This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3341224

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Empirical Study of Software Composition Analysis Tools for C/C++ Binary Programs

YUQIAO NING[1], YANAN ZHANG[2], CHAO MA[1], ZHEN GUO[1], LONGHAI YU[1],
[1]CATARC Intelligent and connected technology Co., Ltd.
[2]Automotive Data of China (Tianjin) Co., Ltd.

ABSTRACT Software composition analysis (SCA) is essential for understanding and optimizing complex C programs, ensuring system reliability and efficiency. Analyzing programs at the binary level provides insights into behavior, performance, and security. However, comprehensive evaluations of both academic and commercialized SCA tools are lacking. To this end, this paper presents a comprehensive evaluation of software composition analysis techniques for accurately identifying components in C/C++ binary programs. The study examines different analysis techniques in terms of accuracy, performance, domain-specific capabilities, and additional abilities such as detecting security vulnerabilities and code reuse potential. The results show that SCA tools reach over 70% accuracy in detecting general libraries and the accuracy drops to less than 45% for libraries in domain-specific software. Commercialized tools exhibit better efficiency and practicalness than academic tools. The evaluation provides insights into the strengths and limitations of various approaches, offering suggestions for SCA development and the selection of the most suitable tools.

INDEX TERMS Binary Program Analysis, Software Composition Analysis

I. INTRODUCTION

Software composition analysis (SCA) assumes a pivotal role in comprehending and enhancing intricate C programs. As software systems grow in complexity, characterized by an intricate interplay of interdependent modules, a critical need arises to examine individual components and their interactions. This scrutiny is indispensable for ensuring the dependability, manageability, and optimal operation of the system. The primary objective of software composition analysis is to deconstruct the program into its elemental constituents, probing their respective functionalities, interdependencies, and interfaces. By studying these constituents, developers can glean insights into the program's organizational framework, identify potential performance bottlenecks or security vulnerabilities, and facilitate efficient code reuse. Through a meticulous approach to software composition analysis, developers can streamline development endeavors, enhance code quality, and elevate the overall efficiency of the system.

Furthermore, binary-level software composition analysis emerges as a critical technique for comprehending and refining C programs at the deepest level of abstraction. By examining programs at the binary level, researchers can dissect the compiled code, thereby extracting insights into the program's behavior, performance attributes, and security characteristics—distinct from the original source code [1].

Various methods have been introduced to improve the precision of component detection across diverse programming languages. In the realm of Java, AtvHunter [2] utilizes control flow graphs from specific Java libraries as distinctive signatures to identify similar components. Backes et al. [3] derive library profiles by leveraging hierarchical code information, enabling them to detect APK packages even in the presence of obfuscation. Libd [4] uses dependencies within functions as signatures for detecting library reuse. LibPecker [5] refines detection accuracy by introducing an adaptive class similarity threshold, calculating similarity between signatures and target libraries. Furthermore, several studies [6]–[9], introduce clone detection techniques, which can also be applied to tasks involving component detection.

Numerous methodologies have been proposed to discern components within C/C++ binary programs. For instance, OSSPolice [10] employs invariant literals and a hierarchical indexing scheme to recognize components, while B2SFinder [11] establishes binary-to-source code mappings using invariants to identify components within commercially available software. Modx [12] adopts a modular approach,

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3341224

dissecting components into modules and gauging module-level resemblances to discern component usage, with a specific focus on partially imported libraries. Moreover, commercialized tools proffer component analysis capabilities. Snyk [13] and SonarQube [14] furnish means to ascertain components, primarily within the source code realm, catering to C/C++ programming as well as languages like Java. Conversely, BlackDuck [15] and Cybellum [16] are tailored to optimize component detection within binary executables. These tools harness diverse feature sets encompassing strings, file names, hashes, and instruction-level details, culminating in the precise and efficient identification of open source components.

Despite the availability of various SCA tools for analyzing C/C++ binary programs, there is a notable absence of comprehensive empirical evaluations for these tools. A pertinent study is presented in [17], which conducts an examination of 61 binary code similarity methods. This survey delves into the features employed for code similarity detection, the methodologies' evaluation criteria, and the downstream applications of similarity detection tasks. It is noteworthy that SCA emerges as one of the applications within the realm of similarity detection endeavors. Additionally, other relevant studies are primarily focused on source code level component analysis. Sheneamer and Kalita [18]extensively explore the domain of code clones, covering common types of clones, phases of clone detection, state-of-the-art techniques and tools in code clone detection, as well as the challenges faced by these techniques. Wang, Jingdong, et al. [19] analyze solutions addressing the similarity search problem. Roy and Cordy [20] conduct an in-depth study of code clone techniques, providing comprehensive clone taxonomies. Rainer Koschke [21] focuses on different clone types and assesses the consequences resulting from code clones. Most of these works primarily focus on surveying code clone techniques at the source code level.

The studies aforementioned exhibit several inherent limitations. Primarily, a notable constraint surfaces in the absence of a standardized benchmark dataset for testing the efficacy of SCA tools. It is noteworthy that the dataset dimensions often remain constrained, comprising fewer than 100 samples [12]. However, this sample scope pales in comparison to the practicalities of real-world software systems that typically encompass over 10,000 files—underscoring a discrepancy of one hundredfold in scale [22]. This incongruity intimates that an evaluation centered on a dataset of 100 samples may fall short of adequately assessing the real-world performance of SCA tools. Secondarily, these studies [17]–[21] encounter a drawback by neglecting to scrutinize the tools within specific application domains. In practice, SCA tools are predominantly employed for the identification of components in distinct software types, deviating from the realm of commonly recognized open-source libraries. A salient illustration is the disparity in component detection performance when transitioning from generic open-source libraries to domain-specific software configurations. Existing tools, often attuned to the broader open-source landscape, tend to exhibit compromised effectiveness when confronted with the intricacies and idiosyncrasies associated with specialized domains. This divergence underscores the need for a more nuanced evaluation framework that encapsulates the challenges intrinsic to targeted application sectors. Lastly, an imbalance exists in the focus of extant research, with greater emphasis placed on academic tools, inadvertently sidelining the exploration of commercialized SCA tools.

To this end, our objective is to conduct a thorough empirical study to assess the effectiveness of academic and commercially available SCA tools in real-world C/C++ binary library detection tasks. Our methodology is outlined as follows: first, we curated and assembled a benchmark dataset comprising 16,713 C/C++ binary libraries from various data sources. These cases encompass standard binary software managers, manually obfuscated binaries, and binaries employed in vehicles manufactured by an automobile company. Subsequently, we selected three industrial SCA tools BlackDuck, Scantist, and Cybellum, and one academic tool for evaluation.

Our main study focuses on the measurement of accuracy and time consumption for each tool in SCA tasks. The results reveal that SCA tools exhibit an accuracy rate of over 70% in detecting general libraries, while their accuracy in detecting domain-specific software libraries drops to less than 45%. The academic SCA too also demonstrates competitive accuracy in library name and version detection. Furthermore, the time consumption varies significantly due to the different usage of light and heavy weighted features. Additionally, we conducted manual testing and in-depth exploration of the advanced security-oriented auxiliary features of these tools. The findings reveal that commercialized SCA tools universally incorporate supplementary functionalities, driven by both competitive market pressures.

