Preserving Privacy in Software Composition Analysis: A Study of Technical Solutions and Enhancements

Huaijin Wang[1], Zhibo Liu[1†], Yanbo Dai[2], Shuai Wang[1], Qiyi Tang[3], Sen Nie[3], Shi Wu[3]
[1] Hong Kong University of Science and Technology, [2] Hong Kong University of Science and Technology (Guangzhou),
[3] Keen Security Lab, Tencent
{hwangdz, zliudc, shuaiw}@cse.ust.hk, ydai851@connect.hkust-gz.edu.cn, {dodgetang, snie, shiwu}@tencent.com

Abstract—**Software composition analysis (SCA) denotes the process of identifying open-source software components in an input software application. SCA has been extensively developed and adopted by academia and industry. However, we notice that the modern SCA techniques in industry scenarios still need to be improved due to *privacy* concerns. Overall, SCA requires the users to upload their applications' source code to a remote SCA server, which then inspects the applications and reports the component usage to users. This process is privacy-sensitive since the applications may contain sensitive information, such as proprietary source code, algorithms, trade secrets, and user data.**

Privacy concerns have prevented the SCA technology from being used in real-world scenarios. Therefore, academia and the industry demand privacy-preserving SCA solutions. For the first time, we analyze the privacy requirements of SCA and provide a landscape depicting possible technical solutions with varying privacy gains and overheads. In particular, given that de facto SCA frameworks are primarily driven by code similarity-based techniques, we explore combining several privacy-preserving protocols to encapsulate the similarity-based SCA framework. Among all viable solutions, we find that multi-party computation (MPC) offers the strongest privacy guarantee and plausible accuracy; it, however, incurs high overhead ($184\times$). We optimize the MPC-based SCA framework by reducing the amount of crypto protocol transactions using program analysis techniques. The evaluation results show that our proposed optimizations can reduce the MPC-based SCA overhead to only 8.5% without sacrificing SCA's privacy guarantee or accuracy.

I. Introduction

Given a piece of software, software composition analysis (SCA) identifies components borrowed from third-party and open-source software (OSS) projects. Hence, according to the component-usage information, developers are aware of the usage of vulnerable or outdated OSS projects and, therefore, are able to take further actions to reduce security risk, ensure license compliance, and promote healthy open-source project usage. In recent years, the industry [1, 2, 6, 7] and academia [16, 20, 21, 45, 46, 57, 68, 76, 77, 84, 92–94] have extensively studied SCA techniques in different settings and achieved promising results regarding accuracy and efficiency.

Despite the prosperous development and adoption of SCA techniques, we notice that the accompanying *privacy concerns* are hindering the usage of SCA in real-world scenarios. Indeed, some authors of this paper have been working in a leading industrial company of SCA solutions, and they are frequently encountering and handling requests for performing SCA without disclosing customer's data in recent years. The typical pipelines of industrial SCA solutions require a customer to upload the software to a remote server possessed by the SCA vendor, where the software is analyzed to produce a report that reveals the potential security and legal risks of the software. The report usually contains a list of reused OSS projects [11, 14, 37], some of which may include known vulnerabilities. This process is privacy-sensitive since the uploaded software may contain sensitive information, such as trade secrets or personal data. Moreover, the source code is proprietary and deemed part of the customer's intellectual property (IP). In reality, few companies are willing to provide commercial products' source code to another party since it is hazardous to leak valuable IPs [81].

To avoid leaking customers' privacy, contemporary SCA vendors may deploy the SCA service in the customer's in-house domain. While this alleviates the privacy concerns from the user side, it requires the SCA vendor to share its IPs (i.e., the SCA frameworks, algorithms, and accompanying data) with the customers. For instance, When the SCA service is deployed on the customer's side, the SCA vendor has to release the OSS database to customers periodically. However, existing SCA vendors are often unwilling to share the valuable OSS database, which requires many resources to build. Existing works [38, 84] show that a comprehensive OSS database with reliable OSS dependencies has an essential impact on the accuracy of SCA, and it is not economical for small and medium-sized companies to build their own databases. For instance, it is disclosed that BinaryAI [11], an industrial SCA vendor, has spent over three years, a dozen (senior) engineers, and PB-level disk space to construct its OSS database. In addition, BinaryAI relies on a large model that has been trained with ten Nvidia Tesla T4 GPUs for years. Thus, exposing the model's parameters is unacceptable.

Since SCA serves as one of the cornerstones of software security and a paramount part of software engineering, the industry and academia are demanding a privacy-preserving SCA solution to promote the practical adoption of SCA in industry scenarios. Accordingly, this paper targets this practical need to study the possible technical solutions for privacy-preserving SCA, and we aim to answer the following two research questions (RQs):

- **RQ1:** How can we effectively protect both customers' and vendors' privacy in SCA?
- **RQ2:** How can a privacy-preserving SCA pipeline be built with satisfactory performance for practical use?

To answer **RQ1**, we systematically study existing technical solutions and their (dis)advantages, exploring and proposing two novel privacy-preserving SCA frameworks. As shown

† Corresponding author.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00055

in Table I, each studied solution exhibits distinct privacy guarantees, required resources, overhead, and potential pitfalls. Furthermore, we concretize the technical solutions and empirically benchmark their overheads and accuracies. Specifically, the two novel solutions we proposed involve advanced privacy-preserving protocols to offer privacy-preserving SCA. One solution relies on similarity-based bucketization (SBB) [35], a scale private similarity testing technique; the other depends on multi-party computation (MPC) [19, 24, 88], which enables multiple parties to jointly perform computation without revealing their private data to each other. Sec. V-E demonstrates that the SBB-based solution, although lightweight, still bears the risk of potential privacy leakage. In contrast, MPC offers a much stronger privacy guarantee by shielding both the SCA customers' and vendors' valuable assets, such as code, model, and data. Overall, our study illustrates the promising potential of MPC-based SCA.

Nevertheless, the MPC-based SCA pipeline we proposed incurs excessively high overhead (184×), which prohibits its practical usage. To answer **RQ2**, we propose three optimization strategies with program analysis techniques, which significantly reduce the amount of crypto protocol transactions. Our optimization strategies, including (1) *symbol filter*, (2) *informative source function selector*, and (3) *assembly function selector*, can vastly reduce the time cost of MPC-based SCA from 184× to 23×, and the most time-consuming process, i.e., encrypted computation with MPC, is reduced to 4.8%, leading to a practical privacy-preserving SCA solution. In sum, this work presents the following contributions:

- Observing the privacy concerns of SCA over both the user and the SCA service provider, we, for the first time, study the privacy-preserving SCA, whose absence is a crucial hindrance to the practical adoption of SCA in industry scenarios.
- We analyze the privacy leakage risks in three typical SCA solutions and formulate two novel privacy-preserving SCA frameworks with SBB and MPC, respectively. Each studied solution exhibits distinct privacy guarantees and required resources. We then present an empirical evaluation of their overheads and accuracy. We find that MPC offers high accuracy with the strongest privacy guarantee.
- We design a set of optimizations that can largely reduce the cost of expensive crypto operations for the MPC-based SCA solution and without sacrificing accuracy, achieving the first practical privacy-preserving SCA.

Artifact Availability. We have released our artifact at [5]. We will maintain it for future research comparison and usage.
Extended version. We present additional details in an extension due to limited space: https://arxiv.org/abs/2412.00898.

II. Background

A. A Common SCA Pipeline

This section introduces the common SCA pipeline, which is widely adopted by state-of-the-art (SoTA) C/C++ SCA works [11, 38, 70, 84, 85, 87]. As shown in Fig. 1, the pipeline comprises a server (usually owned by the SCA vendor) and

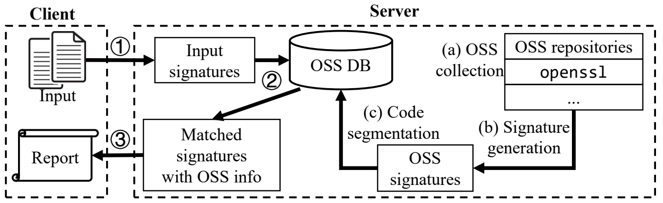


Fig. 1: A common SCA pipeline.

a client for customers to access the SCA service. To perform SCA, the server first builds an OSS database (OSS DB) offline (i.e., (a), (b), and (c)) and then provides an online SCA service (i.e., edges ①, ②, and ③) to the client. Accurate SCA analysis relies on extracting high-quality distinguishable code signatures that can help identify semantic differences between code snippets. Recent SCA works [38, 68, 70, 77, 84, 92] generate signatures with advanced similarity analysis techniques such as locality-sensitive hashing (LSH) [84, 85] and deep learning-based embeddings [37, 70, 92] to extract features from source or binary code. Below, we introduce the major steps in SCA. Moreover, Sec. II-B discusses a concrete example, CENTRIS.

Offline Analysis. The offline process for building OSS DB consists of three steps: *OSS collection*, *signature generation*, and *code segmentation*.

OSS Collection. An extensive OSS DB is inevitable for real-world SCA tasks. Existing SCA techniques (e.g., CENTRIS [84] and TPLite [38]) build their OSS DB by downloading reputed repositories from open-source platforms like GitHub and GitLab. In this work, we build our OSS database from the same sources. More details are discussed in Sec. V-A.

