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

Abstract—**Software composition analysis (SCA) has attracted the attention of the industry and academic community in recent years. Given a piece of source code or Android APKs, SCA facilitates extracting certain components from the input software and match the extracted components with open source software (OSS) libraries. Despite the prosperous development of SCA, binary code SCA (BSCA) is highly challenging and still underdeveloped. Few existing BSCA solutions are closed source and suffer from low precision. Given that said, another line of research, namely binary similarity analysis (BSA), has been developed progressively to decide the similarity of two executables. De facto BSA techniques, often based on deep learning techniques, efficiently analyze large-scale executables with high precision.**

This study explores bridging the gap between state-of-the-art (SOTA) BSA and BSCA. We build the first comprehensive benchmark dataset with considerable manual effort. Then, we establish our BSCA pipeline, by extending the SOTA SCA solution. Particularly, we concretize the key procedure of BSCA, namely matching a binary component with OSS, with 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. With manual inspection on the failed cases, we propose three enhancements, whose combination improves the F1 score of BSCA for nearly 30% and outperforms SOTA commercial BSCA software. We discuss several open challenges and potential solutions to augment BSCA solutions.

I. Introduction

Given a piece of software, software composition analysis (SCA) facilitates identifying potential use of third-party and open-source software (OSS) components. The success adoption of SCA enables localizing potentially vulnerable or outdated OSS components, reducing risk factors, and leading to healthy open source usage. To date, production SCA tools are offered by the industry [5, 7, 8, 19, 22], and many research works [25, 31, 32, 50, 51, 56, 63, 69, 71, 77, 80, 81] have been published in recent few years. Many of them [7, 19, 22, 63, 71] require access to the source code of the input software, the precise package structures [25, 56, 81], class/methods declarations [25, 50, 51, 80, 81], and manifest files [77] to search against OSS databases.

Despite the prosperous development and adoption of SCA techniques, it is worth noting that in many real-world, security-sensitive scenarios, input software's source code is *not* always available. In fact, when performing SCA over commercial off-the-shelf (COTS) software, the primarily available information is the program *binary code*. With this regard, binary software composition analysis (BSCA) becomes a demanding, yet under-explored field [5, 8].

On the other hand, recent years have seen a tremendous progress made by the industry and the research community toward binary similarity analysis (BSA) [35, 37, 57, 72]. BSA quantifies the similarity between two binary code samples, which forms the basis of various important software engineering and security applications. For example, BSA promotes malware analysis by comparing suspicious code with known malware families to determine if it is malicious [38, 46]. BSA also helps discover code clones and algorithm plagiarism in executable [55, 78].

Motivation. From a holistic view, BSA and BSCA shares conceptually similar technical demand about "binary code matching." With over twenty years of development [26, 41], BSA has been enhanced from varying angles using syntactic-based, structure-based, and semantics-based methods. To date, advanced BSA solutions have been extensively using deep neural networks (DNNs) and its enabled representative learning and large language models (LLMs)-based embedding [35, 52, 57, 74, 75]. In contrast, SCA/BSCA, as an emerging demand in our community, is under explored. It is seen that contemporary SCA/BSCA may use either clustering-based methods (e.g., LibD [50, 51]) or similarity-based matching (e.g., LibScout [25]). One state-of-the-art (SOTA) SCA work [77] is based on extracting a hierarchical collection of features.

Overall, this research is motivated by the observation that BSA and SCA/BSCA, two conceptually similar fields, exhibit a notable difference in the technical solution; one may question if the technical solutions of SCA/BSCA is "outdated" and less reliable. And due to the fact that BSCA serves as the cornerstone for many security applications and risk assessment, analysis errors are particularly unwanted, which can have a major impact on the trustworthiness of today's software security land space. It is thus intriguing and urgent to know if de facto BSA solutions can be extended to the demanding field of BSCA and achieving high effectiveness. Overall, this paper aims to provide a systematic understanding and study applying SOTA BSA solutions in the field of BSCA.

This paper conducts the first comprehensive study to analyze the capability of de facto BSA solutions in their support of BSCA. We first spend a considerable manual effort to form the comprehensive dataset for BSCA benchmarking, a dataset containing 35 complex real-world executables (e.g., Chrono Physics Engine [14] and Nano [12]) where each executable contains reuse relations with 12.6 production OSS libraries on average. The totally involved OSS libraries are 255, across two platforms (Windows and Linux) and three compilation toolchains (GCC, Clang, and MSVC). Then, we extend six SOTA BSA solutions for BSCA over our formed

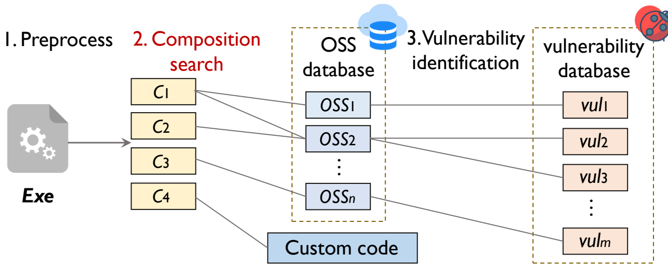


Fig. 1. SCA workflow. In the context of BSCA, this paper primarily explores to concrete the second phase, composition search, with various BSA techniques.

benchmark dataset. We also compare with available commercial SCA/BSCA tools to understand "how far" our BSA-empowered BSCA pipeline can achieve.

We summarize a series of key findings to assess the gaps between BSA in supporting BSCA. We find that BSCA has distinct requirement than standard BSA, for which the latter case may *overly* emphasize the necessity of precisely extracting program semantics. Often, syntactic and CFG features are *sufficient* to attribute a function to real-world third-party OSS libraries. However, identifying the exact version of OSS libraries is very challenging. We also analyze lessons and guidance that can be used to calibrate and extend the current BSA solutions in BSCA. Following, we design three low-cost and highly effective enhancement strategies from different perspectives to enhance BSA for BSCA. Applying individual and the combination of these strategies, we can largely enhance the effectiveness of BSA for BSCA (for over 30% F1 score), *outperforming SOTA commercial BSCA solutions* and achieving a close performance with the SOTA SCA solution. In sum, we make the following contributions:

- This research is primarily motivated by the conceptual similarity and the technical gap between BSA and BSCA. We aim to assess advanced BSA solutions in their usage of BSCA, a highly demanding yet under-explored application field.
- We conduct the first comprehensive study in bridging BSA solutions to BSCA. To do so, we form the first *comprehensive* and *version-representative* BSCA benchmark dataset. We benchmark SOTA, "out-of-the-box" BSA solutions in its support of BSCA, summarizing key findings that can be leveraged as guidance for future research.
- On the basis of our findings, we design three enhancement strategies, where each of them (and their combination) can significantly enhance the effectiveness of BSA solutions in BSCA, outperforming the commercial BSCA software.

Artifact Availability. We have released our artifacts at [6]. We will maintain them for future research comparison and usage.

