Are We There Yet? Filling the Gap Between Binary Similarity Analysis and Binary Software Composition Analysis

Huaijin Wang, Zhibo Liu†, Shuai Wang†
Hong Kong University of Science and Technology
{*hwangdz, zliudc, shuaiw*}*@cse.ust.hk*

Ying Wang
Northeastern University (China)
wangying@swc.neu.edu.cn

Qiyi Tang, Sen Nie, Shi Wu
Keen Security Lab, Tencent
{*dodgetang, snie, shiwu*}*@tencent.com*

***Abstract*—Software composition analysis (SCA) has attracted the attention of the industry and academic community in recent years. Given a piece of program source code, SCA facilitates extracting certain components from the input program and matching the extracted components with open-source software (OSS) libraries. Despite the prosperous development of SCA, binary SCA (BSCA) is highly challenging and still underdeveloped. Few available BSCA solutions are either closed source (for commercial usage) or suffer from low performance. Nevertheless, a related line of research, binary similarity analysis (BSA), which decides the similarity between two pieces of binary code, has been progressively developed in academia for decades. De facto BSA techniques, often based on deep learning, efficiently analyze large-scale executables with high accuracy.**

This study explores bridging the gap between state-of-the-art (SOTA) BSA and BSCA. We spent considerable manual effort building the first large real-world benchmark dataset, containing over 55 million lines of C/C++ code. Then, we establish our BSCA pipeline by extending and calibrating the SOTA SCA pipeline. Particularly, we concretize the key procedure of BSCA, namely matching a binary component with OSS using six SOTA BSA techniques. Evaluation using our benchmark dataset reveals that simply employing BSA in BSCA exhibits less desirable accuracy, as BSCA faces unique challenges. After inspecting the failed cases, we propose three enhancements whose hybrid usage improves the F1 score of BSCA by over 30% and outperforms SOTA commercial BSCA software. Our experiment on 1-day vulnerability detection demonstrates our BSCA framework's effectiveness. We also discuss several open challenges and potential solutions to augment BSCA solutions.

†

1. Introduction

Software composition analysis (SCA) identifies the potential use of third-party and open-source software (OSS) projects in a given software. Adopting SCA enables localizing potentially vulnerable or outdated OSS projects, reducing risk factors, ensuring license compliance, and promoting healthy open-source usage. To date, production SCA tools are offered by the industry [6, 8, 9, 24, 26], and many research works [34, 41, 42, 68, 69, 75, 83, 90, 94, 101, 104, 105] have been published in recent years. Many of them [8, 24, 26, 83, 94] require the source code, precise package structures [34, 75, 105], class/methods declarations [34, 68, 69, 104, 105], and manifest files [101] for OSS & version identification.

Despite the successful adoption of SCA techniques, software's source code is *not* always available in many real-world, security-sensitive scenarios. In fact, *binary code* is the primary information available when performing SCA over commercial off-the-shelf (COTS) software. With this regard, binary software composition analysis (BSCA) has become a demanding yet under-explored field [6, 9].

On the other hand, the industry and the security community have achieved remarkable progress in binary similarity analysis (BSA) [47, 49, 77, 95] in recent years. BSA quantifies the similarity between two pieces of binary code, which forms the basis of various software security and software (re-)engineering applications. For example, BSA promotes malware analysis by comparing suspicious code with known malware families to determine its potential maliciousness [50, 59]. BSA also helps to discover code clones and algorithm plagiarism in executables [74, 102].

Motivation. From a holistic view, BSA and BSCA share conceptually similar technical demand about "binary code matching." With over twenty years of development [35, 55], BSA has been enhanced from various perspectives using syntactic-, structure-, and semantics-based methods. To date, advanced BSA techniques extensively use deep neural networks (DNNs), including representative learning and large language models (LLMs)-based embedding [47, 70, 77, 98, 99]. In contrast, BSCA, as an emerging security demand, has not been well-explored by academia. Contemporary BSCA works [48, 56, 100] mostly rely on string literals and exported symbols. However, string literals can be easily changed, and exported symbols are not always available. We believe they cannot be considered the best practice of BSCA.

Nevertheless, given the conceptual similarity of BSA and BSCA, one natural idea is to use well-developed BSA techniques to solve BSCA. Since BSCA serves as the cornerstone for many security applications and risk assessments, analysis errors are particularly unwanted, which could substantially affect the trustworthiness of today's software security landscape. Thus, it is intriguing and urgent to know if de facto BSA solutions can be extended to the demanding BSCA field to achieve high accuracy. This study is thus motivated to provide a systematic understanding of BSCA and explore the feasibility of BSA-based BSCA solutions.

†Corresponding authors.

This work conducts the first comprehensive study to analyze de facto BSA techniques' capability to support BSCA. We spend great manual effort (one senior Ph.D. student and two security engineers for about a month) to form the first large real-world dataset for BSCA benchmarking. Our dataset encompasses 35 complex real-world executables (including Chrono Physics Engine [21] and a digital payment protocol, Nano [18]), where each executable reuses 12 production OSS projects on average. In total, 255 OSS projects are engaged across two platforms (Windows and Linux) and three compilation toolchains (`gcc`, `clang`, and `MSVC`). We extend six SOTA BSA techniques for BSCA over our dataset. We also compare with commercial SCA/B-SCA tools to understand "how far" our BSA-empowered BSCA pipeline can achieve. A vulnerability identification experiment further demonstrates the usefulness of our BSCA framework for software security.

We assess BSA tools' support for BSCA and summarize key findings. We find that BSCA has distinct requirements from standard BSA. The *overemphasis* on precisely program semantics extraction might be less necessary in the BSCA scenario. Syntactic and control-flow graph (CFG) features are often *sufficient* to attribute a function to third-party OSS libraries. Despite this, identifying exact OSS library versions is still challenging, even with advanced BSA techniques. We further summarize lessons and guidance for calibrating and extending the current BSA techniques for BSCA. Following, we design three low-cost and highly effective enhancement strategies to enhance BSA for BSCA from different perspectives. By applying these strategies, we can vastly enhance BSA tools for BSCA (increasing F1 score by over 30%), *exceeding the SOTA commercial BSCA solution*. In sum, we make the following contributions:

- We conduct the first comprehensive study in bridging BSA techniques to BSCA, a highly demanding yet under-explored application field. We detail a practical BSCA pipeline based on de facto BSA solutions, which shows excellent feasibility via experimental results.
- We form the first *large real-world* and *version-representative* BSCA benchmark dataset containing over 55 million lines of C/C++ code to assess the support of advanced BSA techniques for BSCA. We benchmark six SOTA, "out-of-the-box" BSA tools in their support for BSCA, summarizing findings that can be leveraged as guidance to boost BSA-based BSCA.
- Based on our findings, we design three enhancement strategies that can significantly enhance the effectiveness of BSA techniques in BSCA. According to our evaluation, our research prototype outperforms the SOTA BSCA tools and effectively detects potential 1-day vulnerabilities.

Artifact Availability. Our artifact is available at our website [7].

2. Preliminaries

2.1. Software Composition Analysis (SCA)

2.1.1. SCA Overview. OSS is frequently reused to facilitate the fast development of applications. However, extensive OSS reuse gives attackers opportunities to exploit known vulnerabilities, which may cause severe security issues for software users. On the other hand, developers may not know which OSS is assembled in their software because of many indirect and transitive dependencies. Thus, SCA [29, 32] has become a common practice that helps developers quickly track and analyze any OSS brought into a project. Then, developers can address security risks from known vulnerabilities and avoid unauthorized software usage. Formally, let $C = F(S, D)$ be the SCA procedure, where S denotes the input software, and D denotes databases maintained by the SCA service provider. The output C represents a list of uncovered *software compositions* in S. Each element $c \in C$ forms a 2-tuple (l, v), where l represents an OSS (e.g., a library) on the market, and v denotes valuable information concerned by the users. Given that many SCA tasks are for vulnerability detection, v usually represents CVE vulnerabilities, e.g., Heartbleed [87] and Log4j [39].

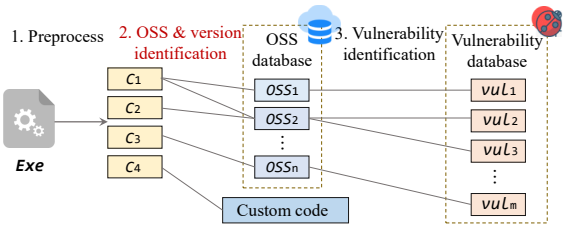


Figure 1. SCA workflow. In the context of BSCA, this paper primarily explores how to concrete the second phase, OSS & version identification, with various BSA techniques.

Fig. 1 depicts the high-level workflow of SCA. Note that this workflow subsumes SCA's different scenarios, where the input software S could be open-source programs, Android APKs, or binary executables (i.e., BSCA). Overall, typical SCA analysis comprises the following three phases, where one phase's output serves as the next phase's input.

① **Component Dissection.** First, the input software S needs to be pre-processed and dissected into a list of code components $S = [c_1, c_2, \cdots, c_n]$. Dissected components will be used for later component identification. However, choosing a suitable code structure unit as the "component" is not as straightforward as it looks. To date, we have seen techniques proposed to dissect S in terms of different hierarchical structures of the software. For instance, in typical Android APKs, the package structure that can reflect rich information on third-party libraries is often leveraged by Android SCA works to dissect Android apps [34, 69, 75, 104, 105]. Besides, source code files, class hierarchies (particularly for object-oriented languages), and functions are also commonly used as components in SOTA SCA works [34, 94, 101].

② **OSS & Version Identification.** Let an OSS database be D_{oss}, where each $\text{oss}_i \in D_{\text{oss}}$ is an OSS project with its version information specified. For each component $c_j \in S$, we need to decide if it originates from any $\text{oss}_i \in D_{\text{oss}}$. This is the *key challenge* for SCA, in the sense that we need to decide the similarity between c_j and records in D_{oss}. Today, most existing SCA works aim at enhancing the accuracy of matching c_j with oss_i. Ideally, each c_j from OSS projects is correctly matched to its corresponding OSS with correct version. A component c_j, however, should not be matched to any $\text{oss}_i \in D_{\text{oss}}$ if it is user-written code, which is referred to as "custom code" in Fig. 1.

③ **1-day Vulnerability Identification.** Once a set of reused OSS projects $L = \{l_1, l_2, \cdots, l_n\}$ has been detected in S, the next step would be exploring the vulnerability v_i within $l_i \in L$, where l_i denotes a specific version of an OSS library with known vulnerabilities [41].

Prior works [34, 48, 69, 75, 94, 101, 104] often assume the availability of mapping between vulnerability and OSS. In Sec. 8, with a vulnerability dataset from previous works [44, 47, 91], we demonstrate the effectiveness of BSCA for detecting 1-day vulnerabilities.

2.1.2. SCA Metrics. Existing works [83, 88, 94, 101, 104] primarily benchmark SCA with standard metrics, including precision, recall, and F1 score. Let the ground truth be R^*, which denotes the OSS projects reused in software S, and those detected by an SCA tool in S be R. Then, true positives are $TP = R \cap R^*$, and precision, recall, and F1 scores are computed by $\frac{|TP|}{|R|}$, $\frac{|TP|}{|R^*|}$, and $\frac{2 \times precision \times recall}{precision + recall}$, respectively. With high precisions, SCA tools will warn of vulnerabilities with high confidence, and they are also expected to have high recalls to avoid missing potential vulnerabilities.

2.1.3. BSCA. BSCA [6, 9] denotes a demanding sub-area of SCA. BSCA aims to conduct SCA over executables compiled from the *C/C++ programs*, assuming the source code is unavailable (particularly in security-related applications, such as legacy code analyzing). Meanwhile, the security community [58] also champions the necessity of directly analyzing the executables to identify all OSS. However, existing SCA solutions for analyzing source code and Android APKs may be inapplicable to C/C++ executables. Precisely extracting features like variable names and types from executables is difficult since compilers always strip such symbols and change the code structure (e.g., CFG) with optimizations. Hence, many source-based SCA techniques are inapplicable. The following section compares analyzing Android APKs and C/C++ executables.

