Demystifying Cross-Language C/C++ Binaries: A Robust Software Component Analysis Approach

Meiqiu Xu
Northeastern University
Shenyang, China
xumeiqiu@outlook.com

Ying Wang*
Northeastern University
Shenyang, China
wangying@swc.neu.edu.cn

Wei Tang
Huawei Technologies Co., Ltd.
Shenzhen, China
tangwei119@huawei.com

Xian Zhan
Huawei Technologies Co., Ltd.
Shenzhen, China
zhanxian2@huawei.com

Shing-Chi Cheung
The Hong Kong University of Science and Technology
Hong Kong, China
scc@cse.ust.hk

Hai Yu
Northeastern University
Shenyang, China
yuhai@mail.neu.edu.cn

Zhiliang Zhu
Northeastern University
Shenyang, China
zhuzhiliang_neu@163.com

***Abstract*—Binary Software Composition Analysis (BSCA) is a technique for identifying the versions of third-party libraries (TPLs) used in compiled binaries, thereby tracing the dependencies and vulnerabilities of software components without access to their source code. However, existing BSCA techniques struggle with cross-language invoked C/C++ binaries in polyglot projects due to two key challenges: (1) interference from heterogeneous Foreign Function Interface (FFI) bindings that obscure distinctive TPL features and generate false positives during matching processes, and (2) the inherent complexity of composite binaries (fused binaries), particularly prevalent in polyglot development where multiple TPLs are frequently compiled into single executable units, resulting in blurred boundaries between libraries and substantially compromising version identification precision.**

We propose DEEPERBIN, a BSCA technique that addresses these challenges through a high-quality, large-scale feature database with four key advantages: (1) high scalability that is capable of analyzing 74,647 C/C++ TPL versions, (2) efficient noise filtering to remove FFI bindings and common functions, (3) automated extraction of version string regexes for 31,855 TPL versions, and (4) generation of distinctive version features using the *Minimum Description Length* (MDL) principle. Evaluated on 418 cross-language binaries, DEEPERBIN achieves 81.2% precision and 84.6% recall for TPL detection, outperforming state-of-the-art (SOTA) techniques by 14.1% and 23.2%, respectively. For version identification, it achieves 70.3% precision, a 12.6% improvement over state-of-the-art techniques. Ablation studies confirm the usefulness of FFI filtering and MDL-based features, boosting precision and recall by 17.1% and 18.8%. DEEPERBIN also maintains competitive efficiency, processing binaries in 364.3 seconds while supporting the largest feature database.

***Index Terms*—Binary Software Composition Analysis, Third-Party Libraries, Polyglot Projects**

I. INTRODUCTION

Binary Software Composition Analysis (BSCA) [1]–[3] is a critical technique for detecting reused third-party library (TPL) versions in compiled binaries (e.g., `.so` files). By comparing extracted binary features against a pre-built ***feature database*** containing characteristic patterns (e.g., function call graphs, control flow structures) of known TPL versions, BSCA enables software dependency and vulnerability tracing without requiring source code access [4]–[6]. The analysis follows a two-phase process: (1) feature extraction from target binaries, and (2) similarity matching against feature database entries. **The effectiveness of BSCA is determined by three key database characteristics: quality (feature precision), coverage (TPL version breadth), and scale (feature volume).** While comprehensive coverage prevents false negatives, high-quality features minimize false positives, making the feature database construction crucial for accurate detection [7]–[9].

Modern software development increasingly incorporates C/C++ third-party libraries (TPLs) within polyglot architectures, where high-level languages (e.g., Python, Java) leverage native binaries to optimize performance-critical operations such as machine learning and image processing. While this approach yields significant computational advantages, it simultaneously creates a dangerous vulnerability propagation pathway. Security flaws in C/C++ TPLs can cross language barriers via *Foreign Function Interface* (FFI) mechanisms, potentially compromising dependent software ecosystems and amplifying attack surfaces [10]–[12]. Prior work, INSIGHT [11], leverages the BSCA tool LibRARIAN [13] to study vulnerability propagation from C/C++ binaries to Java/Python projects. However, existing BSCA techniques, including LibRARIAN [13] and related approaches [3], [4], [14]–[16], exhibit limited effectiveness in detecting cross-language-invoked C/C++ binaries in polyglot projects. This limitation occurs because analyzing binaries in such contexts faces several new challenges:

Challenge 1: Diverse FFI bindings pose significant interference to TPL detection. FFI bindings act as intermediate interfaces for invoking C/C++ TPLs from high-level languages, rather than being distinctive core library functions that characterize TPL identity. This creates detection challenges as FFI tools (e.g., *Cython*, *JNI*) produce highly similar FFI bindings across different TPLs - analogous to standard library functions. When mistakenly used as discriminative features, these generic FFI bindings cause false positives. Our analysis of the top 20,000 most-starred C/C++ projects on GitHub reveals that 48.8% of these TPLs contain Python/Java

*Corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00148

bindings, significantly polluting binaries with FFI artifacts.

Effectively filtering out FFI bindings from feature database is crucial for reducing their interference in TPL detection, yet this task faces three fundamental challenges: (1) differing signature naming conventions and usage patterns across FFI tools hinder unified identification, (2) compilation stripping essential FFI signatures needed for filtering references, (3) post-compilation name mangling or renaming of FFI binding symbols obscures their relationship to original source signatures. These issues cause FFI bindings to persistently contaminate feature databases, significantly reducing BSCA precision for cross-language C/C++ binaries (see § II-A).

Challenge 2: Prevalent fused binaries increase the complexity of TPL detection and version identification. In polyglot projects, fused binaries refer to binaries that compile multiple C/C++ TPLs into a single executable. The fused binary approach has gained popularity in polyglot development as it simultaneously reduces cross-language invocation overhead, prevents runtime linking errors, and enhances cross-platform portability. Our analysis of 410 C/C++ TPLs across 103 polyglot projects reveals that approximately 81.3% of these TPLs exist as fused binaries, with each fused binary containing an average of 5.28 ± 2.04 integrated TPLs.

However, fused binaries introduce three challenges for TPL detection and version identification: (1) the effectiveness of function call graph (FCG)-based matching diminishes as FCGs in fused binaries span multiple TPLs with blurred boundaries; (2) function-level feature matching (using CFGs or binary pseudo-code) becomes unreliable due to blended core features across TPLs and false matches from common functions; (3) version identification fails for partially-reused TPLs when their version strings are absent in the compiled binary (see § II-B).

In this paper, we present DEEPERBIN, a BSCA technique for detecting cross-language invoked C/C++ binaries in polyglot projects and identifying reused TPLs along with their versions. DEEPERBIN is powered by a large-scale, high-quality C/C++ TPL feature database, enabling broad coverage of library detection. It employs fine-grained version identification, extracting embedded version signatures from binaries and leveraging distinctive features to accurately characterize TPL versions (solving *Challenge #2*). Additionally, it reduces interference from FFI bindings through intelligent filtering, ensuring robust cross-language TPL detection (addressing *Challenge #1*). Specifically:

- ***Comprehensive TPL Coverage:*** The feature database collects 832,939 binary files from three mainstream Linux package repositories, covering 74,647 C/C++ TPL versions. Compared to feature databases built from a single repository (69× larger than SOTA technique BINARYAI [3]), DEEPERBIN achieves broader coverage of C/C++ TPLs, significantly reducing the false negatives of TPL detection.
- ***Advanced Noise Filtering:*** By comprehensively filtering noise features such as FFI bindings that interfere with cross-language C/C++ binary detection and common functions (e.g., standard library functions), DEEPERBIN significantly improves the purity of the feature set, markedly enhances the precision of TPL detection.
- ***Automated Version Regex Extraction:*** Version string regular expressions enable direct matching of embedded version identifiers in binary text, offering the most efficient and accurate method for version identification. Our tool automates the extraction of these regex patterns by analyzing diverse version string embeddings, successfully identifying 31,855 TPL versions—a substantial improvement over heuristic-based approaches [13], [17].
- ***MDL-based Distinctive Features:*** For version strings without regex matches, DEEPERBIN identifies the distinctive features that characterize each TPL version based on *Minimum Description Length* (MDL) approach, enabling precise identification even in minimally-reused fused binaries.

DEEPERBIN employs a hierarchical detection approach: (1) lightweight similarity matching against our large-scale C/C++ TPL feature database for initial identification; (2) direct version string regex search for precise version pinpointing; (3) distinctive feature analysis when version strings are unavailable. This three-stage process ensures both efficient and accurate TPL version identification.

To evaluate DEEPERBIN, we conducted assessments based on a high-quality ground-truth dataset containing 418 cross-language invoked C/C++ binaries. DEEPERBIN achieved a *Precision* of 81.2% and a *Recall* of 84.6% for TPL detection, significantly outperforming all SOTA BSCA tools, with improvements of 14.1% and 23.2% over the SOTA tool BINARYAI [3]. For version identification, DEEPERBIN also achieved the best precision (70.3%), which is 12.6% higher than the SOTA tool LIBRARIAN [13]. We evaluated our FFI filtering and MDL-based features by integrating them into SOTA tools and conducting ablation studies on DEEPERBIN. Results show these modules improve precision by 17.1% and recall by 18.8%. DEEPERBIN achieved competitive efficiency (364.3s) while supporting the largest feature database.