II. Preliminary

A. Software Composition Analysis (SCA)

1) SCA Overview: Open-source software (OSS) is frequently reused to facilitate the fast development of applications. However, extensive reuse of OSS gives attackers opportunities to exploit the known vulnerabilities, which causes severe security issues for software users. On the other hand, developers may not know which OSS is assembled in their software because of many direct and transitive dependencies. Thus, SCA [20, 21] has become a common practice that helps developers to quickly track and analyze any OSS brought into a project. Hence, developers can address security risks from known vulnerabilities and license issues. Formally, let $C = F(S, D)$ be the SCA procedure, where S denotes the input software to be analyzed, 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 [67] or Log4j [29].

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

① **Component Dissection.** The input software S needs to be pre-processed and dissected into a list of code components $c \in S$ first. Each component will be leveraged for component identification. Overall, it is not as straightforward as it looks to define "component" in software. To date, we have seen techniques proposed to dissect S in terms of differential hierarchical structures of the software. For instance, in typical Android APKs, the directory structure can often reflect rich information on a third-party library. Hence, prior Android SCA works often leverage such information to dissect Android apps [25, 51, 56, 80, 81]. Furthermore, files, class hierarchies (particularly for object-oriented languages), as well as functions are also leveraged to form components.

② **Component Search.** 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 $c \in S$, we need to decide if it is originated from any OSS $\text{oss}_i \in D_{\text{oss}}$. This is the *key challenge* for SCA, in the sense that we need to decide the similarity between c and records in D_{oss}. Today, most existing SCA works aim at enhancing the accuracy of matching $c \in S$ with $\text{oss}_i \in D_{\text{oss}}$. Ideally, each $c \in S$ from OSS projects is correctly matched to its corresponding OSS project with the correct version. A component c, however, should not be matched to any $\text{oss}_i \in D_{\text{oss}}$ if it is a user-written code; user-written code is referred to as "custom code" in Fig. 1.

③ **Valuable Information Identification.** Once a set of reused OSS libraries $L = \{l_1, l_2, \cdots, l_n\}$ has been detected in S, the next step would be exploring the value information v_i within $l_i \in L$. Depending on the usage scenario, v could be vulnerability information [31], e.g., l_1 denotes a specific version of the OpenSSL library where the heartbleed vulnerability exists. Similarly, v could be certain sensitive license information [42]. Note that OSS does not always imply "unrestricted usage," e.g., software under the BSD license requires users to retain the copyright notice and credit the software's developers. Therefore, detecting the usage of OSS under the BSD license may be a "showstopper" before the distribution of S. Overall, deciding valuable information v

in l is addressed case by case, usually solved with user-specified rules. In prior literature [25, 36, 51, 56, 71, 77, 80], the mapping rules are often assumed to be available in a "valuable information database" for a query.

2) SCA metrics: Existing works [63, 68, 71, 77, 80] essentially leverage the **precision**, **recall** and **F1 score** to benchmark SCA. Let the ground truth, denoting the OSS projects reused in software S, be R^*. Let the OSS projects detected by a SCA tool in S be R. Then, true positive cases are $TP = R \cap R^*$. Thus, the precision score can be computed as $\frac{|TP|}{|R|}$, recall score can be computed as $\frac{|TP|}{|R^*|}$, and F1 score can be computed by $\frac{2 \times precision \times recall}{precision + recall}$. In addition, existing works also measure and report the time cost of launching SCA.

It is easy to see that we need to prepare a ground-truth dataset R^* from input software to benchmark SCA techniques. This is not an easy task, given that small synthetic datasets may hardly demystify the true potential of SCA, whereas building a large-scale dataset requires considerable human effort. We find that recent works like ATVHunter manually analyzed open-source Android applications to annotate specific third-party libraries and version information. For Android apps, such ground truth information is relatively easy to obtain from the configuration files and source code. Nevertheless, it becomes much harder for executable files compiled from C/C++ programs, where most meta information is absent. In fact, one of our contributions in this work is to build a comprehensive real-world dataset for BSCA benchmark; see Sec. IV-C for the details.

3) BSCA: BSCA [5, 8] denotes a demanding sub-area of SCA. BSCA aims to conduct SCA over executables compiled from *C/C++ programs*, assuming that C/C++ source code is unavailable. To date, a number of works have launched SCA toward source code and Android applications. Nevertheless, BSCA is still largely under-explored.

BSCA represents a practical, demanding, yet challenging need, given the indispensable role of BSCA when the source code is unavailable (particularly in security-related applications), such as analyzing legacy code. Similarly, the research community [45] also champions the necessity of directly analyzing the deployed executables and identifying all OSS.

Android APKs vs. C/C++ Executable. To clarify, SCA for Android APKs also frequently targets the low-level code in executable format. Nevertheless, reverse engineering Android apps is deemed as *much simpler* comparing with decompiling x86 executables compiled from C/C++ code. Accordingly, the available information in launching SCA for Android APKs is richer. In fact, when launching SCA for Android APKs [25, 51, 56, 77, 80, 81], various well-established reverse engineering tools, including APKTool [3] , dex2jar [9] and Androguard [1], are employed to recover the package structures, method prototypes, and class inheritance dependencies. Decompiling C/C++ executable is much harder, and recovering those high-level information (e.g., class inheritance dependencies) is not well addressed yet [64]. Performing BSCA has its unique challenge, and this empirical study launches the ever first in-depth study to explore technical solutions that are feasible.

B. Binary Similarity Analysis (BSA)

We have seen many research works emerging in the software engineering, security, and programming language communities about BSA [41]. Overall, given two pieces of code, BSA decides how similar they are. The similarity is usually a score ranging from 0 to 1.

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**). From our study, the research community pays much more attention on B2B than B2S. A recent survey [41] investigated 70 binary code similarity approaches, but we can merely find six tools (i.e., BAT [43], OSSPolice [36], `FIBER` [79], B2SFinder [76], `CodeCMR` [75] and XLIR [40]) aiming at B2S.

Most recent B2B works perform semantics-aware search at the level of assembly functions [24, 27, 35, 41, 52, 72, 74, 82]. That is, they aim to determine the semantical similarity of two functions in binary code, although these two functions may appear syntactically distinct, for example, as a result of compiler optimization. The function search task is similar to that of information retrieval. Given an input assembly function f and a repository of assembly functions RP, binary code search engines retrieve the top-k functions $t \in RP$ ranked by their semantic similarity with f. Similarly, the function-level B2S works [47, 75] can be evaluated by replacing the binary functions in the RP with functions in source code.

Advantages of B2S for BSCA. Though B2B appears to be the mainstream solution for BSA [41], using B2B for BSCA faces an extra and practical challenge: we must compile OSS projects into binary code. In production, BSCA is envisioned to analyze hundreds of OSS projects with various dependencies and compilation toolchains required. Therefore, compiling all OSS projects requires enormous resources and considerable manual effort. Some OSS projects available online (e.g., RapidJSON [16]) do not even provide a `Makefile`. Additionally, OSS can upgrade and change, and it may not be an one-time effort to hash out the compilation procedure. Hence, it appears a dilemma to use either B2B or B2S in BSCA.