Android APKs vs. C/C++ Executables. To clarify, SCA for Android APKs also frequently targets low-level code in executable format. Nevertheless, reverse engineering Android apps is deemed *much simpler* than decompiling x86 executables compiled from C/C++ code. Accordingly, more information is available for Android APKs SCA. When performing SCA for Android APKs [34, 69, 75, 101, 104, 105], various well-established reverse engineering tools. including APKTool [3], dex2jar [12], and Androguard [1], are employed to recover the package structures, method prototypes, and class inheritance dependencies. Decompiling C/C++ executables is much more challenging, and recovering the high-level information (e.g., class inheritance dependencies) is not well addressed yet [84]. Performing BSCA on C/C++ executables has its unique challenge, and we launch the first in-depth study to explore feasible technical solutions.

2.2. Binary Similarity Analysis (BSA)

Many research works about BSA are emerging in the security and software engineering communities [55]. Overall, given two pieces of binary code, BSA decides how similar they are with a score.

B2B and B2S. Based on the types of targets being matched, we classify BSA into 1) *binary-to-binary* similarity analysis (**B2B**) and 2) *binary-to-source* similarity analysis (**B2S**). The research community appears to pay much more attention to B2B than B2S. A recent survey [55] investigated 70 binary code similarity approaches, in which merely six tools [48, 54, 56, 99, 100, 103] aiming at B2S.

Most B2B works perform semantics-aware search for function-level assembly code [37, 47, 55, 70, 95, 98, 106]. They aim to determine the semantical similarity of two functions in binary code, even if they show distinct low-level representations due to different compilers and optimizations. The usage of B2B tools is similar to search engines. Given an input assembly function f, the B2B search engines retrieve and rank the top-k similar functions by their similarities from a repository of assembly functions RP. Similarly, the function-level B2S works [60, 99] are evaluated by replacing the assembly functions in the RP with source code functions.

B2B for BSCA. Though B2B works appear to be the mainstream solution for BSA [55], using B2B for BSCA faces a unique challenge: we must compile OSS projects into binary code to enable existing B2B techniques. In production, BSCA is envisioned to analyze hundreds of OSS projects with various dependencies and compilation toolchains/configurations. Therefore, compiling all OSS projects requires enormous resources and considerable manual effort since most OSS projects cannot be automatically tuned to use different compilation toolchains. Some OSS projects available online (e.g., RapidJSON [22]) do not even provide a `Makefile`. Additionally, OSS can upgrade and change; compilation may not be a one-time effort.

Immaturity of B2S. B2S is much less explored than B2B. In general, B2S faces the cross-modality challenge (i.e., accurately pairing source code and binary code with similar semantics), which is inherently formidable. In contrast, B2B avoids such a problem by only comparing more "regularized" low-level binary code. To overcome the cross-modality challenge, recent B2S papers [56, 100] propose to extract features (e.g., numbers and strings) remaining unchanged after compilation while ignoring program semantics. There are also B2S studies [54, 60] converting assembly and source code into the intermediate representation (IR) for matching; however, compiling OSS source code into IR faces the same difficulties as B2B techniques. Nevertheless, while using B2S for BSCA is less studied, we view it as having great potential (since the labor of compiling large OSS projects is avoided) and advocate for more attention. In this study, we assess `CodeCMR` [99], the SOTA B2S tool developed by the industry (see Sec. 4.2).

3. BSCA Technical Pipeline

Motivation. There is a high demand for accurate and dependable BSCA, envisioning the practical need to analyze closed-source software and track any (unsafe or outdated) open-source component brought into an executable. Given that said, it is still *unclear* about the best practice for performing BSCA in real-world scenarios. The industry-leading security vendors are promoting their BSCA solutions, including CodeSentry, offered by GrammaTech [9]; Black Duck, offered by Synopsys [6]; and Scantist [31]. However, *none* of these commercial tools disclose their technical solutions in detail.

In short, we believe that the academia and our community lack a systematic and in-depth understanding to calibrate the technical solution and uncover the best practice for performing BSCA. This motivates our study.

As reviewed in Sec. 2.1, de facto SCA works primarily undertake a three-step approach, with OSS & version identification being the central technical challenge. We now discuss the potential technical solutions for each step.

① **Component Dissection.** C/C++ executables lack high-level program structures. As noted in Sec. 2.1.3, it is generally challenging to recover high-level program structure information accurately. Hence, as a practical setting, this study frames an assembly function as the minimal program structure unit, i.e., a "component" focused on by the SCA pipeline.

We view treating functions as "components" for matching as a practical and beneficial setting, whose reasons are three-fold. First, such a setting follows the convention of many prior SCA tools (e.g., LibD, ATVHunter, and Centris), which rely on function-level components to identify the reused OSS. Second, while recovering source-level information like classes and variable types is challenging, it is generally feasible to obtain program function information (e.g., function entries) precisely, even in stripped binaries. For instance, [79, 82] achieved above 0.95 F1 scores while identifying function entries. By utilizing the patterns of Intel CET (control-flow enforcement technology) instructions, `FuncSeeker` can achieve over 0.99 F1 score. `IDA pro` [25], the de facto disassembler, claims that it can identify standard function calls in binaries compiled by most mainstream compilation toolchains, and numerous studies [40, 47, 49, 51, 66, 99] use it to extract assembly functions. Third, using the function granularity component can ease the burden of extending various existing works to bridge BSA and BSCA since most existing BSA works focus on function-level similarity [55].

② **OSS & Version Identification.** We extensively studied prior SCA works. Particularly, we refer to recently published works, including LibRadar [75], LibD [69], LibID [104], LibScout [34], Centris [94], and ATVHunter [101]. According to our study, de facto SCA methods are primarily similarity-based detections [101].

The similarity-based SCA performs software similarity analysis to match a code component to known OSS projects. Software similarity comparison methods can be performed at different hierarchical representations, including opcode sequences, code structures, and dependency relations. Recent SCA works [94] also use fuzzy hash or machine learning [101] for higher matching accuracy.

We envision that well-developed BSA techniques can be used as mature technical solutions to improve the accuracy of OSS & version identification in the BSCA scenario. Accordingly, as reviewed in Sec. 2.2, we will benchmark both B2B and B2S, as two mainstreams in BSA, on its support of BSCA.

③ **Valuable Information Identification.** Matching successfully identified OSS with valuable information records is mostly unchanged comparing BSCA with SCA. Thus, we consider step ③ to have no extra difficulties: the "valuable information database" [41, 42] adopted by existing SCA solutions can be reused here. In this study, we consider the task of vulnerability identification.

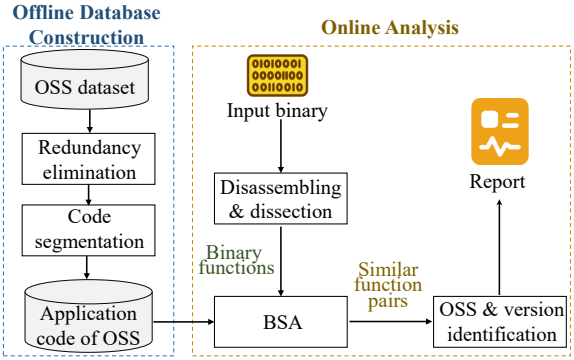


Figure 2. BSCA implementation, including the offline database construction and the online analysis phase (the latter is introduced in Sec. 3).

4. Solving BSCA with BSA

Based on the SOTA source-based SCA work, Centris [94], we propose the BSA-based BSCA workflow in Sec. 4.1. Then, we elaborate on selecting the representative BSA techniques in Sec. 4.2. We also introduce our BSCA benchmarking dataset in Sec. 4.3.

4.1. BSCA Implementation

As illustrated in Fig. 2, there are two primary phases in implementing BSCA: the offline database construction phase and the online analysis phase (the latter phase includes all three steps ① ② ③ introduced in Sec. 3). The offline phase collects and stores OSS projects into a database for query, whereas the online phase iterates over functions in the input executable and queries the database to recognize all reused OSS and the versions. We now discuss implementing these two phases.

Offline Database Construction. While building the OSS database is generally mundane, given its large size, we often need to explore reducing the database to speed up queries. To do so, Centris proposed *redundancy elimination* and *code segmentation* to minimize the database. We extend similar ideas in the BSCA scenario.

version	*functions*	
0.9	*a*	*b*
1.0	*a*	*b*
1.1	*c*	*b*
1.2	*c*	*d*

⇒

function	versions
a	[0.9, 1.0]
b	[0.9, 1.0, 1.1]
c	[1.1, 1.2]
d	[1.2]

Figure 3. Redundancy elimination.

Redundancy Elimination. A common observation is that new versions of OSS libraries are often built on top of their earlier version, leaving many functions unchanged. Hence, storing only one copy of such unchanged functions across all versions is sufficient. Thus, redundancy elimination is proposed to reduce the database size and speed up database queries. As shown in Fig. 3, by saving functions with their relative versions instead of storing each version's functions, the number of functions to be analyzed reduces from 8 to 4. We reuse Centris's implementation for B2S and extend this idea to build a binary OSS database for B2B settings.

Code Segmentation. Observation of real-world OSS shows that an OSS project may borrow code components from

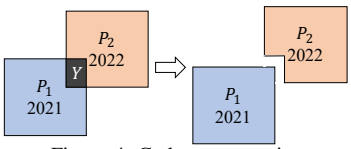


Figure 4. Code segmentation.

other projects. Therefore, segmenting an OSS project's application code can reduce the false positives caused by the borrowed code. Fig. 4 presents an example. Let P_1 and P_2 be two OSS projects, and P_2 reuses some code (i.e., Y) from P_1. Given an input binary reusing P_1, P_2 may be classified as reused due to the existence of the code of Y, which causes a false positive.

The code segmentation cross-compares functions across different OSS projects, and only the functions with an earlier timestamp are kept. In Fig. 4, code segmentation removes Y from P_2 since Y was created before P_2. The remaining code is unique, forming the custom code of P_2. To get functions' creation time, besides using the timestamp of a `git` repository's tag as Centris, we also ensure the reliability of collected timestamps (e.g., zlib [33]) with TPLite [62].

Algorithm 1 OSS identification

```
1: function IDENTIFY_OSS(F_bin, DB_oss, θ)
2:     # F_bin is the set of input functions.
3:     # DB_oss stores all OSS instances.
4:     # θ is the threshold.
5:     R ← set()
6:     for oss ∈ DB_oss do
7:         F_oss ← oss.functions
8:         M ← MATCH_FUNCTIONS(F_bin, F_oss, θ)
9:         p ← IS_REUSED(M, oss)
10:        if p = True then                ▷ oss is identified as reused.
11:            R ← R ∪ {oss}
       return R
12: function MATCH_FUNCTIONS(F_bin, F_oss, θ)
13:    M ← set()
14:    for f ∈ F_bin do
15:        t ← SEARCH_MOST_SIMILIAR(f, F_oss)   ▷ Using Milvus
16:        if SIMILARITY(t, f) > θ then         ▷ f matches t
17:            M ← M ∪ {(t, f)}
       return M
```

Online Analysis. Further to the above steps, we detail the implementation of online BSCA analysis below.

Disassembling & Dissection. Disassemblers turn the input binary into assembly code and extract assembly functions. Disassembling stripped binaries is challenging. Nevertheless, with the development of reverse engineering techniques, existing well-developed disassemblers can identify assembly functions with high precision [79, 82, 97]. This study uses `IDA pro` [25], a widely-used disassembler [40, 47, 49, 51, 99]. After this process, we get F_{bin} as one of the inputs to Alg. 1.

BSA. This process differentiates our BSCA workflow from Centris. Originally, Centris receives the source code of a function and converts the source code function into a hash value with TLSH [81]. TLSH is a locality-sensitive hash library that can reflect the similarity of two functions by the distance of their hash values. If two functions are syntactically similar, their hash values will have a short distance, indicating that one function is likely a copy of the other. Although TLSH effectively detects source code reuse, it is unsuitable for binary code since different compilers and optimizations can generate dramatically different assembly codes with the same source code [47, 70]. Hence, in this study, we replace the TLSH with BSA techniques introduced in Sec. 2.2.