Signature Generation. The SCA server dissects an OSS repository into smaller code pieces with finer granularities (e.g., functions and source files). Then, the server generates signatures for each code piece based on specific underlying SCA techniques. A common practice is to generate function-level signatures with hash algorithms [84, 85] or embedding techniques [37, 70, 92]. Therefore, each OSS repository in the OSS DB has a set of function-level signatures (e.g., hash values or embedding vectors).

Code Segmentation. A common observation is that an OSS project often reuses other OSS projects. Therefore, the signatures of an OSS project may overlap with some other projects' signatures due to the borrowed code. For instance, `SDL` [10], a popular media library, borrows code from `yuv2rgb` [12], an image conversion library. Thus, some function signatures of `yuv2rgb` are also included in `SDL`'s signatures. The overlapped signatures likely result in false positives in SCA; e.g., a project reusing `yuv2rgb` is likely to be identified as reusing `SDL`. Existing works (e.g., CENTRIS [84] and TPLite [38]) segment the sets of OSS projects' signatures into non-overlapped sets to address this issue.

Online Analysis. To conduct SCA, the client first uploads the input source or binary code to the server and generates signatures derived from the input (①). Then, the server searches for similar or identical signatures in the OSS DB and records the matched signatures with their OSS information (②). Finally, the server returns the matched information and the SCA report to the client (③). Details of the three steps are below.

① The client sends the target software's codebase (e.g., source or binary code) to the SCA server, which computes the input signatures in the same way as the offline signature generation. Precisely, the server dissects the input codebase into smaller code pieces (e.g., functions) and then generates signatures (with hash or embedding models) for each code piece.
② Given the input signatures, the server searches for relevant signatures in the OSS DB. Since signatures generated from similar code pieces have short distances, if an input signature is similar or identical to one signature in the OSS DB, the code piece is likely borrowed from the corresponding OSS project.
③ Having identified which code snippets are being borrowed from OSS projects, the server reports the SCA results to the client, which depict the information on reused OSS projects. Specifically, the server can warn the client if the reused OSS projects have known vulnerabilities or license issues.

B. CENTRIS *(TLSH)*

To better understand the pipeline, we take CENTRIS [84], a SoTA source-based SCA framework depending on the Trend Micro Locality Sensitive Hash (TLSH) [66] for signature generation and OSS identification, as an example. For simplicity, we refer to it as "CENTRIS (TLSH)" to distinguish it from other CENTRIS variants.

TLSH. TLSH is a locality-sensitive hash algorithm designed for similarity testing. TLSH generates a hash value for one function and ensures similar functions result in close hash values. In practice, if the distance between two hash values is smaller than a threshold ($\delta = 30$), the two source functions are considered from the same source with high confidence. Specifically, with the distance measure function (dis), the functions f_i and f_j are similar if their hash values h_i and h_j satisfy $dis(h_i, h_j) < \delta$.

Offline Analysis. Given the set of collected OSS repositories $O = \{o_1, o_2, \dots, o_n\}$, CENTRIS first dissects $o_k \in O$ into a set of source functions F_k, and then generates TLSH hash values for each function in F_k, which forms the function hash value set H_k. After processing all OSS repositories, for each OSS o_k, CENTRIS segments the hash value set H_k into a non-overlapped set DB_k, which is stored in the OSS DB for online analysis.

Online Analysis. Assume there are n OSS projects in the OSS DB, and the function hash value set of the OSS project k is DB_k. First, given the target software's codebase, CENTRIS dissects it into source functions and generates TLSH hash values for each function, resulting in the function hash value set I. Then, CENTRIS searches for similar hash values and their original OSS projects in the OSS DB. For each OSS project k, CENTRIS checks the set of similar hash values between I and DB_k with Eq. 1 and gets the set of similar function pairs S_k. Finally, if $\frac{|S_k|}{|DB_k|}$ is greater than a threshold ϵ (set as 0.10 in CENTRIS), which denotes that a certain portion of functions are deemed as reused, CENTRIS reports the OSS project k as reused. By traversing all OSS projects, CENTRIS can report all reused OSS projects in the input codebase.

$$S_k = \{(h_i, h_j) | \forall (h_i, h_j) \in I \times DB_k \text{ and } dis(h_i, h_j) < \delta\} \quad (1)$$

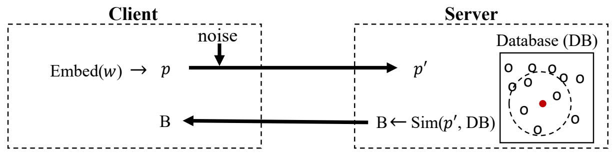


Fig. 2: Similarity-based bucketization protocol.

C. Privacy-Preserving Protocols

Privacy-preserving protocols allow users to protect their privacy while maintaining the service without being affected. In this paper, privacy-preserving protocols aim to protect SCA consumers' and vendors' IPs from being exposed to other parties while allowing a valid and accurate SCA analysis. This section introduces the relevant mainstream privacy-preserving protocols that are relevant to our work.

SBB. Similarity-based bucketization (SBB) [35] provides private similarity testing. Compared with expensive cryptographic protocols, SBB scales well to large databases and does not sacrifice the correctness of matching. Fig. 3 presents the framework of SBB. Given a client's element (w) to be queried, the client first embeds the element and get its representation p. After mutating p with noises, the mutated result p' is sent to the server. Then, the server searches for similar elements with p' (i.e., the red point) in its database and get a bucket (i.e., B) of similar elements, which is then returned by the server. After receiving the bucket, the client searches for the most similar element with the non-mutated p. Since the server can only get the mutated p', the client's privacy is preserved. Sec. IV-A will discuss the application of SBB in the SCA scenario. Nevertheless, the mutated p' may still leak some information to the SCA vendor. Sec. V-E will discuss the SBB leakage in the SCA scenario.

MPC. Multi-party computation (MPC) is a privacy-preserving paradigm that allows participating parties to jointly perform computations over each party's local data. Following exact MPC protocols, nothing except for the correct computation result is revealed to each party. Practical MPC protocols can be either garbled-circuits-based [88] or secret-sharing-based methods [19, 24, 44]. We use CrypTen [44], a secret-sharing method for machine learning, to implement our framework.

While the application of MPC protocols provides cryptographically guaranteed privacy, they often introduce significant overhead, which is influenced by the number of parties and the complexity of the model. There are always two parties in our scenario, and the model remains unchanged after encryption.

III. PRIVACY CONCERNS IN SCA AND MOTIVATION

In this section, we discuss the assets to be protected in SCA (Sec. III-A), the leakage of existing SCA techniques (Sec. III-B), and the motivation for proposing privacy-preserving SCA. In the following discussion, we assume the customer and the vendor hold the client and the server in Fig. 1, respectively, unless otherwise stated (i.e., private deployment).

A. Assets of Concern in SCA

Valuable Assets. As shown in Fig. 1, the online SCA service involves three resources, including the client's input (in source

TABLE I: Leaked resources of various settings. ❌ denotes resources are leaked. "NA" means the SCA solution does not rely on a trained model.

SCA Solution	Valuable Assets					Tool
	Customer-side			Vendor-side		
	Src	Bin	Report	Model	DB	
Source-based	❌	✅	❌	NA	✅	CENTRIS (TLSH)
Private deployment	✅	✅	✅	NA	❌	CENTRIS (TLSH)
Binary-based	✅	❌	❌	✅	✅	BinaryAI
SBB-based	✅	✅	❌	NA	❌	CENTRIS (TLSH + SBB)
MPC-based	✅	✅	✅	✅	✅	CENTRIS (DPCNN + CrypTen)

or binary code format), the SCA report, and the server's OSS database (with any embedding models used to establish the database). We consider those resources (listed in Table I) as valuable assets to be protected for privacy security. As aforementioned in Sec. I, the privacy of customers' code and vendors' OSS database and model are essential. As the importance of protecting customers' source code and the vendor's model is self-evident, and securing the vendor's database has been discussed in Sec. I (e.g., BinaryAI [11, 37] keeps the database private), we discuss the necessity of protecting the client's binary code and SCA reports below.

Privacy Concerns for Binary Code. Although the compiled binary code is less comprehensible than the source code, it still contains sensitive information that can be reverse-engineered and disclosed. Therefore, customers are unwilling to share binaries due to potential security and IP risks [59, 60]. For instance, leaking an online game's server executable results in deploying private servers easily and causing catastrophic financial loss [13]. Even if a binary is publicly available, developers may only release the secured binaries to protect their IP by hindering reverse engineering [17, 25, 41, 55, 56, 80]. For instance, many Android apps are packed and obfuscated before release. Since reverse engineering on those binaries is challenging, binary-based SCA works can hardly extract accurate features [70], resulting in unreliable SCA reports. Thus, it is necessary to consider binary code as a valuable asset to be protected.

Privacy Concerns for SCA Reports. Leaking the SCA report allows a potential attacker to know what OSS is used by the customer; thus, if the customer uses an OSS with a known vulnerability (or weakness), the attacker can directly exploit it instead of searching for vulnerabilities blindly, which is often a time-consuming step in real-world attacks [23, 67]. Hence, leaking the SCA report can ease the attacker's effort in searching and exploiting software vulnerabilities. For example, once an attacker knows a web server uses the vulnerable Log4j [73], the attacker can reuse the disclosed attack resources (e.g., crafted payloads) for other systems depending on the vulnerable Log4j to attack the target web server.