In summary, our contribution is as follows:

- We collect C/C++ binaries from different sources to build a benchmark with over 16,000 ground truth samples.
- We perform empirical studies on both academic and industrial SCA tools to evaluate their accuracy and performance on SCA tasks.
- We are the first to evaluate the SCA capability in a specific application domain. The result shows that the precision of existing tools drops to less than 45% when detecting non-general purpose open-source libraries.

II. OVERVIEW

In this section, we furnish an overarching synopsis of our study, represented in Figure 1. This figure depiction concurrently introduces the pivotal research questions (RQs) that serve as the compass directing our investigative journey. To commence, our primary focus, as delineated in RQ1, resides in the meticulous evaluation of the precision with which SCA tools discern components within diverse binary files. Subsequently, we embark on a quantitative exploration in

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3341224

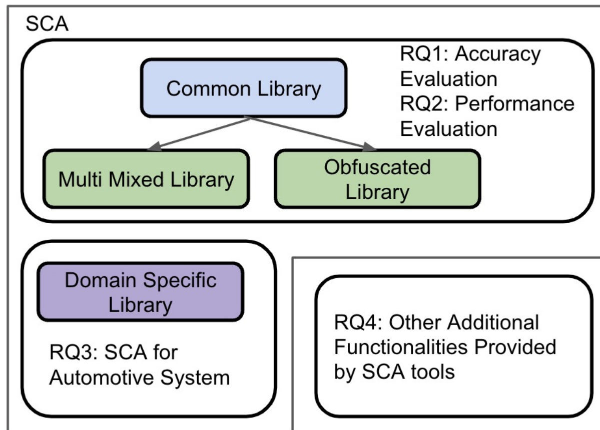

FIGURE 1: Overview of Empirical Study on Software Composition Analysis

RQ2, wherein we gauge the temporal investment entailed in scanning individual binary files. This temporal scrutiny serves as a conduit to deduce the underlying methodologies intrinsic to the process of component detection. The trajectory of inquiry escalates with RQ3, wherein we meticulously scrutinize the accuracy of component identification within a specific application domain. This granular examination furnishes us with indispensable insights into the pragmatic feasibility of these tools within authentic SCA scenarios. Guided by the findings encapsulated within RQ1 through RQ3, we embark on a comprehensive exposition elucidating the detection algorithms employed by each distinct SCA tool. This analytical venture serves to unravel the intricacies underpinning their operational frameworks and contextual performance. Culminating our analysis is the inquiry encapsulated in RQ4, wherein we undertake a thorough appraisal of supplementary functionalities embedded within the SCA tools. This research question stands as a pivotal juncture, contributing to the holistic comprehension of the real-world security paradigms orchestrated by the SCA solution.

The subsequent sections of this paper are organized as follows: In Section III, we provide a comprehensive elucidation of the intricate procedures employed to either generate or gather libraries in binary format. Section IV, delineates the rationale underlying the judicious selection of the specific SCA tools employed within this study. Moving forward, Section V unveils the empirical study's outcomes, accompanied by our comprehensive interpretation of the discerned findings. Engaging in a critical dialogue, Section VI delves into the manifold insights extracted from the study, elucidating the assimilated lessons and proffering viable directions for future research endeavors. Section VII and Section VIII are devoted to the thorough evaluation of pertinent prior works and culminating conclusions, respectively.

III. BENCHMARK DATA SET

This section elucidates the methodology employed to construct the benchmark ground truth dataset for our empirical study. The dataset comprises three distinct subsets of binary programs, each accompanied by ground truth references. Figure 2 presents the overview of the three approaches used to construct the subsets in the benchmark. First, we build the subset that encompasses general-purpose open-source libraries via directly crawling the packages from online package managers. Second, we download the source code from the library repositories and apply obfuscation strategies to modify the code. Then, we compile the source code into binary files to build the second subset. Third, we extract the binary files from the automotive firmware to obtain the domain-specific libraries. Since the files collected from this approach lack ground truth, we employ security experts to manually verify the library name and version information.

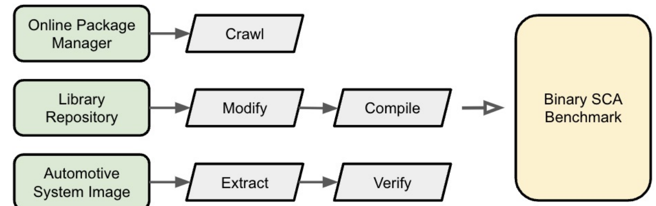

FIGURE 2: Overview of Benchmark Construction Approaches

A. GENERAL PURPOSE BINARY COLLECTION

Constructing an encompassing binary Software Composition Analysis test suite presents a notable challenge, centered primarily around identifying pertinent and authentic components within the intended package. To address this intricate task, we performed a comprehensive review of practices spanning both academic [23]–[25] and industrial spheres. We summarize three approaches, which can provide third-party library binaries with their names and versions from different aspects.

The first approach involves downloading the source code and performing compilation. This method offers the advantage of selecting specific versions, particularly those with known high-severity vulnerabilities [26], [27]. Specifically, we crawled all the vulnerability records from the online source [28]. Each record of the vulnerability is associated with one Common Platform Enumeration (CPE). The CPE contains information on the libraries that are affected by the vulnerabilities. We ranked the libraries by the number of vulnerabilities they include and selected the top libraries as our test data. However, this approach entails a higher cost due to the substantial manual effort required.

The second approach entails downloading packages that provide explicit component or software version information, such as component packages, software installation packages, and container images. This approach offers the advantage of having a well-defined target for analysis. However, it may limit the flexibility of the test suite since it relies on the availability of specific packages with known information.

The third approach relies on textual analysis techniques

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

to determine component names, versions, and additional information. We implemented the algorithm proposed by [29], which includes analyzing website data, examining metadata within component packages, and extracting relevant details. Textual analysis provides a more comprehensive view of the components, including dependencies on other components. Nonetheless, this approach may introduce errors or omissions in the analysis process due to potential inaccuracies in the textual information.

The previously outlined three steps culminate in the generation of three distinct sets of library lists, amalgamated to create an extensive dataset. The subsequent phase is the selection of representative candidates, forming the benchmark. To ensure the data completeness of the benchmark, we leverage two distinct strategies. The initial strategy revolves around random sampling, wherein data is randomly chosen from diverse sources, engendering a test dataset that embraces diversity and represents a broad spectrum of scenarios. In parallel, the second strategy, termed dimension coverage, is invoked. This strategy engulfs a multitude of dimensions, including platform architecture, file attributes, dimensions, nesting intricacies, indirect dependency tiers, size, vulnerability presence, update frequency, temporal span, minor version variations, and manual alterations (such as the removal of file information and symbol tables). The overarching goal is to meticulously embrace all conceivable subtypes nested within each dimension.