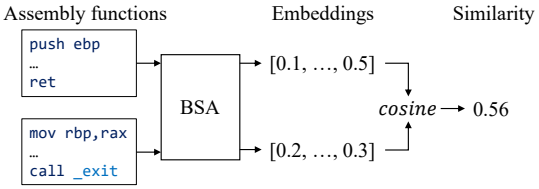


Figure 5. BSA common pipeline.

BSA techniques explored in this study have a common pipeline. As depicted in Fig. 5, given two assembly functions, BSA tools convert them into two embedding vectors. Then the similarity score is computed with these vectors via *cosine similarity*. The B2S tool, `CodeCMR`, can embed the source code functions similarly. The details of how they are selected will be elaborated in Sec. 4.2.

Note that the embedding vector of a function can be used repeatedly. In our implementation, we convert all functions of the OSS database into embedding vectors and store them in `Milvus` [92], a vector database. Given an embedding vector for querying, `Milvus` can efficiently search highly similar vectors in the database.

As shown in MATCH_FUNCTIONS() in Alg. 1, we search the most similar functions in the function set of an OSS (i.e., F_{oss}) for each input function (line 14). The search process is accelerated by the well-developed vector database `Milvus` (line 15). The embedding vectors of F_{oss} are stored in `Milvus`, and the function embedding vectors of an input binary are used as query inputs. Given an input function f, the function t with a similarity score higher than the threshold θ will be deemed similar (line 16). Note that each selected BSA tool has its own θ. We build a hyper-parameter selection dataset to determine the proper θ for each BSA tool (see Sec. 4.3). After iterating all functions of the input binary, we get the matching results M (i.e., the similar function pairs of Fig. 2). Then, we identify whether an OSS is reused and its version with M.

OSS & Version Identification. We decide whether an OSS instance is reused with similar function pairs (line 9). At this step, Centris calculates the proportion of the matched functions M to the entire functions of oss. oss is deemed as reused (i.e., IS_REUSED() returns `true`) if the proportion is higher than a threshold (i.e., 0.1 in Centris). Then, the exact reused OSS version is determined by exploring if certain functions of M belong to a unique version. Details of version identification are elaborated in Sec. 5.3. Sec. 5 uses this approach to benchmark BSA. Moreover, Sec. 6 proposes three enhancements to replace Centris's implementation of IS_REUSED() (line 9) with notably improved BSCA accuracy.

4.2. BSA Tools Selection

A BSA tool searches similar functions in the OSS database with the input functions; then, we can analyze the similar functions to produce the knowledge of used OSS and the specific versions associated. Overall, besides the mundane requirements of *high accuracy* and *available implementation*, the BSA tool must be *highly efficient* since BSCA requires comparing each assembly function of the input binary against every function in the OSS database. The OSS database is extremely large for production usage,

TABLE 1. STUDIED VISIBLE BSA TOOLS.

	Venue	Interal	Tool
SAFE [77]	DIMVA'19	B2B	SAFE [23]
Asm2vec [47]	S&P'19	B2B	asm2vec-pytorch [4]
PalmTree$_G$ [70]	CCS'21	B2B	PalmTree [20] and Gemini [13]
PalmTree$_B$ [70]	CCS'21	B2B	PalmTree [20] and Commercial$_B$
Commercial$_B$		B2B	Commercial$_B$
CodeCMR [99]	NIPS'20	B2S	CodeCMR

with millions of functions from various OSS projects and versions. As a feasible setup, when selecting BSA tools, we require the average pair-wise similarity comparison to take less than 10^{-5} seconds; see Sec. 7 for the cost assessment.

Selection of B2B Tools. With many B2B tools published at top-tier conferences, we first collected 15 works [45, 47, 49, 52, 65, 70, 72, 77, 80, 85, 86, 93, 95, 96, 106] published in the last five years, from top security and software engineering conferences, including USENIX Security, IEEE S&P, NDSS, CCS, ICSE, PLDI, ASE, and DIMVA. After excluding works without publicly-available implementation, we retain seven works, including Gemini [95], VulSeeker [52], SAFE [77], InnerEye [106], Asm2vec [47], DeepBinDiff [49], and PalmTree [70], GMN [76]. In addition, we also select an industrial tool. Commercial$_B$ is a widely-used BSA tool developed by a leading company in the industry, and we blind its name for legal reasons. They all employ machine-learning techniques to measure assembly code similarity. After studying their publications and released code, we eventually select five B2B tools (see their publications in Table 1). Explanations of including or excluding B2B tools are presented in Sec. B.1 of Appendix.

Selection of B2S Tools. As described in Sec. 2.2, the choices of B2S tools are limited (i.e., BAT [56], OSSPolice [48], B2SFinder [100], FIBER [103], CodeCMR [99], BugGraph [60], and XLIR [54]). We eventually select the SOTA B2S work developed by industry, CodeCMR [99]. We also elaborate on the excluded B2S tools in Sec. B.2.

Existing BSCA Solutions. In addition to the analysis of SOTA BSA tools, we also consider three existing BSCA solutions, including an anonymized commercial tool, as the baselines for comparison. Commercial$_A$ is a widely-used commercial BSCA tool developed by a leading company to ensure application security. Commercial$_A$ is generally deemed as the SOTA BSCA solution with presumably the best performance, though its internal implementation is not publicly known. To clarify, soon in our study (as in Sec. 5), we show that de facto BSA tools, when being used for BSCA, are less effective than Commercial$_A$. Nevertheless, we report highly encouraging results that with our optimizations (Sec. 6), BSCA empowered by BSA can surpass the SOTA commercial tool Commercial$_A$.

BAT [56] is a representative research work identifying the reused OSS project with collected string literals. LibDB [88] is a SOTA BSCA tool using function retrieval with BSA techniques (i.e., so-called "function vector channel" in [88]) and basic features like strings and exported function names (i.e., "basic feature channel" in [88]). Its released codebase is incomplete and lacks the code for its basic feature channel. To compare with LibDB, we strictly follow its paper to complete its implementation.

TABLE 2. STATISTICS OF EXECUTABLE IN OUR BENCHMARK DATASET.

Toolchain	*# Binaries*	*# Functions*	*# Reuses*	*Avg. size*
gcc	14	218,773	198	9,386KB
clang	10	151,730	136	7,470KB
MSVC	11	618,846	86	10,182KB
Total	35	989,349	420	9,089KB

4.3. BSCA Dataset Preparation

We introduce our dataset used for BSCA benchmarking here. Overall, one of our key contributions is delivering the first large real-world dataset specifically designed for BSCA.

Necessity of Building the Dataset. While existing datasets are mostly used for BSA scenarios, our tentative exploration shows that those datasets are incompatible with BSCA tasks. First, the datasets (e.g., Coreutils [11], Binutils [5], and OpenSSL [19]) widely used by existing BSA tools [47, 70, 95, 99] usually provide full functionalities without using much code from other OSS projects. Thus, they are not suitable for BSCA benchmarking. Second, the most frequently-evaluated cross-optimization challenge in the BSA field may be less critical in the BSCA scenario. Existing BSA tools [47, 70, 99] aim at learning semantics from binary code. Hence, they focus on challenges introduced by different optimizations, compilers, and platforms. The most challenging setting in BSA is often comparing binaries compiled without optimization (i.e., O0) and binaries with heavy optimization (i.e., O3). In reality, however, developers seldom release executables without optimization. Thus, besides benchmarking BSCA using various compilers and platforms, we believe it is more urgent to use optimized binaries to study the status quo of BSCA.

Benchmark Dataset. We establish the first benchmark dataset for BSCA, whose establishment involves a considerable amount of manual effort. Three authors, including a senior Ph.D. student and two security engineers from the industry, spent about a month on the dataset collection and manual ground truth marking. All three authors are highly experienced in reverse engineering, software security, BSA/SCA, and have constantly published relevant papers in the community.

This study aims at real-world software with complex software supply chains. Hence, we first search GitHub repositories with more than five git submodules. The explicitly declared git submodules help us identify precise reuses since we can know their URLs and check their tags. There are also reuses in the form of copy (e.g., ControlBlock [10] uses code copied from libmcp23s17 [17]). Developers usually locate their reused packages in a directory like deps/ and third-party/. Thus, we also identify reused OSS by searching these directories. While building collected software, we noticed that some OSS submodules are not compiled into released binaries since they do not provide functionalities but are used for other tasks like testing and code formatting. For instance, googletest [14] is a popular testing framework, but developers will exclude it from released binaries. Hence, we should not label it as reused even if it is a submodule of a repository. Additionally, a project can produce multiple executables, and a submodule is likely in merely one of them. Therefore, to collect reliable ground truth, two authors manually identify the submodules taking

part in the compilation process of an executable and make sure their labeled ground truth has no conflicts.

Our dataset comprises 35 binaries across two platforms with nearly one million assembly functions. The total lines of C/C++ code of the repositories for compiling these binaries are 55,530,527. All these 35 binaries are large-size, widely-used applications, including the physical engine [21] and payment protocol [18]. All information about these binaries can be found in [7] and Table 7 in Appendix.

We report our dataset's statistics in Table 2. ELF executables are compiled with `gcc` and `clang`, while PE executables are compiled with `MSVC`. The datasets are all complex binaries with an average of 12 reused OSS projects. Specifically, 24 out of 35 binaries are Linux ELF binaries with 370,503 assembly functions in total. With manual investigation, we identify that these 24 ELF binaries reuse 334 OSS versions (of 157 OSS). The average size of each stripped ELF binary is 8,588 KB, indicating those binaries are complex (in comparison, the stripped `gcc-7.5.0` executable is 1,023KB).

OSS Database Collection. We build the OSS database with all identified OSS projects used by benchmark and ten for hyperparameter tuning. In sum, the OSS database consists of 255 OSS projects, 16,266 versions, and 11,638,109 source functions. These OSS projects, such as APR [2], are also large-size and frequently linked in daily development. As aforementioned, evaluated executables have an average of 12 reused OSS projects. In other words, for each executable, we benchmark if BSCA solutions can correctly flag those 12 OSS out of 255 OSS projects in the database. Overall, we view our setting as challenging and practical in real-world scenarios.

Hyperparameter Selection Dataset. It is worth noting that our selected BSA tools all require employing a validation dataset for hyperparameter tuning. To do so, we prepare ten large OSS projects and compile all their versions (a total of 524 binaries) into executables. We fine-tune each employed BSA tool's hyperparameter over a simple OSS classification task using these 524 binaries. When the classification accuracy is optimal, we select the best hyperparameter (i.e., θ of Alg. 1) for each BSA tool.

These ten OSS projects are part of our OSS database for the following study. It is reasonable since a BSCA service provider always wants a comprehensive OSS database with as many instances as possible. To make our study convincing, we avoid querying the OSS database with binaries that reuse these ten OSS projects. We report that all 35 executables in Table 2 do not reuse these ten projects.

5. Study

The following sections evaluate how de facto BSA tools perform in the BSCA scenarios. We first introduce the experimental setups. Then, we illustrate the performance of our BSCA framework. Sec. 6 presents useful enhancements, and Sec. 7 shows the time cost.

5.1. Experimental Setups of BSA Tools

We reuse officially released well-trained models if available. The official `SAFE` model was trained with OpenSSL compiled by `gcc` and `clang` with optimizations from `O0` to `O3`, which is frequently used in prior works [70, 95]. `Asm2vec` follows an unsupervised training paradigm with the PV-DM model [67]. Given an assembly function, its embedding is generated after the training process. Thus, all functions in our OSS database are used for training. $\texttt{PalmTree}_G$ uses two models, including an instruction-level embedding model released by `PalmTree`'s authors and a function CFG embedding model. We train the second model in the same method as `SAFE`. The graph network of $\texttt{PalmTree}_B$ is trained in the same way as $\texttt{PalmTree}_G$. $\texttt{Commercial}_B$ and `CodeCMR` are not publicly available. Therefore, we sent the binaries to the authors and received the embeddings for the following study.

TABLE 3. AUC SCORES OF BSA TOOLS ON BINUTILS.