B. Existing SCA Solutions

Based on the input types, we categorize existing SCA techniques into *source-based SCA* and *binary-based SCA*. Besides, we also observe that some companies deploy the server of source-based SCA on the customer side (we refer to it as *private deployment*) to protect customers' proprietary source code. These representative SCA solutions and their privacy leakage are summarized in Table I.

Assume a customer develops a software project P; the source code of P is P_{src}, and the compiled (and stripped) binary is P_{bin}. The vendor owns an OSS database DB and provides an SCA service to detect the reused OSS in P. Besides, binary-based SCA may use an embedding model M for signature generation due to the difficulties in analyzing binaries [37]. Initially, the customer owns P_{src} and P_{bin}, and the vendor owns DB and M. After the analysis, the customer receives an SCA report R. When P_{src}, P_{bin}, and R are available to the vendor, we mark them as ❌ in Table I. The vendor's OSS database DB and model M are marked as ❌ if they are leaked to the customer. The three SCA solutions and their leakages are discussed separately below.

Source-Based SCA. To perform SCA, the customer's client uploads P_{src} to the vendor's server, which then generates the SCA report R and sends it back to the client. While OSS DB is not disclosed to the customer's client, the SCA server can access P_{src} and R. Therefore, the customer's P_{src} and R are leaked, while the vendor's DB is secure.

Private Deployment (of Source-Based SCA). To address the customer's privacy concerns, some vendors deploy the SCA server on the customer side. In this way, the customer has full control over the SCA client and server, while the vendor needs to provide the OSS database DB. Hence, the customer's P_{src}, P_{bin}, and R are secure, but the vendor's DB is leaked.

Binary-Based SCA. The customer's binary P_{bin} is uploaded to the vendor's server, while the source code P_{src} is not disclosed. After receiving P_{bin}, the vendor performs reverse engineering and SCA analysis, generates the SCA report R, and sends it back to the customer. As a result, the vendor has access to P_{bin} and R, while its OSS database DB and model M for signature generation are secure.

Source-Based SCA vs. Binary-Based SCA. Binary-based SCA is mainly proposed for scenarios where source code is unavailable to the vendor, which can be caused by the usage of closed-source libraries or by the customer's privacy concerns. Compared with source-based SCA, binary-based SCA only leaks obscure binary code to the vendor. However, as discussed earlier, a binary still exposes sensitive information when it is reverse-engineered. On the other hand, binary analysis is difficult [25, 37, 52–54, 70, 75, 83, 91], leading to a relatively low SCA accuracy. In Sec. V-D, we show that BinaryAI [11, 37], a commercial binary-based SCA tool, has relatively low accuracy compared with source-based techniques. We thus deem binary-based SCA is far from an ideal privacy-preserving solution.

Motivation. Existing SCA solutions have different privacy concerns. Source- and binary-based SCA leaks customers' assets, while private deployment leaks vendors' assets. Thus, there is a dilemma between customers' and vendors' privacy in SCA, and addressing the dilemma with a privacy-preserving SCA solution is an urgent need.

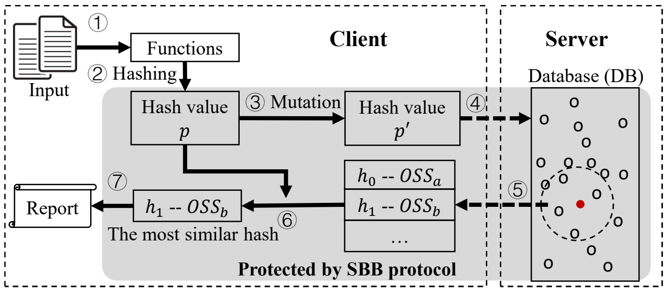


Fig. 3: Workflow of SBB-based SCA.

IV. Privacy-Preserving SCA Frameworks

This section illustrates the privacy-preserving SCA frameworks, where we expect that neither the client nor the server can learn the other's valuable assets (see Sec. III-A). Specifically, the vendor cannot learn the client's source code, binary code, and the SCA report, while the client cannot learn the vendor's database and model (if any). Before introducing their designs in detail, we first clarify the threat model.

Threat Model. We consider a *semi-honest* threat model, where the client and the server are assumed to follow the protocol specification exactly [35, 50]. However, they are curious about each other's private information and record all intermediate results [31]. We assume the customer and the SCA vendor are entities with a reputation and are not likely to conduct maliciously adversarial behaviors.

In order to design privacy-preserving SCA frameworks, we adapt the privacy protocols to an existing SoTA SCA solution. Specifically, we enhance Centris [84], which is introduced in Sec. II-B. We select advanced and representative privacy-preserving protocols to protect the privacy of the SCA pipeline in two aspects: (1) protecting the signature matching process based on similarity algorithm (i.e., ② in Fig. 1) and (2) encrypting the package transactions between the client and the server (i.e., ① and ③ in Fig. 1).

We tentatively adapt SBB to protect the signature matching process, resulting in an SCA framework referred to as "Centris (TLSH + SBB)", whose details are presented in Sec. IV-A. However, we notice that such a solution leaves SCA reports and OSS DB unprotected (explained in Sec. V-E), we resort to the other approach that employs MPC to protect the privacy of different parties with encryption. Specifically, we enhance Centris with code embedding techniques [34, 64, 65] (referred to as "Centris (DPCNN)" in Sec. IV-B) and then adapt the popular MPC framework, CrypTen [44], to protect it (referred to as "Centris (DPCNN + CrypTen)" in Sec. IV-C).

A. *SBB-Based SCA (*Centris *(TLSH + SBB))*

Fig. 3 presents the workflow of SBB-based SCA. The client first dissects the input source repository into functions (①) and then generates the signatures with TLSH (②). By employing the SBB protocol to protect the private data, a function's hash value p will not be sent to the server directly. Instead, the client mutates the hash value p into a new hash value p' (③), and sends p' to the server (④). The server then searches for similar signatures of p' (i.e., the red point) in the OSS DB and replies to the client with a bucket of signatures close to p' (⑤). After receiving the bucket, the client can identify the most similar signature in the bucket with p (⑥). After repeating the process from ② to ⑥ for all functions, the client can report the SCA results to the customer (⑦).

Hyperparameters of SBB. The SBB protocol has two hyperparameters: the distance threshold γ for mutating the hash value p to p', and the bucket size θ for the server to reply to the client. Intuitively, if γ is too small, the mutated hash value p' is close to p, which discloses the information of the client's source code; however, if γ is too large, the server has to use a large θ, which can result in high overhead in identifying the most similar signature in the bucket (⑥); otherwise, the exact similar signatures of p may be missed in the replied bucket.

Privacy Concerns of SBB-Based SCA. As shown in Fig. 3, the SBB protocol protects the client's private data by mutating the signatures before sending them to the server. The client locally generates SCA results according to the embeddings returned by the server. Thus, the server has no access to the original signatures of source code and SCA results. However, the mutated signatures are similar to the original ones and still disclose some information about the client's source code. In Sec. V-E, we demonstrate that the server can infer the client's SCA report, given only the mutated signatures. Moreover, because the server replies to the client with a bucket of similar signatures (i.e., thousands of signatures) for each query, the client can efficiently steal the server's OSS DB by repeatedly querying the OSS DB. As a result, we mark the customer's SCA report and the server's OSS DB as leaked (✖) in Table I.

B. Centris *(DPCNN)*

Since there is no MPC solution for TLSH, we design a variant of Centris (TLSH) that replaces TLSH with an embedding model, which is a deep parallel convolutional neural network (DPCNN) [40] that maps a source function to an embedding. We use the Euclidean distance between two embeddings to evaluate the similarity between two functions.

As mentioned in Sec. II-B, Centris (TLSH) identifies an OSS project as reused when 10% of its functions are recognized in the input. An observation is that the threshold is too high for many OSS projects [38]. Hence, like many existing static analysis methods [8, 11, 30, 72, 74], Centris (TLSH) struggles to avoid false negatives, and lowering the threshold cause a significant increase in false positives [84]. To further explore unleashing the full potential and achieving higher accuracy, we consider assigning weights to the source functions, following the idea of BAT [33] for weighted strings.

Weight Enhancements. Given a function f with similar functions in the OSS database, let the number of OSS repositories with similar functions be N, and the lines of f be LoC. We define the weight of f as $w = \frac{LoC}{5^{N-1}}$. The intuition behind this formula is that a function with many similar functions is less representative (e.g., helper functions) and should consequently have a lower weight. On the other hand, a longer function usually contains rich semantics and should have a higher weight. We then score each OSS project in the OSS DB by the sum of the weights of its similar functions. If an OSS project's score exceeds the threshold β, it is identified

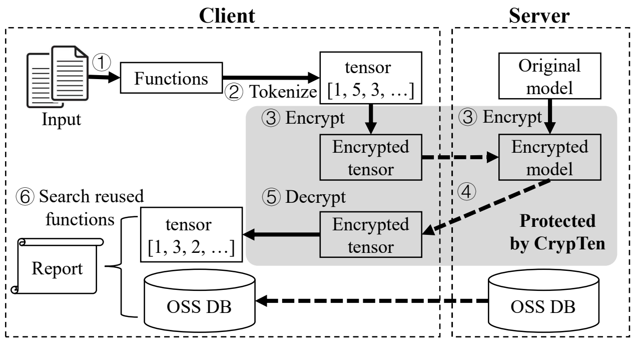


Fig. 4: Workflow of MPC-based SCA.

as reused. Table II shows the result when $\beta = 100$, which is the threshold used by BAT. To efficiently search similar embeddings, we use Faiss [39], an efficient similarity search library provided by Meta, to build an index for all embeddings.