After selecting the target libraries, we ensure their format diversity by collecting 5 different versions for each library from a pool of available versions. The selection of 5 versions is done randomly from the library's available versions. If a library has fewer than 5 versions, we replace it randomly with another library to maintain consistency. For the formats, for each target library we collect one copy of the amd64, i386, AArch64, MIPS, and PowerPC binary file as the test cases. These formats are commonly provided by software vendors for different software platforms, making them representative of real-world systems. In total, for each library candidate, we select 25 (5 versions * 5 formats) variants to form the final benchmark.

B. MANUALLY CRAFTED BINARY COLLECTION

To comprehensively assess SCA tools across various scenarios, we undertake the manual compilation of library source code to procure the corresponding expected binary files. This compilation process diverges from the standard procedure mentioned in the previous section, involving the manipulation of compilation settings or source code to elicit mutations in the resultant binary outputs.

Concretely, we initiate this process by amalgamating multiple libraries into a unified binary file. This fusion is orchestrated through compilation settings that facilitate the static incorporation of source code from the libraries into the software. Consequently, the compiler orchestrates the amalgamation of all libraries into a singular binary entity. The rationale underlying this approach is grounded in real-world practices where libraries are often bundled and disseminated together as an integral system. Separating them into individual files without intensive program analysis is a formidable task. Therefore, the component detection process necessitates the ability to discriminate each library within a confluence of libraries and self-generated code. The mixed features from multiple libraries will increase the difficulties for SCA since the features from one library may become noises in detecting other libraries. We would like to test the ability of SCA tools, which are often designed to detect individual libraries, in mixed library detection.

Subsequently, embracing the concept of obfuscation [30], [31], we engage in binary file modifications to yield variants of the original library. This initiative stems from the observation that software vendors may modify the source code of the libraries to achieve customizations, leading to deviations from the standard binaries. Thus, our anticipation is that SCA tools should exhibit robustness in detecting libraries even when customizations are introduced. This dual-pronged approach amplifies the breadth of our evaluation, shedding light on the tools' adaptability across multifarious real-world scenarios.

To obtain different kinds of binary files, we apply the following mutation strategies. We assign two researchers to build the data set, which takes them 18 hours.

- **Code Flattening:** Code flattening involves transforming the code's nested structures into a single flat structure, making it harder to follow the logic. For each loop in the source code, if the number of the iteration is constant, we replace the loop with repeating the loop content to flatten the loop.
- **Code Substitution:** Replace standard library functions with custom implementations that have similar functionality but different internal logic.
- **Conditional Compiling:** Use conditional compilation to include or exclude certain code sections based on predefined macros, making the code more complex. Duration the compilation, we first detect the available macros and flip the flag of these macros.
- **Constant Obfuscation:** Convert constant values into expressions or use arithmetic operations to hide their original values. For each constant number we express it with the summation of one randomly generated number and another calculated number (the two numbers add up to the original constant).
- **Control Flow Obfuscation:** Change the order of statements, insert dead code or use goto statements to create confusing control flow paths. For each function, we insert a random number of lines of dead code. If the function contains if statement, we flip the condition of the statement and the corresponding code blocks.
- **String Stripping:** Stripe strings in the binary to prevent easy identification of sensitive strings and features.
- **Symbol Renaming:** All variable and function names have been replaced with arbitrary and meaningless names (e.g. rttfsmpkrnwi, xrsqrkgilfl, etc.).

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3341224

- **Code Splitting:** Split functions or methods into smaller parts and obfuscate their relationships to make the code harder to understand. For each function with over 100 lines of code, we construct two sub-functions with one calling the other. We copy the code of the original function into the sub-functions and build the call graphs by replacing the original function with sub-functions.

C. AUTOMOTIVE SYSTEM BINARY COLLECTION

To collect data from the automotive system, we collaborate with industrial partners to obtain firmware from real-world car hardware units and the update packages.

To ensure the authenticity and real-world applicability of our research, we established collaborations with industrial partners to collect data from automotive systems. Through these partnerships, we were able to obtain firmware directly from real-world car hardware units, encompassing a diverse range of vehicle models and configurations. This approach allowed us to access firmware that accurately represents the software landscape of modern automotive systems.

In addition to firmware from car hardware units, we also acquired update packages used in over-the-air (OTA) updates. These packages play a critical role in delivering software updates to vehicles, making them valuable sources of data for our research. By collaborating with our industrial partners, we gained access to a wide variety of OTA update packages, covering different vehicle models, software versions, and component configurations.

The collaborative nature of our data collection process ensured that we obtained a comprehensive and representative dataset that reflects the complexities and nuances of real-world automotive software. It enabled us to capture the evolution of software components, the presence of potential vulnerabilities, and the impact of updates on the system.

To ensure the privacy and confidentiality of the data obtained from our industrial partners, stringent measures were implemented. Non-disclosure agreements were signed, and data handling protocols were strictly followed throughout the collaboration. All data used in our research was anonymized and securely stored, adhering to best practices and industry standards for data protection.

D. DATA STATISTICS

Table 1 displays the total number of data collected through the aforementioned procedures. We gathered 1190 libraries from the general package manager. With each library, we generate multiple test cases with different architectures and versions. In total, we generated 16713 test cases from the libraries. Moreover, we collected 64 self-compiled and 432 libraries from automotive systems. Each library is regarded as one test case for the experiment. For the general-purpose binaries, the ground truth is readily available from the sources. The names of the files, obtained when downloading the source code or binary files, serve as the ground truth knowledge. In the case of automotive systems, we employed three binary security experts to manually examine the binary files and extract their names. In most instances, the file names provided indicative information about their components, which we carefully verified to ensure accurate ground truth. We have published our benchmark at [32].

IV. SCA TOOL SELECTION AND IMPLEMENTATION

In this section, we provide a detailed description of the tool we selected for our empirical study. We have selected three industrial tools that are developed specifically for SCA tasks. We focus on the problem of analyzing software components in the automotive domain and selecting leading and hot emerging software in the SCA domain, as well as leading software in the automotive safety domain. Blackduck is a Leader in the 2023 Gartner Magic QuadrantTM for Application Security Testing (AST) for the seventh year in a row. Based on their Ability to Execute and our Completeness of Vision, we are positioned highest and furthest right in the Leaders Quadrant among the 12 AST vendors evaluated by Gartner. Cybellum has a very high market share in the field of safety in the automotive industry. Cybellum has won the following awards: Best Product and Device Cybersecurity Platform, Best SBOM Management Product, Best Vulnerability Management Product, Best SBOM Management Product, and Best Vulnerability Management Product. Scantist is a start-up company that has been recognized by many media outlets as the most promising software component analysis and vulnerability management platform among emerging software.