Interal	Tool	`gcc -O3`	`clang -O2`	`mingw32 -O2`
B2B	`SAFE`	.953	.973	.971
	`Asm2vec`	.951	.856	.861
	$\texttt{PalmTree}_G$	.969	.961	.979
	$\texttt{PalmTree}_B$	.989	.991	.989
	$\texttt{Commercial}_B$	.982	.990	.996
B2S	`CodeCMR`	.972	.992	.996

B2B tools search binary functions compiled by `gcc -O2`.
B2S tool searches the binary functions in the source code.

Table 3 presents the AUC score of all BSA tools on Binutils. Both the measurement and the dataset are frequently used in previous studies [44, 47, 66, 70, 77, 95]. Comparing binaries compiled by `gcc -O2` with binaries compiled by `gcc -O3`, `clang -O2`, and `mingw32 -O2` correspond to cross-optimization, cross-compiler, and cross-platform challenges of B2B works. We observe that all BSA tools perform well on the original BSA task. Almost all tools achieve AUC scores over 0.95, except `Asm2vec` on cross-compiler and cross-platform settings. The relatively poor performance of `Asm2vec` is caused by its training data since we compile the OSS database with `gcc` only. Analyzing the binaries compiled by unseen compilers introduces additional challenges to `Asm2vec`. However, as elaborated in Sec. 2.2, compiling the database with another compiler is costly, and there may be unsupported features (e.g., using the `auto` type arguments in a function declaration is not supported by `clang` contemporarily).

5.2. OSS Identification

We clarify that OSS identification is much easier than identifying specific OSS versions reused in the input executable. Thus, we first report the accuracy of the OSS identification and then report the result of version identification. The input binaries have been described in Sec. 4.3. All binaries are stripped before being analyzed. Regarding the hyperparameters of Alg. 1, θ is fine-tuned with the dataset used by each BSA tool.

Recall that the settings of using B2B and B2S are distinct. When using B2B, we need to compile OSS projects in our OSS database into assembly code. To do so, we use the default setting of each OSS project for compilation. Typically, OSS projects are compiled by `gcc/g++`. `O2` optimization is usually used by default. On the other hand, we save the effort of compiling OSS projects when using B2S.

Table 4 presents the *precision*, *recall*, and *F1 score* values when using different BSA tools for BSCA. As introduced in Sec. 4.2, we also consider three SOTA BSCA

TABLE 4. OSS IDENTIFICATION ACCURACY OF DIFFERENT TOOLS.

Tool	S_1			S_2			S_3		
	P	*R*	*F1*	*P*	*R*	*F1*	*P*	*R*	*F1*
T1	.281	.074	.117	.231	.065	.102	.000	.000	.000
T2	.103	.475	.169	.110	.411	.174	.007	.018	.010
T3	.124	.450	.195	.122	.330	.178	.023	.018	.020
T4	.191	.465	.271	.207	.264	.232	.500	.020	.038
T5	.422	.322	.365	.342	.238	.281	.138	.214	.168
T6	.246	.065	.103	.346	.083	.134	.231	.188	.207
C_A	.731	.320	.445	.755	.328	.457	.750	.325	.454
BAT	.560	.351	.432	.647	.341	.447	.746	.371	.495
LibDB	.250	.606	.354	.316	.602	.415	.219	.526	.309

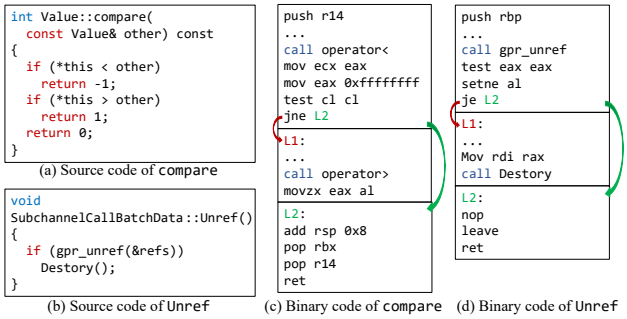
S_1: Binaries are compiled with gcc and no extra flags.
S_2: Binaries are compiled with clang and no extra flags.
S_3: Binaries are compiled with MSVC and no extra flags.
P, *R*, and *F1* denote *precision*, *recall* and *F1 score*, respectively.
T1-**T6** denote SAFE, Asm2vec, PalmTree_G, PalmTree_B, Commercial_B, and CodeCMR, respectively.
C_A is the abbreviation for Commercial_A.

solutions (i.e., C_A, BAT, and LibDB) obtained from the commercial market and academia for comparison.

F1 score is the harmonic mean of precision and recall, and existing academic works use precision and recall to measure SCA solutions' performance [48, 56, 88, 104]. Low precision indicates that many unused OSS projects are falsely identified, likely resulting in warning developers with irrelevant 1-day vulnerabilities from unused OSS projects. Low recall indicates that many reused OSS projects are missed, which may lead to missing critical vulnerabilities from missed OSS projects. Therefore, high precision and recall are critical for a practical SCA solution.

To comprehensively evaluate the feasibility and performance of BSCA with the support of BSA techniques, we classify the input binaries by their compilation toolchains, as shown in Table 2. Since the projects of our OSS database are compiled with gcc/g++, we use the ELF binaries compiled by clang (S_2) to represent the cross-compiler challenge and the PE binaries compiled by MSVC (S_3) to denote the cross-platform challenge. For comparison, the ELF binaries compiled by gcc/g++ (S_1) present the BSCA performance when input binaries and the OSS database are compiled with the same toolchain. Since developers always release software compiled with optimizations, we do not build a cross-optimization dataset specifically.

Comparison with Existing Solutions. Overall, the results are not promising when directly employing BSA (either B2B or B2S) in the pipeline of SCA. It is seen that the F1 scores are lower than 0.4 when using different BSA tools. In comparison, Commercial_A and BAT manifest higher accuracy, achieving F1 scores over 0.4 across all settings. With manual analysis, we find that LibDB's suboptimal performance is primarily caused by its employed BSA tool, Gemini. As noted in [70, 77], Gemini relies on manually-selected features and is less superior to de facto semantics-aware BSA techniques. Hence, when analyzing our benchmark dataset, LibDB primarily resorts to its extracted "basic features" (e.g., strings and exported function names). However, compared with BAT, which allocates heavy weights to long and unique strings, LibDB treats strings equally, making its precision notably lower than that of BAT. By comparing the precision and recall of our BSA tools with existing BSCA solutions, we attribute the low F1 scores to the *low precision* of our BSCA framework with existing BSA techniques. In a total of 18 settings (6 BSA tools with three binary compilation settings), all settings have significantly lower precision in identifying reused OSS than BAT and Commercial_A.



(a) Source code of compare (b) Source code of Unref (c) Binary code of compare (d) Binary code of Unref

Figure 6. False positive case study.

Low Precision of BSA Tools. As shown in Alg. 1, the similar function pairs produced by BSA tools determine the results of OSS identification. Here, we analyze similar function pairs provided by BSA tools and compare them with similar source function pairs generated by Centris using TLSH. In short, we find that our studied BSA tools yield *a noticeable number of false positives*, resulting in *low precision* on BSCA and falsely alarming vulnerabilities. Since we compare the assembly functions of the input binary with the functions of each OSS project in the database, due to numerous OSS projects in our database, a function $f \in F_{bin}$ might find similar functions in multiple OSS projects. As illustrated in Fig. 1, f can be an OSS project's custom code or reused function. Thus, a maximum of one function similar to f is the reuse (true positive), while other similar functions are false positives. On the other hand, a false negative occurs when a reused function is not identified as similar to a function from its reused OSS. False negatives result in low recall of our BSCA framework and may miss essential vulnerabilities.

Unsurprisingly, our study shows that BSA tools frequently produce false positives, i.e., identifying functions from unused OSS projects as similar. Since two functions are similar when their similarity score is greater than the threshold θ (see line 16 of Alg. 1), increasing θ is a possible solution to reduce false positives. Unfortunately, our study demonstrates that raising θ cannot significantly decrease false positives to support BSCA. For example, when using Commercial_B with the optimal threshold $\theta = 0.95$, which is selected with our hyperparameter selection dataset (see Sec. 4.3), an assembly function can find its similar function in 3.43 OSS projects on average, which means at least 70.8% (2.43/3.43) of similar function pairs are false positives. After raising θ to 0.99, close to the upper bound of *cosine similarity*, the average number of similar functions for each binary function is still 2.7, i.e., the false positive rate is 63.0% (1.7/2.7). In contrast, TLSH used by Centris is much more precise, with an average of 1.45 similar functions for each source function.

False Positive Case Study. Fig. 6 provides two functions classified as similar by Commercial_B with a similarity score over 0.95. Fig. 6(a) and Fig. 6(c) present function compare's source and binary code, respectively. Function compare belongs to the OSS jsoncpp [16], which is reused by ControlBlock [10] (i.e., a driver for an extension board). Fig. 6(b) and Fig. 6(d) show the source and binary code of function Unref of gRPC [15] (i.e., a remote procedure call framework), respectively. The binary code of Unref is stored in the OSS database.

While disassembling, IDA pro, the disassembler of Commercial_B, successfully extracts the binary function of

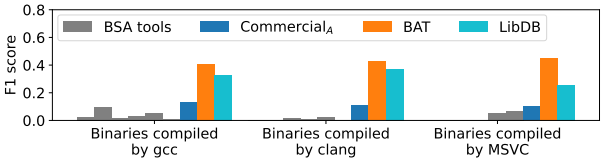


Figure 7. Version identification F1 scores of different tools.

`compare` from the whole executable. As illustrated in Fig. 5, `Commercial`$_B$ will convert the binary code of Fig. 6(c) into an embedding vector to search for a similar function in the OSS database. The binary code of `Unref` (i.e., Fig. 6(d)) is also similarly converted into an embedding vector by `Commercial`$_B$. To decide whether `ControlBlock` reuses `gRPC`, our BSCA framework will search if there is a function of `gRPC` similar to `compare`. Unexpectedly, the cosine similarity between vectors of `Unref` and `compare` is sufficiently high.

The source code (Fig. 6(a) and Fig. 6(b)) illustrates notable differences between `compare` and `Unref`. However, they look similar after compilation (Fig. 6(c) and Fig. 6(d)). Both functions contain three basic blocks with similar CFGs. After entering the function, they both invoke a callee; the following conditional jump instructions (i.e., `jne` and `je`) control the CPU to execute the second (**L1**) or third basic block (**L2**). Moreover, their second blocks invoke another callee, and their third blocks are functions' epilogues. Note that we add the callee symbols (i.e., `operator<` and `operator>`) for readability, and they are not known in the stripped binary code of `compare`. Hence, we conclude that `Commercial`$_B$ treats `compare` as similar to `Unref` because of their similar CFGs. Although the source codes of these two functions are semantically and syntactically distinct, classifying their assembly functions as similar seems reasonable with existing BSA tools. Such false positives imply the gap between binary code similarity and reuse relations, i.e., a reuse relation is not necessary for a high similarity score.

Comparison across Different Settings. Comparing the F1 scores of our BSA solutions across three settings shows that cross-platform is still challenging. In Sec. 5.1, we report that they all gain high AUC scores on the frequently-used Binutils dataset across all settings. However, all B2B tools (**T1**-**T5**) showed notable performance degradation in the cross-platform setting (S_3) of Table 4. Their F1 scores decrease by over 0.1 compared with the other two settings. Regarding the cross-compiler setting (S_2), we can see a slight reduction of B2B tools' F1 scores compared with S_1, which denotes that our BSCA framework inherits the difficulties of B2B tools.

5.3. Version Identification

We implement Centris's version identification in our BSCA framework. Centris assigns weights to all functions; then, it scores a version by summing weights of similar functions belonging to that version. The version with the highest score is identified as reused. To improve the precision, Centris assigns a larger weight to the function existing in fewer versions. Given a function f of an OSS, its weight is computed with $W(f) = log(n/|V(f)|)$, where n is the total number of versions of the OSS, and $|V(f)|$ is the number of versions containing f.

Fig. 7 presents the results of version identification. Our experiment shows that `BAT` achieves the best F1 score in the version identification task. Since OSS identification is the pre-condition of version identification, LibDB's version identification accuracy is lower than that of `BAT`.

