CENTRIS: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse

Abstract—**Open-source software (OSS) is widely reused as it provides convenience and efficiency in software development. Despite evident benefits, unmanaged OSS components can introduce threats, such as vulnerability propagation and license violation. Unfortunately, however, identifying reused OSS components is a challenge as the reused OSS is predominantly modified and nested. In this paper, we propose CENTRIS, a precise and scalable mechanism for identifying modified OSS reuse. By segmenting an OSS code base and detecting the reuse of a unique part of the OSS only, CENTRIS is capable of precisely identifying modified OSS reuse in the presence of nested OSS components. For scalability, CENTRIS eliminates redundant code comparisons and accelerates the search using hash functions. When we applied CENTRIS on 10,241 widely-employed GitHub projects, comprising 229,326 versions and 80 billion lines of code, we observed that modified OSS reuse is a norm in software development, occurring 20 times more frequently than exact reuse. Nonetheless, CENTRIS identified reused OSS components with 91% precision and 94% recall in less than a minute per application on average, whereas a recent clone detection technique, which does not take into account modified and nested OSS reuse, hardly reached 10% precision and 40% recall.**

Index Terms—**Open-Source Software, Software Composition Analysis, Software Security**

I. INTRODUCTION

Recent years have seen a dramatic surge in the number and use of open-source software (OSS) [1]–[4]. Not to mention the immediate benefit of reusing the functionalities of existing OSS projects, using OSS in software development generally leads to improved reliability because OSS is publicly scrutinized by multiple parties. At the same time, however, reusing OSS without proper management can impair the maintainability and security of software [5]–[7], especially when a piece of code is reused over various projects.

One effective solution to prevent this undesirable situation is to undertake *software composition analysis* (SCA) [7]–[9]. The aim of SCA process is to identify the OSS components contained in a target program. With an SCA tool, developers can systematically keep track of what and how OSS components are reused in their software, and can therefore mitigate security threats (by patching known vulnerabilities) and avoid potential license violations.

Unfortunately, precisely identifying OSS components in the target software is becoming increasingly challenging, mainly owing to the following recent trends in software development practice regarding OSS:

1) **Modified OSS reuse:** Instead of reusing existing OSS in its entirety, developers commonly utilize only a portion of it, or modify the source code or structure.
2) **Nested OSS components:** The reused OSS may contain multiple sub-OSS components, and even the sub-OSS components may include other OSS components.
3) **Growth of OSS projects and their code size:** The number of OSS projects is rapidly increasing [4], along with the growing code size [10].

These three factors collectively affect the accuracy and scalability of SCA tools. To our knowledge, no existing techniques are capable of precise and scalable detection of modified OSS reuse in the presence of nested OSS components.

Limitations of existing techniques. Existing SCA techniques [7], [11]–[15] assume that the reused OSS is essentially unmodified (or modified to a limited fashion), thereby producing false negatives when it comes to identifying modified reuse. For example, OSSPolice [7], a recent SCA technique that aims to identify partially reused components, cannot identify OSS components when their directory structures are modified. On the other hand, existing code clone detection techniques [6], [16]–[35] can, in principle, be used for identifying modified reuse of OSS components, but they easily produce false positives if an OSS project is nested. When only a nested third-party software component of an OSS is used in the target software, clone detection techniques falsely report them as reuse of the original OSS. Also, as we demonstrate in this paper, existing SCA and code clone detection techniques are hardly scalable for large OSS code bases.

Our approach. In this paper, we present CENTRIS, a new SCA technique that aims to overcome the above limitations. CENTRIS can effectively detect modified OSS reuse in a precise and scalable manner even when OSS components are arbitrarily nested. CENTRIS uses the recent signature-based, function-level clone detection technique [6]. To identify reused OSS components, we generate signatures of all versions of OSS projects, search for matching signatures in the target software, and subsequently interpret the matched signatures. Given this simple approach, however, we are more concerned with making it precise and scalable.

