of the Software Supply Chain through Secure Software Development Practices. https://www.whitehouse.gov/wp-content/uploads/2022/09/M-22-18.pdf. Memorandum M-22-18.

[20] OpenAI. 2025. Introducing gpt-oss. https://openai.com/index/introducing-gpt-oss/.

[21] OpenRouter. 2025. OpenRouter: A unified interface for LLMs. https://openrouter.ai/. Accessed: 2025-01-26.

[22] Quarkslab. 2024. LIEF: Library to Instrument Executable Formats. https://lief.re/. Version 0.16.6.

[23] Meng Rao. 2024. fmtlog: A performant fmtlib-style logging library with latency in nanoseconds. https://github.com/MengRao/fmtlog/blob/308b2317d4e9c21d3b203d104007fde127dcdbfa/fmtlog.h#L249.

[24] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165–1199. doi:10.1016/j.infsof.2013.01.008

[25] Chanchal Kumar Roy and James R Cordy. 2007. A survey on software clone detection research. *Queen's School of Computing TR* 541, 115 (2007), 64–68.

[26] Qige Song, Yongzheng Zhang, Linshu Ouyang, and Yige Chen. 2022. BinMLM: Binary Authorship Verification with Flow-aware Mixture-of-Shared Language Model. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 1023–1033. doi:10.1109/saner53432.2022.00120

[27] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639117

[28] Wei Tang, Ping Luo, Jialiang Fu, and Dan Zhang. 2020. Libdx: A cross-platform and accurate system to detect third-party libraries in binary code. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 104–115. doi:10.1109/saner48275.2020.9054845

[29] Wei Tang, Yanlin Wang, Hongyu Zhang, Shi Han, Ping Luo, and Dongmei Zhang. 2022. Libdb: An effective and efficient framework for detecting third-party libraries in binaries. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 423–434. doi:10.1145/3524842.3528442

[30] Wei Tang, Zhengzi Xu, Chengwei Liu, Jiahui Wu, Shouguo Yang, Yi Li, Ping Luo, and Yang Liu. 2022. Towards understanding third-party library dependency in c/c++ ecosystem. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12. doi:10.1145/3551349.3560432

[31] Tree-sitter Contributors. 2024. Tree-sitter - A parser generator tool and an incremental parsing library. https://tree-sitter.github.io/tree-sitter/

[32] Zhilong Wang, Lan Zhang, Chen Cao, and Peng Liu. 2023. The Effectiveness of Large Language Models (ChatGPT and CodeBERT) for Security-Oriented Code Analysis. *arXiv preprint arXiv:2307.12488* (2023).

[33] Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. 2021. CENTRIS: A precise and scalable approach for identifying modified open-source software reuse. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 860–872. doi:10.1109/icse43902.2021.00083

[34] Jiahui Wu, Zhengzi Xu, Wei Tang, Lyuye Zhang, Yueming Wu, Chengyue Liu, Kairan Sun, Lida Zhao, and Yang Liu. 2023. Ossfp: Precise and scalable c/c++ third-party library detection using fingerprinting functions. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 270–282. doi:10.1109/icse48619.2023.00034

[35] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639121

[36] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational Automated Program Repair. *arXiv preprint arXiv:2301.13246* (2023). doi:10.48550/arXiv.2301.13246

[37] HanXiang Xu, ShenAo Wang, Ningke Li, Kailong Wang, Yanjie Zhao, Kai Chen, Ting Yu, Yang Liu, and HaoYu Wang. 2025. Large language models for cyber security: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* (2025). doi:10.1145/3769676

[38] Aidan Z. H. Yang, Claire Le Goues, Ruben Martins, and Vincent J. Hellendoorn. 2024. Large Language Models for Test-Free Fault Localization. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12. doi:10.1145/3597503.3623342

[39] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024. WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024). doi:10.1145/3689736

[40] Can Yang, Zhengzi Xu, Hongxu Chen, Yang Liu, Xiaorui Gong, and Baoxu Liu. 2022. ModX: binary level partially imported third-party library detection via program modularization and semantic matching. In *Proceedings of the 44th International Conference on Software Engineering*. 1393–1405. doi:10.1145/3510003.3510627

[41] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 1145–1152. doi:10.1609/aaai.v34i01.5466

[42] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2020. Codecmr: Cross-modal retrieval for function-level binary source code matching. *Advances in Neural Information Processing Systems* 33 (2020), 3872–3883.

[43] Zimu Yuan, Muyue Feng, Feng Li, Gu Ban, Yang Xiao, Shiyang Wang, Qian Tang, He Su, Chendong Yu, Jiahuan Xu, et al. 2019. B2sfinder: Detecting open-source software reuse in cots software. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1038–1049. doi:10.1109/ase.2019.00100

[44] Chenyuan Zhang, Hao Liu, Jiutian Zeng, Kejing Yang, Yuhong Li, and Hui Li. 2024. Prompt-Enhanced Software Vulnerability Detection Using ChatGPT. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 276–277. doi:10.1145/3639478.3643065

[45] Weidong Zhang, Yu Chen, Hong Li, Zhi Li, and Limin Sun. 2018. PANDORA: A scalable and efficient scheme to extract version of binaries in IoT firmwares. In *2018 IEEE International Conference on Communications (ICC)*. IEEE, 1–6. doi:10.1109/icc.2018.8423015

[46] Xiaoya Zhu, Junfeng Wang, Zhiyang Fang, Xiaokang Yin, and Shengli Liu. 2022. BBDetector: A precise and scalable third-party library detection in binary executables with fine-grained function-level features. *Applied Sciences* 13, 1 (2022), 413. doi:10.3390/app13010413

Received 2026-01-30; accepted 2026-04-16