The main contributions of this paper are as follows:

- ***An effective BSCA tool.*** We developed DEEPERBIN, a robust BSCA technique based on a high-quality large-scale feature database which includes 832,939 binaries from 74,647 known C/C++ TPL versions, with version string regexes collected for 31,855 TPL versions.
- ***Thorough comparisons.*** We conduct systematic and thorough comparisons between DEEPERBIN and five SOTA BSCA techniques from different perspectives.
- ***Data availability.*** We provide a reproduction package, including our ground-truth dataset, an available tool, and experimental raw data, on the website (***https://deeperbin.github.io/***) for facilitating future research.

II. MOTIVATION AND CHALLENGES

A. Varied FFI Binding Patterns Interfere with TPL Detection

FFI bindings act as intermediate interfaces for calling C/C++ TPLs from high-level languages but lack the distinctive features that characterize TPL identity, and their presence actually disrupts TPL detection in two ways: (1) ***False Positives***, as uniform FFI tools generate nearly identical bindings across different TPLs, increasing incorrect matches; and (2) ***False Negatives***, since FFI bindings introduce redundant features,

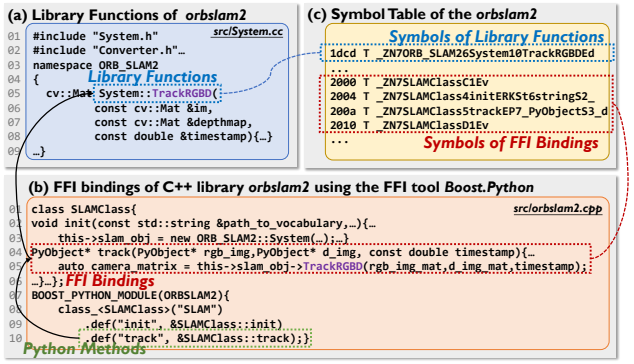

Fig. 1: Identifying FFI bindings in binaries poses key challenges

diluting the weight of core functions and lowering similarity scores. To ensure accurate and reliable BSCA, FFI bindings should be filtered out before TPL detection.

While FFI bindings and library functions can be clearly differentiated at the source code level through their distinct usage patterns in FFI tools, this distinction becomes obscured in compiled binaries. At the binary level, both types of functions appear interleaved in the symbol table, where the absence of source context, compiler-induced name transformations, and variations across different FFI implementations collectively hinder accurate identification.

Figure 1 illustrates the challenge of identifying FFI binding symbols in binaries. Figure 1(a) shows the library function *System::TrackRGBD ()* from the C++ library `orbslam2` (`lines 5-8`). Figure 1(b) presents the FFI binding *SLAMClass::track ()* of *System::TrackRGBD ()* (`lines 4-5`). The FFI tool *Boost.Python* maps the Python method *SLAM.track ()* to the *SLAMClass::track ()* via the FFI tool function *BOOST_PYTHON_MODULE.def ()* (`line 10`), establishing the cross-language invocation chain: *SLAM.track ()* (Python function) ⇒ *SLAMCLASS::track ()* (FFI binding) ⇒ *System::track ()* (C++ library function). FFI bindings and library functions are easily distinguishable in source code.

When FFI bindings and library functions are compiled into the binary, they become intermixed in the symbol table. As illustrated in Figure 1(c), the symbol *1dcd:_ZN7ORB_SLAM26System10TrackRGBDEd* represents the library function System::TrackRGBD(), while *200a:_ZN7SLAMClass5trackEP7_PyObjectS3_d* corresponds to the FFI binding *SLAMCLass::track ()*. Their naming conventions make differentiation non-trivial. Moreover, adjacent symbols (e.g., at addresses *2000*, *2004*, and *2010*) are also FFI bindings used for cross-language invocation, further blurring the distinction. This challenge is exacerbated by the varying FFI binding formats generated by different FFI tools.

B. Prevalent Fused Binaries Complicate TPL Detection

Fused binaries -single executables combining multiple C/C++ TPLs -have become prevalent in polyglot development due to their advantages in reducing cross-language overhead, preventing linking errors, and improving portability. As shown in Figure 2, `PySuperTuxKart` [18] exemplifies this approach by merging 11 C/C++ TPLs (including 10 widely-used libraries and a custom

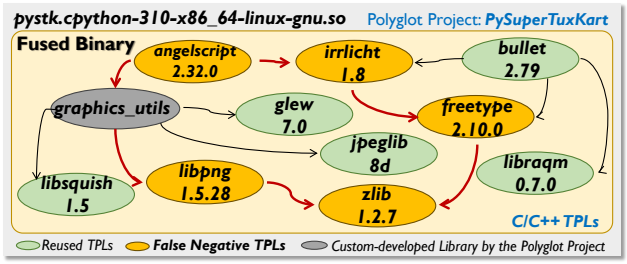

Fig. 2: Challenges arising from the complexity of fused binaries

`graphics_utils` component without version information) into `pystk.cpython-310-x86_64-linux-gnu.so`, exhibiting complex reuse patterns ranging from full-library to function-level granularity. To assess how this fusion affects BSCA techniques, we evaluated three state-of-the-art approaches that leverage different code features to perform TPL detection: LIBDB [16] (*FCG-based*), BINARYAI (*function-based*), and LIBRARIAN (*symbol-based*). The analysis revealed that:

- ***FCG-based matching loses effectiveness in fused binaries due to cross-TPL graph merging.*** LIBDB failed to detect 5 TPLs because, in the fused binary, the FCG of `angelscript` spans five TPLs: (`graphics_utils`→`libpng`→`zlib`→`irrlicht`→`freetype`→`zlib`), where `graphics_utils` (a custom library) was absent from the database. This creates a structural mismatch between the database's isolated TPL FCGs and the binary's merged cross-TPL graphs, generating false negatives.
- ***Function-level matching proves unreliable due to feature blending and common function collisions.*** In BINARYAI's case, it produced 2 false positives (`assimp` and `kodi`) - all multimedia TPLs sharing similar graphics functions with `irrlicht` in this fused binary.
- ***Partial TPL reuse reduces version identification precision.*** LIBRARIAN missed all 11 versions by assuming full-library reuse, failing for fused binaries with partial reuse.

This study shows that in fused binaries, existing BSCA methods have lower TPL version identification accuracy due to lacking robust binary features, thorough noise filtering, and effective version identifiers.

III. DEEPERBIN APPROACH

To address the challenge of BSCA in polyglot projects that incorporate C/C++ libraries, we developed DEEPERBIN, a novel BSCA technique designed to trace TPLs and their versions, enabling precise security vulnerability impact assessment. DEEPERBIN takes polyglot projects (e.g., `.whl/.jar`) as input and automatically extracts the cross-language invoked binaries to perform TPL version identification. Figure 3 gives an overview of DEEPERBIN.

DEEPERBIN is empowered by a high-quality C/C++ TPL feature database: (1) ***Comprehensive TPL Coverage:*** By aggregating data from three Linux repositories (832,939 binaries, 74,647 TPL versions - 69× larger than BINARYAI), it significantly improves recall over single-repository approaches; (2) ***Advanced Noise Filtering:*** The expanded database introduces noise (e.g., common functions and FFI bindings) that increases

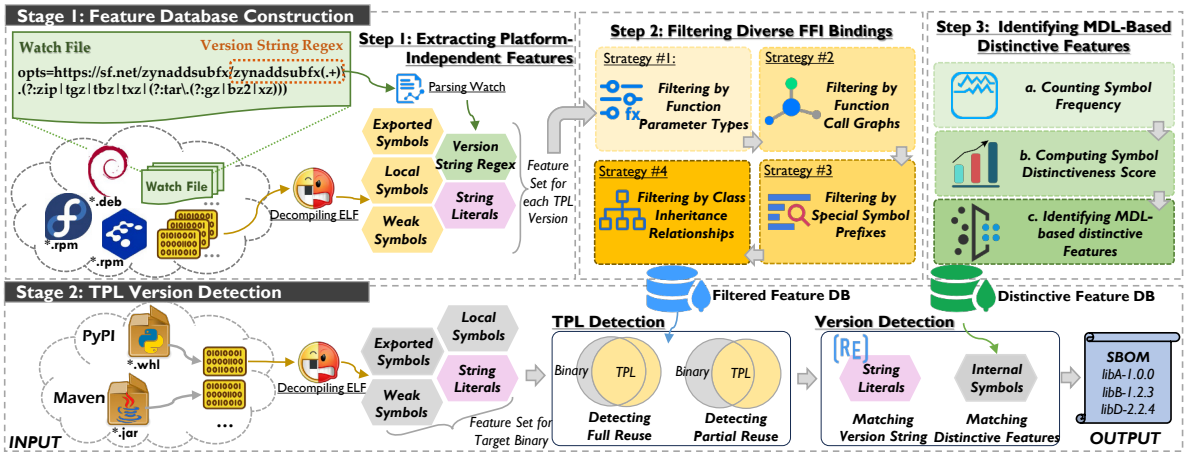

Fig. 3: The overall architecture of DEEPERBIN

false positives. DEEPERBIN addresses this through targeted noise filtering, with four specialized strategies for FFI binding identification (*Step 2*); (3) ***Automated Version Regex Extraction:*** DEEPERBIN enhances version identification through automated regex extraction (covering 31,855 TPL versions vs. 15 and 126 in prior works [13], [19]), enabling direct binary matching of version identifiers for precise identification; (4) ***MDL-based Distinctive Features:*** For missing version strings in fused binaries, DEEPERBIN employs MDL-based distinctive features to enable precise version identification (*Step 3*).