Immaturity of B2S. B2S is much less explored than B2B. Recent B2S papers [42, 76] propose to extract features (e.g., constants, strings) that are unchanged after compilation. Some studies [40, 47] convert assembly and source code into the compiler intermediate language (IR) for matching; however, compiling OSS source code into IR is also challenging, as noted above. Overall, we view B2S solutions are much fewer and immature. In this study, we assess `CodeCMR` [75], the SOTA B2S solution developed by industry (see Sec. IV-A).

III. Demystifying BSCA Technical Pipeline

Motivation. There is a high demand of performing accurate and dependable BSCA, envisioning the practical need of analyzing closed-source software and track any (unsafe or outdated) open-source component brought into executable. Given that said, it is still *unclear* about the best practice of launching BSCA in real-world scenarios. The industry-leading security vendors are promoting their BSCA solutions, including CodeSentry offered by GrammaTech [8], Black Duck offered by Synopsys [5],

and Scantist [18]. However, *none* of these commercial tools disclose their technical solutions in detail.

> *In short, we believe that the academy and our community lack a systematic and in-depth understanding to calibrate the technical solution and uncover the best practical of performing BSCA. This motivates our study in this paper.*

As reviewed in Sec. II-A, de facto SCA works primarily undertake a three-step approach, where the second step, component identification, denotes the central technical challenge. We now analyze the potential technical solutions for each step.

① **Component Dissection.** Typical executables compiled from C/C++ programs lack high-level program structures. As noted in Sec. II-A3, it is generally challenging and inaccurate to recover such information. In contrast, program function information can usually be obtained accurately in even stripped binaries. For instance, [59, 62] achieve above 95% F1 score while identifying function entries. Hence, we give a clear definition that *assembly functions should form the "components" in an executable.*

We argue that it is a practical and beneficial setting to take functions as the components, whose reasons are three-fold. First, many prior SCA tools (e.g., LibD, ATVHunter and Centris) also rely on function-level component to identify the reused OSS. Second, considering each OSS project has multiple functions, treating functions as components to match denotes a fine-grained focus, which marks an OSS as "reused", only if the specific function is within the analyzed executable. *Partial reuses* are prevalent [71], where only a few functions exist in the executable. Third, nearly all existing BSA works focus on function-level similarity. Hence, treating function as components is reasonable and ease the burden of bridging various BSA techniques for BSCA.

② **Component Identification.** It is the key challenge of SCA/BSCA. At this step, we extensively studied prior studies on SCA. Particularly, we refer to recently published SCA works, including LibRadar [56], LibD [51], LibID [80], LibScout [25], Centris [71] and ATVHunter [77]. We classify the component identification solutions used by existing SCA into the following two categories (the taxonomy is from [77]):

ⓐ *Clustering-based detection*, which relies on extracted lightweight signatures to identify and extract components (e.g., Android third-party libraries) from application code [50, 51]. Typical signatures may include package names and structures, API calls, and other obvious meta information in the software.

ⓑ *Similarity-based detection*, which performs software similarity analysis to match a code component to known OSS. Software similarity comparison methods can be performed at different software hierarchical representations, including opcode sequences, code structures, and dependency relations. Recent SCA works also explore using fuzzy hash or machine learning for higher matching accuracy.

Clustering-based methods are believed to suffer from various limitations such as missing some niche and new OSSs, and low accuracy in distinguishing different OSS versions [77]. Accordingly, de facto SCA methods are primarily based on similarity-based detections. Therefore, we hypothesize that well-performing BSA shall be used to form the technical solution of component identification. As reviewed in Sec. II-B, we will benchmark both B2B and B2S solutions at this step.

③ **Valuable Information Identification.** We believe there is no extra difficulty for ③ comparing BSCA with existing SCA solutions. The "valuable information database" [31, 32] adopted by existing SCA can be simply reused here. In this study, we consider the task of vulnerabilities as the "value information" a BSCA pipeline will finally output. Nevertheless, other tasks like license checking can also be the target of BSCA.

IV. Solving Component Identification with BSA

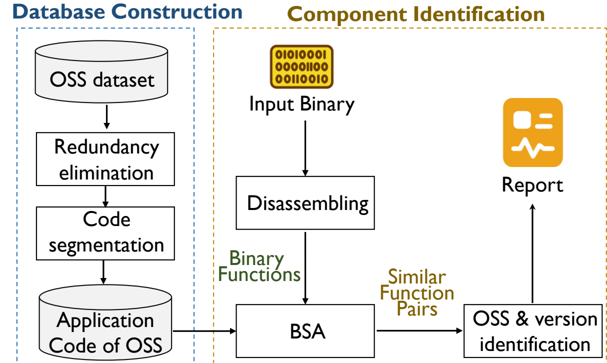


Fig. 2. Workflow of BSCA.

Sec. III has clarified that the second step of BSCA, component identification, denotes the central technical challenge that needs to be addressed by BSA. We present the component identification technical pipeline in Fig. 2; note that this is derived from [71], the SOTA SCA work. Overall, given a list of assembly functions extracted from the input executable (due to component dissection), this step identifies all reused OSS projects and their exact versions.[1]

As shown in Fig. 2, there are two primary phases to conduct component identification, namely, database construction and component search. The database construction phase extracts and stores OSS into a database for query, whereas the component identification phase iterate over functions in the input executable and query the database. Two optimization techniques, redundancy elimination and code segmentation, are adopted in the first phase to shrink the size of formed database and improve search accuracy. We introduce them in Sec. IV-B.

When performing component search, we use the equipped BSA solution to compare each assembly function f (a component) with OSS functions in the database. BSA solutions, in their out-of-the-box setup, yields a similarity score between f and OSS functions. Finally, given the matched function pairs, we identify the reused OSS projects and generate reports. Before describing the detailed workflow in Sec. IV-B, we introduce how we select representative BSA solutions.

A. BSA Solution Selection

A BSA tool compares assembly functions with functions in the OSS database, then produces the knowledge of reused OSS and specific versions associated. Overall, besides the mundane requirements of *high accuracy* and *available implementation*,

[1]In our study, the component dissection phase (recovering assembly functions from an executable), is carried by well-developed decompilers (e.g., IDA-Pro [44], Radare2 [15]) required by the employed BSA techniques.

TABLE I
STUDIED VISIBLE BSA SOLUTIONS.

	Venue	Interal	Tool
SAFE [57]	DIMVA'19	B2B	SAFE [17]
Asm2vec [35]	S&P'19	B2B	asm2vec-pytorch [4]
PalmTree$_G$ [52]	CCS'21	B2B	PalmTree [13] and Gemini [10]
PalmTree$_B$ [52]	CCS'21	B2B	PalmTree [13] and Commercial$_B$
Commercial$_B$		B2B	Commercial$_B$
CodeCMR [75]	NIPS'20	B2S	CodeCMR

the BSA tool must be *highly efficient*. As shown in Fig. 2, BSCA requires to compare each assembly function against every record in the OSS database. For production usage, the OSS database is typically extremely large, with millions of functions from various OSS instances and different versions. As a practical setup, when selecting BSA, we require the average pair-wise similarity comparison takes less than 10^{-5} second; see Sec. VII for the cost assessment of BSCA.