After a detailed manual analysis, we conclude that identifying the exact reused version is difficult for existing BSA tools since they are *not sufficiently sensitive to minor changes*. Minor changes like small patches do not significantly change a function's functionality and control-flow structure. Thus, an old version function can still be treated as similar to the updated one. For instance, the notorious `HeartBleed` vulnerability affects the function `tls1_process_heartbeat` of `OpenSSL` from version `1.0.1a` to `1.0.1f`. When we feed a binary compiled from `OpenSSL-1.0.1g` (without the vulnerability), `SAFE` can successfully discover the existence of function `tls1_process_heartbeat` but report a wrong version since the vulnerable version gains a higher similarity than the patched function (0.959 to 0.944). Therefore, we believe that a future direction of BSA works is to perform *minor-change-sensitive* matching. This way, BSA tools could accurately solve tasks like patch presence detection and version identification.

In sum, directly employing BSA techniques in the BSCA pipeline cannot gain satisfactory accuracy in identifying OSS. Our analysis reveals the low precision problem of BSA tools in the BSCA scenario. Moreover, existing BSA tools can hardly work on version identification since they are insensitive to minor changes.

6. Enhancement

With findings obtained in Sec. 5, we explore enhancements over our BSCA pipeline from three aspects. In particular, we first present three enhancements in Sec. 6.1, Sec. 6.2, and Sec. 6.3, and then discuss their effectiveness.

6.1. Enhancement – Signature

Originating from the primary setup of modern BSA techniques, our study pipeline in Sec. 5 performs function-level matching. Nevertheless, our observation shows that string-level signatures likely facilitate a more accurate matching. The string-level signatures have been seen to be employed by existing SCA tools [48, 57, 100]. Those tools do not extract and compare program semantics but merely rely on the extracted string-level signatures, which are expected to be generally unchanged across different compilers, optimizations, and platforms. Therefore, we expect to use string-level signatures to *identify the false negative function pairs as positives while maintaining a low false positive rate*. Also, extracting these string-level features usually poses few additional challenges compared with recovering assembly functions.

We use `IDA pro` to search and extract strings from the input software and OSS executables at this step. All strings referred to by a function composes its signature. When OSS source code is used, we directly extract them from the source code with `BAT`'s string extraction utility. Given a function f of the input binary and a function t

recorded in the OSS database, we compute their similarity score by Eq. 1,

$$sim(t, f) = cos(vec(t), vec(f)) + \frac{\alpha|sig(t) \cap sig(f)|}{|sig(t)|} \quad (1)$$

where $vec(f)$ is the embedding vector of f, and $sig(f)$ is the set of referred string signatures of f. The original similarity computed by a BSA tool is $cos(vec(t), vec(f))$; we add it with their signature similarity, in which α is a weight factor. Since our goal with this enhancement is identifying the false negatives as positives, we use a relatively large factor ($\alpha = 10$) to largely raise the similarity between functions with low cosine similarity. Unlike using strings for BSCA directly, we still focus on functionality code for BSCA, and the signature enhancement can be viewed as a shortcut to better BSA results.

While collecting string-level signatures, we notice some strings are frequently used by various OSS instances, such as "%s." To ensure the matched string-level signatures are strong evidence of reuses, strings in more than 100 OSS projects will be omitted. We leave presenting and discussing the enhancement results in Fig. 9. In short, the F1 scores for all B2B tools are increased with the extracted string-level signatures.

6.2. Enhancement – Global Information

Different from function-level BSA works that aim at assembly functions, BSCA focuses on the whole binary. Hence, it is reasonable to use richer information to improve the BSCA results. Specifically, we exploit relations between functions.

Call graph. A call graph is a directed graph. It can be represented by $G = (V, E)$, where V is the set of all functions, and E is the set of all calling relations. Since we analyze binaries compiled by different toolchains, the input binary's and an OSS project's call graphs can be different due to different optimization strategies like function inlining [61], even if they are compiled from the same source code. To design a robust enhancement, we extend the set of calling relations E. If E contains (f_u, f_v) and (f_v, f_w), we add (f_u, f_w) to E. We repeat this process until E reaches a fixed point.

To reduce the falsely identified similar functions using calling relations, a straightforward idea is that *a function without a matched calling relation is likely a false match*. By changing matching functions (nodes) to matching calling relations (edges), we successfully improve the BSCA results with B2B tools.

Layout. Although call graph is effective in improving BSCA, existing tools like CodeQL [53] need to compile a project before generating its precise call graph. However, as mentioned in Sec. 2.2, B2S BSA is designed to save the effort for compiling the OSS database. Therefore, we use the layout of a binary as a replacement when we use B2S solutions.

When compiling C/C++ programs into executables, compilers first compile each C/C++ source file into an object file and then link all object files into an executable. A common observation is that functions in the same source file will be put into the same object file, and then *very likely* placed closely in the executable. Hence, during OSS detection with B2S tools (e.g., `CodeCMR`), we check if more than one functions from the same source file can be detected closely from the input executable.

Algorithm 2 Update the similar function pairs with layout.

```
1: function UPDATE_WITH_LAYOUT(M, L)
2:     # M is the set of function matching results.
3:     # L is the set of source files of an OSS.
4:     R ← ∅
5:     M' ← sorted(M)   ▷ Sort M by the order of input functions.
6:     for (t_i, f_i) ∈ M' do   ▷ t_i is the similar function in database.
7:         Q ← {l | l ∈ L and t_i ∈ l and length(l) > 1}
8:         L ← L − Q            ▷ Every object's layout is used once.
9:         for l ∈ Q do
10:            x ← [f_i, f_{i+1}, ..., f_{i+length(l)−1}]
11:            y ← [t_i, t_{i+1}, ..., t_{i+length(l)−1}]
12:            s ← LONGEST_COMMON_SUBSEQUENCE(y,l)
13:            if length(s) > 1 then
14:                R ← R ∪ s
15:    return R
```

Alg. 2 illustrates the idea of layout enhancement. Given the similar function pairs produced by BSA tools, we first sort them by function memory addresses in the input binary (line 5). While iterating the ordered functions of input binary, given f_i and its similar function t_i in the database, we can get Q, the set of source files containing t_i from the OSS source file set L (line 7). While iterating $l \in Q$, we can generate a "window" with the same size as object l, which contains a list of functions. x is the "window" with ordered functions of the input binary (line 10), and y is the "window" with similar functions according to x (line 11). Since l and y are lists of functions in the OSS database, we can compute their longest common subsequence as s (line 12). When s has more than one element (line 13), there must be another function $t_j \in l$ that is close to t_i. Since their matched functions (f_i and f_j) in the input binary are also close, t_i and t_j are likely to be true matches.

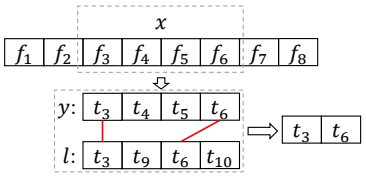


Figure 8. Layout matching processing.

Fig. 8 depicts an example. Given the similar function pair $(f_3; t_3)$, we get l containing t_3 with four functions. Then we extract the consecutive four functions from the input binary, and their similar functions compose y. After comparing y and l, t_3 and t_6 are deemed correct matches, whereas t_4 and t_5 will be ignored. This enhancement successfully *reduces many false positives* in matching OSS with global layout knowledge.

6.3. Enhancement – Distinguishability

An insight of this enhancement is that certain complex functions with unique semantics may serve as a strong indicator to confirm an OSS reuse. Hence, we design the distinguishability to describe the degree to which a function can represent a particular OSS. Specifically, we adapt `BAT`'s string literals ranking function to our function-level distinguishability enhancement. We use $complexity(f)$,

the number of instructions of f, as the dividend. Intuitively, a utility function likely matches similar functions across various OSS projects, resulting in unavoidable false positives. Therefore, we use $\beta^{|match(f)|-1}$ as the divisor to decrease the importance of such functions. The distinguishability is computed by $Dis(f) = \frac{complexity(f)}{\beta^{|match(f)|-1}}$, where $match(f)$ is the set of OSS projects with functions being matched to f. We require $match(f)$ to have at least one element; otherwise, we treat f as the custom code. $\beta > 1$ is a constant whose value is seen to have a small impact on the results; we set β as 5 following the prior work's setting [56].

Given an input binary and an OSS, let M be the set of similar function pairs and $(t_i, f_i) \in M$ is a pair where t_i and f_i belong to the OSS and input binary, respectively. The score of the given OSS is $\sum_{(t_i,f_i)\in M} Dis(f_i)$. The OSS is deemed reused if the score exceeds the threshold hyperparameter γ. We set $\gamma = 100$, which is consistent with the BAT's setting. The threshold γ is only involved in the OSS identification phase when the distinguishability enhancement is enabled. Otherwise, we still use the setting of Centris described in Sec. 4.1, i.e., an OSS project is identified as reused when we find that 10% of its functions are similar to functions of the input binary.

Unlike the distinguishability described above, Centris assigns weights to functions in the OSS database for version identification; differently, this step aims to assign a distinguishability score for each function of the input binary during OSS identification. The sum of Centris's weights ranks the likely reused version of an OSS, but we sum the distinguishability scores to determine whether an OSS is reused. If a function with a higher distinguishability score is matched to an OSS's function by BSA, the chance of reusing this OSS is also higher. Fig. 9 reports the effectiveness of this enhancement. This enhancement improves the performance of all tools on the OSS identification task. The F1 scores of all BSA tools are further increased by over 0.2 on average.

6.4. Utilizing Enhancements

Since three enhancements can work individually or jointly, we evaluate their performance with different modes in the following section. When three enhancements work jointly, given an OSS project, we first employ string signatures (sig) to update the similarity scores calculated by BSA techniques and to produce updated similar function pairs. Then, the global enhancement reduces the false positives by detecting similar edges or comparing the memory layout of function pairs (glb). After getting similar function pairs with all OSS projects, we calculate the distinguishability scores for each input function (dis) and decide the reuse relation for each OSS project.

6.5. Enhancement Results Discussion

OSS Identification. Fig. 9 depicts how the F1 scores are changed according to various enhancements. In 14 of the 18 settings (6 BSA tools with binaries compiled by 3 toolchains), enabling three enhancements results in better performance than existing BSCA solutions, which depend on string literals and syntactic-based BSA technique (i.e., Gemini). The performance of the rest four cross-platform settings does not exceed string-based BSCA solutions, but our enhancements still raise their F1 scores by over 0.2.

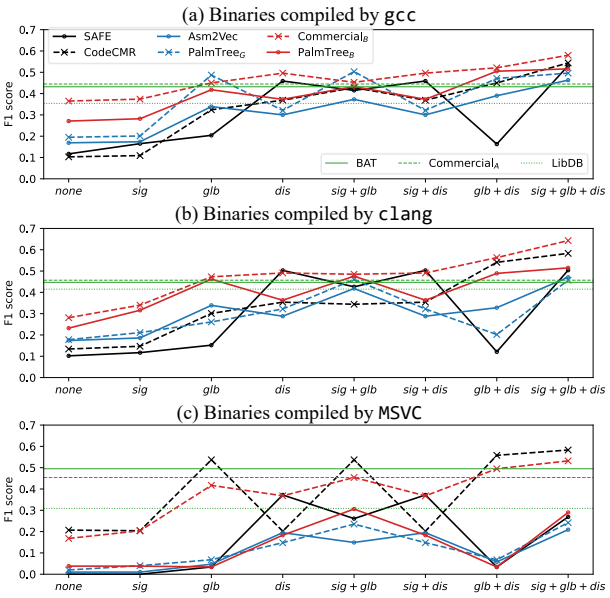


Figure 9. OSS identification F1 scores with enhancements. sig, glb, and dis denote string signature, global information, and distinguishability enhancements, respectively. + means their combination.

Amoung the three enhancements, dis usually brings the most significant improvement. Enabling dis raises an average F1 score of all settings by 0.186, and the maximum improvement is 0.405. In addition, although we set the threshold for distinguishability scores with a fixed value ($\gamma = 100$) for six BSA tools, developers can fine-tune γ with their BSA tool to achieve an optimal BSCA result. We present a study of the impact of different values of γ in Fig. 11 and Sec. A of Appendix.