DEEPERBIN employs a hierarchical detection approach: (1) lightweight similarity matching against our large-scale feature database, (2) direct version string extraction for identified TPLs, and (3) MDL-based distinctive feature matching when version strings are unavailable. This three-stage process ensures both efficient and accurate version identification.

A. Feature Database Construction

To identify reused TPLs in binaries, the BSCA technique requires a reference database of known TPL code features. DEEPERBIN builds this database by extracting features from TPL binaries and indexing them with version metadata in the format: <*binary feature set*, *binary file name*, *TPL name*, *TPL version*>. Table I summarizes DEEPERBIN's feature database, which includes 832,939 binaries covering 74,647 C/C++ TPL versions (as of March 1, 2025). Notably, DEEPERBIN prioritizes platform-independent features to generate the feature set for each TPL version, as these features remain stable across platforms regardless of underlying architecture or compilation options, making feature matching more scalable for large-scale feature databases. For efficiency, we extract features exclusively from x86 binaries since platform-independent features generalize well across architectures, eliminating redundant processing of multi-arch binaries.

DEEPERBIN constructs the *feature database* in three steps:

Step 1: Extracting Platform-independent Features.

DEEPERBIN crawls all available binary packages of C/C++ TPL versions (`.deb`/`.rpm`) from *Debian* [20], *Fedora* [21], and *OpenEuler* [22] package repositories. It decompresses all binaries from each package using the archive file analysis tool UNIX-AR [23]. For each binary, DEEPERBIN further extracts the following platform-independent features.

- ***Internal Symbols.*** Internal symbols represent functions and variables defined within the binary. Based on visibility and linkage, they fall into three categories: (1) ***exported symbols*** (externally visible and referenceable by other binaries), (2) ***local symbols*** (visible only within the binary for internal encapsulation), and (3) ***weak symbols*** (externally visible but with lower precedence during linking). These symbols serve as effective TPL identifiers, as they often exhibit unique patterns across different TPLs and versions [24].

  Existing approaches typically only extract *exported symbols*, significantly limiting recall. DEEPERBIN overcomes this by extracting all three symbol types using ANGR [25], a platform-independent binary analysis framework. To ensure comprehensive symbol recovery, DEEPERBIN extracts from both `.symtab` and `.dynsym` sections of binaries—while *strip* removes most `.symtab` symbols, `.dynsym` retains those critical for dynamic linking.
- ***String Literals.*** String literals are sequences of characters derived from string constants in the source code, stored in read-only sections (e.g., `.rodata`) of binaries. Because they directly mirror source-level strings and are platform-independent, they serve as highly reliable features for TPL detection [4], [5], [14]–[16]. DEEPERBIN extracts string literals as features to further enhance the comprehensiveness and robustness of the feature database.

For each C/C++ TPL package, DEEPERBIN automatically extracts *version string regexes*.

- ***Version String Regexes.*** A *version string* is a source-code identifier for a TPL's version, typically compiled into the binary's *read-only* section (e.g., `.rodata`). These strings provide highly reliable version identification since they explicitly encode version information. While prior work [13] showed that regex-based matching can accurately identify such strings in binaries, the diversity of version formats across TPLs necessitates custom regex patterns for each case. Manual regex construction is infeasible at scale; consequently, existing tools like LIBRARIAN [13] and VES [19] resort to heuristic methods, extracting only 15 and 126

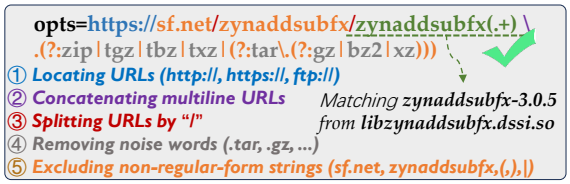

Fig. 4: Parsing version string regexes from WATCH files

TABLE I: Statistics of DEEPERBIN's Feature Database

Linux Distribution	*#Binaries*	*#TPL Versions**	*#WATCH Files*	*#Version Regexes*
Debian	374,092	47,460	41,023	29,221
Fedora	321,045	23,457	-	1,563
OpenEuler	137,802	3,730	-	1,071
Summary	832,939	74,647	41,023	31,855

* All versions are retained to maximize C/C++ TPL version coverage.

version string regexes, respectively—a limited coverage that highlights the need for automation.

Our study found that *Debian* packages often include version string regexes in a packaging configuration file named `WATCH` [26], which track upstream source updates for *Debian* packages. The regexes are embedded in URLs under the *opts* field, but their inconsistent formatting complicates automated extraction.

To address this, DEEPERBIN implements a parsing algorithm to extract version string regexes from the *opt* field of the `WATCH` file. As described in Figure 4, it combines *URL pre-processing*, *noise word removing*, and *non-regular-form strings excluding* to automate the extraction of version string regexes from the configuration files.

Our statistics show that 41,023 (86.4%) *Debian* packages in the feature database contain `WATCH` files, from which DEEPERBIN successfully extracts 29,211 version string regexes. For *Fedora/OpenEuler* packages without `WATCH` files, DEEPERBIN shares version string regexes when *Fedora/OpenEuler* has the same TPL version as *Debian*. In total, 31,855 version string regexes were collected, covering 42.7% of the TPL versions in our large-scale feature database, significantly better than prior studies [13], [19]. For cases where version string regexes cannot be used for version identification, DEEPERBIN confirms the version by identifying the distinctive features that characterize the TPL version (see *Step 3*).

Step 2: Filtering Diverse FFI Bindings. Distinguishing FFI bindings from native C/C++ TPL symbols in binary symbol tables poses significant challenges (see § II-A), further compounded by the substantial diversity of FFI binding patterns across different tools. To identify and filter out FFI bindings, as shown in Table II, we inspected eight widely used FFI tools by polyglot projects: *Boost.python* [27], *pybind11* [28], *Python/C API* [29], *Cython* [30], *CFFI (API Mode)* [31], *SIP* [32], *SWIG (Python/Java-C/C++)* [33] and *JNI* [34]. Guided by tools' documentation and tracking their generated FFI binding symbols in binaries, we found that FFI binding symbols in binaries originate from two types of functions: (1) non-member functions defined in source files (e.g., `.c/.cpp`), and (2) member functions encapsulated within classes.

To effectively filter ***symbols of non-member functions***, our approach analyzes three key characteristics in binary files: (1) function signatures (*Strategy #1*), (2) call graph patterns (*Strategy #2*), and (3) specific naming prefixes (*Strategy #3*). These features serve as reliable indicators of functions participating in cross-language interactions through FFI bindings. For ***symbols of member functions originating from binding classes***, which cannot be identified using the aforementioned methods (as detailed in *Strategy #4*), our solution incorporates virtual table analysis to reconstruct class hierarchies. This enables accurate tracing of language-binding classes involved in cross-language operations. The comprehensive filtering methodology operates as follows:

- ***Strategy #1: Function Parameter Types.*** For C++ binaries, DEEPERBIN identifies FFI bindings by analyzing preserved function signatures, specifically detecting cross-language parameter types (e.g., *PyObject** for Python, *jobject* for Java) that FFI mechanisms introduce to enable object manipulation across languages. These distinctive type signatures remain observable in compiled symbols and serve as reliable indicators of language-binding interfaces.
- ***Strategy #2: Function Call Graphs.*** For C binaries where function signatures are stripped during compilation, our approach employs call graph analysis to identify FFI bindings through two reliable indicators: (1) invocation by language-specific registration functions (e.g., Python's *PyMethodDef* or Java's *JNI* registration methods), which explicitly declare binding interfaces; and (2) usage of FFI-specific runtime functions, detected through external symbol analysis. This methodology is grounded in the behavior of FFI mechanisms, which necessarily create these observable call patterns when establishing cross-language interoperability. The first pattern confirms explicit registration as a binding interface, while the second reveals implicit participation in cross-language operations through FFI runtime dependencies.
- ***Strategy #3: Special Symbol Prefixes.*** Some FFI tools

TABLE II: Overview of Diverse FFI Bindings Generated by Different FFI Tools

FFI Tool	*Binding Languages*	*FFI Bindings in Source Code*	*Symbols of FFI Bindings in Binaries*
CFFI (API mode)	Python-C	`int curl_ws_recv(void *curl,...)`	`_cffi_d_curl_ws_recv`
JNI	Java-C/C++	`void LASreader::set_transform()`	`Java_com_LASreader_setTransform`
SWIG	Java-C/C++	`bool arm_neon_support()`	`Java_libtorrent_jni_arm_1neon_1support`
pybind11	Python-C++	`PyObject* PyStatusNotOkOrNone()`	`_ZZL19PyStatusNotOkOrNoneE13kImportedObj`
Python/C API	Python-C/C++	`PyObject *aacgmv2_setDateTime(...)`	`aacgmv2_setDateTime`
Boost-python	Python-C/C++	`PyObject* track(PyObject* ...)`	`ZN7SLAMClass5trackEP7_PyObjectS3_d`
Cython	Python-C/C++	`cdef __set__(self, int val)`	`__pyx_pw_6LBFGS_2m___set___`
SWIG	Python-C/C++	`MEMBER_FUNCTION(gp_camera_autodetect`	`_wrap_gp_camera_autodetect`
SIP	Python-C/C++	`Soprano::Util::AsyncResult *isEmptyAsync() const;`	`_Z24PyAsyncModel_isEmptyAsyncP18Py AsyncModelObjectP7_object`

add standardized prefixes (e.g., *cffi_*, *jni_*) to generated binding symbols to prevent naming collisions. DEEPERBIN utilizes these distinctive prefixes for detection, as they serve as dedicated markers for binding symbols and maintain consistent patterns across versions and application scenarios.