For scalability, CENTRIS uses a technique called *redundancy elimination*. Instead of generating signatures from all functions in all versions of the entire OSS code base, CENTRIS first collects all functions in all versions of an OSS project, and then removes all redundancies in functions across versions. Here, redundant functions refer to two or more functions having the same source code shared across different versions. This approach is effective in reducing space complexity; most of the time, the delta across versions is significantly smaller than the size of the unchanged code base.

*CENTRIS will be open-sourced with a public web service at the publication time.

For precision, we employ a technique called *code segmentation*. To identify modified components, we basically use loose matching that checks whether the code similarity between the target software and the OSS is greater than a predefined threshold. As we previously mentioned, however, simply applying this method suffers from false alarms especially when an OSS is nested. Therefore, we segment an OSS into the *application* code and the *borrowed* code; we analyze whether each function in the OSS belongs to the borrowed code (*e.g.*, a part of the nested third-party software) or the application code (*i.e.*, a unique part of the OSS). We remove the borrowed code of an OSS and only analyze the reuse patterns of the application code of the OSS for component identification. This code segmentation enables CENTRIS to drastically filter out false alarms while still identifying desired OSS components even when they are heavily modified or nested.

Evaluation. For the experiment, we first collected a large OSS dataset from 10,241 public C/C++ repositories on GitHub comprising 229,326 different versions and 80 billion lines of code (LoC) in total (our dataset is 25 times larger than that used in a related approach [7]). From a cross-comparison experiment, we discovered that 95% of the detected components were reused with modifications. Nevertheless, CENTRIS successfully identified the reused OSS components with 91% precision and 94% recall, whereas a recent clone detection technique, DéjàVu [29], yielded less than 10% precision and at most 40% recall because DéjàVu neither identifies heavily modified components nor filters out false alarms caused by nested components (see Section V-B). Furthermore, CENTRIS reduced the matching time to tens of seconds when comparing a software with one million LoC to the dataset with 80 billion LoC, while DéjàVu requires more than three weeks (see Section V-C) because they perform matching against all lines of code from all versions of every OSS in the dataset.

Contributions. This paper makes the following contributions:

- We propose CENTRIS, the first approach capable of precisely and scalably identifying modified OSS reuse in the presence of nested OSS components. The key enabling technical contributions include redundancy elimination and code segmentation.
- We applied CENTRIS in an industrial setting with a large OSS dataset. As a result, we confirmed that most (95%) of the OSS components are reused with modification.
- CENTRIS can identify reused OSS components from 10K widely-utilized software projects on GitHub with 91% precision and 94% recall, even though modified OSS reuse is prominent. CENTRIS takes less than a minute on average to identify components in a software project.

II. TERMINOLOGY AND MOTIVATION

A. Terminology

Basic terms. We first define a few terms upfront. *Target software* denotes the software from which we want to identify reused OSS components. An *OSS component* refers to an entire OSS package or sometimes the functions contained in the OSS; called component for short. Lastly, *OSS reuse* refers to utilizing all or some of the OSS functionalities [36], [37].

TABLE I: Examples of identified components in ArangoDB using CENTRIS.

Name	Version	#Reused functions		#Unused functions	Structure change	Reuse patterns†
		Identical	Modified			
Curl	v7.50.3	2,211	26	1	X	P & CC
GoogleTest	v1.7.0	1,197	11	33	O	P & SC & CC
Asio	v1.61.0	941	0	0	X	E
Velocypack	OLD‡	134	0	3,765	X	P
TZ	v2014b	89	0	26	X	P

†E: Exact reuse, P: Partial reuse, SC: Structure-changed reuse, CC: Code-changed reuse
‡OLD version: Velocypack code committed in 2016.

A software project. We define a software project as the combined set of *application* and *borrowed* codes. The borrowed code denotes the part comprised of reused OSS, *i.e.*, a set of third-party software, which we aim to identify within the target software. The application code refers to the original part of the software project excluding the code from another OSS.

OSS reuse patterns. We classify OSS reuse patterns into four categories according to the code and structural changes:

1) **Exact reuse (E):** The case where the entire OSS is reused in the target software without any modification.
2) **Partial reuse (P):** The case where only some parts of an OSS are reused in the target software.
3) **Structure-changed reuse (SC):** The case where an OSS is reused in the target software with structural changes, *i.e.*, the name or location of an original file or directory is changed, such as code amalgamation.
4) **Code-changed reuse (CC):** The case where an OSS is reused with source code changes.