When enabling sig alone, the F1 scores of all B2B tools are slightly increased by 0.0197 on average, which means that merely a small portion of functions are identified by string-level signatures. In comparison, glb and dis enhancements can notably raise the F1 scores by 0.144 and 0.186, respectively. However, the low improvement of sig alone does not mean that it is useless. We notice a synergy effect between sig and glb enhancements (i.e., $sig + glb$), which further increases the F1 scores by 0.1 on average compared with enabling glb alone (0.244 to 0.144). Recall in Sec. 6.2 that given a matched function f, if no callers (or callees) of f (i.e., call graph-based enhancement) or functions close to f in memory (i.e., layout enhancement) match the functions of the OSS project, f will be omitted. Hence, when glb is enabled, a more stringent criterion is used to determine whether two functions are similar. Hence, while reducing the number of false positives, some true positives are also judged as dissimilar. To alleviate this problem, using string-level signatures to precisely identify additional functions is a solution.

Overall, enabling all three enhancements can achieve the best performance. The F1 scores of all tools are increased by 0.315 on average, and the maximum improvement is 0.45 while analyzing binaries compiled by clang with CodeCMR (i.e., the black curve of Fig. 9(b)).

Among the three binary settings, the cross-platform compiled binary poses a critical challenge to BSCA. SAFE, Asm2vec, PalmTree$_B$, and PalmTree$_G$ were orig-

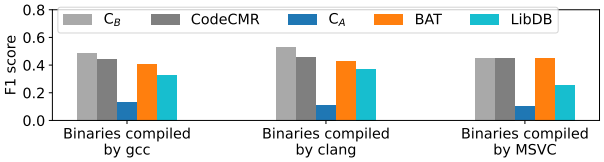


Figure 10. Version identification results with enhancements.

inally trained with binaries compiled by `gcc` and `clang`. Although our enhancements vastly improve their F1 scores, their performance is still incomparable to `BAT` and $\texttt{Commercial}_A$ when identifying binaries compiled by `MSVC`. $\texttt{Commercial}_B$ is also trained with binaries compiled by `MSVC`; however, its F1 score is also largely decreased from over 0.3 to less than 0.2 when disabling enhancements, resulting in relatively poor performance compared to `CodeCMR`, i.e., the B2S solution.

One may question if BSCA developers shall consider training a separate model for each platform to alleviate the challenges of cross-platform BSCA. We argue that this is unlikely to promote B2B BSA, if not making it impractical. In practice, migrating OSS to platforms that were not originally supported often requires significant manual effort from software developers. Consequently, BSCA service providers may not have access to OSS binaries for an originally unsupported platform, making cross-platform capability indispensable for B2B solutions. Regarding B2S BSA, the primary focus lies in addressing the cross-modality challenge. Conventional approaches extract platform-independent features like strings and exported function names, while the machine learning methods (e.g., `CodeCMR`) use binaries of various platforms for training to improve their ability to overcome the cross-modality challenge.

On the other hand, the performances with enhancements of six tools are consistent with their initial results (i.e., *none* in Fig. 9). $\texttt{Commercial}_B$ and `CodeCMR` are still the two best BSA tools with enhancements applied. With this observation, we infer that our enhancements should still work and bring a promising result when a more advanced BSA tool is available.

Version Identification. A correct OSS identification is the pre-condition of an accurate version identification. Hence, improving OSS identification will result in a growth in version identification accuracy. Due to the incapability of existing BSA tools to identify functions of different versions, we view version identification as a completely new task. After identifying all reused OSS instances with our BSCA framework, we rely on `BAT` to rank the most likely version for each identified OSS instance with string signatures. Fig. 10 presents the F1 scores of the version identification task. Both $\texttt{Commercial}_B$ and `CodeCMR` outperform `BAT` when enabling all enhancements. Besides string-level signatures, we argue that techniques that are sensitive to minor changes (e.g., patch presence detectors [96, 103]) can take part in the version identification process, and the existence of a specific unique code snippet can be used to determine the exact version.

7. Time Cost of Online Analysis

Compared with signature-based SCA works like `BAT`, whose online analysis phase is efficient due to fast hash matching, the time cost of function-level granularity matching used in our BSCA framework is not trivial. Centris, having the same workflow but deciding similar function pairs with the distance of TLSH, takes nearly 8 hours for the component identification with our benchmark dataset. LibDB spends most of its time on time-consuming call graph-based analysis, which takes over a hundred CPU hours for 35 binaries. As depicted in Fig. 2, the online analysis phase can be further split into (1) Disassembling & dissection, (2) BSA, and (3) OSS & version identification. We do all experiments with an AMD 3970X server with 256GB memory and an RTX3090.

Disassembling & Dissection. We implement BSA tools with the `IDA pro` disassembler. The size of binaries in our dataset is 9,089KB on average. The `IDA pro` (ver 7.5) successfully analyzed 35 binaries within 4.5 hours.

BSA. The BSA process consists of two stages. Given a binary code embedding tool, we first **(a)** encode the input binary's functions. Then, we employ `Milvus`, the vector database, to **(b)** search for similar functions. The efficiency of **(a)** depends on the embedding tool, while the size of vectors influences the query speed in **(b)**. $\texttt{Commercial}_B$, the most accurate but least efficient tool, consumes 10 hours in **(a)** and 22 minutes in **(b)** with 256-dimension vectors. The time cost of BSA with $\texttt{Commercial}_B$ for online analysis is about 10.5 hours.

OSS & Version Identification. BSCA with $\texttt{Commercial}_B$ costs half an hour to analyze the matching results without any enhancement. The overhead of signature and distinguishability enhancement is negligible. Global enhancement rises the overhead by 315% (about 1.5 hours).

Overall, analyzing 35 binaries with six evaluated BSA tools takes 10 to 18 hours. Given that Centris takes about eight hours to finish the corresponding SCA task, the time cost of our BSCA framework is comparable and reasonable.

8. Vulnerability Detection with BSCA

After identifying the reused OSS projects and their versions, as mentioned in Sec. 2.1.1, the next is to extract valuable information from the reused information. In this study, we reuse the vulnerability dataset from previous works [44, 47, 91] to show an example of using SCA to detect vulnerabilities. The dataset contains eight vulnerabilities from seven OSS projects. Our experiment is more challenging than previous works since our database (millions of functions) to be searched is much larger than previous works' databases (3,015 functions).

Experimental Setup. To perform SCA on the dataset, we first compile all versions of those projects with `gcc` and add them to our OSS database. Given the eight CVEs, we build the benchmark dataset with `clang` by compiling those seven projects' last affected versions (i.e., **Affected Ver.** of Table 5) and the versions after the affected versions (i.e., **Non-affected Ver.** of Table 5), resulting in 16 binaries. As presented in Sec. 5.2, this setting poses cross-compiler challenges to our BSCA framework. We do not include cross-platform binaries in this experiment since some projects have no cross-platform support. We use the BSCA framework with $\texttt{Commercial}_B$, the best-performing setting for ELF binaries. Table 5 shows the results of vulnerability detection with BSCA.

TABLE 5. VULNERABILITY DETECTION WITH BSCA

Vulnerability	CVE	OSS	Affected Ver.	Non-affected Ver.
Shellshock #1	2014-6271	✓	✓	✓*
Shellshock #2	2014-7169	✓	✓	✓*
FFmpeg	2015-6826	✓	×	✓*
Clobberin' Time	2014-9295	✓	✓	✓
Heartbleed	2014-0160	✓	✓	✓
wget	2014-4877	✓	✓	✓
ws-snmp	2011-0444	✓	✓*	×
Venom	2015-3456	✓	✓*	✓*

✓ Identified the exact OSS and version.
✓* Identified the (non-)affected version but a wrong version.
× Failed to identify the (non-)affected version.

All reused OSS projects are correctly identified since all the binaries reuse hundreds of functions from the OSS projects. Thus, our BSCA framework can always identify sufficient similar functions for OSS identification. Among 16 binaries, half are identified with the exact version (marked by ✓). Although the identified versions for the remaining may be wrong, six (marked by ✓*) still show the existence of the vulnerabilities. There is only one missing alarm and one false alarm eventually (marked by ×).

Missing Alarms of BSCA. A "×" in the **Affected Ver.** column indicates a missing alarm. False negative (FN) of BSCA (i.e., incorrectly identifying the affected version as non-affected) can cause missing alarms. For the FN of BSCA presented in Table 5 (i.e., FFmpeg), the identified version is very close to the affected version. The vulnerable FFmpeg version is n2.6.3, and the detected version is n2.7.0, released just one month after n2.6.3. In practice, the BSCA solutions can still report vulnerabilities to avoid missing alarms when the identified version is close enough to a vulnerable version.

False Alarms of BSCA. A "×" in the **Non-affected Ver.** column indicates a false alarm. False positive (FP) of BSCA (i.e., incorrectly identifying the non-affected version as affected) can cause false alarms. When analyzing non-affected binaries, our framework incorrectly classifies one (i.e., ws-snmp of Wireshark [27]) as vulnerable. The differences between the affected (v1.4.2) and non-affected version (v1.4.3) are small, i.e., only 0.6% of functions are newly added or changed in v1.4.3. However, due to BSA's insensitivity to minor changes, 2.6% of functions from v1.4.3 are falsely identified as the functions from versions prior to v1.4.2, leading to the false alarm. Nevertheless, we believe that the false alarm will not substantially waste developers' time since developers can post-check those warnings easily with the source code. Besides, a patch presence detector can also verify the alarm. We report that FIBER [103] successfully identifies this patch, eliminating the false alarm.

9. Impact of Compilation Optimizations

Sec. 4.3 mentions that few developers release non-optimized binaries. To be close to the real-world scenario, our benchmark dataset is compiled with default optimizations. In this section, we investigate the impact of compilation optimizations on our BSCA framework. Since iterating all optimization options is impractical, we focus on the (possibly) most attention-drawing optimization for BSA techniques, i.e., function inlining [61].

Impact on B2B tools. Existing BSA works demonstrate that comparing optimized and non-optimized binaries is challenging [47, 61, 70, 77, 98, 99]. Since we build our OSS database with optimized binaries, analyzing non-optimized binaries is likely more challenging than analyzing optimized binaries. However, we argue that changing the optimization options of the query dataset will not prevent BSCA. As a demonstration, we re-compile the setting of searching Heartbleed in Sec. 8 with `-fno-inline` to disable function inlining. We select the notorious Heartbleed since the number of affected versions is the greatest among eight CVEs, which potentially denotes the most difficult in identifying the exact reused version of the binaries after changing the compilation option and reveals the impact of function inlining. We then use $\texttt{Commercial}_B$ to identify the OSS and versions in this study.

TABLE 6. INLINING'S IMPACT FOR HEARTBLEED DETECTION

Inline setting	$\texttt{Commercial}_B$ (B2B)		CodeCMR (B2S)	
	Aff-Ver	Non-Ver	Aff-Ver	Non-Ver
Enabled	✓	✓	×	✓*
Disabled	✓*	×	✓	✓

Aff-Ver and **Non-Ver** denote affected version (1.0.1f) and non-affected version (1.0.1g), respectively.

The second and third columns of Table 6 show $\texttt{Commercial}_B$'s results. We observe that $\texttt{Commercial}_B$ identifies the reused OSS (i.e., OpenSSL), while the version identification accuracy decreases. Specifically, after disabling inlining for compiling the binaries, although the number of similar function pairs decreases from 306 to 126, the remaining 126 similar function pairs are still sufficient for identifying the reused OSS (i.e., OpenSSL). The version identification is also useful for further vulnerability detection since the identified version (1.0.1d) is close to the exact versions (1.0.1f and 1.0.1g).

Impact on B2S tools. Since B2S tools' OSS databases are built from the source code directly, the source functions in the database are not inlined. Hence, for B2S tools, analyzing binaries compiled with function inlining is more difficult than binaries without inlining [61]. However, our selected B2S tool, CodeCMR, is well-trained with inlined assembly functions. It can still identify 56% of reused functions, accurately detecting the existence of OpenSSL.