- ***Strategy #4: Class Inheritance Relationships.*** To address these cases, DEEPERBIN employs virtual table analysis to reconstruct class hierarchies and verify inheritance from FFI-specific base classes (e.g., *PyObject* in Python or *JNI wrapper* classes in Java). This approach is grounded in the observation that FFI mechanisms universally require binding classes to extend particular base classes for runtime integration. By detecting these characteristic inheritance patterns, DEEPERBIN can reliably identify both binding classes and their member functions, significantly improving filtering completeness and robustness.

Step 3: Identifying MDL-based Distinctive Features. To reliably identify reused C/C++ TPL versions in cross-language binaries, DEEPERBIN employs a dual-phase approach. First, it utilizes pre-extracted version string regular expressions for 31,855 TPL versions to perform direct matching against string literals in binaries. When version strings are unavailable - either due to their absence in source code or removal during compilation - DEEPERBIN activates its secondary MDL-based (***Minimum Description Length***) algorithm. This advanced technique identifies version-specific distinctive features, enabling accurate version identification even in absence of explicit version strings.

The MDL principle [35] is an information-theoretic approach to model selection, where the *best model* minimizes the *total description length*—the sum of the model's complexity (description length) and the encoded data length (fit to the model). In DEEPERBIN, the best model corresponds to the selected set of distinctive features for each TPL version. To identify this set, DEEPERBIN iteratively computes the total description length, incrementally adding symbols from the binary feature set until the description length is minimized. The resulting set contains only the most distinctive symbols—eliminating redundancy while preserving uniqueness—by balancing model complexity (symbol count) and encoding efficiency (symbol distinctiveness). The detailed procedure is as follows.

a. Counting Symbol Frequency. First, DEEPERBIN computes the *frequency* of each symbol f across the entire feature database as $freq(f) = |\{TPL_i : f \in S_i\}|$. This step identifies how commonly a symbol appears across all TPL versions, providing a basis to distinguish common from unique symbols.

b. Computing Symbol Distinctiveness Score. Then, DEEPERBIN calculates the *Distinctiveness Score(DS)* for each f within the specific TPL version TPL_i using the formula:

$$DS(f, TPL_i) = \frac{1}{freq(f)} \times \log_2\left(\frac{|S_i|}{freq(f)}\right) \quad (1)$$

This step quantifies each symbol's distinctiveness by considering both its global frequency and local concentration within the TPL version. Unlike traditional *Inverse Document Frequency (IDF)*, which only measures global frequency, *DS* adaptively integrates both global and local factors, providing a more effective measure of a symbol's uniqueness.

Algorithm 1: Identifying MDL-driven Distinctive Features

Input: A list of symbols $(f, freq, DS)$ sorted by descending DS
Output: Distinctive feature set S_i^* for the current TPL version

```
Initialize S_i* ← ∅, L_total ← ∞;        // L represents the length
for each symbol (f, freq, DS) in the sorted list do
    if freq > 0 then
        S_temp ← S_i* ∪ {f}              // S_temp is a temporary set
        L_model ← |S_temp| · log2|F|
        L_data ← Σ_{f_k ∈ S_temp} log2 freq(f_k)
        L_prev ← L_model + L_data
        if L_prev ≤ L_total then
            Add f to S_i*
            Update L_total ← L_prev
return S_i*
```

Fig. 5: Identifying MDL-based distinctive features

c. Identifying MDL-based Distinctive Features. Based on the *DS*, DEEPERBIN ranks all symbols in the current TPL version. This ranking prioritizes symbols most likely to serve as distinctive features, enabling an efficient selection process. As Figure 5 shows, the algorithm begins by initializing the distinctive feature set S_i^* as empty and setting the initial total description length L_{total} to infinity (`Line 1`). For each f, it calculates its log frequency to quantify global frequency and determine its contribution to both *model complexity* and *data encoding*. A symbol with low frequency is more likely to reduce L_{data} (`Line 6`), while adding more symbols increases L_{model} (`Line 5`). The algorithm computes the new *total description length* $L_{total} = L_{model} + L_{data}$ and compares it to L_{prev} (`Line 7`). If L_{total} is reduced, f is retained in S_i^* and L_{total} is updated; otherwise, f is discarded (`Lines 8-10`). This process continues until all candidates are evaluated, yielding a distinctive feature set for the given TPL version.

B. TPL Version identification

For a polyglot project containing embedded C/C++ binaries (e.g., `.whl/.jar` files), DEEPERBIN extracts these binaries and leverages ANGR tool to analyze both *internal symbols* and *string literals* for TPL detection and version identification.

TPL Detection. Based on the number of matched internal symbols, the similarity of two binaries are defined as follows:

$$similarity\ score = \frac{FV_{target} \cap FV_{TPL}}{FV_{TPL}} \quad (2)$$

where FV_{target} is a feature set of the C/C++ binary embedded in the polyglot project, FV_{TPL} is a feature set from the feature database. DEEPERBIN employs a dual-threshold strategy for TPL detection: (a) the *similarity score* threshold and (b) the *file proportion* threshold. Detection results meeting threshold (a) are considered candidate TPLs, though some remain false positives due to code clones. To mitigate these, DEEPERBIN heuristically applies threshold (b) to identify the prime TPL most likely associated with a binary and filter out clones.**For threshold (a)**, we conducted thorough threshold sensitivity analysis (examining values from 0.3 to 0.9) and found 0.5 optimally balances precision and recall for DEEPERBIN. **For threshold (b)**, DEEPERBIN introduces the novel *file pro-*

portion-the ratio of a binary file to its containing software package. A binary typically shows a higher *file proportion* in its prime TPL package (e.g., `libz.so` with 1.0 in `zlib`), but a lower ratio when reused in another TPL (e.g., 0.25 in `maqview`). This dual-threshold strategy effectively distinguishes prime TPLs from cloned code segments.

Version Identification. For detected C/C++ TPLs, DEEPERBIN further utilizes their version string regexes to match unique version numbers in the binary's string literals. However, for generic regexes (e.g., `v?(\d\.\d\.\d)`), multiple candidate versions may be produced from a *fused binary*. To resolve this ambiguity, DEEPERBIN leverages the *version distance* (*VD*) proposed by LIBDB [16], which measures the *distance* between the reported and correct versions. Given two versions $a_1.b_1.c_1$ and $a_2.b_2.c_2$ (*Major.Minor.Patch*), *VD* is calculated as $10\times|a_1-a_2|+b\times|b_1-b_2|+c\times0.1\times|c_1-c_2|$. DEEPERBIN resolves version ambiguity by comparing candidate versions against the TPL's version sequence in the feature database, selecting the version with minimal *VD* as the correct match.

When the binary lacks version strings, DEEPERBIN compares the symbols of the detected TPL against the distinctive feature set for each TPL version (see *Step 3*). If a detected symbol matches in a distinctive feature set, the corresponding version is pinpointed. Finally, DEEPERBIN outputs a software bill of materials (SBOM) listing the C/C++ TPL versions reused in the polyglot project.

IV. EVALUATION

We study four research questions in our evaluation section:

- **RQ1 (Effectiveness of TPL Detection)**: *How effective is* DEEPERBIN *in TPL detection?* To answer RQ1, we quantitatively assessed the tool's effectiveness in TPL detection through *Precision*, *Recall*, and *F1-score* metrics, followed by a comparative analysis against SOTA BSCA techniques.
- **RQ2 (Effectiveness of TPL Version Identification)**: *How effective is* DEEPERBIN *in TPL version identification?* To answer RQ2, we conducted a comparative evaluation of DEEPERBIN against SOTA BSCA techniques regarding their *Precision* in TPL version identification.
- **RQ3 (Ablation Study)**: *To what extent do the three core modules — filtering diverse FFI bindings, matching version strings and identifying MDL-based distinctive features — contribute to* DEEPERBIN *'s effectiveness in TPL detection and version identification?* To address RQ3, we quantified the impact of these three modules on effectiveness metrics by: (1) incrementally integrating each module into SOTA BSCA techniques, and (2) ablating them from DEEPERBIN.
- **RQ4 (Efficiency Evaluation)**: *How efficient is* DEEPERBIN *in the overall BSCA process?* To answer RQ4, our efficiency evaluation examines DEEPERBIN's execution time across two critical phases: (1) feature extraction, and (2) TPL detection and TPL version identification.