C. *MPC-Based SCA (*CENTRIS *(DPCNN + CrypTen))*

To adapt MPC to SCA, we build a SCA framework whose workflow is presented in Fig. 4. Compared with CENTRIS (DPCNN), the data transaction between the client and the server is protected by CrypTen. Before the online analysis, the client first downloads the OSS DB from the server. During online analysis, the client initially dissects the input source repository into source functions (①) and then tokenizes and converts each function into a plaintext tensor (②). The client then encrypts the tensor with CrypTen (③), and the server encrypts the original model with CrypTen (③). Then, the client sends the encrypted tensor to the server for encrypted computation and receives the encrypted results (④). After decrypting the results, the client gets the plaintext embedding vectors of the functions (⑤), with which the client can search for similar embeddings in the OSS DB and report the SCA results to the customer (⑥).

Shared OSS Vector Database. To avoid leaking the customer's privacy (e.g., SCA reports) to the SCA vendor, the client needs to download the OSS DB from the vendor's server and query the DB locally. Otherwise, if the vendor takes responsibility for conducting DB queries, SCA reports will be revealed directly to the vendor. As summarized in Table I, source- and binary-based SCA have the same issue when the DB is deployed on the vendor's server side. Nevertheless, note that the OSS DB contains only embedded vectors (i.e., signatures) of OSS projects. Consequently, without the embedding model, which is exclusive to the client, the client cannot exploit the DB or hurt the vendor's privacy, which will be introduced in the next few paragraphs. On the other hand, downloading the OSS DB requires additional storage space and network bandwidth. We clarify that the cost is acceptable since the client only downloads the OSS vector database (around 10 GBs) instead of OSS repositories (over 800 GBs).

Privacy Concerns of MPC-Based SCA. From the client's perspective, the CrypTen protocol protects the client's private data by encrypting the data before sending it to the server. Thus, the server cannot access the client's source code and SCA results. Moreover, no information about the client's data is leaked as the MPC protocol guarantees clients' privacy.

A notable concern from the server's perspective is that the server's OSS DB is exposed to the client. However, we emphasize that the meaningful information of the OSS database is *still protected*. To clarify, the client uses the OSS database to search for similar functions from OSS projects. Each record in the OSS database stores the embedding vector of a function, and the information about the project to which this function belongs. To use the OSS database, the client needs access to the embedding model to embed functions for similarity search. However, the model itself is *not* disclosed to the client. Consequently, even if the client has downloaded the OSS database, the client still needs to request the server's service to use the database. In this way, the only meaningful information accessible to clients is the set of included OSS projects; therefore, the privacy of the OSS database is protected.

Another concern is that the client may approximate the server's plaintext model with sufficient query results; we view this as *orthogonal* and mostly beyond the consideration of MPC-based model protection. To understand the mitigation, we observe that the community has developed various techniques to alleviate such model stealing attacks [29, 42, 43, 48], such as limiting the query frequency, updating the model periodically, and detecting malicious queries. Second, in our context, issuing sufficient queries is time-consuming due to the MPC protocol's overhead. As a proof-of-concept, we studied stealing 1% of the OSS database to approximate the source model. The result is not promising (about 0.589 of the plaintext model F1 score), and it takes about three days to finish all queries.

V. BENCHMARKING PRIVACY-PRESERVING SCA

In line with the landscape of privacy-preserving SCA solutions presented in Sec. IV, this section performs empirical evaluations and discusses the results. We first introduce the dataset and the OSS database used in our evaluation and then present the evaluation results. We additionally discuss the privacy leakage of the SBB-based SCA at the end.

A. *OSS Database Construction*

Building a large-scale OSS database is inevitable for real-world SCA tasks. Existing SCA techniques (e.g., CENTRIS [84] and TPLite [38]) build their OSS database by downloading reputed OSSs from open-source platforms like GitHub and GitLab. Since prior works [38, 84] do not release their database or the dependency ground truth due to privacy protection policies, we establish a new large-scale database in this work. In particular, we build the database by identifying all C/C++ software with GitHub URL references collected from the CVE database. Initially, we collected 1,166 C/C++ software that was referred by a CVE number. After that, we also cloned their Git submodules recursively and removed repeated repositories. Eventually, we collected 1,832 C/C++ OSS repositories from GitHub. Those OSS projects have significantly different sizes (ranging from 10 KB to 2.3 GB) and are developed for various purposes (e.g., NumPy for scientific computing and RocksDB for efficient database). Our database contains many more OSS repositories than TPLite (1,036 repositories), demonstrating the comprehensiveness of our setup.

B. Evaluation Dataset

We pick 15 representative OSS projects, each with over 5 `Git` submodules, from our collected OSS database as the evaluation dataset. We use the `Git` submodules to ease the effort of marking the ground truth of reused OSS projects. For each project, we compile it with `GCC` with default compiler flags. We successfully built 14 projects and formed a test dataset of 14 binaries. The projects of 14 binaries contain about 37 million LoC with 211 identified OSS dependencies in our OSS database. Their diverse purposes include but are not limited to multimedia processing, databases, drivers, and coin miners.

It is worth noting that the ground truth marked with `Git` submodules is not perfectly accurate; we still need to manually check each project's source and binary code due to the complex C/C++ software ecosystem [71]. Many OSS projects are reused in the form of copy, and there may be no announcement/documents detailing this in the original OSS projects. To build a reliable ground truth, we first tried to use CENTRIS, the SoTA source code-based SCA framework, to detect OSS reuses. However, the results are unsatisfactory due to imperfect code segmentation and the incapability of TLSH [66] used by CENTRIS. We then manually refine the results by searching for URLs, keywords, file names, and representative function names in the source code. Regarding directly reused binary code, we manually check libraries that are statically linked to the binary. Conducting this manual check took two authors about 40 man-hours. Each author has an in-depth knowledge of reverse engineering, SCA, and rich experience in binary code analysis. This ensures the credibility and accuracy of our dataset to a great extent. Note that to avoid possible biases in the experiments and be close to the real-world scenario, we remove these 14 OSS projects from our OSS database.

C. Experimental Setup

In this section, we elaborate on the setup of CENTRIS and its variants, and the selection of binary-based SCA tool (i.e., BinaryAI [11, 37]).

CENTRIS (TLSH). The official implementation of CENTRIS [9] is not sufficiently efficient to be applied to our extensive OSS database since it searches close hash values in the OSS database one by one. To accelerate the search process, we equip CENTRIS with the HNSW index algorithm [58] for TLSH. Other settings of CENTRIS (TLSH) are kept the same as the official implementation. We present its performance in Table II as a "lite" SCA solution that can be deployed on a customer's side due to its limited requirement for computation resources and public availability.

CENTRIS (TLSH + SBB). As described in Sec. IV-A, we propose CENTRIS (TLSH + SBB) as a variant of CENTRIS (TLSH) with SBB protected. It has two additional parameters for using the SBB protocol. γ is the mutation bias for the client, and θ is the bucket size returned by the server. In the experiment, we set $\gamma = 50$ and $\theta = 1,000$.

CENTRIS (DPCNN). We enhance the native CENTRIS by replacing TLSH with a well-trained DPCNN model and improve the OSS identification process by assigning weights to the source functions. To train the DPCNN model, we follow the approach of Code2Vec [15] by training the DPCNN model to predict the name of a function with the function's implementation. All source functions collected in our OSS database (see Sec. V-A) are split into training (80%), validation (10%), and testing (10%) data. We clarify that we do not choose widely used CodeBert [28] and Code2Vec due to their inefficiency and limited support of MPC frameworks.

CENTRIS (DPCNN + CrypTen). As introduced in Sec. IV-C, this variant uses CrypTen to encrypt the client's tensors before sending them to the server. It uses the same DPCNN model as CENTRIS (DPCNN). In the experiment, we set up two parties, the client and the server, and we use the official interface of CrypTen to encrypt the DPCNN model.

BinaryAI [11, 37] is a commercialized binary-based SCA tool from the industry. We use its official service in an out-of-the-box setting. BinaryAI takes stripped binary executables (without symbols/debug information) as its inputs, which is reasonable as few closed-source binaries are unstripped.

TABLE II: Precision (*P*), recall (*R*), F1 score (*F1*), and time cost of various SCA tools.