Selection of B2B solutions. There are many B2B works published at top-tier conferences. We first collect relative publications on ICSE, CCS, USENIX, IEEE S&P, PLDI, ASE, NDSS and DIMVA from 2017 and get 15 works [33, 35, 37, 39, 48, 52, 54, 57, 60, 65, 66, 70, 72, 73, 82]. After checking their availabilities, the remaining 7 works are Gemini [72], VulSeeker [39], SAFE [57], InnerEye [82], Asm2vec [35], DeepBinDiff [37], and PalmTree [52]. They all employ machine learning techniques.

We exclude DeepBinDiff, which produces block-level embeddings and is unsuitable for our workflow. InnerEye released the majority of their code without scripts for preprocessing [11]. Gemini (CCS' 17) and VulSeeker (ASE'18) employ manually-selected features to represent basic blocks. However, latter solutions like SAFE (DIMVA'19), InnerEye (NDSS'19), and DeepBinDiff (NDSS'20) mostly depend on instruction-level embeddings and show better performance. Moreover, PalmTree (CCS'21) aims at embedding instructions for applications like BSA. PalmTree's authors assembled it with Gemini and proved its efficiency; hence, we implement their idea as a variant namely PalmTree$_G$. We thus exclude Gemini and VulSeeker.

Finally, we select five B2B solutions (see their publications in Table I). SAFE employs word2vec [58] 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 structure of a function. PalmTree utilizes BERT [34] to produce context-sensitive instruction embeddings. Then the embedding of a basic block for PalmTree$_G$ is computed by the mean of embeddings of its instructions. Lastly, PalmTree$_G$ uses a Siamese network [28] to generate the embedding for a function's CFG. PalmTree$_B$ is a combination of PalmTree and Commercial$_B$. Similar to PalmTree$_G$, it first generates basic block embeddings with a recurrent neural network and then embeds the function CFG with GGNN [53]. Commercial$_B$ also employs GGNN, while it utilizes additional knowledge from a decompiler to produce the basic block embeddings. Asm2vec does not use the CFG structure of a function directly. It collects paths with random walks on the CFG and produces function embeddings by encoding the collected paths with a PV-DM model [49].

We reuse the officially-released models of SAFE and PalmTree. The GNN of PalmTree$_G$ and PalmTree$_B$ are trained with OpenSSL compiled by gcc and clang. We train Asm2vec with our OSS database as it is unsupervised learning.

Selection of B2S Solutions. As described in Sec. II-B, the choices of B2S solutions are few (i.e., BAT [42], OSSPolice [36], B2SFinder [76], FIBER [79], CodeCMR [75],BugGraph [47], and XLIR [40]). BAT and OSSPolice, relying on the string-level signature, are designed as BSCA tools. We evaluate BAT as a baseline. B2SFinder requires compiling source code to LLVM-IR for extracting switch and if structures. However, due to the complex composition of our OSS projects dataset (e.g., no Makefile and features not supported by LLVM), it is too complicate to get their LLVM-IR. We exclude XLIR for the same reason. Besides, FIBER and BugGraph are B2B solutions indeed. They compile the database to binaries with various configurations, which cannot utilize the advantage of B2S techniques.

Eventually, we select CodeCMR, which uses cross-modality deep learning to compare the latent representations of assembly and source code embeddings. It does **not** require compiling OSS, and exhibits high accuracy in comparing with contemporary B2B works [75]. Since CodeCMR is not publicly available, we send the data to CodeCMR's author and get the embeddings.

Considering existing BSCA Solutions. In addition to the analysis of SOTA BSA tools, we also take into account existing BSCA solutions, including a commercial tool, as a baseline for comparison. Commercial$_A$ is a wildly-used commercial BSCA tool, and BAT [42] is a research work that identifies the reused OSS project with collected string-level signatures.

B. Workflow of BSCA

Building the OSS database is essential to component identification. A SOTA SCA, Centris [71], proposed *redundancy elimination* and *code segmentation* to reduce the size of the database without undermining the accuracy. We extend them in our BSCA scenarios.

Redundancy Elimination. This scheme reduces the size of the database, whose observation is that a new OSS version is often built on top of its earlier version, leaving many functions unchanged. Hence, storing only one copy of such unchanged functions across all versions is sufficient. We reuse the implementation of Centris for B2S solutions and extend this idea to build a binary OSS database for B2B solutions.

Code Segmentation. Observation over real-world OSS shows that an OSS project itself may borrow code components from other OSS instances. Therefore, segmenting an OSS project's application code out can reduce the false positives caused by the borrowed code. The code segmentation strategy cross-compares functions across different OSS instances, and only the function with an earlier check-in timestamp is kept when two functions are similar. Centris uses the distance of TLSH [61] to evaluate the similarity between two source functions. Nevertheless, given the assembly code can change largely with different compilation flags, we take a conservative approach that two functions are deemed similar only when they have identical hash values.

Component Identification. With redundancy elimination and code segmentation performed, we build the database with OSS components stored. The algorithm for component identification is given in Alg. 1. We iterate assembly functions F_{bin} in the

Algorithm 1 Component identification

```
1: function IDENTIFY_COMPONENT(F_bin, DB_oss, θ, k)
2:     # F_bin is the set of input functions.
3:     # DB_oss stores all OSS instances.
4:     # θ is the threshold.
5:     # k is the number of similar functions being searched.
6:     R ← list()
7:     for oss ∈ DB_oss do
8:         F_oss ← oss.functions
9:         M ← MATCH_FUNCTIONS(F_bin, F_oss, θ, k)
10:        p ← IS_REUSED(M, oss)
11:        if p = True then                ▷ The oss is identified as reused.
12:            R ← R ∪ {(oss, v)}          ▷ Return the "value info" v of oss.
       return R
13: function MATCH_FUNCTIONS(F_bin, F_oss, θ, k)
14:    M ← set()
15:    for f ∈ F_bin do
16:        T ← SEARCH_TOP_K_SIMILIAR(f, F_oss, k)     ▷ Using BSA
17:        for t ∈ T do
18:            if SIMILARITY(t, f) > θ then          ▷ f matches t
19:                M ← M ∪ {(t, f)}
       return M
```

TABLE II
STATISTICS OF EXECUTABLE IN OUR BENCHMARK DATASET.

Toolchain	*# binaries*	*# functions*	*# reuses*	*Avg. size*
gcc	14	216,828	198	9,386KB
clang	10	151,730	138	7,470KB
MSVC	11	618,846	86	10,182KB
Total	35	987,404	422	9,089KB

input executable and DB_{oss} is the OSS database. For each OSS instance oss, DB_{oss} stores a set of functions F_{oss} (after performing two optimizations mentioned above) extracted from oss (line 8). We then use MATCH_FUNCTIONS() to identify matched components (line 9). The matched components (if any), and the oss are analyzed to decide if oss deems as reused by the input executable (line 10; see below for details). If so, we extract the "value information" associated with oss as the final analysis output back to users (line 12). In MATCH_FUNCTIONS(), we search the top-k ($k = 1$) most similar functions in oss for each input function (line 16). Note that this step is conducted by **BSA** tools, while Centris relies on the distance of TLSH. Then, a matched component pair is identified, if the similarity between the input and the detected function is greater than a configurable threshold θ (lines 18-19). We select the θ for each BSA work by performing BSCA using a hyper-parameter tuning dataset.