A. Data Collection

In the absence of publicly available ground-truth datasets, we constructed our evaluation dataset by extracting cross-language invoked C/C++ binaries from polyglot projects in *PyPI* and *Maven* repositories. Through rigorous manual verification, we identified reused C/C++ TPLs and their specific versions within these binaries. The dataset construction involved two key steps:

TABLE III: Statistics of the Ground-Truth Dataset

Central Repository	*#Polyglot Projects*	*#C/C++ Binaries*	*#C/C++ TPLs*	*#C/C++ TPL Versions*
PyPI	49	207	321	263
Maven	54	211	531	490
Summary	103	418	852	753

(1) *Collecting polyglot projects.* To ensure dataset quality, we selected the top 10% most-downloaded polyglot projects from *PyPI* and *Maven* using *PyPISimple* [36] and *MavenIndexer* [37], then extracted archive files (`.whl`/`.jar`). C/C++ binaries were identified via *pyelftools* [38] with ELF header verification (`magic: 0x7F45446, type: ET_DYN`).

(2) *Inspecting C/C++ TPL versions.* To identify C/C++ TPL versions within binaries, we manually inspected the source code of polyglot projects, employing distinct strategies based on the three common TPL incorporation patterns:

- *Remote Package Dependencies.* By examining build configuration files (e.g., CI scripts in workflows/`*.yml`), we extracted explicit package installation commands. For instance, the directive `CIBW_BEFORE_BUILD_LINUX: yum install -y freetype-devel`, definitely establishes the `freetype` library dependency.
- *Git Submodule References.* We determined TPL versions by analyzing standard project hierarchies, specifically examining canonical dependency directories (`/vendors/`, `/third_party/`, or `/external/`).
- *Locally copied TPLs.* When TPLs are copied directly into projects, we verify versions through examination of the libraries' documentation files, particularly `README` (installation notes), and `CHANGELOG` (version history).

We collected polyglot projects from the *PyPI* and *Maven* repository snapshots dated March 1, 2025. After filtering out projects with inaccessible source code, we performed manual analysis of 938 qualifying projects to: (1) detect embedded C/C++ TPLs, and (2) recover their version metadata. Three authors of this paper, each with over three years of experience in SCA, independently analyzed each polyglot project and annotated the reused C/C++ TPLs. Through cross-validation, the inter-rater agreement, as measured by *Cohen's kappa* [39], stands at 0.93, signaling a near-perfect agreement between the raters. Our analysis revealed that only 10.9% of these projects exhibited explicit version management for their C/C++ TPL dependencies. As shown in Table III, from 103 polyglot projects, we obtained 418 C/C++ binaries that were originally built from 852 C/C++ TPLs, referred to as ***Dataset-Libs***. Of these TPLs, 753 have identifiable versions, forming ***Dataset-Vers***. This distinction was made to retain as much evaluation data as possible for all RQs.

B. Baselines

We compare DEEPERBIN with five SOTA BSCA tools:

- **B2SFINDER [4]** is a SOTA ***binary-to-source (B2S) SCA tool***, specializes in ***identifying embedded C/C++ TPLs***

TABLE IV: Overview of All Baselines

Baseline (Year)	*Type*	*Origin of C/C++ Binaries*	*Version Identification*	*OSS?*
B2SFinder(*ASE'19*)	B2S	**COTS software**	*MFN*	✓
LibRARIAN(*ICSE'21*)	**B2B**	**Android Apps**	*MFN+VS*	✓
LibDB(*MSR'22*)	**B2B**	C/C++ projects	*MFN*	✓
BinaryAI(*ICSE'24*)	B2S	C/C++ projects	*MFN*	✗
LibvDiff(*ICSE'24*)	B2S	C/C++ projects	*CD*	✓
DeeperBin	**B2B**	**Polyglot projects**	*MFN+VS*	✓

MFN* (*Number of Matched Features: determines the version based on the number of matched features; ***VS* (*Version String*)**: matches the version string directly from the binary (e.g., `.rodata` section); ***CD* (*Code Differences*)**: detects the TPL version using code differences between adjacent versions.

within commercial off-the-shelf (COTS) software through seven distinctive feature comparisons. Since the original feature database was inaccessible, we faithfully reconstructed it according to the published methodology. Leveraging the *Ubuntu* packages, we significantly expanded the C/C++ TPL version coverage from 2,189 to 6,800 (increase 3×).

- **LibRARIAN [13]** is a SOTA ***binary-to-binary (B2B) BSCA tool***, specializing in ***identifying C/C++ TPL versions*** in Android applications through five distinct metadata features and version regex matching. To address incompleteness in the original feature database, we reconstructed it using the *Debian* package repository, expanding TPL version coverage from 904 to 19,193 (increase 21×). LibRARIAN was employed by Insight [11] to investigate vulnerability propagation impact cross different programming ecosystems.
- **LibDB [16]** is a SOTA ***binary-to-binary BSCA tool*** that leverages neural network-based function embeddings and call graphs to detect C/C++ TPLs and their versions.
- **BinaryAI [3]** is a SOTA ***commercial BSCA tool*** that employs a two-phase matching strategy combining transformer-based function embeddings and link-time locality to capture both syntactic and semantic features of the C/C++ TPLs.
- **LibvDiff [40]** is a SOTA ***version identification tool for C/C++ binaries*** that leverages fine-grained code differences between adjacent versions to detect C/C++ TPL versions.

All baseline tools are summarized in Table IV. For a comprehensive and fair comparison, our baselines cover four dimensions: **(1) *BSCA type.*** Both B2B and B2S techniques are included to reflect diverse detection strategies. **(2) *Binary origins.*** The target C/C++ binaries are drawn from both pure and polyglot projects, capturing the challenges introduced by cross-language invocation. **(3) *Version identification strategies.*** The baselines vary in strategy for version identification; for instance, LibRARIAN employs limited version regex matching (15 TPLs), while DeeperBin automatically extracts regex from large-scale repositories (34,855 TPLs). **(4) *Source code accessibility.*** The baselines span open-source and commercial tools, covering academic and industry solutions.

C. RQ1: Effectiveness of TPL Detection

Study Methodology. To evaluate TPL detection, we compare DeeperBin against the baselines using our ground-truth *Dataset-Libs*, which measures each tool's capability to identify reused C/C++ TPLs regardless of their specific versions. LibvDiff is excluded in this evaluation, as its publicly available implementation supports only TPL version identification.

TABLE V: Comparison Results of TPL Detection (RQ1)

Baseline	*#TP*	*#FP*	*#FN*	*Precision*	*Recall*	*F1-score*
B2SFinder′	262	471	590	35.7%	30.8%	0.33
LibRARIAN′	98	72	754	57.6%	11.5%	0.19
LibDB	295	313	557	48.7%	34.6%	0.40
BinaryAI	523	256	329	67.1%	61.4%	0.64
DeeperBin	**721**	**167**	**131**	**81.2%** ⇑	**84.6%** ⇑	**0.83** ⇑

Evalution Metrics. We employ six evaluation metrics that are widely used in previous studies [3], [4], [16]: (1) ***True Positive (TP)***: C/C++ TPLs correctly detected by the tools; (2) ***False Positive (FP)***: C/C++ TPLs incorrectly detected by the tools; (3) ***False Negative (FN)***: C/C++ TPLs missed by the tools. Based on the above three metrics, we can obtain the *Precison*, *Recall* and *F1-score* as follows: (a) ***Precision*** = *TP / (TP + FP)*; (b) ***Recall*** = *TP / (TP + FN)*; (c) ***F-measure*** = $2 \times Precision \times Recall$ *(Precision + Recall)*.

Results. Table V shows the experiment results of RQ1. DeeperBin successfully detected 721 TPLs with a *Precision* of 81.2%, a *Recall* of 84.6%, and an *F1-score* of 0.83. In contrast, B2SFinder, LibRARIAN, LibDB and BinaryAI report a large number of false positives and false negatives, resulting in *Precision* = 35.7%, 57.6%, 48.7% and 67.1%, and *Recall* = 30.8%, 11.5%, 34.6% and 61.4%, respectively. Based on the results, we can see DeeperBin outperforms other baselines regarding all the evaluation metrics.

To analyze the root causes of *FP*s and *FN*s, we applied the open coding procedure [41] with triple-author cross-validation. Two authors independently analyzed each case by writing 1–2 phrases describing causes (e.g., `matched symbols generated by FFI tool pybind11`) without predefined categories. The identified causes were then consolidated into preliminary categories (e.g., `FFI binding symbols matched TPL`). Disagreements were adjudicated by the third author until consensus, and each *FP*/*FN* was assigned at least one clear-cut root cause.