- **Blackduck [15].** Blackduck by Synopsys is the leading provider of Software Composition Analysis solutions. It enables comprehensive visibility into open source and third-party dependencies compiled into binaries and firmware. Key capabilities include GUI and automated multifactor scanning to detect dependencies, coupled with fuzzy matching against Black Duck's extensive knowledge base of over 6.3 million open source projects, 2,750+ licenses, and 208,000+ vulnerabilities. By combining fast and reliable static and string analysis with an unmatched open source component database, Black Duck delivers precise software composition, license compliance, and vulnerability management for binaries and containers. It aims to help developers build secure software while managing open source at scale. In the experiment, we use Blackduck with version 2023.1.0.
- **Cybellum [16].** Cybellum is an automotive cybersecurity company that provides continuous threat monitoring and protection for connected vehicles. Its key product is a cybersecurity platform that leverages AI and machine learning to automatically identify risks and vulnerabilities in vehicle firmware and networks. Core capabilities span firmware security assessments, compliance checks, agent-based threat detection, and an extensive automotive threat intelligence database. With one of the largest datasets in the industry encompassing hundreds of millions of vehicles, Cybellum aims to secure vehicle

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3341224

TABLE 1: Infomation of the Constructed Dataset

Data Source	Description	Number of Library
	General Dataset	
Debian	By downloading and extracting C and C++ packages from Debian operating system package mirrors.	383 (7113 test cases)
RPM	Downloading C and C++ related component packages from RPM mirrors.	539 (7215 test cases)
NetBSD	Downloading binary components maintained by the well-known C package manager Conan.	268 (2385 test cases)
	Manually Compiled Dataset	
Mixed Library	Combine multiple libraries into one binary file.	56
Obfuscated Library	Modify the source code and compilation settings to mutate the generated binary file.	8 types of libraries
	Automotive Dataset	
Total Number	Dataset collected from reverse-engineered firmware of the real-world automotive system.	432

cybersecurity throughout the entire product lifecycle. Its automated, intelligent solutions help Original Equipment Manufacturers (OEMs) and suppliers assess cyber risks, enforce security standards, monitor networks for threats, and maintain fleet-wide security. By combining data-driven analytics with adaptive security controls, Cybellum is driving the future of automotive cyber safety. In the experiment, we use Cybellum with version 2.13.

- **Scantist [33].** Scantist is a cybersecurity company that provides network asset management and vulnerability management platforms to help organizations improve their security posture. Key functions include asset discovery, vulnerability scanning, vulnerability risk management, threat intelligence integration, and security compliance assessments. The platform utilizes both active and passive scanning, provides vulnerability scoring and remediation advice, enables granular asset tagging, and has customizable workflows. With an extensive vulnerability database of over 1.8 million entries, a library database covering over 7.6 million libraries and 985 million versions, Scantist leverages automation and intelligence to empower organizations with continuous network asset visibility, vulnerability control, and compliance assurance. The aim is to strengthen cyber defense through automated, comprehensive vulnerability and asset management. In the experiment, we use Scantist with version 4.1.

Among academic tools, we identified BinaryAI [34], [35] as a relevant candidate, offering a dedicated interface for component detection. Our assessment of BinaryAI's accuracy was based on the available interface. However, the constrained usage limits of BinaryAI restricted the scale of our testing, limiting our ability to perform comprehensive evaluations. All the industrial and academic SCA tools are set up with their default settings.

Additionally, our endeavor to access the source code of other academic tools, such as ModX [12], was undertaken with the intent of reimplementation for evaluation purposes. Yet, these attempts were met with challenges arising from scalability concerns. The majority of these tools grappled with limitations in scalability, attributable to their training with constrained datasets. Consequently, their scope for detecting libraries beyond this confined realm remained circumscribed. The expansion of training data proved to be an arduous endeavor, given the substantial time costs associated with the existing size of open-source datasets. In light of these constraints, the absence of viable scalability rendered them unsuitable for our evaluation, as they yielded accuracy levels close to zero due to data limitations.

V. EMPIRICAL STUDY

A. STUDY OVERVIEW

In this section, we address four key research questions (RQs) aimed at exploring different aspects of software composition analysis. There are listed as follows:

- RQ1: What is the component detection accuracy of SCA tools in detecting different types of binary libraries?
- RQ2: What is the time cost of each SCA tool in the component detection?
- RQ3: How do the SCA tools perform in detecting the domain-specific third-party libraries?
- RQ4: What are the additional functionalities provided by the SCA tools?

Firstly, in response to RQ1, we investigate the accuracy of various analysis techniques and compare their effectiveness in identifying software components accurately. This evaluation will provide insights into the strengths and limitations of different approaches and help practitioners choose the most accurate technique for their specific needs. Moving on to RQ2, we focus on the performance aspect, examining the efficiency and speed of different analysis techniques. By comparing their runtime and resource utilization, we aim to identify the techniques that offer optimal performance for large-scale software systems. Additionally, RQ3 delves into the domain-specific component detection capabilities of these techniques. We analyze their ability to identify and understand software components that are specific to particular domains, such as healthcare, finance, or automotive systems. This evaluation will shed light on the suitability of different techniques for specialized software domains. Lastly, in response to RQ4, we evaluate the additional abilities provided by the analysis techniques. These abilities may include the detection of security vulnerabilities, code reuse potential, or maintainability metrics. By examining these additional capabilities, we can determine the broader benefits and practical utility of each technique. Through these investigations, this paper aims to provide a comprehensive understanding of software composition analysis techniques, their accuracy, performance, domain-specific capabilities, and additional abilities.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

B. RQ1: ACCURACY COMPARISON

In this section, we aim to evaluate the accuracy of the SCA tools in detecting the components in each of the benchmark datasets. We run each of the SCA tools on each of the components individually. In total, we have three kinds of data (i.e. common libraries, combined libraries, and obfuscated libraries) to test the SCA accuracy under different scenarios. Moreover, we also evaluate the accuracy of academic SCA tools and compare the accuracy with commercialized tools. We use precision and recall to measure the accuracy of the SCA tools. The precision and recall are calculated by the following equations.

$$Precision = TP/(TP + FP) \tag{1}$$

$$Recall = TPs/(TP + FN) \tag{2}$$

True positive (TP) refers to the cases where the tool correctly detects the name or version of the given binary. False positive (FP) refers to the cases where the tool gives a wrong name for the binary. False negative (FN) refers to the cases where the tool cannot predict a name or version, while the given binary is actually a known library in our database.

TABLE 2: Precision and Recall for Component Detection on Benchmark Dataset

SCA Tool	Data Source	No. of Libraries	Precision	Recall
Blackduck	debian	383	0.922	0.981
	rpm	539	0.625	0.825
	netbsd	268	0.737	0.052
Scantist	debian	383	1	1
	rpm	539	0.902	0.909
	netbsd	268	0.96	0.301
Cybellum	debian	383	0.614	0.987
	rpm	539	0.514	0.938
	netbsd	268	0.435	0.500

TABLE 3: Precision and Recall for Component Version Detection on Benchmark Dataset

SCA Tool	Data Source	Number of Files	Precision	Recall
Blackduck	debian	7113	0.705	0.961
	rpm	7215	0.54	0.642
	netbsd	2385	0.416	0.037
Scantist	debian	7113	0.922	0.903
	rpm	7215	0.884	0.711
	netbsd	2385	0.13	0.06
Cybellum	debian	7113	0.612	0.913
	rpm	7215	0.6	0.938
	netbsd	2385	0.132	0.165

Common Library Detection. In this experiment, we aim to evaluate the general component detection ability of the SCA tools. To achieve this goal, we run the three SCA tools on individual libraries. Each library is well-labeled with its name and version. Each binary file contains and only contains the instructions from one library package. In total, we collected 1190 such libraries with 16713 individual files for evaluation. Table 2 presents the results of the library detection accuracy evaluation. We choose one file from each library to evaluate the name prediction accuracy since all the files in one library share the same library name. Table 3 presents the results of the library version detection accuracy based on each individual file. Since each file has its own version, we use all 16713 files to test the version prediction accuracy.