The BSA tool is invoked at line 16 to identify top-k functions similar with a function in the input executable; similar functions are ranked by their similarity scores from the highest.

Given a set of matched functions M (M may be empty), IS_REUSED() decides if oss is reused and the exact reused OSS version. At this step, Centris calculates the proportion of the matched functions M to the entire functions in oss. oss is deemed as reused if the proportion is higher than a threshold (i.e., 0.1 in Centris). Moreover, Sec. VI proposes three optimizations to replace Centris's implementation of IS_REUSED() with notably improved BSCA accuracy.

C. BSCA Dataset Preparation

In this section, we introduce our dataset used for BSCA benchmark. Overall, one key contribution of this research is to deliver the ever-first comprehensive dataset specifically designed for BSCA. While existing datasets are mostly used for BSA scenarios, our tentative exploration shows that those datasets are not suitable for BSCA. As revealed in Sec. II-B, modern SOTA BSA solutions not necessarily manifest high performance for BSCA, whereas some other syntactic feature-based methods, though lightweight, achieves plausible accuracy.

Dataset Preparation. We establish our ever-first benchmark dataset for BSCA, whose establishment involves a considerable amount of manual efforts. Three authors of these paper spent about a month for the dataset collection and manual ground truth marking: one author is a senior Ph.D. student and two authors are security engineers from industry. All three authors are highly experienced in reverse engineering, software security and BSA/SCA and have constantly published relevant papers in the community. This way, we ensure the accuracy of our study and the credibility of our benchmark dataset to a great extent. Our the dataset comprise 35 software across two platforms, whose compiled executable include totally over one million assembly functions. All these 35 software are large-size, widely-used applications, such as Nano [12]. The OSS database to be searched has 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.

We report the statistics and ground truth in Table II. ELF and executables are compiled with `gcc` and `clang`, while PE executables are compiled with `MSVC`. For instance, out of 35 software, 24 are on the Linux platform and are compiled into ELF binary format; they have in total 368,558 assembly functions. with manual investigation, we find that 336 OSS versions (and 157 OSS) are reused by these 24 ELF binaries. The average size of each stripped executable is 8,588KB, denoting executable with very large size (in comparison, the stripped `gcc-7.5.0` executable is 1,023KB).

We determine the OSS reuse in the application software (i.e., the ground truth) by first checking the reused submodules of their repositories. Then, each author manually compares the involved source code in the application code with our OSS database to confirm the reused OSS component and all involved OSS versions. Finally, authors discuss and cross-compare their findings to reach consensus.

Hyper-Parameter Selection Dataset. It is worth noting that our selected BSA solutions all require to employ a validation dataset for hyper-parameter tuning. To do so, we randomly select ten OSS projects from our benchmark dataset, and compile all their versions (in total 524 OSS instances) into executable. We fine-tune hyper-parameters of each employed BSA over a simple task of OSS classification using these 524 OSS instances. We select the best hyper-parameters of each BSA tool when the classification accuracy is optimal.

V. STUDY

The following sections evaluate how de facto BSA solutions perform in the BSCA scenarios. In particular, we try to answer the following research questions (RQs):

- **RQ1**: *how do de facto BSA techniques support BSCA?*
- **RQ2**: *what improvement can be made over standard BSA solutions for BSCA?*
- **RQ3**: *how efficient is using BSA solutions for BSCA?*

TABLE III
OSS IDENTIFICATION ACCURACY OF DIFFERENT BSA TOOLS.

Tool	S_1			S_2			S_3		
	P	*R*	*F1*	*P*	*R*	*F1*	*P*	*R*	*F1*
T1	.230	.583	.330	.223	.519	.312	.062	.214	.096
T2	.107	.611	.183	.137	.425	.207	.045	.071	.055
T3	.081	.916	.148	.098	.856	.176	.032	.964	.062
T4	.194	.708	.304	.237	.481	.318	.073	.480	.126
T5	.422	.322	.365	.237	.552	.331	.126	.518	.203
T6	.330	.463	.385	.272	.454	.340	.145	.600	.233
$\texttt{C}_A$	.731	.320	.445	.755	.328	.457	.750	.325	.454
BAT	.560	.351	.432	.647	.341	.447	.746	.371	.495

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, $\texttt{PalmTree}_G$, $\texttt{PalmTree}_B$, $\texttt{Commercial}_B$, and CodeCMR, respectively.
$\texttt{C}_A$ is the abbreviation for $\texttt{Commercial}_A$.

We clarify that OSS identification is indeed much easier than identifying specific OSS versions reused in the input executable. Thus, in the rest of the section we first report the accuracy of the OSS identification, and then report version identification.

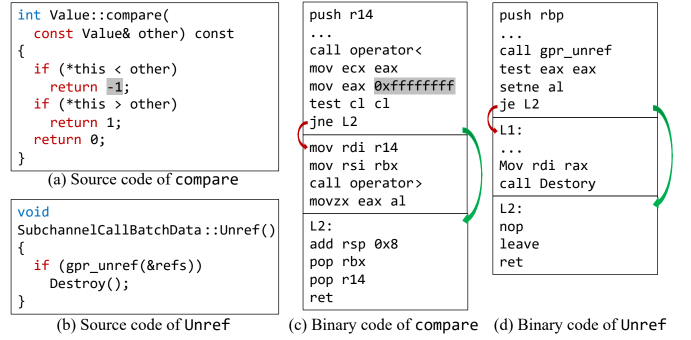
A. OSS Identification

Recall the settings of using B2B and B2S are distinct. Particularly, when using B2B, we need to compile OSS in our benchmark dataset into assembly functions for use. To do so, we use the default setting of each OSS project for compilation. Typically on Linux, OSS projects are compiled by gcc/g++. As for Windows, OSS are compiled with MSVC. -O2 optimization levels are usually required by default. When using B2S, we save the effort of compiling OSS projects.

Table III presents the *precision*, *recall*, and *F1 score* values when using different BSA solutions for BSCA. Recall as introduced in Sec. IV-A, we also take into account two SOTA commercial or academic BSCA solutions for comparison. The results are reported in the last two rows of Table III.

Comparison with Existing Solutions. Overall, we find that when directly employing BSA (either B2B or B2S) in the pipeline of SCA, the results are not promising. It is seen that the F1 scores when using different BSA methods are lower than 0.4. In comparison, existing BSCA solutions (i.e., BAT and $\texttt{Commercial}_A$) manifest high accuracy, with the achieved F1 scores over 0.4 across all settings. By comparing the precision and recall of our BSA solutions with existing BSCA solutions, we realize the low F1 scores are mainly caused by the *low precision*. In total 18 (3×6) settings of selected BSA solutions, 15 settings gain recalls higher than existing works, but all settings have significantly lower precisions.