***FP Analysis of* DeeperBin.** We analyzed all false positives reported by DeeperBin and concluded two primary reasons: **(1) *Code Clone Across Different TPLs* (115/167=68.9% of cases)**. DeeperBin may generate false positives primarily due to widespread internal code cloning in TPLs. This occurs because: (1) code clones create identical features across different TPLs; (2) Binary-based analysis cannot inherently distinguish which TPL originally contained the cloned code (Prime TPL). DeeperBin uses *file proportion* - the ratio of a binary file to total files in its software package - as a heuristic to identify Prime TPLs. While *file proportion* effectively improves detection accuracy in typical scenarios, the static linking approach for binary library reuse can artificially elevate this metric, resulting in false positives. **(2) *Code Duplication Across Different Branches of the TPL* (52/167=31.1% of cases).** Many mature C/C++ TPLs maintain multiple branches tailored for specific platforms, configurations, or use cases. Although these branches differ in build settings or extensions, they share a common codebase and exhibit high feature similarity. As a result, if the binary reuses such TPLs, it may incorrectly report

multiple branches originating from the same TPL.

FP Analysis of Other Baselines. When applied to cross-language binaries, baseline methods exhibit significantly higher false positive rates compared to their performance on original datasets. This degradation stems primarily from: **(1) *Misidentification of FFI Bindings as C/C++ TPL Features* (460/1112=41.4% of cases).** All BSCA techniques that fail to properly filter out FFI bindings from C/C++ TPLs inadvertently preserve these bindings as features in their databases. Consequently, when both the target binary and a TPL contain similar FFI bindings (e.g., those generated by *Cython*), tools may produce incorrect TPL matches. **(2) *Failure to Distinguish Original TPLs from Reused Ones in Feature Databases* (652/1112=58.6% of cases).** Despite our efforts to reconstruct larger and more up-to-date databases for B2SFINDER and LIBRARIAN, both tools still reported numerous false positives. The root cause lies in their architectural limitation of failing to discriminate between original TPL implementations and their derivative reused versions within the feature database. For BSCA techniques, the absence of explicitly modeled reuse relationships among TPLs in the feature database can introduce noise and reduce detection precision.

***FN Analysis of* DEEPERBIN.** DEEPERBIN failed to detect 131 TPLs, all of which were custom-developed libraries internal to polyglot projects and absent from standard Linux package repositories. A representative case is the polyglot project `hgdb` [42], where the binary `_hgdb.cpython-310-x86_64-linux-gnu.so` reuses 12 C/C++ TPLs—11 of which were successfully identified via package repositories. The remaining library, `vlstd` [43], was missed because it is an internal C/C++ TPL developed by `hgdb` maintainers and unavailable in any public repository.

FN Analysis of Baselines. Most false negatives in baseline tools stem from detecting partially reused TPLs within *fused binaries*. While existing approaches attempt to address this challenge, they exhibit critical limitations: (1) B2SFINDER relies on *compilation dependency layered graphs* (CDLGs), but repeated recompilation in polyglot projects distorts the original CDLG structure, invalidating its partial reuse detection. (2) LIBRARIAN's similarity metric $\frac{FV_1 \cap FV_2}{FV_2 \cup FV_2}$ effectively identifies fully reused TPLs but fails to reliably detect partial reuse in *fused binaries* due to feature dilution. (3) Threshold-based tools (LIBDB and BINARYAI) miss partial reuse instances when the shared feature proportion falls below predefined thresholds, generating false negatives.

> ***Conclusion:*** DEEPERBIN *achieves a Precision of 81.2% and a Recall of 84.6% in TPL detection, which significantly outperforms all baselines.*

D. RQ2: Effectiveness of TPL Version Identification

Study Methodology. To evaluate TPL version identification, we compare DEEPERBIN against the baselines using our ground-truth *Dataset-Vers*, which measures each tool's capability to correctly detect the versions of reused TPLs.

Evaluation Metrics. Following previous studies [14], [44], we adopt two evaluation metrics: (1) ***True Versions (TV)***: Versions of C/C++ TPLs correctly detected by the tools. (2) ***Precision′*** = *TV* / *TP′*, since not all *TP* detected in RQ1 have version records in *Dataset-Vers*, the *TP′* used for version evaluation is defined as the intersection of the two sets.

TABLE VI: Comparison Results of TPL Version Identification (RQ2)

Baseline	*#TP′*	*#TV*	*Precision′*
B2SFINDER′	126	39	31.0%
LIBRARIAN′	78	45	57.7%
LIBDB	189	82	43.4%
LIBVDIFF	41	21	51.2%
BINARYAI	280	151	53.9%
DEEPERBIN	660	**464** ⇑	**70.3%** ⇑

Results. As shown in Table VI, DEEPERBIN correctly detected 464 versions out of 660, achieving the highest *Precision′* of 70.3%. The other baselines yielded comparatively lower *Precision′*: 53.9%, 51.2%, 43.4%, and 31.0% for BINARYAI, LIBVDIFF, LIBDB, and B2SFINDER, respectively.LIBRARIAN attained a *Precision′* of 57.7% through its hybrid approach combining feature matching (*MFN*) and version string matching (*VS*). However, its effectiveness was fundamentally limited by only supporting version string regexes for 15 C/C++ TPLs.

***False Version Analysis of* DEEPERBIN.** Although DEEPERBIN uses version string regexes to assist in version identification, it fails when version strings are absent in the binary. We identify two key scenarios: **(1) *Missing Version Strings in Source Code* (104/196 = 53.1% of cases).** Many C/C++ TPLs do not embed version information in their source code, leaving binaries without identifiable version strings. Despite DEEPERBIN's extensive regex collection, it covers only 31,855 of the known C/C++ TPLs. **(2) *Version Strings Omitted in Reused Code* (92/196 = 46.9% of cases).** Even when version strings exist in the full source, they may be excluded from the reused portions of the binary. This frequently occurs in *fused binaries*, where regex-based detection fails because version strings are absent in the partially reused code.

False Version Analysis of Baselines. We conclude two primary causes of false versions reported by the baselines. **(1) *Incomplete Version Coverage* (159/376=42.3% of cases).** The fragmented C/C++ ecosystem prevents any single repository from maintaining complete historical versions. When the true version (e.g., `llhttp-6.0.6`) is absent from the feature database, tools default to the nearest available version (e.g., `llhttp-6.0.9` in LIBDB's *Fedora*-based detection of `_http_parser.cpython-37m-x86_64-linux-gnu.so` from `aiohttp` [45]). **(2) *Minor Feature Variations Across Adjacent Versions* (217/376=57.7% of cases).** The feature matching approach proves inadequate for distinguishing adjacent versions exhibiting minor feature divergence. This is exemplified in BINARYAI's analysis of `OpenCV-4.5.4` binaries from `daisykit` [46], where the tool's GitHub-derived feature database generated 13 false version matches—including historically inconsistent versions (`3.4.19`, `3.4.2`, and `4.7.0`)—due to insufficient sensitivity to minor variations between sequential releases.

> ***Conclusion:*** DEEPERBIN *establishes new SOTA performance with 70.3% version identification precision, surpassing all baselines by 12.6-39.3% across diverse tool methodologies.*

TABLE VII: Detection Performance on Various FFI Bindings (RQ3)

FFI Tool	*Boost.py*	*pybind11*	*Python/C*	*Cython*	*CFFI*	*SIP*	*SWIG*	*JNI*
#Projects	3	10	14	14	3	3	5	51
#Bindings	65	82	41	87	49	90	110	95
Precision	84.2%	84.1%	84.2%	80.3%	85.7%	85.2%	80.7%	80.2%
Recall	84.2%	80.2%	80.0%	85.7%	80.0%	85.2%	85.2%	85.8%

TABLE VIII: Detection Performance on Fused Binaries (RQ3)

	Fused Binary			*Non-fused*
	#TPLs (2-4)	*#TPLs (5-9)*	*#TPLs (10-19)*	*Binary*
#Binaries	72	24	10	312
#TPLs	192	173	175	312
Precision	82.9%	80.4%	78.1%	82.4%
Recall	83.3%	81.8%	79.4%	90.8%

E. RQ3: Ablation Study

Study Methodology. To evaluate DEEPERBIN's performance in filtering diverse FFI bindings and analyzing complex fused binaries, our ground-truth dataset includes FFI bindings generated by most mainstream FFI tools (41–110 per project on average) and identifies 106 fused binaries integrating 540 C/C++ TPLs (about 5 per binary). We further assess the effectiveness of DEEPERBIN's three core modules—*Filtering Diverse FFI Bindings*, *Matching Version Strings*, and *Identifying MDL-based Distinctive Features*—by analyzing the impact of these three modules on effectiveness metrics by: (1) incrementally integrating each module into SOTA BSCA techniques, and (2) ablating them from DEEPERBIN. Our ablation study is conducted on *Dataset-Libs* and *Dataset-Vers* datasets to quantify the contributions of these three core modules to DEEPERBIN 's effectiveness in TPL detection and version identification.