Nevertheless, as shown in Table 6, function inlining causes inaccurate version identification, and there is a FN in identifying the affected version. We also observe that the identified version (1.0.1l) is close to a vulnerable version (1.0.1f). As mentioned in Sec. 8, we can invoke a patch presence detector to verify the existence of Heartbleed and eliminate the FN. When analyzing binaries compiled without inlining, CodeCMR can detect nearly 95% of similar functions from OpenSSL, identifying the exact reused versions.

10. Threats to Validity

Internal Validity. Although we have spent tremendous efforts on labeling the ground truth of the reuse relations, we cannot ensure whether all reused OSS projects are identified, especially modified reuses. However, all identified reuse relations are reliable and sufficient to present our BSCA framework's performance and reveal the incapability of existing BSA solutions.

External Validity. We extend the workflow of advanced SCA solutions, Centris [94], for our BSCA analysis. Although Centris's authors reported that function-level granularity is the best for source-based SCA compared

with file-level and line-level granularities, it may differ in BSCA. Nevertheless, we still work on the function-level granularity since it is the most frequently studied. We admit that others, like block-level granularity, may also work. However, existing de facto block-level embedding technique (i.e., `DeepBinDiff` [49]) relies on expensive algorithms to produce block embedding vectors. Moreover, `DeepBinDiff` relies on k-hop greedy matching algorithm, which is slow and cannot be accelerated by vector database. Thus, we do not evaluate it in this study.

11. limitations and Future Work

Software Obfuscation. Obfuscation techniques can produce binaries dramatically different from normally-compiled ones [36, 64]. Some obfuscation techniques, like opaque prediction, can significantly change the CFG of a function; thus, the function embeddings depending on CFG structure, which is the most frequently used feature, become unreliable, and the similar function pairs are meaningless. Moreover, existing obfuscation techniques like data encryption, function reordering, and garbage code insertion can make our three enhancements ineffective. A potential solution is restricting obfuscation to custom code only. Since obfuscation inevitably introduces extra execution overhead, obfuscating only custom code can protect developers' intellectual property while not incurring a huge overhead, and our BSCA framework can still identify the OSS projects being used.

C/C++ software. Many other programming languages, like C#, Rust, and Go, can also be compiled into binary executables. However, this study focuses on C/C++ software, as existing BSA techniques also primarily target C/C++ software. Furthermore, the complex and long-standing usage of toolchains in the C/C++ ecosystem [89] makes the analysis of C/C++ software particularly challenging. This work could potentially be extended to software developed with other programming languages, and we leave such exploration for future research.

Backported Patches. Software developers often backport patches from newer versions to older versions to fix vulnerabilities instead of upgrading the software due to compatibility issues. This practice, however, places a challenge on all SCA tools since an identified "vulnerable" component may be patched already and might result in false alarms. Due to the one-time effort of patching a vulnerability and the catastrophic consequences of not patching, we argue that reporting false alarms is more acceptable than missing alarms. Moreover, verifying the existence of a patch should not be a difficult task for developers with source code, and existing works [63, 96, 103] can be employed as a post-check to verify BSCA alarms by identifying the existence of patches.

12. Related Work

Binary Software Composition Analysis (BSCA). `BAT` [56] and OSSPolice [48] employ string-level signatures like strings and exported function names. B2SFinder [100] additionally extracts `if/else` and `switch/case` structures for higher accuracy. Although their selected features are helpful, they discard semantic knowledge and cannot work when the OSS project has few unique strings (e.g., RapidJSON [22]). LibDB [88], a recent BSCA work, employs a function-level embedding technique using manually selected features (i.e., `Gemini` [95]) to decide binary function similarity, and it uses function call graphs as global knowledge to improve matching precision. However, LibDB does not learn from SOTA SCA methods to establish the BSCA pipeline, and it lacks using SOTA BSA tools as the cornerstone, as clarified in Sec. 5.2. Comparing to LibDB, we also propose a set of optimizations to largely improve our BSA-based BSCA performance and surpass the commercial solution. Moreover, LibDB requires compiling its OSS database, while we extensively explore comparing the binary with source code (the B2S-based BSCA); this solution alleviates the compilation overhead and is generally more extendable.

Binary Similarity Analysis (BSA). We now discuss relevant but not selected BSA solutions. Existing methods usually extract semantic features from binary since the compiler can change the structure information (e.g., CFG) significantly. `CoP` [73, 74], `BinGo` [40], `BinSim` [80], `IMF-SIM` [93], [65], and FIBER [103] extract semantic features via symbolic or dynamic execution, which may bring unacceptable overhead [106]. `Gitz` [45] lifts binary code to LLVM-IR, then employs the optimizer to transform the IR with the same options. It heavily relies on the quality of the lifter. Recent works often take advantage of machine-learning techniques. α`diff` [72] treats a binary function as an image, then uses a CNN model to produce the embedding of a binary function. `InnerEye` [106] treats instructions as words and blocks as sentences, then produces their embeddings with `word2vec` [78]. The program's CFG is decomposed into paths for similarity detection.

Program Clone Search. A recent work named PSS [38] considers this problem, i.e., given a target program and a repository of known programs, the goal is to find the binary in the repository most similar to the target. Compared with SCA, which produces a list of reused OSS projects, PSS only returns one program. We clarify that SCA itself is not a similarity analysis problem, although many existing SCA works are supported by similarity analysis techniques (e.g, SourcererCC [83], LibID [104], and Centris [94]).

13. Conclusion

In view of the prolific development of BSA, we conducted the first comprehensive analysis of BSA to promote the development of BSCA, a critical application widely needed in security and software re-engineering tasks. Our experiment revealed that directly using BSA techniques in the SCA pipeline does not lead to satisfactory performance, and is incapable of version identification. We then proposed enhancements from three aspects, which largely improved the accuracy of OSS identification with moderate costs.

Acknowledgements

This work was supported in part by CCF-Tencent Open Research Fund and NSFC/RGC Joint Research Scheme (JRS) under the contract N_HKUST605/23. We are grateful to the anonymous reviewers for their valuable comments.

References

[1] Androguard. https://github.com/androguard/androguard.

[2] Apache Portable Runtime Project. https://apr.apache.org/.

[3] Apktool. https://ibotpeaches.github.io/Apktool/.

[4] asm2vec-pytorch. https://github.com/oalieno/asm2vec-pytorch.

[5] Binutils. https://www.gnu.org/software/binutils/.

[6] Black Duck Binary Analysis. https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis/binary-analysis.html.

[7] BSA2BSCA. https://sites.google.com/view/bsa2bsca/home/.

[8] Bytesafe. https://bytesafe.dev/.

[9] CODESentry: Binary Software Composition Analysis. https://www.grammatech.com/binary-software-composition-analysis-sca.

[10] ControlBlockService2. https://github.com/petrockblog/ControlBlockService2.

[11] Coreutils. https://www.gnu.org/software/coreutils/.

[12] dex2jar. https://github.com/pxb1988/dex2jar.

[13] Gemini. https://github.com/xiaojunxu/dnn-binary-code-similarity.

[14] GoogleTest. https://github.com/google/googletest.

[15] gRPC. https://github.com/grpc/grpc.

[16] JsonCpp. https://github.com/open-source-parsers/jsoncpp.

[17] libmcp23s17. https://github.com/piface/libmcp23s17.

[18] Nano. https://github.com/nanocurrency/nano-node.

[19] OpenSSL. https://www.openssl.org/.

[20] PalmTree. https://github.com/palmtreemodel/PalmTree.

[21] Project Chrono. https://github.com/projectchrono/chrono.

[22] RapidJSON. https://github.com/Tencent/rapidjson/.

[23] SAFE. https://github.com/gadiluna/SAFE.

[24] Snyk. https://snyk.io/what-is-snyk/.

[25] The IDA Pro disassembler. https://www.hex-rays.com/products/ida/index.shtml.

[26] Whitesource. https://www.whitesourcesoftware.com/product-overview/.

[27] Wireshark. https://www.wireshark.org/.

[28] Can't run OSSPolice. https://github.com/osssanitizer/osspolice/issues/1, 2023.

[29] Guide to Software Composition Analysis. https://snyk.io/series/open-source-security/software-composition-analysis-sca/, 2023.

[30] Problem installing Redis of OSSPolice. https://github.com/osssanitizer/osspolice/issues/2, 2023.

[31] Scantist. https://scantist.io/, 2023.

[32] Software Composition Analysis Explained. https://www.whitesourcesoftware.com/resources/blog/software-composition-analysis/, 2023.

[33] zlib. https://zlib.net/, 2023.

[34] Michael Backes, Sven Bugiel, and Erik 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*, pages 356–367, 2016.

[35] Brenda S Baker, Udi Manber, and Robert Muth. Compressing differences of executable code. In *ACMSIGPLAN Workshop on Compiler Support for System Software (WCSS)*, pages 1–10. Citeseer, 1999.

[36] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 189–200, 2016.

[37] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. NIPS 2018, 2018.

[38] Tristan Benoit, Jean-Yves Marion, and Sébastien Bardin. Scalable program clone search through spectral analysis. In *Proceedings of the 31th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023.

[39] The National Cyber Security Centre. Log4j vulnerability. https://www.ncsc.gov.uk/information/log4j-vulnerability-what-everyone-needs-to-know, 2021.

[40] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. BinGo: Cross-architecture cross-OS binary search. FSE, 2016.

[41] Yang Chen, Andrew E Santosa, Asankhaya Sharma, and David Lo. Automated identification of libraries from vulnerability data. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, pages 90–99, 2020.

[42] Yang Chen, Andrew E Santosa, Ang Ming Yi, Abhishek Sharma, Asankhaya Sharma, and David Lo. A machine learning approach for vulnerability curation. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 32–42, 2020.

[43] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*, pages 2702–2711. PMLR, 2016.

[44] Yaniv David, Nimrod Partush, and Eran Yahav. Statistical similarity of binaries. PLDI, 2016.

[45] Yaniv David, Nimrod Partush, and Eran Yahav. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 79–94, 2017.

[46] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[47] S. H. Ding, B. M. Fung, and P. Charland. Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *IEEE S&P*, 2019.

[48] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke 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*, pages 2169–2185, 2017.

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

[50] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 2008.

[51] Han Gao, Shaoyin Cheng, Yinxing Xue, and Weiming Zhang. A lightweight framework for function name reassignment based on large-scale stripped binaries. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 607–619, 2021.

[52] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary. ASE, 2018.

[53] Microsoft & GitHub. CodeQL. https://codeql.github.com/, 2021.

[54] Yi Gui, Yao Wan, Hongyu Zhang, Huifang Huang, Yulei Sui, Guandong Xu, Zhiyuan Shao, and Hai Jin. Cross-language binary-source code matching with intermediate representations. *arXiv preprint arXiv:2201.07420*, 2022.

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

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

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

[58] Nasif Imtiaz, Seaver Thorn, and Laurie Williams. A comparative study of vulnerability reporting by software composition analysis tools. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2021.

[59] Jiyong Jang, Maverick Woo, and David Brumley. Towards automatic software lineage inference. In *USENIX Security*, 2013.

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

[61] Ang Jia, Ming Fan, Wuxia Jin, Xi Xu, Zhaohui Zhou, Qiyi Tang, Sen Nie, Shi Wu, and Ting Liu. 1-to-1 or 1-to-n? investigating the effect of function inlining on binary similarity analysis. *ACM Transactions on Software Engineering and Methodology*, 32(4):1–26, 2023.

[62] Ling Jiang, Hengchen Yuan, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun 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*, pages 1383–1395, 2023.

[63] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. Pdiff: Semantic-based patch presence testing for downstream kernels. In

Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pages 1149–1163, 2020.

[64] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM: Software protection for the masses. SPRO, 2015.

[65] Ulf Kargén and Nahid Shahmehri. Towards robust instruction-level trace alignment of binary code. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 342–352. IEEE, 2017.

[66] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *IEEE Transactions on Software Engineering*, 2022.

[67] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196. PMLR, 2014.

[68] Menghao Li, Pei Wang, Wei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, Wei Huo, and Wei Zou. Large-scale third-party library detection in android markets. *IEEE Transactions on Software Engineering*, 46(9):981–1003, 2018.

[69] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. Libd: Scalable and precise third-party library detection in android markets. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 335–346. IEEE, 2017.