Tool	Type[1]	*P*	*R*	*F1*	Time (s)	Overhead
CENTRIS (TLSH)	S	.555	.592	.573	239.74	-
CENTRIS (TLSH + SBB)	S & SBB	.554	.593	.573	10149.45	4133%
CENTRIS (DPCNN)	S	.838	.769	.802	715.21	198%
CENTRIS (DPCNN + CrypTen)	S & MPC	.838	.769	.802	131332.66	54681%[2]
BinaryAI	B	.881	.477	.619	NA[3]	NA

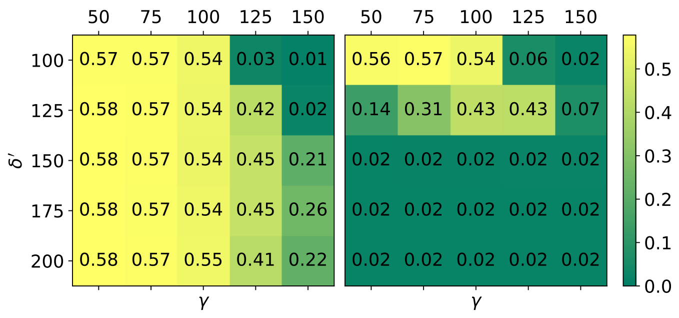
[1] Tools are classified into source-based (S), binary-based (B), SBB-based (SBB), and MPC-based (MPC).
[2] Compared with CENTRIS (DPCNN), the overhead is 18283%, nearly 184× slower.
[3] We are unable to measure the time cost of BinaryAI via its official web service.

D. Performance on SCA

Evaluation Metrics. Precision, recall, and F1 scores are used to evaluate the accuracy of SCA tools. Those metrics are widely used in the SCA scenario [37, 70, 84, 92]. Moreover, since privacy protocols like SBB and CrypTen introduce additional overhead, we also measure the time cost of each SCA tool. In the experiment, we use the database constructed in Sec. V-A and the dataset introduced in Sec. V-B for evaluation.

Accuracy. Table II presents the precision, recall, and F1 scores of CENTRIS's variants and BinaryAI. We observe that CENTRIS (TLSH) and CENTRIS (TLSH + SBB) have similar accuracies. A similar phenomenon is observed between CENTRIS (DPCNN) and CENTRIS (DPCNN + CrypTen), denoting that privacy protocols do not undermine the SCA analysis. We also observed that CENTRIS (DPCNN) largely outperforms CENTRIS (TLSH). We attribute the improvement to the weight enhancements in CENTRIS (DPCNN) elaborated in Sec. IV-B. Indeed, CENTRIS (DPCNN) has a similar accuracy to CENTRIS (TLSH) when the weight enhancement is disabled.

By comparing the industrial binary-based SCA tool, BinaryAI, with the source-based CENTRIS (DPCNN), we observe that BinaryAI's precision is surprisingly high, close to CENTRIS (DPCNN). However, the recall is relatively low, leading to a lower F1 score. We noticed that BinaryAI is much more accurate when analyzing binaries with exported symbols, and BinaryAI's OSS database already contains the 14 OSS repositories used as our test dataset. While this indicates certain



(a) Client-side F1 scores. (b) Server-side F1 scores.

Fig. 5: SBB-based CENTRIS's F1 according to γ and δ'.

"unfairness," we are unable to modify BinaryAI as it is close-source. However, as shown in Table II, it still gains relatively poor accuracy compared to analyzing source code.

Overall, CENTRIS (DPCNN) and CENTRIS (DPCNN + CrypTen) achieve the best F1 score among all evaluated SCA tools, indicating that the source-based SCA framework is accurate and effective, and the privacy-preserving protocols introduced in Sec. IV do not notably affect the accuracy.

Time Cost. By comparing the time cost of CENTRIS (TLSH) and CENTRIS (TLSH + SBB), we observe that the time cost is significantly increased by $41\times$ due to the extra computations incurred by SBB. Similarly, the time cost of CENTRIS (DPCNN + CrypTen) is $184\times$ slower than CENTRIS (DPCNN) due to the overhead of CrypTen. Thus, we conclude that the privacy-preserving protocols (i.e., SBB and CrypTen) introduce an enormous overhead to the SCA tools.

On the other hand, by comparing CENTRIS (TLSH) and CENTRIS (DPCNN), we observe that the time cost of CENTRIS (DPCNN) is nearly tripled due to the embedding process. However, CENTRIS (DPCNN) is still efficient enough for large-scale SCA tasks, and more efficient hardware can further reduce the overhead. We view the time cost of using DPCNN as acceptable for SCA tasks.

E. Leakage of SBB-Based SCA

As mentioned in Sec. IV-A, although the client does not send the original hash value p to the server, the mutated hash value p' is still close to the p. Thus, p' may disclose some information about p to the server. In this section, we uncover the subtle privacy leakage of SBB-based SCA, i.e., CENTRIS (TLSH + SBB). The experiment follows the semi-honest threat model, and we assume the customer (holding the client) and vendor (holding the server) follow the SBB protocol faithfully. The vendor stores all mutated function hash values and is curious about the client's SCA report.

Following the workflow of SBB-based SCA shown in Fig. 3, the client first dissects the input (①) and gets the hash value set I (②). During the SBB-based analysis, the client mutates the hash value $h_i \in I$ with the mutation bias γ to get h'_i, which is sent to the server (③). Thus, the server has access to the set of mutated hash values I', from which the curious vendor can predict the client's SCA report.

Formally, let DB_k be the hash value set of the k-th OSS project, and the function dis evaluates the distance between two hash values. Given the mutated hash values set I', the curious vendor can infer whether the k-th OSS project is reused in the following way. Firstly, the vendor computes a set S'_k with I', DB_k, and a threshold δ' as Eq. 2.

$$S'_k = \{(h'_i, h_j) | \forall (h'_i, h_j) \in I' \times DB_k \text{ and } dis(h'_i, h_j) < \delta'\} \quad (2)$$

Secondly, by iterating all OSS projects and computing $S'_k (k \in [1, n])$, the vendor can rank all OSS projects by the size of S'_k, i.e., $|S'_k|$. A larger $|S'_k|$ denotes a higher probability that the consumer's software reuses k-th OSS project. Thus, the curious vendor can infer the client's privacy. Intuitively, a larger mutation bias γ increases the uncertainty of the vendor's inference. Fig. 5a presents the relations between mutation bias (γ), vendor's threshold (δ'), and F1 scores of a client. Fig. 5b illustrates the predicted F1 scores of a server. Visually, the yellow cells in the same positions of Fig. 5(a) and Fig. 5(b) indicate that the client's and server's F1 scores are close and high, demonstrating the vendor can predict the SCA report, thereby leaking the OSS projects reused by the client.

When the mutation bias γ is relatively small, the SCA service works functionally, and the client has decent F1 scores, while a large γ leads to a poor F1 score for the client. To ensure the client's accuracy and privacy, the client's γ should be small (i.e., yellow cells in Fig. 5a), and the vendor's threshold δ' should be large (i.e., green cells in Fig. 5b). However, δ' is selected by the vendor. From the vendor's perspective, it can intentionally use a relatively small δ' (e.g., 100 or 125) for prediction. Thus, the server-side F1 score is close to the client-side F1 score, which means the vendor could infer the SCA results as well as the client, posing a serious threat to the customer privacy protection of SBB-based SCA.

Overall, by observing the different privacy guarantees and accuracies between CENTRIS (TLSH + SBB) and CENTRIS (DPCNN + CrypTen) in Table II, we believe that MPC-based SCA is more suitable for privacy-preserving SCA. Nevertheless, the MPC-based SCA incurs a high overhead, which we will address in Sec. VI.

VI. OPTIMIZING MPC-BASED SCA

The promising result in Sec. V-D of the MPC-based SCA framework in accuracy and principled privacy guarantee motivates us to build a practical privacy-preserving SCA framework based on MPC. However, the MPC-based SCA framework comes with a high overhead. This section optimizes the standard setup of MPC-based SCA by reducing its cost. The insight of this optimization is selecting only necessary data with software analysis approaches for expensive encrypted computation; thus, we significantly reduce the volume of data required for server queries and achieve a reasonable overhead with high accuracy and privacy guarantee.

Assumption. We assume the customers of SAFESCA are developers, and the client has access to the *source code* and the compiled *binary executables* with debug information. The assumption is reasonable due to the following reasons: (1) Since the MPC-based SCA does not disclose any clients' information, utilizing both the source and binary code does not harm clients' privacy. (2) Consequently, SAFESCA can access the developing environment safely; hence, the developers can easily provide

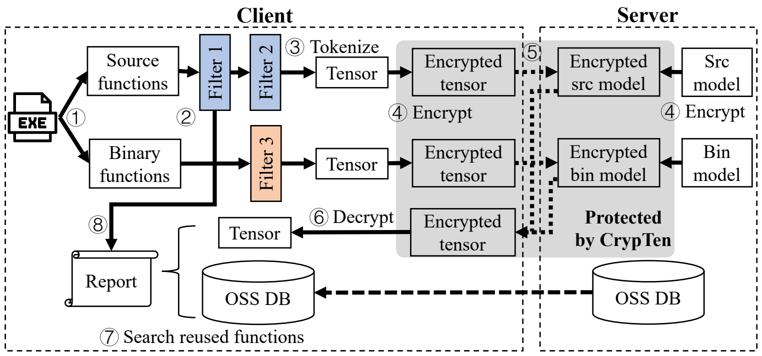


Fig. 6: Workflow of SAFESCA.