Results. As shown in Table VII, **DEEPERBIN** ***consistently achieved high detection accuracy for binaries containing various FFI binding types***, with *precision* ranging 80.2%-85.7% and *recall* 80.0%-85.8%. DEEPERBIN is particularly pronounced for bindings generated by *Cython*, *SWIG*, and *JNI*, which follow consistent naming patterns, but remains more challenging for *Boost.Python*, *Python/C API*, and *pybind11*, where manually written code introduces greater variability.

Table VIII shows that **DEEPERBIN** ***maintains high precision (78.1%–82.9%) and recall (79.4%–83.3%) across fused binaries of varying complexity***, where higher numbers of integrated C/C++ TPLs indicate greater complexity. Recall is lower than on non-fused binaries, largely because partially reused TPLs may be missed under a fixed similarity threshold.

As shown in Table IX, ***filtering out FFI bindings from the feature database significantly improves the F1-score for all evaluated methods.*** For tools that support partial TPL reuse detection, removing FFI bindings reduces false positives and improves precision—LIBDB and DEEPERBIN shown *Precision* improvements of 17.1% and 12.0%, respectively. For tools that assume full TPL reuse, such as LIBRARIAN, filtering out FFI bindings increases the similarity score of feature matching and helps reduce false negatives, resulting in an 18.1% increase in *Recall*.

The evaluation results demonstrate that matching version strings for TPL version identification significantly improves precision′. DEEPERBIN achieved precision′ improvements of 59.5%. This demonstrates that version regexes automatically extracted by DEEPERBIN can accurately match version from binaries. By incorporating distinctive features, LIBDB and DEEPERBIN achieved precision′ improvements of 9.2% and 10.8%, respectively. This demonstrates that the *MDL-based distinctive features* effectively capture the core characteristics that uniquely identify a given TPL version. By leveraging these discriminative features, DEEPERBIN accurately identifies TPL versions even when binaries lack explicit version information and traditional regex-based string matching fails, significantly improving version identification precision.

TABLE IX: Comparison Results of Ablation Study (RQ3)

Method	*Precision*	*Recall*	*F1-score*	*Precision′*
LIBRARIAN′	57.6%	11.5%	0.19	-
LIBRARIAN′ *+ filtering*	59.3%	**29.6%** ⇑	**0.39**	-
LIBDB	48.5%	34.6%	0.40	43.4%
LIBDB *+ filtering*	**65.6%** ⇑	35.1%	**0.46**	-
LIBDB *+ distinctive*	-	-	-	**52.6%** ⇑
DEEPERBIN	81.2%	84.6%	0.83	70.3%
DEEPERBIN *–filtering*	**69.2%** ⇓	83.4%	**0.76**	-
DEEPERBIN *–version regex*	-	-	-	**10.8%** ⇓
DEEPERBIN *–distinctive*	-	-	-	**59.5%** ⇓

'-' denotes the metric is inapplicable for the integration/ablation experiment.

Conclusion: *Ablation results demonstrate that filtering FFI bindings can significantly enhance* ***TPL detection performance****.* LIBRARIAN *achieved an 18.1% increase in Recall.* LIBDB *and* DEEPERBIN *achieved 17.1% and 12.0% increases in Precision. Moreover, incorporating distinctive features improves* ***version identification performance****, with* LIBDB *and* DEEPERBIN *achieving 9.2% and 10.8% increases in Precision′ metric.*

F. RQ4: Efficiency of DEEPERBIN

Study Methodology. To evaluate the efficiency of DEEPERBIN, we measure its time cost on *Dataset-Vers*. The total execution time of BSCA techniques comprises two phases: (1) ***Feature Extraction***, which reflects the efficiency of extracting features from target binaries, and (2) ***TPL Version Identification***, covering both TPL detection and version identification. Notably, BINARYAI is a commercial tool, so we report its combined feature extraction and detection time without separation. We excluded LIBVDIFF due to the absence of TPL detection time, which limits comparability with other baselines. Given the strong correlation between binary size and processing time, we report size-normalized average execution times per binary.

Results. Table X shows the comparison result of time cost. DEEPERBIN and LIBRARIAN achieved the shortest feature extraction times—4.4s and 4.2s, respectively—as both extract platform-independent binary features directly from ELF sections. In contrast, B2SFINDER and LIBDB require disassembling binaries and constructing ASTs or FCGs, which is significantly more time-consuming.

For TPL version identification, the size of the feature database directly affects efficiency—the larger the database, the longer the detection time. DEEPERBIN maintains the largest feature database (74,647 TPL versions) yet achieves efficient detection in merely 359.9 seconds. This performance comes from: (1) a set-based feature representation that simplifies matching, and (2) a lightweight comparison algorithm that

TABLE X: Comparison Results of Time Cost (RQ4)

Baseline	*Feature Extraction(s)*	*TPL Version Identification(s)*	*Total(s)*
B2SFinder′	50.7	8800.2	8850.9
LibRARIAN′	4.2	736.1	739.3
LibDB	94.7	183.5	278.2
BinaryAI	-	-	707.1
DeeperBin	**4.4**	**359.9**	**364.3**

'-' means the commercial BinaryAI doesn't provide separate timing data for feature generation and TPL version identification.

speeds up detection without losing accuracy. In comparison, LibDB completes detection in 278.2s with a smaller database (25,000 TPL versions). B2SFinder and LibRARIAN show a notable drop in efficiency when scaling to larger databases.

In terms of total analysis time, DeeperBin significantly outperforms BinaryAI. This advantage stems primarily from DeeperBin's ability to process binaries of unlimited size, whereas BinaryAI consistently fails to analyze files exceeding 20MB, frequently resulting in timeout errors.

Conclusion: DeeperBin *achieves competitive efficiency with the shortest feature extraction time (4.4s) and outperforms baselines in total time cost (364.3s) while supporting the largest feature database.*

V. Discussions

Limitations. DeeperBin currently faces two key limitations: (1) its FFI binding analysis is currently limited to Python/Java-C/C++ interactions, lacking support for other high-level languages (Go/Rust/JavaScript) which may compromise TPL detection accuracy in general polyglot environments; and (2) scalability issues emerge as the growing feature database for tracking C/C++ TPL versions renders the current full-pairwise comparison method computationally expensive, necessitating future optimizations through indexing or approximate similarity techniques.

Threats to Validity. We identify two main threats: ***(1) Manual Inspection Subjectivity*** - While using manually verified C/C++ dependencies as ground truth introduces potential bias, we mitigated this through: (a) systematic open coding [41], (b) triple-author verification, and (c) consensus-based discrepancy resolution. ***(2) Threshold Enhancement*** - While the *file proportion* threshold mitigates many false positives, some remain, suggesting the incorporation of additional evidence for judging the prime TPL to further enhance detection accuracy.

VI. Related Works

Software Composition Analysis. Prior SCA research has developed diverse methods for source and binary-level detection. Source approaches excel at large-scale analysis: SourcererCC [47] pioneered efficient clone detection via optimized indexing, later extended by DéjàVu [48] for repository-wide analysis. Centris [49] improved accuracy using temporal function signatures. For binaries, B2SFinder [50] introduced weighted control-flow analysis while ModX [51] employed modular semantic clustering. Because ModX's implementation remains proprietary, we were unable to reproduce its results and consequently could not include it as a baseline comparision. Obfuscation-resistant techniques include Xu et al.'s multi-tier birthmarks [52] and LibDB's hybrid approach [53] combining syntactic embeddings with call graphs. System-level innovations feature OSSPolice's hierarchical indexing [17] and TPLite's dual-path analysis [54].

Recent SCA advances have expanded into Android ecosystems, with LibScout [55] enhancing obfuscation resilience through class hierarchy analysis, LibID [56] utilizing class signatures with triple-stage verification, and ATVHunter [57] combining control-flow and opcode analysis. While Insight [11] investigates cross-language vulnerability propagation from C/C++ to Java/Python, it directly adopts LibRARIAN [13] for binary analysis without optimization. In this paper, we propose DeeperBin, a novel BSCA technique that accurately detects reused C/C++ TPLs and their versions in cross-language binaries by constructing a comprehensive feature database specifically designed to address polyglot project challenges.

Library Version Identification. Most existing approaches for OSS version identification rely on version strings in target binaries. For instance, Ves [19] identifies version strings through pattern matching and extracts version numbers via control/data flow analysis. Similarly, Pandora [58] leverages version strings but focuses specifically on IoT firmware in embedded environments, unlike our Linux-oriented C/C++ binary analysis. Other approaches, such as OSSPolice [17] and LibRARIAN [13], generate regex patterns for version matching. However, their heuristic methods cover only 15 and 126 TPLs, respectively, and fail when version strings are absent. Recent work explores diverse binary and source code features. B2SFinder [59] employs seven cross-format code features, while LibDB [60] and FirmSec [61] combine basic features with call graph matching for version identification.

Our study found that Debian's configuration files named `WATCH`, contain version regexes for tracking upstream updates. DeeperBin automatically extracts these patterns from `WATCH` files through an advanced parsing algorithm, obtaining 31,855 regexes covering 42.7% of TPL versions - a significant improvement over prior work [13], [17]. For cases where version string regexes cannot be used for version identification, DeeperBin confirms the version by identifying the distinctive feature set that characterizes the TPL version, achieving state-of-the-art precision in our evaluation.