[70] Xuezixiang Li, Yu Qu, and Heng Yin. Palmtree: Learning an assembly language model for instruction embedding. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3236–3251, 2021.

[71] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[72] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. αdiff: Cross-version binary code similarity detection with DNN. In *ASE*, 2018.

[73] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 389–400, 2014.

[74] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering*, 43(12):1157–1177, 2017.

[75] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. Libradar: fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th international conference on software engineering companion*, pages 653–656, 2016.

[76] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. How machine learning is solving the binary function similarity problem. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2099–2116, 2022.

[77] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. Safe: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 309–329. Springer, 2019.

[78] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[79] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. Probabilistic disassembly. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1187–1198, 2019.

[80] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

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

[82] Kexin Pei, Jonas Guan, David Williams King, Junfeng Yang, and Suman Jana. Xda: Accurate, robust disassembly with transfer learning. *arXiv preprint arXiv:2010.00770*, 2020.

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

[84] Edward J Schwartz, Cory F Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S Havrilla, and Charles Hines. Using logic programming to recover c++ classes and methods from compiled executables. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 426–441, 2018.

[85] Paria Shirani, Leo Collard, Basile L Agba, Bernard Lebel, Mourad Debbabi, Lingyu Wang, and Aiman Hanna. Binarm: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 114–138. Springer, 2018.

[86] Paria Shirani, Lingyu Wang, and Mourad Debbabi. Binshape: Scalable and robust binary library function identification using function shape. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 301–324. Springer, 2017.

[87] Synopsys. The heartbleed bug. https://heartbleed.com/, 2020.

[88] Wei Tang, Yanlin Wang, Hongyu Zhang, Shi Han, Ping Luo, and Dongmei Zhang. Libdb: An effective and efficient framework for detecting third-party libraries in binaries. *19th International Conference on Mining Software Repositories*, 2022.

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

[90] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. Wukong: A scalable and accurate two-phase approach to android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 71–82, 2015.

[91] Huaijin Wang, Pingchuan Ma, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. sem2vec: Semantics-aware assembly tracelet embedding. *ACM Transactions on Software Engineering and Methodology*, 32(4):1–34, 2023.

[92] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2614–2627, 2021.

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

[94] Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo 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)*, pages 860–872. IEEE, 2021.

[95] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *CCS*, 2017.

[96] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. Spain: security patch analysis for binaries towards understanding the pain and pills. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 462–472. IEEE, 2017.

[97] Sheng Yu, Yu Qu, Xunchao Hu, and Heng Yin. Deepdi: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly. In *Proc. of the USENIX Security Symposium*, 2022.

[98] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1145–1152, 2020.

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

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

[101] Xian Zhan, Lingling Fan, Sen Chen, Feng Wu, Tianming Liu, Xiapu Luo, and Yang 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)*, pages 1695–1707. IEEE, 2021.
[102] Fangfang Zhang, Yoon-Chan Jhi, Dinghao Wu, Peng Liu, and Sencun Zhu. A first step towards algorithm plagiarism detection. In *ISSTA*, 2012.
[103] Hang Zhang and Zhiyun Qian. Precise and accurate patch presence test for binaries. In *USENIX Security*, 2018.
[104] Jiexin Zhang, Alastair R Beresford, and Stephan 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*, pages 55–65, 2019.
[105] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. Detecting third-party libraries in android applications with high precision and recall. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 141–152. IEEE, 2018.
[106] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *NDSS*, 2019.

TABLE 7. EXECUTABLES IN OUR BENCHMARK DATASET.

Binaries	*#Reuses*	*#Toolchains*	*#Functions*	*Sizes (Bytes)*
controlblock	7	gcc clang	1,013 1,126	721,840 699,592
db_bench	14	gcc clang	10,525 10,148	6,847,776 5,764,976
dosbox_core	18	gcc clang	11,978 8,723	9,656,088 7,418,784
eth_sc	13	gcc	4,896	3,462,128
example	5	gcc clang	1,879 1,892	811,176 790,648
hyriseSystemTest	10 8	gcc clang	41,640 4,995	16,137,960 4,229,344
kvrocks	8	gcc clang	12,614 63,283	12,484,216 15,327,528
nano_node	13	gcc	21,183	15,327,528
pagespeed_automatic_test	33	gcc clang	50,290 41,638	29,856,672 30,514,480
prometheus_test	4	gcc clang	9,075 8,608	2,627,368 2,443,216
replay-sorcery	10	gcc clang	7,938 8,506	4,603,136 5,759,864
tendisplus	12	gcc	11,960	10,206,528
turbobench	29	gcc clang	2,188 2,811	4,288,064 4,597,768
yuzu-cmd	22	gcc	31,594	19,735,192
ChronoEngine.dll	4	MSVC	242,797	27,265,536
TortoiseGitMerge.exe	7	MSVC	28,227	6,081,536
TortoiseGitProc.exe	6	MSVC	76,902	20,257,792
WinSparkle.dll	4	MSVC	41,209	6,030,336
grpc_csharp_ext.dll	7	MSVC	92,799	16,130,048
iw4x.dll	9	MSVC	81,482	10,910,208
k4a.dll	6	MSVC	2,130	651,648
k4arecord.dll	8	MSVC	6,690	1,722,248
k4aviewer.exe	7	MSVC	5,967	2,403,704
libclamavd.dll	13	MSVC	18,212	11,654,144
tic80.exe	15	MSVC	22,431	8,899,072

Among 25 software, ten of them (e.g., `ControlBlock`) were compiled using both `gcc` and `clang`, resulting in a total of 35 executables being analyzed.

A. Possible improvements by fine-tuning γ

According to γ varying from 0 to 1000, Fig. 11 shows the changes of BSA tools' F1 scores for analyzing all 35 executables, with all enhancements enabled. Except $\texttt{Commercial}_B$ happened to achieve the best F1 scores with $\gamma = 100$, the performance of other BSA tools can be further improved with a fine-tuned γ.

B. BSA Tools Selection

B.1. B2B Tools. Finally, we select five B2B tools (see their publications in Table 1). They all share the same pipeline illustrated in Fig. 5.

`SAFE` employs `word2vec` [78] to produce instruction embeddings; then, it treats a function as an instruction sequence and embeds it with a self-attentive network. Other selected works use the CFG to generate the embedding vector of a function.

`Asm2vec` collects paths with random walks on the CFG and produces function embeddings by encoding the collected paths with a PV-DM model [67].

`PalmTree` uses BERT [46] to produce context-sensitive instruction embeddings. Then, the embedding of a basic block for $\texttt{PalmTree}_G$ is computed by the mean of embeddings of its instructions. Specifically, $\texttt{PalmTree}_G$ uses Structure2Vec [43] to generate the embedding for a function's CFG.

$\texttt{PalmTree}_B$ is a combination of `PalmTree` and $\texttt{Commercial}_B$. Similar to $\texttt{PalmTree}_G$, it first uses a recurrent neural network and instruction-level embeddings to generate basic block embeddings, then embeds the function CFG with a gated graph neural network (GGNN) [71].

$\texttt{Commercial}_B$ also employs GGNN, and it further uses additional knowledge from a decompiler to produce the basic block embeddings.

The following elaborates on the excluded B2B tools:

`DeepBinDiff` is excluded since it produces block-level embeddings, which is unsuitable for our workflow.

`InnerEye` decomposes the CFG into multiple paths and computes the path similarity score with the longest common sequence (LCS) algorithm; thus, we also exclude it since it does not produce vector representations for assembly functions.

`Gemini` (CCS'17) and `VulSeeker` (ASE'18) employ manually-selected features to represent basic blocks, but later tools like `SAFE` (DIMVA'19), `InnerEye` (NDSS'19), and `DeepBinDiff` (NDSS'20) all depend on instruction-level embeddings and show better performance. In addition, the `PalmTree` (CCS'21) paper presents a combo of `PalmTree` and `Gemini`, which is highly effective in function-level BSA. Hence, we exclude `Gemini` and `VulSeeker` but implement the combo of `PalmTree` and `Gemini` as a variant (i.e., $\texttt{PalmTree}_G$).

GMN is a recent work that shows promising results in BSA. However, we do not evaluate GMN due to inefficiency. GMN does not generate an embedding vector for each function but instead computes the similarity between two functions with a learned model. Hence, it is not scalable to large-scale BSCA since the similarity calculation process of GMN cannot be accelerated with a vector database.

B.2. B2S Tools. We eventually select the industry-developed work, `CodeCMR` [99], which uses cross-modality deep learning to compare the latent representations of assembly and source code embeddings. It does *not* require compiling OSS instances and exhibits high accuracy compared with modern B2B tools. Since `CodeCMR` is not publicly available, we send the data to its authors and get the embeddings of functions. The following list excluded B2S tools.

`BAT` and OSSPolice, which both rely on the string literals and exported function symbols for C/C++ executables, are designed to support BSCA tasks. We evaluate `BAT` as a baseline. We exclude OSSPolice since we fail to set it up

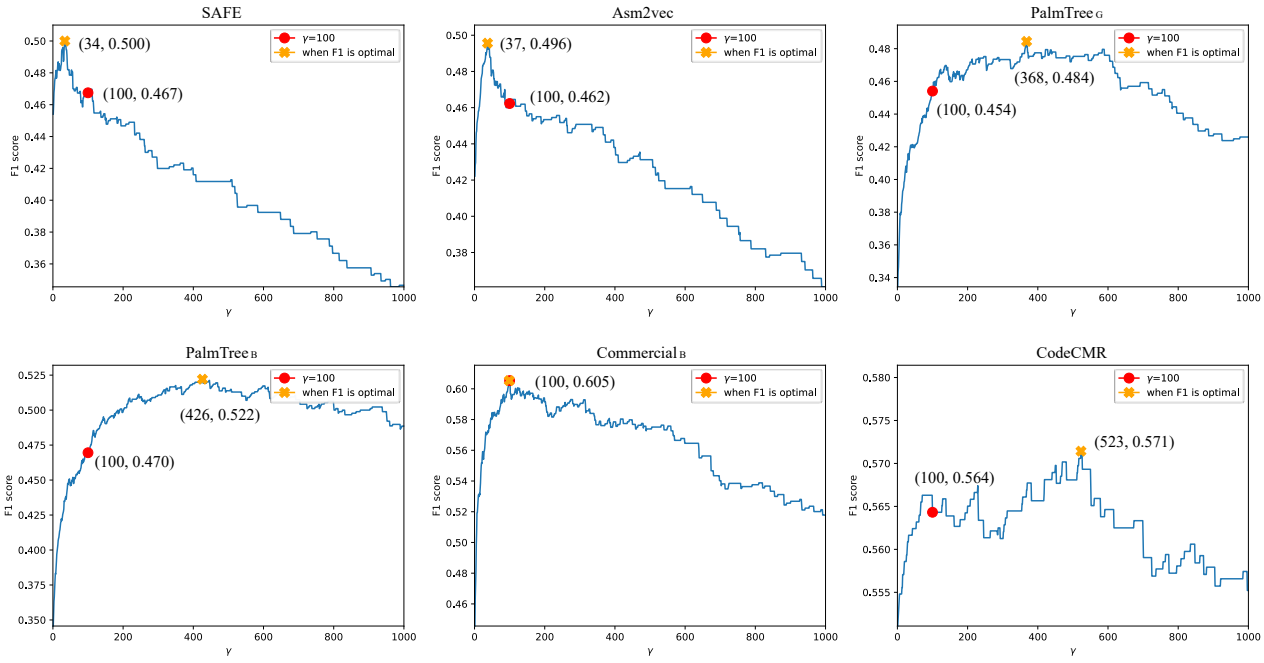


Figure 11. γ-F1 relations of six BSA tools.

due to its dependency issues [30] and likely incomplete code [28].

B2SFinder and XLIR require compiling source code to LLVM-IR for extracting `switch` and `if` structures. However, due to the complex composition of our OSS database (e.g., OSS without `Makefile` or with features not supported by LLVM), it is impracticable to compile all these projects into LLVM-IR.

FIBER and BugGraph still need to compile the database to binaries with various configurations, which cannot leverage the advantage of B2S techniques mentioned in Sec. 2.2.