From Table 2, we can conclude that all three tools have reasonably good accuracy in detecting the libraries from Debian and RPM. Scantist and Cybellum have similar precision and recall in detecting the component. While Backduck has slightly worse precision and recall. Moreover, the precision and recall drop significantly when detecting libraries from NetBSD. The reason is that the libraries in NetBSD have less information such as standard library file name and package structure. The SCA tools rely heavily on this information to predict the names.

In the context of version detection, the observed accuracy registers an average decline from 70.35% (library detection accuracy) to 66.96%. This reduction can be attributed to a twofold rationale. Primarily, the absence of version-specific strings serves as a pivotal determinant. The crux of version determination often hinges on the identification of strings consisting of version information. However, the absence of these strings within many libraries poses a notable challenge. Furthermore, certain libraries incorporate multiple version strings, contributing to the ambiguity that confounds SCA tools' capacity to precisely predict versions. In the absence of string-based cues, SCA tools are compelled to extrapolate features from alternative strings or binary instructions. Regrettably, these alternate features may not exhibit the robustness requisite for accurate version prediction, ultimately resulting in a dip in accuracy.

Secondly, the inadequacy of discernible features across patch versions presents an additional obstacle. Numerous libraries adhere to the semantic versioning convention [36] encompassing major, minor, and patch versions. For instance, a library's version may assume the form x.y.z, with x, y, and z denoting major, minor, and patch versions respectively. During patch-level updates, developers tend to introduce minimal code changes geared toward rectifying specific issues. In comparison to other updates, the extent of code modification is significantly subdued. Consequently, when endeavoring to identify patch versions during SCA tasks, the intricate task of ascertaining the precise version arises due to the subtle nature of code alterations. This intricate landscape adds to the challenge of accurate version detection.

Multi-library Detection. In this experiment, we aim to evaluate the ability of SCA tools in detect libraries that are combined together to form one big binary. The reason for this experiment is that, in the real-world use case, the developer may choose to compile the libraries into their software system. Without completed binary reverse engineering, it is difficult to separate and extract the libraries from the system binary. Therefore, during the SCA task, it is important for the SCA tools to detect all the libraries when they are integrated into one binary file. To achieve the goal, we manually compile multiple libraries and mix them into one binary file. In total, we obtain 10 such binary files with 56 libraries involved.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3341224

Table 4 presents the accuracy of the SCA tools in detecting these libraries.

TABLE 4: Precision and Recall for Multiple Library Detection

SCA Tools	Number of Libraries	Precision	Recall
Blackduck	56	0.52	0.19
Scantist		0.29	0.19
Cybellum		0.84	0.19

The findings exhibit a notable decrease in the recall of the three tools as compared to common library detection. The major reason for this phenomenon is because of the fact that SCA tools tend to predict one library rather than multiple entities. In most cases, SCA tools will try to divide the mixed binary into smaller files and predict the library name on each of the files. Therefore, if they fail to divide the mixed binary, they may predict only one library, which leads to a significant decrease in the recall. This observation underscores that the SCA tools are designed to anticipate the library with the most compatible features. In scenarios where the binary encompasses multiple libraries, the principal libraries are disproportionately favored in terms of detection likelihood compared to secondary ones. This observation underscores a research gap in the realm of multi-library detection tasks, necessitating attention to precisely identify components within such contexts.

Obfuscated Library Detection. In this experiment, we plan to evaluate the accuracy of detecting libraries with obfuscation. It is because, in the real world, developers may modify the program to achieve customization and avoid information leakage. For example, due to security concerns, programmers will apply patches to the program to fix vulnerabilities. The application of the patches can be considered as an obfuscation of the component. Moreover, programmers often strip the strings in the binary file so that the SCA tools cannot use strings to guess the library names and versions. We select 8 strategies to mutate the original library and manually apply the mutation to the binary files. Table 5 presents the SCA accuracy in detecting the obfuscated library version under different strategies.

TABLE 5: Obfuscated Library and Version Detection Ability

Obfuscation Method	Blackduck	Scantist	Cybellum
Code Flattening	N	Y	Y
Code Substitution	N	Y	Y
Conditional Compiling	N	Y	Y
Constant Obfuscation	N	Y	Y
Control Flow Obfuscation	N	Y	Y
String Stripping	N	Y	Y
Symbol Renaming	N	Y	Y
Code Splitting	N	Y	Y

The result shows strong differences in detecting the obfuscated binaries. Blackduck has the worst detection accuracy since it cannot detect the library when obfuscation is present. It may be because it relies heavily on syntax information to predict the library names. For example, it may use the hash value of the entire binary file to match the binary or it may use the function name inside the binary to predict the library. These features are extremely unstable when the obfuscation is applied. Whereas, Scantist and Cybellum have 100% accuracy in detecting such binaries. We believe it uses semantic features such as the function call relationship to determine the library name and versions. This approach will have relatively higher accuracy in handling the obfuscated files. However, as more heavy program analysis algorithms are involved the time consumption also increases. Moreover, it may produce more false positive cases by miss-predicting non-library binaries as third-party libraries.

TABLE 6: Precision and Recall for Comparison with BinaryAI

SCA Tools	No. of Libs	Lib Detection		Version Detection	
		Precision	Recall	Precision	Recall
BinaryAI	22	0.88	0.68	0.7	0.55
Blackduck		1	0.27	1	0.09
Scantist		0.91	0.50	0.91	0.46
Cybellum		0.86	0.29	0.57	0.21

Academic Tool Accuracy Evaluation.

In this section, our objective is to assess an academic SCA tool and juxtapose it with commercial counterparts. We evaluated one academic tool, BinaryAI, with a limited scope due to its constraints. Specifically, BinaryAI limits the total size of the input files. Therefore, we constructed a small testing data set with 22 libraries to compare the accuracy with the industrial tools. The outcomes of this assessment are presented in Table 6. Analysis of the table reveals that BinaryAI achieves a commendable equilibrium between precision and recall in library name and version detection, often matching or slightly surpassing the performance of commercial tools. An aspect of concern is its processing speed, given its reliance on instruction-level data and machine learning models. While potentially time-consuming, we withhold conclusive judgment due to the absence of tests on larger binary files.

Answer to RQ1: The results reveal that while all three tools perform well in detecting Debian and RPM libraries, accuracy declines for NetBSD due to lesser information. For version detection, there is a drop in accuracy attributed to the absence of version-specific strings and challenges in discerning patch versions. Detection of libraries within combined binaries is evaluated, demonstrating variations among tools in accuracy. Detection of obfuscated libraries is assessed with varied strategies, revealing challenges for Blackduck and high accuracy for Cybellum due to semantic features, despite potential time and false positive implications. The academic SCA tool BinaryAI is evaluated, demonstrating competitive accuracy in library name and version detection.