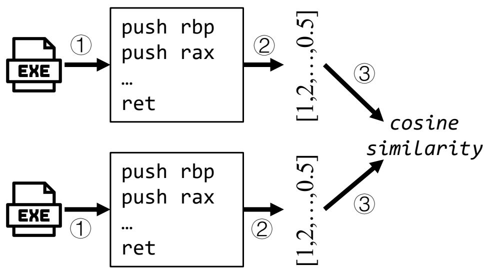


Fig. 7: A common pipeline of binary code similarity analysis (BCSA).

SAFESCA with the source code and the building products (i.e., binary executables). (3) Building products with debug information for testing is a common practice [95], and symbols are usually available. As a result, by analyzing both the source and binary code, SAFESCA can leverage the source code to achieve better accuracy and identify the reused components without source code, e.g., statically linked libraries.

Workflow. Fig. 6 presents the workflow of SAFESCA. Like CENTRIS (DPCNN + CrypTen), a client needs to download the OSS DB first. Given the input binary with debug information, we preprocess it to extract functions from the source codebase, i.e., "source functions", and functions reused without source code, i.e., "binary functions" (①). Then, the functions are filtered and selected for expensive encrypted computation (②). The selected source functions are processed by the source embedding model, which is the same as the process of CENTRIS (DPCNN + CrypTen). Similarly, the selected binary functions are processed by the binary embedding model (③-⑥). To generate the SCA report, the client searches for the reused functions in the database (⑦) and combines the reused OSS knowledge detected by the symbol filter (⑧).

Compared with the workflow of CENTRIS (DPCNN + CrypTen) (Fig. 4), the main optimizations are conducted on the client side. SAFESCA's workflow is more complex and involves an extra binary embedding model (Sec. VI-A) and three filters (Sec. VI-C). Besides, we also accelerate the offline signature generation process, as will be discussed in Sec. VI-B.

A. Binary Embedding Model

Similar to embedding source code for analysis, we need to embed binary code pieces into embedding vectors. Fortunately, many existing binary code similarity analysis techniques (BCSA) can produce function-level embeddings to capture the semantics of assembly code [25, 49, 62, 78, 79, 82, 89, 90]. Therefore, we can leverage BCSA in SAFESCA's workflow. Fig. 7 presents a frequently used pipeline of binary similarity analysis frameworks. Given two binary executables, a BCSA tool disassembles binaries to assembly code and produces the embedding vectors of assembly functions. The similarity score is usually computed as the cosine similarity of embeddings.

Following the training pipeline presented by recent research in relevant topics, GEMINI [86] and SAFE [62], we train a DPCNN model to embed binary code. We report that the AUC score, a widely-used metric for binary similarity works [47, 62, 89, 90], of our binary code embedding model is 0.935, which is comparable to GEMINI (0.932) and SAFE (0.990), demonstrating the effectiveness of our setting. We, however, do not use an existing model due to implementation difficulties. Existing binary code embedding techniques often utilize graph structures [18, 25, 27, 75, 86, 89, 96] and recurrent neural networks [62] to embed assembly code. However, we find that few (if at all any) existing crypto protocols support those models due to computation complexity.

B. Improvements in Signature Generation

Existing works (e.g., CENTRIS and TPLite) treat each `tag` of a Git repository as a unique version. Given a repository's specific version, they analyze its code and generate signatures with extracted features (e.g., strings, magic numbers, functions, and file hashes) for the given version. However, since real-world OSS developers may not treat a tag as a release version, their approach is impractical for our complex OSS database due to the huge cost. For instance, `ClickHouse` [22] generates a new tag every week, resulting in thousands of versions; consequently, generating signatures for `ClickHouse` can take more than four months, per our estimation.

As a practical setting, we build signatures of a repository by commits. Given a repository, after generating the signature for the first version, we focus on the files changed by the subsequent commits, whereas the unchanged files will not be analyzed again. This way, the OSS signature generation process is much more efficient. Following existing works (CENTRIS and OSSFP [85]), we extract function-level features. Thus, the changes between different commits will include newly created functions, changed functions, and deleted functions. In this study, a function's identifier (i.e., ID) is composed of its name and scope; hence, a newly created ID denotes a new function. Functions with the same ID but different implementations between two commits are changed functions. A missing ID denotes a deleted function. Moreover, we also record the direct call relations between two functions, which results in a large call graph for each repository.

C. Client-Side Filters

As shown in Fig. 6, after preprocessing an input binary and extracting its source and binary functions, the client will tokenize those functions for expensive encrypted computation. To reduce the overhead incurred by crypto, we propose three filters to identify suspect functions on the client side before tokenization and analysis. Specifically, the symbol filter first conservatively pinpoints functions copied from OSS projects; such identified functions do not go through subsequent SCA analysis. For the remaining functions that are not identified, the source function filter and assembly function filter are employed to pick representative functions to be tokenized and embedded for further SCA analysis.

Filter 1: Symbol Filter. SAFESCA's client can safely employ sensitive information, including symbols since all data is encrypted before being sent to the server. Therefore, given the symbols provided by customers, a straightforward idea is to pinpoint functions that are likely copied from existing OSS projects by matching their symbols, such as function names and scopes. To do so, we also collect names and scopes (usually namespace names) of functions while building the OSS database. Ideally, given an input function, if it has identical symbols with one record in the OSS database, it is likely to be a reused function, and we forward it to further analysis.

However, directly collecting accurate symbol names (including scope names) from the source code is difficult due to the complexity of C/C++ projects. Strictly matching symbols (i.e., names and scopes) will likely lead to an extensive amount of false negatives (FNs). Thus, we add a more flexible rule as a remedy for symbol matching to reduce FNs. Given that matches of complex (e.g., over 20 characters) and unique function names are usually strong indicators of direct function reuses, we view a function as copied from OSS projects if it has a complex name and is matched to record in the database. Besides, to increase the accuracy of name matching, we further check the function call relationship between potentially matched functions. A function is identified as reused from an OSS project only if the function and its matched function have the same call relationship, i.e., both have the same edge in the call graph.

Our evaluation shows that the proposed symbol filtering method can reach an F1 score of 0.547. Some reused functions cannot be recognized with symbol matching because of modifications in function names and implementations during code reusing. In order to locate the reused components that are not matched in this step, we further involve source- and binary-based embedding models to analyze the remaining functions.

Filter 2: Informative Source Function Filter. When the source code is available, we tokenize and embed source functions for further similarity-based SCA. Before that, the informative source filter is used to determine and skip inconsequential functions. One insight behind this filter is that some trivial functions hardly contribute to the SCA results. Thus, we can only consider complex and representative functions for time-consuming analysis.

Given functions that are not matched by the symbol filter, we select those that are informative and embed their source code for SCA analysis. Specifically, we use the maintainability index [63, 85] to decide if a function is "informative." In short, this metric is negatively correlated to the cyclomatic complexity [69], Halstead volume [32], and the lines of code in a function. We configure our tool to keep a proportion of functions of each source file with the least maintainability indexes for further analysis. For simplicity, we define the proportion of kept functions as the hyperparameter θ_1.

Filter 3: Assembly Function Filter. Similar to Filter 2, when a binary is provided for SCA, we first disassemble it into assembly code and then identify representative functions for embedding and analysis. However, binary code analysis is less accurate than source code analysis in general. Therefore, the assembly function filter more strictly restricts the conditions for picking a function. It selects functions whose names exist in the collected OSS projects. Also, we avoid selecting overly long functions due to the limitation of existing binary code similarity analysis techniques on lengthy input [36, 51, 61, 62]. In particular, to evaluate if an assembly function should be selected, we use the following equation,

$$score = 1/(ln(OSS\_num + 2) * ln(block\_num + 2)) \quad (3)$$

where OSS_num is the number of OSS projects containing the function's name. $block\_num$ is the number of basic blocks in the function. We add 2 to the denominator to avoid zero division. We also define another hyperparameter θ_2 to decide the proportion of functions for embedding and SCA analysis.

D. *Privacy Analysis of* SAFESCA

Similar to CENTRIS (DPCNN + CrypTen), SAFESCA protects the client's privacy with the MPC protocol. The client's valuable assets (e.g., source and binary code) are not leaked to the server. From the server's perspective, the client downloads a symbol database to enable the symbol filter (Filter 1), and the OSS DB also contains the binary embeddings of OSS projects. However, the symbol database only contains function names and namespaces. We thus argue that the symbol database is not a highly valuable asset since it can be easily obtained by parsing an OSS project. Additionally, merely using the symbol database cannot achieve a high-quality SCA result (i.e., 0.547 F1 score as evaluated in Sec. VII-B). Regarding the OSS DB, the binary embeddings are used the same way as the source embeddings; therefore, the OSS DB does not leak any additional information since the client has no access to the code embedding model.

VII. EVALUATION

To evaluate SAFESCA, we reuse our large-scale OSS database described in Sec. V-A and the manually-built evaluation dataset (Sec. V-B). We also reuse the source-based OSS scoring method and source code embedding model of CENTRIS (DPCNN) (see Sec. V-D) for SAFESCA. Sec. VII-A shows the impact of threshold value β on SAFESCA's performance. To ensure the efficiency of SAFESCA, we use a small threshold $\theta_1 = 0.02$ for the informative source function filter, and $\theta_2 = 0.02$ for the assembly function filter. Due to the space limit, we present a detailed analysis of the impact of θ_1 and θ_2 on SAFESCA's performance on our website [5].