VII. Conclusion

We present DeeperBin, an advanced BSCA technique for detecting cross-language C/C++ TPLs that tackles FFI-binding interference and fused binaries using a high-quality feature database with 74,647 TPL versions, noise filtering, automated version regexes (covering 31,855 TPL versions), and MDL-based features. DeeperBin achieves 81.2% precision/84.6% recall (TPL detection) and 70.3% version precision, outperforming SOTA tools while remaining efficient. The solution advances BSCA for polyglot ecosystems where vulnerabilities cross-language boundaries.

Acknowledgments

This work was supported by the National Key Research and Development Program of China (Grant No. 2024YFF0908000) and the China Postdoctoral Science Foundation (Grant No. 2024M750375).

REFERENCES

[1] "binary-sca-what-why," https://blog.binare.io/binary-sca-what-why/, 2025, accessed: 2025-03-01.

[2] "binary-software-composition-analysis," https://finitestate.io/blog/binary-software-composition-analysis-sca/, 2025, accessed: 2025-03-01.

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

[4] Z. Yuan, M. Feng, F. Li, G. Ban, Y. Xiao, S. Wang, Q. Tang, H. Su, C. Yu, J. Xu *et al.*, "B2sfinder: Detecting open-source software reuse in cots software," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1038–1049.

[5] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, "Finding software license violations through binary code clone detection," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 63–72.

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

[7] S. Woo, S. Park, S. Kim, H. Lee, and H. Oh, "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, 2021, pp. 860–872.

[8] J. Wu, Z. Xu, W. Tang, L. Zhang, Y. Wu, C. Liu, K. Sun, L. Zhao, and Y. Liu, "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, 2023, pp. 270–282.

[9] L. Jiang, H. Yuan, Q. Tang, S. Nie, S. Wu, and Y. Zhang, "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*, 2023, pp. 1383–1395.

[10] M. Sun and G. Tan, "Nativeguard: Protecting android applications from third-party native libraries," in *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, 2014, pp. 165–176.

[11] M. Xu, Y. Wang, S.-C. Cheung, H. Yu, and Z. Zhu, "Insight: Exploring cross-ecosystem vulnerability impacts," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[12] J. Cao, F. Guo, and Y. Qu, "Jnfuzz-droid: a lightweight fuzzing and taint analysis framework for native code of android applications," *Empirical Software Engineering*, vol. 30, no. 5, pp. 1–38, 2025.

[13] S. Almanee, A. Ünal, M. Payer, and J. Garcia, "Too quiet in the library: An empirical study of security updates in android apps' native code," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1347–1359.

[14] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "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*, 2017, pp. 2169–2185.

[15] W. Tang, P. Luo, J. Fu, and D. Zhang, "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, 2020, pp. 104–115.

[16] W. Tang, Y. Wang, H. Zhang, S. Han, P. Luo, and D. Zhang, "Libdb: An effective and efficient framework for detecting third-party libraries in binaries," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 423–434.

[17] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "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*, 2017, pp. 2169–2185.

[18] "Pysupertuxkart," https://pypi.org/project/PySuperTuxKart/, 2025, accessed: 2025-03-01.

[19] X. Hu, W. Zhang, H. Li, Y. Hu, Z. Yan, X. Wang, and L. Sun, "Ves: A component version extracting system for large-scale iot firmwares," in *Wireless Algorithms, Systems, and Applications: 15th International Conference, WASA 2020, Qingdao, China, September 13–15, 2020, Proceedings, Part II 15*. Springer, 2020, pp. 39–48.

[20] "Debian," https://www.debian.org/distrib/packages/, 2025, accessed: 2025-03-01.

[21] "Fedora," https://packages.fedoraproject.org/, 2025, accessed: 2025-03-01.

[22] "Openeuler," https://oepkgs.net/, 2025, accessed: 2025-03-01.

[23] "unix-ar," https://pypi.org/project/unix-ar/, 2025, accessed: 2025-03-01.

[24] A. Hemel, "Using elf symbols extracted from dynamically linked elf binaries for fingerprinting," 2021.

[25] "Angr," https://github.com/angr/angr/, 2025, accessed: 2025-03-01.

[26] "Watch file," https://manpages.debian.org/testing/devscripts/uscan.1.en.html#FORMAT_OF_THE_WATCH_FILE/, 2025, accessed: 2025-03-01.

[27] "Boost.python," https://github.com/boostorg/python/, 2025, accessed: 2025-03-01.

[28] "pybind11," https://github.com/pybind/pybind11/, 2025, accessed: 2025-03-01.

[29] "Python/c api," https://docs.python.org/3/c-api/concrete.html/, 2025, accessed: 2025-03-01.

[30] "Cython," https://github.com/cython/cython/, 2025, accessed: 2025-03-01.

[31] "Cffi," https://github.com/cffi/cffi/, 2025, accessed: 2025-03-01.

[32] "Sip," https://github.com/Python-SIP/sip/, 2025, accessed: 2025-03-01.

[33] "Swig," https://www.swig.org/, 2025, accessed: 2025-03-01.

[34] "Jni," https://docs.oracle.com/en/java/javase/23/docs/specs/jni/types.html/, 2025, accessed: 2025-03-01.

[35] K. Yamanishi, *Learning with the minimum description length principle*. Springer, 2023.

[36] "Pypisample," https://pypi.org/project/pypi-sample/, 2025, accessed: 2025-03-01.

[37] "Maven-indexer," https://maven.apache.org/maven-indexer-archives/maven-indexer-LATEST/, 2025, accessed: 2025-03-01.

[38] "pyelftools," https://github.com/eliben/pyelftools/, 2025, accessed: 2025-03-01.

[39] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.

[40] C. Dong, S. Li, S. Yang, Y. Xiao, Y. Wang, H. Li, Z. Li, and L. Sun, "Libvdiff: Library version difference guided oss version identification in binaries," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.

[41] C. Jw, "Qualitative inquiry and research design," *Choosing among five traditions*, 1998.

[42] "hgdb," https://github.com/Kuree/hgdb/, 2025, accessed: 2025-03-01.

[43] "vlstd," https://github.com/Kuree/vlstd/, 2025, accessed: 2025-03-01.

[44] J. Zhang, A. R. Beresford, and S. A. Kollmann, "Libid: reliable identification of obfuscated third-party android libraries," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 55–65.

[45] "aiohttp," https://github.com/aio-libs/aiohttp/, 2025, accessed: 2025-03-01.

[46] "daisykit," https://github.com/MDuc-ai/daisykit/, 2025, accessed: 2025-03-01.

[47] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 1157–1168.

[48] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "Déjàvu: a map of code duplicates on github," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, 2017.

[49] S. Woo, S. Park, S. Kim, H. Lee, and H. Oh, "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, 2021, pp. 860–872.

[50] Z. Yuan, M. Feng, F. Li, G. Ban, Y. Xiao, S. Wang, Q. Tang, H. Su, C. Yu, J. Xu *et al.*, "B2sfinder: Detecting open-source software reuse in cots software," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1038–1049.

[51] C. Yang, Z. Xu, H. Chen, Y. Liu, X. Gong, and B. Liu, "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*, 2022, pp. 1393–1405.

[52] X. Xu, Q. Zheng, Z. Yan, M. Fan, A. Jia, and T. Liu, "Interpretation-enabled software reuse detection based on a multi-level birthmark model," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 873–884.

[53] W. Tang, Y. Wang, H. Zhang, S. Han, P. Luo, and D. Zhang, "Libdb: An effective and efficient framework for detecting third-party libraries in binaries," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 423–434.
[54] L. Jiang, H. Yuan, Q. Tang, S. Nie, S. Wu, and Y. Zhang, "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*, 2023, pp. 1383–1395.
[55] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 356–367.
[56] J. Zhang, A. R. Beresford, and S. A. Kollmann, "Libid: reliable identification of obfuscated third-party android libraries," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 55–65.
[57] X. Zhan, L. Fan, S. Chen, F. We, T. Liu, X. Luo, and Y. Liu, "Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1695–1707.
[58] W. Zhang, Y. Chen, H. Li, Z. Li, and L. Sun, "Pandora: A scalable and efficient scheme to extract version of binaries in iot firmwares," in *2018 IEEE International Conference on Communications (ICC)*. IEEE, 2018, pp. 1–6.
[59] Z. Yuan, M. Feng, F. Li, G. Ban, Y. Xiao, S. Wang, Q. Tang, H. Su, C. Yu, J. Xu *et al.*, "B2sfinder: Detecting open-source software reuse in cots software," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1038–1049.
[60] W. Tang, Y. Wang, H. Zhang, S. Han, P. Luo, and D. Zhang, "Libdb: An effective and efficient framework for detecting third-party libraries in binaries," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 423–434.
[61] B. Zhao, S. Ji, J. Xu, Y. Tian, Q. Wei, Q. Wang, C. Lyu, X. Zhang, C. Lin, J. Wu *et al.*, "A large-scale empirical analysis of the vulnerabilities introduced by third-party components in iot firmware," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 442–454.