C. RQ2: PERFORMANCE COMPARISON

In this RQ, we aim to measure the time cost of scanning the binary files to detect the components. Table 7 shows the results of the average time used to scan one binary file for each of the SCA tools. From the table, it is evident that

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3341224

Scantist demonstrates the best overall performance, taking an average of less than 1 second to scan one item with a size of 0.84 to 4.55 MB. On the other hand, Cybellum proves to be the slowest, taking 9.358s to 34.428s to complete the scan of one item.

TABLE 7: Time Cost of Component Detection on Benchmark Dataset

SCA Tool	Data Source	Average File Size	Time
Blackduck	debian	0.84 Mb	1.518s
	rpm	2.83 Mb	4.490s
	netbsd	4.55 Mb	0.578
Scantist	debian	0.84 Mb	0.0168s
	rpm	2.83 Mb	0.0332s
	netbsd	4.55 Mb	0.679s
Cybellum	debian	0.84 Mb	13.159s
	rpm	2.83 Mb	34.428s
	netbsd	4.55 Mb	9.358s

Upon investigating the reasons behind relatively short and long scan times, we have identified the following factors. All of the SCA tools employ a drill-down approach during scanning, indicating that they utilize multiple strategies to determine components with varying levels of time cost, recall, and precision. They begin with strategies that have the lowest time cost. If they find a match with high confidence, they stop the process and report the result immediately, without going through further strategies. An example of such a strategy is hash matching, where they pre-collect hash values for standard components in their database and perform a lookup of the target component's hash. This strategy exhibits high precision and minimal time cost. However, it suffers from limited coverage, leading to numerous false negative cases. Consequently, they proceed to try other strategies until they find the component's name or exhaust all strategies. The rapid hash matching explains why some large binary files can be processed in a short time by SCA tools. Conversely, for certain binaries, the tools may take significantly longer as they employ more time-consuming analysis methods.

Scantist stands out as it boasts a well-built binary database and implements lightweight strategies, such as hash, string, and file name matching. This contributes to its excellent overall performance. It also uses the meta-information of the packages to rapidly determine the library names and versions, which avoids time-consuming feature extraction and comparison tasks. Conversely, Cybellum's use of time-consuming strategies like string set matching hinders its efficiency. Moreover, it introduces heavy program analysis methods, such as symbolic execution, to further detect the vulnerabilities inside the components. The additional functionalities cost much time to slow down the process.

In addition, BinaryAI has a relatively much higher time cost with over 100 seconds per library. We have also tested the time cost for other academic tools such as ModX [12]. Similarly to BinaryAI, other academic tools also exhibit high time costs as they involve reverse engineering and program analysis techniques, such as symbolic executions, which substantially increase time consumption.

Answer to RQ2: Scantist demonstrates exceptional performance, completing scans in less than 1 second on average for files ranging from 0.84 to 4.55 MB, while Cybellum proves to be the slowest, taking between 9.358s and 34.428s per scan. All tools prioritize strategies with minimal time cost and high-confidence matches, such as hash matching. Scantist's success stems from its streamlined strategies, lightweight methods like hash, string, and file name matching, and efficient utilization of package meta-information. In contrast, Cybellum is coupled with resource-intensive methods like symbolic execution for vulnerability detection, contributing to its slower performance.

D. RQ3: DOMAIN SPECIFIC COMPONENT DETECTION COMPARISON

In this research question, our objective is to evaluate the component detection capabilities of each SCA tool within a specific application domain. We executed each tool on a dataset collected from the automotive system, and the accuracy results are presented in Table 8.

TABLE 8: Precision and Recall for Automotive-related Library Detection

SCA Tools	Number of Libraries	Precision	Recall
Blackduck		0.45	0.10
Scantist	432	0.40	0.08
Cybellum		0.43	0.09

Analyzing the table reveals a notable decrease in accuracy compared to the detection of general libraries. This is primarily due to the increased false negative rate, indicating that all tools failed to detect certain components. The unique characteristics of components used in automotive systems require SCA tools to possess comprehensive signatures to successfully identify them. However, commercialized tools may lack complete coverage of libraries specific to this domain, leading to a higher rate of false negatives in component detection.

On the other hand, all tools maintain a relatively low false positive rate. This can be attributed to two factors. First, components in automotive systems are typically not obfuscated and are compiled using standard compilers and settings. Additionally, the strings within these components are generally preserved, providing sufficient features for the SCA tools to determine their identity. Second, there is an overlap between commonly used components and specific components in automotive systems, allowing the SCA tools to effectively detect them.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3341224

Answer to RQ3: The results indicate a decrease in accuracy compared to general library detection due to a higher false negative rate. This suggests that the tools struggle to identify certain components unique to the automotive domain. Commercial tools might lack comprehensive coverage of domain-specific libraries, leading to more false negatives. However, all tools maintain a low false positive rate due to the absence of obfuscation and the preservation of strings within automotive components.

E. RQ4: ADDITIONAL ABILITY EVALUATION

In this RQ, we explore the supplementary capabilities offered by SCA tools beyond component detection. To achieve this, we conducted manual evaluations of each SCA tool, executing them with their complete functionalities and collecting the resulting reports. Subsequently, we thoroughly examined these reports to identify and summarize the additional functionalities provided by each tool. The comprehensive list of these functionalities is presented in Table 9. This analysis sheds light on the diverse capabilities offered by SCA tools and contributes to a comprehensive understanding of their value beyond component detection.

From the table, we can conclude that all commercialized SCA tools offer additional functionalities. There are two reasons for this. First, as commercial tools, they strive to provide enhanced capabilities to outperform their competitors. Second, SCA alone is not a complete security operation solution. Knowing what components that have been used in the software is the first step to managing the security risk. Therefore, additional functionalities are provided to provide full security protection.

The second notable finding is that vulnerability detection is the most commonly provided functionality. This is because the primary goal of SCA is to manage the security risks associated with using open-source components and identifying vulnerabilities is a critical aspect of this process. Therefore, all commercial tools include vulnerability detection as part of their security solutions.

Thirdly, some of the minor functionalities prove to be more valuable than anticipated. For example, detecting the various interfaces used in the software is crucial as it enables security experts to quickly identify potential attack surfaces. Conversely, functions such as detecting password files and unencrypted files are more commonly provided but offer limited assistance, as most developers already acknowledge the risks associated with these issues.

Answer to RQ4: The analysis reveals that commercialized SCA tools universally incorporate supplementary functionalities, driven by both competitive market pressures and the recognition that SCA alone is not a comprehensive security solution. These tools align with the primary aim of SCA by prominently emphasizing vulnerability detection, a crucial facet for managing security risks. Notably, some minor functionalities, such as interface detection, provide useful information for security expertise.

VI. DISCUSSION

A. LESSONS LEARNED

For SCA tool developers, two key factors can enhance the capabilities of their tools. Firstly, the quality of the data directly impacts the detection accuracy, necessitating careful preprocessing of open source library data. Developers should strive to improve data coverage to enhance the recall of SCA tools. Additionally, as data size increases, signatures may become less distinguishable, leading to a higher false positive rate. A major reason for this issue is the presence of dependencies between open source packages in software. For instance, when a package P_a uses another package P_b as a dependency, it may confuse the tools in determining whether the software utilizes P_a or P_b. To alleviate such situations, incorporating duplicated code removal during the construction of the open source database can be beneficial. Therefore, developers should pay close attention to this preprocessing step to further enhance the performance of SCA tools.