A. *Performance of* SAFESCA

Fig. 8 presents the precision, recall, and F1 score of SAFESCA with different β values. SAFESCA's performance is not sensitive to β's changes, especially the F1 score (i.e., green line). When β is greater than 5, SAFESCA's F1 score

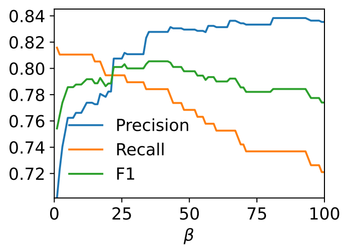


Fig. 8: Precision, recall, and F1 score of SAFESCA with different values of β.

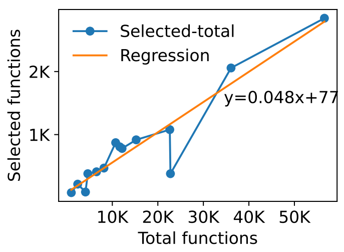


Fig. 9: Number of selected functions of SAFESCA to total functions of each binary.

is greater than 0.78. The highest F1 score is 0.81 when β is 40. Compared with CENTRIS (DPCNN), SAFESCA achieves a similar F1 score, which demonstrates the effectiveness of our proposed three filters. Comparing SAFESCA with and without optimizations (see Table III), we observe that the proposed optimizations do not sacrifice accuracy.

Fig. 9 presents the total number of input functions and the number of functions SAFESCA selected to perform encrypted computation. The regression line (i.e., orange line) has a 0.048 gradient, indicating that SAFESCA selects less than 5% of functions to generate embeddings for SCA. As shown in Table III, compared with SAFESCA without optimization (i.e., all functions are passed to the embedding models), three filters reduce the time cost to 8.47%, i.e., 11.8× faster. Compared with naively using MPC protocol in the SCA framework (184× overhead), SAFESCA's overhead is merely 23×. In total, SAFESCA takes 4.6 hours to identify reused OSS projects of our dataset (i.e., 20 minutes per binary on average), demonstrating the scalability of SAFESCA. Our dataset's largest binary, `mod_pagespeed` [3], whose repository reaches 543 MB without history files, takes 72 minutes for SAFESCA.

B. Impacts of Filters

As we design three filters to accelerate SAFESCA and report the effects of enabling all of them, we further analyze the impact of each filter separately. The overhead of each filter is negligible compared with computing the embedding vectors; thus, we focus on analyzing the accuracy impacts of each filter.

Filter 1. After extracting functions and their symbols from an input software, we first analyze the software with the symbol filter, which is expected to have high precision since we only consider representative symbols and require their direct function calls to be matched. In other words, the symbol filter is conservative and only rules out functions that can be skipped with high confidence. Our experiments show that applying the symbol filter alone (i.e., ①②⑧ of Fig. 6 only; no functions are analyzed by Filter 2 and Filter 3) achieves 0.918 precision but 0.4 recall, resulting in a 0.547 F1 score. Below, we show that the following two filters can further improve the SCA accuracy based on the results of the symbol filter.

Filter 2. After Filter 1, the remaining source functions are sent to Filter 2 for further analysis with the source embedding model. To evaluate the effect of Filter 2, Filter 3 and the following binary embedding model are not used. We report that SCA's precision and recall are 0.829 and 0.747 when $\beta = 40$. Similar to the conclusion in Sec. VII-A, changing β has little impact on the F1 score at this step. This filter can effectively improve the recall (from 0.40 to 0.747), achieving similar accuracy to analyzing all source functions (i.e., CENTRIS (DPCNN)).

TABLE III: Performance of SAFESCA with different settings.

	P	*R*	*F1*	Time cost (s)	Cost ratio
SAFESCA (w/o opt)[1]	.802	.780	.791	195125.06	100%
SAFESCA	.828	.791	.809	16527.3	8.47%
CENTRIS (DPCNN)[2]	.838	.769	.802	715.2	0.37%

[1] SAFESCA (w/o opt) denotes SAFESCA without three filters.
[2] To ease comparison with Table II, we present CENTRIS (DPCNN).

Filter 3. This filter applies to disassembled assembly functions when binary is provided as input. Functions selected by this filter will be embedded into vectors using BCSA techniques. A function is identified as reused from an OSS project when its matched symbol and the most similar embedding vector are both from that OSS repository. This requirement facilitates SAFESCA in eliminating false positives caused by assembly function embeddings. Our evaluation reveals that it improves the recall from 0.747 to 0.791. While the improvement is limited due to the difficulty of analyzing binary code, we view it as a critical replenishment when binary is the only input.

VIII. RELATED WORK

Recent SCA works target different programming languages and different types of software. However, these tools do not naturally offer a privacy-preserving solution. Conventional binary-based SCA tools [26, 33, 38] primarily rely on string-level signatures. B2SFinder [91] additionally uses `if/else` and `switch` structures. However, these signatures may not exist in some OSS projects (e.g., RapidJson [4]) and are not robust to code changes. Recent binary-based SCA [11, 70, 87] also explores using embedding techniques, which are often limited by the hurdles of reverse engineering. Source-based SCA tools [68, 84, 85] for C/C++ software usually rely on code-block- and function-level signatures. [68] uses code-block-level signatures and is relatively time-consuming. Recent works use function-level signatures. CENTRIS [84] uses code segmentation to reduce FPs and leverages redundancy elimination to reduce the size of the OSS database. OSSFP [85] employs the maintainability index to reduce the OSS database's size and improve SCA scalability. Our work also uses the maintainability index to reduce the expensive encrypted computation.

IX. CONCLUSION

We have presented the first study on privacy-preserving SCA. We review SCA privacy requirements and depict viable technical solutions with varying privacy gains and overheads. We also empirically benchmark the proposed solutions and discuss issues in privacy leakage and cost. Accordingly, we optimize the MPC-based SCA by reducing its overhead significantly without sacrificing the privacy guarantee or accuracy. This work provides guides for researchers and users who aim to use and improve SCA in practice.

X. ACKNOWLEDGMENTS

This work was supported in part by CCF-Tencent Open Research Fund. We are grateful to the anonymous reviewers for their valuable comments.

References

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

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

[3] "mod_pagespeed," https://github.com/apache/incubator-pagespeed-mod.

[4] "RapidJSON," https://github.com/Tencent/rapidjson/.

[5] "SafeSCA," https://sites.google.com/view/safesca.

[6] "Snyk," https://snyk.io/what-is-snyk/.

[7] "Whitesource," https://www.whitesourcesoftware.com/product-overview/.

[8] "Black Duck Binary Analysis," https://community.synopsys.com/s/black-duck-binary-analysis, 2023.

[9] "Centris," https://github.com/wooseunghoon/Centris-public, 2023.

[10] "SDL," https://github.com/libsdl-org/SDL, 2023.

[11] "Tencent BinaryAI," https://www.binaryai.cn/, 2023.

[12] "yuv2rgb," https://github.com/descampsa/yuv2rgb, 2023.

[13] "Private server lawsuit of maplestory," https://maplenewsnetwork.wordpress.com/2012/04/24/nexon-awarded-3-6-million-in-private-server-lawsuit/, 2024.

[14] A. C. D. Agency, "Software bill of materials," https://www.cisa.gov/sbom, 2024.

[15] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2Vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019.

[16] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *ACM CCS*, 2016, pp. 356–367.

[17] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 189–200.

[18] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," ser. NIPS, 2018.

[19] R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias, "Semi-homomorphic encryption and multiparty computation," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2011, pp. 169–188.

[20] Y. Chen, A. E. Santosa, A. Sharma, and D. Lo, "Automated identification of libraries from vulnerability data," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, 2020, pp. 90–99.

[21] Y. Chen, A. E. Santosa, A. M. Yi, A. Sharma, A. Sharma, and D. Lo, "A machine learning approach for vulnerability curation," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 32–42.

[22] ClickHouse, "Clickhouse," https://clickhouse.com/, 2023.

[23] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks," in *12th USENIX Security Symposium (USENIX Security 03)*, 2003.

[24] I. Damgård, V. Pastro, N. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *Annual Cryptology Conference*. Springer, 2012, pp. 643–662.

[25] 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.

[26] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "Identifying open-source license violation and 1-day security risk at large scale," in *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*, 2017, pp. 2169–2185.

[27] Y. Duan, X. Li, J. Wang, and H. Yin, "DEEPBINDIFF: Learning program-wide code representations for binary diffing," 2020.

[28] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[29] M. A. Q. Flooding-Based, "Monitoring-based differential privacy mechanism against query flooding-based model extraction attack," 2021.

[30] M. . GitHub, "CodeQL," https://codeql.github.com/, 2021.

[31] O. Goldreich, *Foundations of Cryptography, Volume 2*. Cambridge university press Cambridge, 2004.

[32] T. Hariprasad, G. Vidhyagaran, K. Seenu, and C. Thirumalai, "Software complexity analysis using halstead metrics," in *International Conference on Trends in Electronics and Informatics (ICEI)*, 2017, pp. 1109–1113.