We analyze the similar function pairs provided by $\texttt{Commercial}_B$ and original Centris. Recall we fine-tune its hyper-parameters over an OSS/version classification task. We note that under the optimal hyper-parameters, the conducting BSCA yields *a noticeable number of false positives*. That is, an assembly function in the input executable are "matched" with multiple functions in the OSS database, where each of the matched pair has a high similarity score. The average number of matched functions for each assembly function in the input is 3.43. However, with the redundancy elimination and code segmentation performed in Sec. IV-B, each assembly function in the OSS database is deemed as unique. This indicates the high volume of false positives.



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

Fig. 3. False positive case study.

Fig. 3 provides two functions classified as similar by $\texttt{Commercial}_B$ with a similarity score over 0.95. They are significantly different from the perspective of the source code (i.e., Fig. 3(a) and Fig. 3(b)); however, after compilation, they have the same CFG structure (i.e., Fig. 3(c) and Fig. 3(d)) and $\texttt{Commercial}_B$ failed to notice their distinct semantics. To correct such false positive samples, identifying the unique features may be a solution. By observing the missing constant -1 (i.e., 0xffffffff of Fig. 3(c)), we could treat function Unref as similar but not reused. On the other hand, we can raise the similarity score of binary functions having notable features (e.g., strings and unique constants) but being treated as dissimilar. Eventually, we do not use constants since they could be optimized out (e.g., 1 and 0 disappear in Fig. 3(c)).

In contrast, Centris (recall we extend BSCA from it by replacing its distance of TLSH with BSA solutions) is much more precise. The average number of similar functions for each source function is 1.45. Increasing the θ (line 18 of Alg. 1) does not improve the precision of $\texttt{Commercial}_B$ to the precision of TLSH. When we set $\theta = 0.99$, close to the upper bound of *cosine similarity*, the average number of similar functions for each binary function is still 2.7. Note that detecting multiple similar functions is reasonable. For example, a patched function is functionally similar to its buggy version. Therefore, the low precision problem is deemed to be a result of the design of BSA solutions. In other words, existing BSA solutions are designed to find similar semantics rather than reuse relations.

To improve the F1 score, an approach is to *identify the false negatives as positives*. Table III shows that BAT, relying on strings, presents a high precision performance. We follow its idea and use strings to identify the reused functions. Sec. VI-A describes this enhancement. The other approach is *reducing the false positives*. We consider using global knowledge of a binary and elaborate our design in Sec. VI-C. Additionally, since the custom code of an input binary may inevitably match similar functions in the database, we come up with the distinguishability enhancement (see Sec. VI-B). For example, we find that destructors usually be treated as similar to each other due to their similar functionalities.

Comparison across Different Settings. By comparing the F1 scores of our BSA solutions across three settings, we notice the cross-platform situation is still a critical challenge. We report

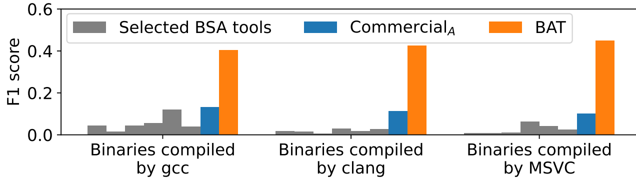


Fig. 4. Version identification F1 scores of different tools.

all of them gain AUC scores over 0.95 on frequently-used `binutils` dataset [52, 57] across all settings. However, the F1 scores on binaries compiled with `MSVC` (S_3) of all selected BSA solutions are the least and reduced over 0.1 compared with the other two settings. We interpret the F1 score of BSCA is a strong metric to reveal the incapability of a BSA solution. Although developers can overcome the cross-platform challenge by compiling the database for each platform, the cost of manual efforts could be unacceptable, and the database size grows with the number of supported platforms.

B. Version Identification

We implement the version identification process of Centris in our BSCA framework and present the result in Fig. 4. Although we have pointed out that BAT, which uses strings to identify the exact version, is not a thorough solution since strings may not change between different versions, our experiment shows BAT achieves the best *F1 score* in the version identification task. The F1 score of $\texttt{Commercial}_A$ becomes incomparable to BAT since it does not provide the exact version of identified OSS projects in many cases. In terms of our selected BSA tools, Fig. 4 shows their F1 scores are always significantly lower than existing BSCA works.

After detailed analysis, we conclude identifying the exact reused version is difficult for existing BSA tools since they are not sufficiently sensitive to minor changes, which may not significantly change the functionality of a function. 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`; however, it reports a wrong version since the vulnerable version has a higher similarity score (0.959 to 0.944). A future direction of BSA works is to perform minor change-sensitive matching. This way, BSA works could solve tasks like patch presence detection and version identification.

> ***Answer to RQ1:*** *Directly employing BSA solutions in the pipeline of SCA cannot gain a comparable accuracy on the OSS identification to existing BSCA tools. Our in-depth analysis reveals the low precision problem of BSA solutions in BSCA scenarios. Moreover, existing BSA solutions can hardly work on version identification since they are insufficiently sensitive to minor changes.*

VI. Enhancement

With lessons obtained in Sec. V, we explore optimizations over the formed BSCA pipeline from three aspects. In particular, we first present three enhancements in Sec. VI-A, Sec. VI-B, and Sec. VI-C, and then discuss their effectiveness.

A. Enhancement – Signature

Originated from the primary setup of modern BSA techniques, our study pipeline in Sec. V performs function-level matching. Nevertheless, our observation shows that string-level signatures likely facilitate a more accurate matching. It is disclosed that string-level signatures are employed by existing SCA tools [36, 43, 76]. Those tools do not extract and compare program semantics, but merely rely on the extracted string-level signatures. It is expected that string-level signatures are generally unchanged when the source code is compiled using different compilers, optimizations, and even toward different platforms. Also, string-level signatures are generally easy to extract from either source code or binary code.

In contrast, existing BSA tools aim at detecting the similarity between pieces of arbitrary assembly functions; this is a more ambitious and challenging goal, given that string-level signatures may not be available in arbitrary assembly functions. Our study and reflection in Sec. V shows that chasing an extreme accuracy in matching every OSS functions may not be needed. Rather, strings may offer a short-cut to match OSS projects, which often use strings to encode OSS names, vendors, and even version information.

At this step, we use `IDA pro` to search and extract strings in input executable and OSS executable. When OSS source code is used, we can directly extract them from the source code (we use BAT's string extraction utility). We extend the similarity scores computed over an assembly function f and a function f' in OSS database by adding the similarity score yieled by the BSA tool and the proportion of exactly matched strings from f and f'. We use a weight $\alpha = 10$ to represent the importance of string signature matching.

We leave presenting and discussing the enhancement results in Fig. 6. In short, with the extracted string signatures, the F1 scores for all B2B tools are increased.