For end users of SCA tools, it is crucial to recognize that the detection results may not guarantee the identification of all open source libraries in the software. Failures in detection can occur due to factors such as data gaps, non-robust features, and limited reverse engineering capabilities. Therefore, it becomes imperative to establish an open source security management system throughout the entire software development lifecycle, shifting the security operation leftwards. SCA tools can serve as the final step to ensure that no harmful code is introduced into the system. However, prior to this stage, building a secure and trusted open source library warehouse can prevent the use of dangerous libraries from the outset. By adopting such proactive measures, end users can significantly enhance the security of their software projects.

B. FUTURE RESEARCH DIRECTION

In this section, we outline potential research directions for future work in the field.

First, there is ample room to improve the overall accuracy of SCA tools. Existing industrial SCA tools rely on simple features to build component fingerprints, such as strings and constants. However, these features suffer from coverage issues, as not all components have distinguishable strings and constants. Moreover, they lack robustness, as they can be easily changed, making them less effective in detecting different versions of target components. To address these limitations, it is essential to incorporate more robust features that can better handle the variability and versioning of components.

Second, the generalizability of academic SCA tools needs improvement. Our empirical study revealed challenges in reproducing the results of academic SCA tools across different datasets. Most academic tools are not open-sourced and require sophisticated software engineering techniques like reverse engineering and program analysis to extract fine-grained features. While these features can enhance detection accuracy, obtaining them can be time-consuming and non-scalable. Future research should focus on developing SCA tools that can effectively detect libraries in real-world sys-

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3341224

TABLE 9: Additional Ability. (Bd for Blackduck, Cy for Cybellum, and Sc for Scantist)

Function	Description	Bd	Cy	Sc
Unknown Vulnerability Perception	Detect unknown vulnerabilities based on predefined rules.	N	Y	Y
Known Vulnerability Scanning	Identify known vulnerabilities by referencing publicly disclosed vulnerabilities.	Y	Y	Y
Fortification Method Perception	Detect fortification methods employed by binary programs in the firmware.	Y	Y	Y
Password File Detection	Analyze the presence of public and private key files in the scanned package.	Y	Y	Y
Unencrypted Communication Perception	Identify executable files using specific communication functions.	Y	Y	Y
Coding Standard Detection	List files not complying with coding standards.	N	Y	Y
Password Information Leakage Detection	Identify files that may contain leaked account credentials.	Y	Y	Y
Sensitive Information Leakage Detection	List identified sensitive information in the firmware package.	Y	Y	Y
Error Security Configuration Detection	Identify potential erroneous system configurations.	Y	Y	Y
Firmware Component Attribution	List potential third-party components present in the firmware package.	Y	Y	N
License Recognition	Identify open-source software licenses associated with third-party components.	Y	Y	Y
Interface Recognition	List files potentially related to various interfaces.	N	Y	N
Remediation	Give advice for vulnerability fixes and software updates	Y	Y	Y

tems using larger datasets, and the tools should be designed with better generalization capabilities.

Third, given the current challenges faced by existing SCA tools in effectively detecting components in automotive systems due to limited data support, researchers can address this issue by proposing data scope analysis. This analysis aims to define the specific data that SCA tools should include to reduce false negatives in automotive systems. It involves outlining the data available from various sources, such as libraries hosted on Github, individual websites, and other repositories. Additionally, researchers should devise appropriate methods to select important libraries from this vast quantity of data to ensure comprehensive coverage and accuracy in component detection.

VII. RELATED WORKS

In this section, we list the related works that focus on the SCA tools and SCA tool evaluation.

SCA Tool Evaluation. There are a few works that focus on evaluating the accuracy of the SCA tools. For example, [37] conducting an empirical study to analyze the vulnerability reporting functionality of 9 SCA tools for Java and JavaScript projects. Haq and Caballero [17] analyze 61 binary code similarity methods to provide the characteristics, the implementation, and the scope of these works. Sheneamer and Kalita [18] discuss code clones, common types of clones, phases of clone detection, the state-of-the-art in code clone detection techniques and tools, and challenges faced by clone detection techniques. Wang, Jingdong, et al. [19] analyze the solutions to address the similarity search problem. Roy and Cordy [20] study the code clone techniques and provide clone taxonomies. Rainer Koschke [21] focuses on the different clone types and measures the consequences resulting from code clones. Most of these works focus on surveying source code-level code clone techniques. Our work aims to provide a study on SCA tools, which provide the downstream task of the code clone. We aim to examine the industrial usage of the SCA tools and provide suggestions for open-source software management tasks. Moreover, we perform the study at the binary level, which is close to the real-world use case.

SCA Tools. Numerous approaches have been proposed to enhance the accuracy of component detection across various programming languages. In the context of Java, AtvHunter [2] utilizes the control flow graph of specific Java libraries as signatures for detecting similar components. Backes et al. [3] extract library profiles by leveraging code hierarchy information, enabling them to detect APK packages despite obfuscation. Libd [4] employs dependencies within functions as signatures for library reuse detection. LibPecker [5] further enhances detection accuracy by introducing an adaptive class similarity threshold to calculate the similarity between signatures and target libraries. Additionally, several works [6]–[9] propose clone detection techniques, which can also be applied to component detection tasks.

For the C programming language at the binary level, BAT [38] employs clone detection with string literal matching algorithms to detect binary libraries. OSSPolice [10] leverages invariant literals and a hierarchical indexing scheme to identify components. B2SFinder [11] maps binaries to source code using invariants to detect components in commercial-off-the-shelf software. ModX [12] introduces modularization techniques, dividing libraries into smaller parts to enable finer-grained matching for component detection. Furthermore, there are works that focus on detecting libraries at the source code level. For instance, Centris [39] proposes a duplication removal technique to filter out clones between two libraries, thereby improving signature quality. TPLite [40] performs detailed analysis on source code level SCA tool Centris and proposes TPLite, which outperforms it via accurately determining the function birth time. OSSFP [41] enhances Centris by categorizing functions into core and non-core types, applying duplication removal only to less important functions.

Additionally, research efforts have aimed at component detection in languages such as JavaScript, Python, etc., with a focus on improving accuracy and performance. However, none of these works have been evaluated using widely agreed-upon benchmarks, making cross-comparisons challenging. Our work addresses this issue by providing a benchmark for evaluating SCA tools and facilitating future comparisons and assessments of component detection capabilities.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2023.3341224

VIII. CONCLUSION

In this paper, we present an empirical study evaluating the component detection capabilities of software composition analysis tools in the C/C++ programming language. We curate and introduce two datasets comprising over 16,000 individual binary components, establishing ground truth benchmarks for evaluation. Our study covers three industrial SCA tools and examines the performance of the academic tool BinaryAI. The results indicate that industrial tools exhibit higher accuracy (70.35%) in detecting commonly used components, with a significant decline in precision to less than 45% when handling automotive binaries. Moreover, industrial tools have relatively equal accuracy with BinaryAI. However, due to the inability to reproduce claimed results and limited access to open-source libraries, other academic tools fail to have satisfactory results as the industrial tools. Additionally, all tools demonstrate acceptable performance and offer a wide array of security-related functionalities to support open-source security management. Overall, our study provides valuable insights into the capabilities and limitations of SCA tools, aiding software engineers in making informed choices for effective component detection.