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

[34] J. Henkel, S. K. Lahiri, B. Liblit, and T. Reps, "Code vectors: understanding programs through embedded abstracted symbolic traces," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 163–174.

[35] Y. Hua, A. Namavari, K. Cheng, M. Naaman, and T. Ristenpart, "Increasing adversarial uncertainty to scale private similarity testing," in *31st USENIX Security Symposium*, 2022, pp. 1777–1794.

[36] A. Jia, M. Fan, W. Jin, X. Xu, Z. Zhou, Q. Tang, S. Nie, S. Wu, and T. Liu, "1-to-1 or 1-to-n? investigating the effect of function inlining on binary similarity analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, pp. 1–26, 2023.

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

[38] L. Jiang, H. Yuan, Q. Tang, S. Nie, S. Wu, and Y. Zhang, "Third-party library dependency for large-scale sca in the c/c++ ecosystem: How far are we?" in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1383–1395.

[39] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2019.

[40] R. Johnson and T. Zhang, "Deep pyramid convolutional neural networks for text categorization," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2017, pp. 562–570.

[41] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM: Software protection for the masses," ser. SPRO, 2015.

[42] M. Juuti, S. Szyller, S. Marchal, and N. Asokan, "Prada: protecting against dnn model stealing attacks," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 512–527.

[43] M. Kesarwani, B. Mukhoty, V. Arya, and S. Mehta, "Model extraction warning in mlaas paradigm," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 371–380.

[44] B. Knott, S. Venkataraman, A. Hannun, S. Sengupta, M. Ibrahim, and L. van der Maaten, "Crypten: Secure multi-party computation meets machine learning," in *arXiv 2109.00984*, 2021.

[45] M. Li, P. Wang, W. Wang, S. Wang, D. Wu, J. Liu, R. Xue, W. Huo, and W. Zou, "Large-scale third-party library detection in android markets," *IEEE Transactions on Software Engineering*, vol. 46, no. 9, pp. 981–1003, 2018.

[46] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "Libd: Scalable and precise third-party library detection in android markets," in *2017 IEEE/ACM 39th International Conference on Software Engineering*, 2017, pp. 335–346.

[47] X. Li, Y. Qu, and H. Yin, "Palmtree: Learning an assembly language model for instruction embedding," in *ACM CCS*, 2021, pp. 3236–3251.

[48] Y. Li, L. Zhu, X. Jia, Y. Jiang, S.-T. Xia, and X. Cao, "Defending against model stealing via verifying embedded external features," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 36, no. 2, 2022, pp. 1464–1472.

[49] Z. Li, P. Ma, H. Wang, S. Wang, Q. Tang, S. Nie, and S. Wu, "Unleashing the power of compiler intermediate representation to enhance neural program embeddings," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2253–2265.

[50] Y. Lindell, "How to simulate it–a tutorial on the simulation proof technique," *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*, pp. 277–346, 2017.

[51] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αdiff: cross-version binary code similarity detection with dnn," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 667–678.

[52] Z. Liu and S. Wang, "How far we have come: Testing decompilation correctness of c decompilers," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 475–487.

[53] Z. Liu, Y. Yuan, S. Wang, and Y. Bao, "Sok: Demystifying binary lifters through the lens of downstream applications," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 1100–1119.

[54] Z. Liu, Y. Yuan, S. Wang, X. Xie, and L. Ma, "Decompiling x86 deep neural network executables," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 7357–7374.

[55] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 389–400.

[56] ——, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection," *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1157–1177, 2017.

[57] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: fast and accurate detection of third-party libraries in android apps," in *Proceedings of the 38th international conference on software engineering companion*, 2016, pp. 653–656.

[58] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 42, no. 4, pp. 824–836, 2018.

[59] P. Manadhata and J. M. Wing, *Measuring a system's attack surface*. School of Computer Science, Carnegie Mellon University Pittsburgh, PA, USA, 2004.

[60] P. K. Manadhata and J. M. Wing, "An attack surface metric," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 371–386, 2010.

[61] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, "How machine learning is solving the binary function similarity problem," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2099–2116.

[62] L. Massarelli, G. A. D. Luna, F. Petroni, R. Baldoni, and L. Querzoni, "Safe: Self-attentive function embeddings for binary similarity," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2019, pp. 309–329.

[63] Microsoft, "Code metrics - maintainability index range and meaning," https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning, 2023.

[64] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[65] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[66] J. Oliver, C. Cheng, and Y. Chen, "Tlsh–a locality sensitive hash," in *2013 Fourth Cybercrime and Trustworthy Computing Workshop*, 2013, pp. 7–13.

[67] A. Praseed and P. S. Thilagam, "Ddos attacks at the application layer: Challenges and research perspectives for safeguarding web applications," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 661–685, 2018.

[68] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 1157–1168.

[69] M. Shepperd, "A critique of cyclomatic complexity as a software metric," *Software Engineering Journal*, vol. 3, no. 2, pp. 30–36, 1988.

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

[71] W. Tang, Z. Xu, C. Liu, J. Wu, S. Yang, Y. Li, P. Luo, and Y. Liu, "Towards understanding third-party library dependency in c/c++ ecosystem," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[72] T. J. Team, "Joern," https://joern.io, 2021.

[73] The Apache Software Foundation, "Apache log4j," https://logging.apache.org/log4j/2.x/, 2024.

[74] C. Wang, W. Wang, P. Yao, Q. Shi, J. Zhou, X. Xiao, and C. Zhang, "Anchor: Fast and precise value-flow analysis for containers via memory orientation," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 3, pp. 66:1–66:39, 2023.

[75] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, "jtrans: Jump-aware transformer for binary code similarity," *arXiv preprint arXiv:2205.12713*, 2022.
the 2015 International Symposium on Software Testing and Analysis, 2015, pp. 71–82.

[76] H. Wang, Y. Guo, Z. Ma, and X. Chen, "Wukong: A scalable and accurate two-phase approach to android app clone detection," in *Proceedings of*

[77] H. Wang, Z. Liu, S. Wang, Y. Wang, Q. Tang, S. Nie, and S. Wu, "Are we there yet? filling the gap between binary similarity analysis and binary software composition analysis," in *2024 IEEE 9th European Symposium on Security and Privacy (Euro S&P)*, 2024, pp. 506–523.

[78] H. Wang, P. Ma, S. Wang, Q. Tang, S. Nie, and S. Wu, "sem2vec: Semantics-aware assembly tracelet embedding," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, pp. 1–34, 2023.

[79] H. Wang, P. Ma, Y. Yuan, Z. Liu, S. Wang, Q. Tang, S. Nie, and S. Wu, "Enhancing DNN-based binary code function search with low-cost equivalence checking," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 226–250, 2022.

[80] H. Wang, S. Wang, D. Xu, X. Zhang, and X. Liu, "Generating effective software obfuscation sequences with reinforcement learning," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1900–1917, 2020.

[81] Z. Wang, P. Ma, H. Wang, and S. Wang, "PP-CSA: Practical privacy-preserving software call stack analysis," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 1264–1293, 2024.

[82] W. K. Wong, H. Wang, Z. Li, and S. Wang, "BinAug: Enhancing binary similarity analysis with low-cost input repairing," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2024, pp. 1–13.

[83] W. K. Wong, H. Wang, P. Ma, S. Wang, M. Jiang, T. Y. Chen, Q. Tang, S. Nie, and S. Wu, "Deceiving deep neural networks-based binary code matching with adversarial programs," in *2022 IEEE International Conference on Software Maintenance and Evolution*, 2022, pp. 117–128.

[84] S. Woo, S. Park, S. Kim, H. Lee, and H. Oh, "Centris: A precise and scalable approach for identifying modified open-source software reuse," in *2021 IEEE/ACM 43rd International Conference on Software Engineering*, 2021, pp. 860–872.

[85] J. Wu, Z. Xu, W. Tang, L. Zhang, Y. Wu, C. Liu, K. Sun, L. Zhao, and Y. Liu, "Ossfp: Precise and scalable c/c++ third-party library detection using fingerprinting functions," in *2023 IEEE/ACM 45th International Conference on Software Engineering*, 2023, pp. 270–282.

[86] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *CCS*, 2017.

[87] C. Yang, Z. Xu, H. Chen, Y. Liu, X. Gong, and B. Liu, "Modx: binary level partially imported third-party library detection via program modularization and semantic matching," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1393–1405.

[88] A. C.-C. Yao, "How to generate and exchange secrets," in *27th annual symposium on foundations of computer science*, 1986, pp. 162–167.

[89] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 01, 2020, pp. 1145–1152.

[90] Z. Yu, W. Zheng, J. Wang, Q. Tang, S. Nie, and S. Wu, "Codecmr: Cross-modal retrieval for function-level binary source code matching," *Advances in Neural Information Processing Systems*, vol. 33, pp. 3872–3883, 2020.

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

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

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

[94] Y. Zhang, J. Dai, X. Zhang, S. Huang, Z. Yang, M. Yang, and H. 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)*, 2018, pp. 141–152.

[95] A. Zhou, C. Ye, H. Huang, Y. Cai, and C. Zhang, "Plankton: Reconciling binary code and debug information," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 912–928.

[96] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *NDSS*, 2019.