B. Enhancement – Distinguishability

Another insight from the study in Sec. V is that certain complex functions may serve as a strong indicator to confirm an OSS reuse. This step aims to assign a distinguishability score for each function, such that if a function with higher distinguishability score is matched to an OSS's function by BSA, then the chance of this OSS being reused is also higher. In particular, we consider the following metric,

$$Dis(f) = \frac{complexity(f)}{\beta^{|match(f)|-1}} \tag{1}$$

where given an assembly function f in the input executable, $complexity(f)$ is the number of instructions in f, and $match(f)$ is the set of OSS projects with functions being matched to f by the employed BSA. We require $match(f)$ has at least one element, because f needs to be matched with at least one function from an OSS. $\beta > 1$ is a constant whose value is seen to have small impact on the results; we use $\beta = 5$. Fig. 6 reports the effectiveness of this enhancement. It rises precision, recall and F1 scores of all tools. The F1 score of the best-performing BSA tool is further increased by over 0.2.

C. Enhancement – Layout

Recall when compiling C/C++ programs into executable, 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 together in the executable file. Thus, the layout of assembly functions establishes correlations, which can be leveraged to improve the BSCA matching accuracy.

Overall, besides generating the OSS database of assembly functions, we also store all object files generated during compilation. Then, during OSS detection, we search if a reasonable portion of functions from the same object can all be detected from the input executable.

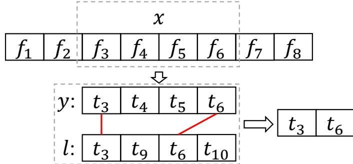


Fig. 5. Layout matching processing.

Fig. 5 illustrates the idea of layout matching. Let the Y be a set of functions from the input executable that match functions in the OSS database, we first sort these functions using their memory addresses. We then iterate each $f_i \in Y$, and for f_i' matching with f_i, we extract the object file containing f_i'. We then check if there exists at least one more function $f_j \in Y$ that matches a function in the object file. We also require the memory distance between f_i and f_j must be smaller than the size of the object file. This optimization tactic successfully reduces many false positive cases in matching OSS.

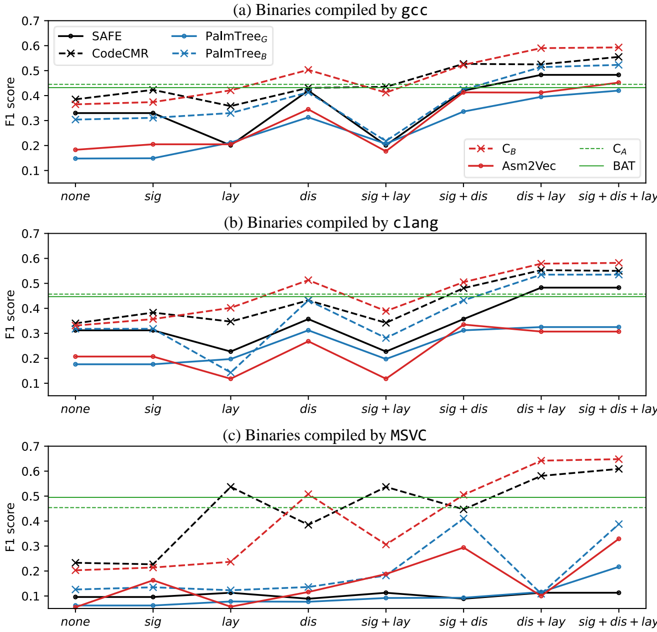


Fig. 6. OSS identification F1 scores of tools with enhancements.[4]

[4] sig, dis, and lay denote string-level signature, distinguishability, and layout enhancement; $+$ represents their combinations. C$_A$ and C$_B$ are abbreviations of `Commercial`$_A$ and `Commercial`$_B$, respectively.

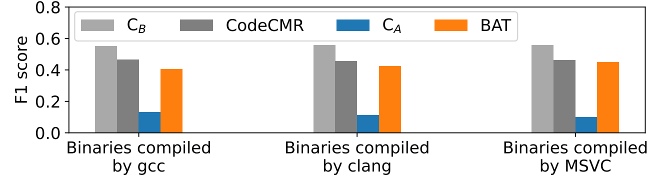


Fig. 7. Version identification F1 scores of BSA tools with enhancements.

D. Optimization Results Discussion

OSS Identification. Fig. 6 depicts how the F1 scores are changed according to various enhancements. In a total of 18 (3×6) settings, we get 15 the highest and 3 the second highest F1 scores with all enhancements enabled, and 11 F1 scores surpass that of BAT and `Commercial`$_A$. Among three enhancements, dis usually brings the largest improvement.

We analyze the three settings where all enhancements do not achieve the best F1 scores. lay decreases F1 scores sometimes by comparing $sig + dis$ and $sig + dis + lay$. We interpret relatively poor performances of `Asm2vec` (solid red curve of Fig. 6(b)) and `PalmTree`$_B$ (dotted blue curve of Fig. 6(c)) on the unseen compilers result in the reductions. The model of `Asm2vec` is trained with all binaries of our OSS, which are compiled with `gcc`. `PalmTree`$_B$ is trained with binaries compiled with `gcc` and `clang`, but the binaries of Fig. 6(c) are compiled with `MSVC`. Therefore, few functions are identified as similar. Recall as specified in the layout optimization (Sec. VI-C), given a matched function f, if the following function matched to the OSS database is far from f, f will be trimmed off. Hence, the functions being classified as similar become much smaller (though many false positives are correctly reduced) after lay, resulting in the negative effect. sig causes the remaining reduction. While analyzing binaries compiled by `clang`, the final F1 score (0.550) of `CodeCMR` (dotted black curve of Fig. 6(b)) is slightly reduced compared to $dis + lay$ (0.553). Indeed, it may be unnecessary to take into account sig when using B2S, since strings have been learned by B2S's embedding models. In terms of B2B solutions, sig can always improve the F1 scores, especially when the B2B tool has a poor performance initially (e.g., the solid red curve of Fig. 6(c)).

On the other hand, the performances with enhancements of six tools are consistent with their initial results (i.e., *none*). `Commercial`$_B$ and `CodeCMR` are still the best two BSA with enhancements applied. `SAFE` and `PalmTree`$_B$ surpass existing works when binaries are compiled by `gcc` and `clang`. With this observation, we infer our enhancement methods should still work and bring a promising result with a better BSA solution.

Version Identification. A correct OSS identification is the pre-condition of a true version identification. Hence, improving OSS identification may bring a rising in version identification. Due to the incapability of BSA solutions to identify functions of different versions, we rely on string-level signatures to rank the most likely version for each identified OSS project. Fig. 7 presents the version identification F1 scores of `Commercial`$_B$ and `CodeCMR` with all enhancements enabled. They outperform BAT since they are more accurate in OSS identification task.

Besides improving the capability of BSA solutions for version identification, using other techniques could be a

substitution. Fig. 7 has demonstrated that strings are effective. We argue that techniques sensitive to minor changes (e.g., patch presence detectors [73, 79]) can take part in the version identification process and the existence of a code snippet can determine the exact reused version.