REFERENCES

[1] Kaiping Liu, Hee Beng Kuan Tan, and Xu Chen. Binary code analysis. Computer, 46(8):60–68, 2013.

[2] Xian Zhan, Lingling Fan, Sen Chen, Feng We, 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.

[3] 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.

[4] 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.

[5] 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.

[6] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In Proceedings of the 36th International Conference on Software Engineering, pages 175–186, 2014.

[7] Jonathan Crussell, Clint Gibler, and Hao Chen. Andarwin: Scalable detection of android application clones based on semantics. IEEE Transactions on Mobile Computing, 14(10):2007–2019, 2014.

[8] Mario Linares-Vásquez, Andrew Holtzhauer, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Revisiting android reuse studies in the context of code obfuscation and library usages. In Proceedings of the 11th Working Conference on Mining Software Repositories, pages 242–251, 2014.

[9] 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.

[10] 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.

[11] 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.

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

[13] Snyk. https://snyk.io/, 2023.

[14] Sonarqube. https://www.sonarsource.com/, 2023.

[15] Black duck software. https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis.html, 2023.

[16] Cybellum. https://cybellum.com/, 2023.

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

[18] Abdullah Sheneamer and Jugal Kalita. A survey of software clone detection techniques. International Journal of Computer Applications, 137(10):1–21, 2016.

[19] Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. Hashing for similarity search: A survey. arXiv preprint arXiv:1408.2927, 2014.

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

[21] Rainer Koschke. Survey of research on software clones. In Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[22] Israel Herraiz, Gregorio Robles, Jesús M González-Barahona, Andrea Capiluppi, and Juan F Ramil. Comparison between slocs and number of files as size metrics for software evolution analysis. In Conference on Software Maintenance and Reengineering (CSMR'06), pages 8–pp. IEEE, 2006.

[23] André Miranda and João Pimentel. On the use of package managers by the c++ open-source community. In Proceedings of the 33rd Annual ACM Symposium on Applied Computing, pages 1483–1491, 2018.

[24] 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 Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pages 1–12, 2022.

[25] Di Wu, Lin Chen, Yuming Zhou, and Baowen Xu. How do developers use c++ libraries? an empirical study. In Proceedings of the Twenty-Seventh International Conference on Software Engineering and Knowledge Engineering, pages 260–265, 2015.

[26] Yung-Yu Chang, Pavol Zavarsky, Ron Ruhl, and Dale Lindskog. Trend analysis of the cve for software vulnerability management. In 2011 IEEE third international conference on privacy, security, risk and trust and 2011 IEEE third international conference on social computing, pages 1290–1293. IEEE, 2011.

[27] Matthew Finifter, Devdatta Akhawe, and David Wagner. An empirical study of vulnerability rewards programs. In 22nd USENIX Security Symposium (USENIX Security 13), pages 273–288, 2013.

[28] Nvd. https://nvd.nist.gov/, 2023.

[29] 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. In Proceedings of the 19th International Conference on Mining Software Repositories, pages 423–434, 2022.

[30] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. IEEE Transactions on software engineering, 28(8):735–746, 2002.

[31] Mariano Ceccato, Andrea Capiluppi, Paolo Falcarin, and Cornelia Boldyreff. A large study on the effect of code obfuscation on the quality of java code. Empirical Software Engineering, 20:1486–1524, 2015.

[32] Benchmark link. https://drive.google.com/drive/folders/1AtjYQ_qQtJdHtmRwPdtvm61IbN5C0YV3?usp=sharing, 2023.

[33] Scantist. https://scantist.com/, 2023.

[34] 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.

[35] 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.

[36] Semantic versioning. https://semver.org/, 2023.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] Jiahui Wu, Xu Zhengzi, Tang Wei, Zhang Lyuye, Wu Yueming, Liu Chengyue, Sun Kairan, ZHAO LIDA, and Liu Yang. Ossfp: Precise and scalable c/c++ third-party library detection using fingerprinting functions. In Proceedings of the 45th IEEE/ACM International Conference on Software Engineering, 2023.

YUQIAO NING is the Technical Director of CATARC Intelligent and connected technology Co., Ltd. He has rich experience in the computer system and software security research, mainly responsible for the research of automotive penetration technology and the development of automated detection tools, participated in the construction and operation of the China Automotive Vulnerability Database (CAVD), focusing on researching the impact of open source software's security vulnerabilities on automobiles, meanwhile, he once discovered security vulnerabilities in Tesla, Huawei, Alipay, General Motors and Xiaomi, etc., and supported the relevant departments of the state in carrying out the security inspection and assessment of the Internet of Vehicles (IoV), and participated in writing a number of national standards on automotive information security.

YANAN ZHANG received her Master of Science degree in Beihang University. She is the chief expert of China Automotive Technology and Research Center Co., Ltd., Deputy chief engineer of Automotive Data of China (Tianjin) Co., Ltd., concurrently ISO/TC22 international registration expert, Ministry of Industry and Information Technology network security special evaluation expert, mainly conducted research on information security technologies and policies such as network security and data security for intelligent connected vehicles, lead the formulation of GB/T40856-2021 and other national standards, participated in more than 10 major provincial and ministerial scientific research projects, and publish 2 volumes of professional books. Her research achievements have won the second prize of Science and Technology Progress of China Intelligent Transportation Association, the first prize and the second prize of Tianjin Science and Technology Progress, etc.

CHAO MA is the senior engineer, director of the Information Security Office of CATARC Intelligent and connected technology Co.,LTd.. He is concurrently an expert in network security supervision and review of the Ministry of Industry and Information Technology. His main research fields are intelligent networked vehicle information and network security, data security, etc. Participated in the construction of the Ministry of Industry and Information Technology's IoV product vulnerability database and IOV network trust support platform, participated in the research and compilation of "Voluntary Product Certification Implementation Rules - Automobile (Information Security)", and participated in the construction of China Automobile Information Security Sharing and Analysis Center (C-AUTO- ISAC), etc. He has led or participated in more than 10 major provincial and ministerial scientific research projects and has published more than 20 papers in the field of automotive information security.

ZHEN GUO received his master's degree in mechanical engineering from Tianjin University in 2020. He is working at CATARC Intelligent and connected technology Co.,LTd. as the supervisor of the information security office. His current research fields include Internet of vehicles cybersecurity, software security, and vulnerability detection. Completing car firmware vulnerability discovery and vulnerability verification procedure. He is also skilled at the establishment of vehicle information security management systems and automotive information security regulatory interpretation, helping enterprise clients build their automotive information security capabilities.

LONGHAI YU works in the Information Security Office of CATARC Intelligent and connected technology Co., Ltd. His main research interests include automotive information security and lightweight encryption for the Internet of Things. Graduated from the School of Computer Science at Guangdong University of Technology, with three years of work experience, including participation in the completion of 5G private network security testing, smart meter security testing, IPTV security testing, and vehicle security testing.

• • •

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4