> ***Answer to RQ2:*** *We show that when taking account three optimization opportunities, including string-level signature, functions with potentially high distinguishability, and object file layouts, BSA-based BSCA solutions can be largely improved with much higher accuracy. In fact, after employing the enhancements, all settings improve notably and 11 out of 18 settings surpass SOTA commercial BSCA solutions.*

VII. Time Cost of Component Detection

Compared with signature-based SCA works like BAT, whose component detection phase is efficient due to repidly hash matching, the time cost of function-level granularity matching is not trivial. Centris, having the same workflow in Fig. 2 but deciding similar function pairs with the distance of TLSH, takes nearly 8 hours to finish the component detection phase of our benchmark dataset. Our BSCA component detection costs more time due to the complexity of binary analysis. As depicted in Fig. 2, the component detection phase can be further split into (1) Disassembling, (2) BSA, and (3) OSS & version identification. We do all experiments with a server with AMD 3970X 32-core, 256GB memory and a RTX3090.

Disassembling. In this study, we implement most BSA tools with `IDA pro` disassembler, except `SAFE`, which uses `Radare2` [15] originally. The size of our binaries is 9,089KB on average. The `IDA pro` (ver 7.5) successfully analyzed 35 binaries within 4.5 hours.

BSA. There are three stages of BSA. Given a binary code embedding tool, it has to **(a)** encode the input binary's functions, and **(b)** encode binary functions in the database. Then, we employ `Milvus`, the vector database, to **(c)** search for similar functions. **(a)** and **(b)** depend on the efficiency of the embedding tool, while the size of vectors influences **(c)**. **(b)** is a one-time effort. $\texttt{Commercial}_B$, the tool with the best accuracy but least efficiency, consumes 10 hours in **(a)** and over 30 hours in **(b)**. Stage **(c)** with 256-dimension vectors takes 22 minutes.

OSS & Version Identification. BSCA with $\texttt{Commercial}_B$ costs a half hour to analyze the searching results without any enhancement. The overhead caused by signature and distinguishability enhancement is negligible. After using layout enhancement, the overhead rises by 83.7% (around 25 minutes). On average, this phase takes about two minutes for each sample.

> ***Answer to RQ3:*** *With well-developed toolchains (e.g., disassemblers) and efficient GPUs, the time cost of component identification using SOTA BSA tools ranges from 8 to 16 hours. Given Centris takes about eight hours to finish the corresponding SCA task, the time cost of our BSCA framework is comparable and reasonable.*

VIII. Threat to Validity

Missing reused OSS in our benchmark. 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 prove our BSCA framework's performance and reveal the incapability of existing BSA solutions.

Function-level Granularity. We extend the workflow of advanced SCA solutions, Centris [71], for our BSCA analysis. Although Centris's authors reported that function-level granularity is the best compared with file-level and line-level granularities, it may differ in BSCA scenarios. Nevertheless, we still work on the function-level granularity since it is the most frequently studied. We admit others like block-level and binary-level granularities may also work. However, it is too time-consuming to employ de facto block-level embedding techniques (e.g., `DeepBinDiff` [37]), and binary comparison tools like `bindiff` [23] still split a whole binary into functions and basic blocks for analysis.

IX. Related Work

In this section, we review existing binary software composition analysis and binary similarity analysis techniques.

Binary Software Composition Analysis. Several BSCA works focus on extracting features unchanged after compilation. BAT and OSSPolice employ string-level signatures like string arrays and names of exported functions. B2SFinder extracts `if/else` and `switch/case` structures additionally. Although their selected features are useful, they discard semantic knowledge and cannot work when the OSS project has few unique strings (e.g., RapidJSON [16]). LibDB [68], a recent BSCA work, performs similarly to our B2B situation with `Gemini` and proposes employing function call graphs as global knowledge to improve precision. However, it does not take advantage of SOTA SCA methods on data processing and does not comprehensively evaluate its BSCA performance on binaries compiled with various configurations.

Binary Similarity Analysis. Here we discuss relevant but not selected BSA solutions. They usually extract semantic features from binary since the compiler can change the structure information (e.g., CFG) significantly. `CoP` [55], `BinGo` [30], `BinSim` [60], `IMF-SIM` [70], [48], and `FIBER` extract semantic features via symbolic or dynamic execution, which may bring unacceptable overhead. `Gitz` lifts binary code to LLVM-IR, then applies the optimizer to transform the IR with the same options. It heavily relies on the quality of the lifter. Recent works often rely on machine learning. α`diff` [54] treats a binary function as an image, then uses a CNN model to produce the embedding of a binary function. `InnerEye` treats instructions as words and basic blocks as sentences, then produces their embeddings. The CFG of a function is decomposed into paths for function matching.

X. Conclusion

In the view of the prosperous 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 directly using BSA solutions in the pipeline of SCA got poor performance, and they were incapable in version identification. We then proposed optimization tactics from three aspects, which largely improved the accuracy of OSS identification with moderate costs.

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

[6] Bsa2bsca. https://sites.google.com/view/bsa2bsca/home/.

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

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

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

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

[11] InnerEye. https://github.com/nmt4binaries/nmt4binaries.github.io/issues/8.

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

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

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

[15] Radare2. https://github.com/radareorg/radare2.

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

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

[18] Scantist. https://scantist.io/.

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

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

[21] Software composition analysis (SCA): what is it and does your company need it? https://snyk.io/blog/what-is-software-composition-analysis-sca-and-does-my-company-need-it/.

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

[23] Bindiff. https://www.zynamics.com/bindiff.html, 2014.

[24] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2Vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[25] 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.

[26] 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.

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

[28] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. Signature verification using a" siamese" time delay neural network. *Advances in neural information processing systems*, 6, 1993.

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

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

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

[40] 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.

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

[42] 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.

[43] 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.

[44] SA Hex-Rays. IDA Pro: a cross-platform multi-processor disassembler and debugger, 2014.

[45] 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.

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

[47] 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.

[48] 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.

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

[50] 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.

[51] 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.

[52] Xuezixiang Li, Qu Yu, and Heng Yin. PalmTree: Learning an assembly language model for instruction embedding. 2021.

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

[54] 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.

[55] 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 Trans. Softw. Eng.*, 43(12):1157–1177, December 2017.

[56] 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.

[57] 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.

[58] 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.

[59] 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. IEEE, 2019.
[60] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. USENIX, 2017.
[61] 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.
[62] 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.
[63] 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.
[64] 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.
[65] 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.
[66] 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.
[67] Synopsys. The heartbleed bug. https://heartbleed.com/, 2020.
[68] 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.
[69] 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.
[70] Shuai Wang and Dinghao Wu. In-memory fuzzing for binary code similarity analysis. In *ASE*, 2017.
[80] 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.
[71] 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.
[72] 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.
[73] 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.
[74] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. Order matters: Semantic-aware neural networks for binary code similarity detection. 2020.
[75] 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.
[76] 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.
[77] 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.
[78] Fangfang Zhang, Yoon-Chan Jhi, Dinghao Wu, Peng Liu, and Sencun Zhu. A first step towards algorithm plagiarism detection. In *ISSTA*, 2012.
[79] Hang Zhang and Zhiyun Qian. Precise and accurate patch presence test for binaries. In *USENIX Security*, 2018.
[81] 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.
[82] 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.