When an OSS is reused with modification (*i.e.*, partial, structure-changed, and code-changed reuse), we refer to this as *modified OSS reuse*. In the modified reuse, P, SC, and CC can occur simultaneously.

B. Motivating example

Suppose we want to identify OSS components reused in ArangoDB v3.1.0 (3.5 million LoC), a native multi-model database software[1]. Given a large OSS dataset (80 billion LoC), CENTRIS took less than a minute and identified a total of 29 C/C++ OSS components in ArangoDB. Table I elaborates on five of the identified OSS components.

The modified reuse pattern is very prominent in ArangoDB. Among the 29 identified OSS components, 22 were modified, wherein the reused functions were located in directories different from those in the original OSS, *e.g.*, GoogleTest, or the code base was partially updated, *e.g.*, Curl. Also in most cases, ArangoDB reused a fraction of the OSS code base, *e.g.*, 3.6% of Velocypack, with unnecessary features such as testing infrastructure removed. Moreover, 21 components were reused in the form of nested components; for example, TZ was reused by the V8 engine, and V8 was in turn reused in ArangoDB.

[1]https://github.com/arangodb/arangodb

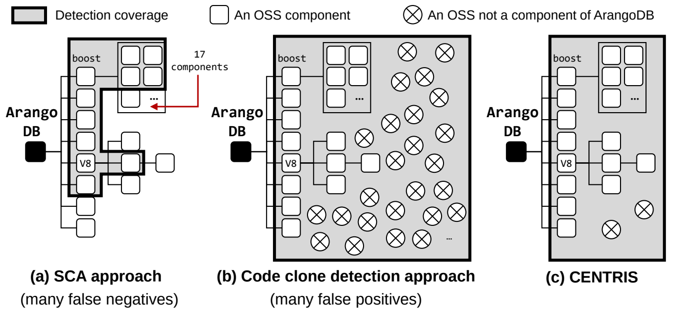


Fig. 1: Illustration of the component detection coverage of the SCA approach, code clone detection approach, and CENTRIS. Compared to CENTRIS, which identified 29 components, existing SCA approaches could not detect components where structural modification occurs (*e.g.*, OSSPolice [7]) or when only a small portion of an OSS code base is reused, whereas code clone detection approaches (*e.g.*, DéjàVu [29]) reported numerous false positives.

Existing SCA techniques are not designed for handling such code bases with modified components. For example, six components were reused in ArangoDB with structural changes, as OSSPolice [7] relies on the original OSS structure for component detection, it fails to identify such structure-changed components. In contrast, code clone detection techniques report numerous false alarms in identifying modified and nested components. For instance, DéjàVu [29] reported that 422 OSS were reused in ArangoDB, among which 411 were confirmed as false alarms as we investigated (see Section V-B). This is because DéjàVu reports any OSS as a reused component if the OSS contains the same third-party software reused in ArangoDB. One example is Ripple, a cryptocurrency-related OSS that contains RocksDB as a sub-component. ArangoDB also reuses multiple functions from RocksDB, thereby having shared functions with Ripple, and DéjàVu misinterprets this relation as ArangoDB reusing Ripple.

III. DESIGN OF CENTRIS

In this section, we describe the design of CENTRIS.

A. Overview

Figure 2 depicts the workflow of CENTRIS. CENTRIS comprises two phases: (1) **P1** for constructing the OSS component database (DB), and (2) **P2** for identifying OSS components reused in the target software. In **P1**, we use a technique, called redundancy elimination, which enables scalable component identification; CENTRIS reduces the space complexity in component identification by eliminating redundancies across the versions of each OSS project. All functions of an OSS project are converted into the OSS signature, which is a set of functions without redundancies, and subsequently stored in the component DB. In **P2**, we use a technique, code segmentation, for precise component detection. Specifically, CENTRIS minimizes false alarms in component detection by only analyzing the patterns wherein the application code of an OSS is reused in the target software.

Design assumptions. CENTRIS is designed to identify OSS components at the source code level; that is, our goal is to

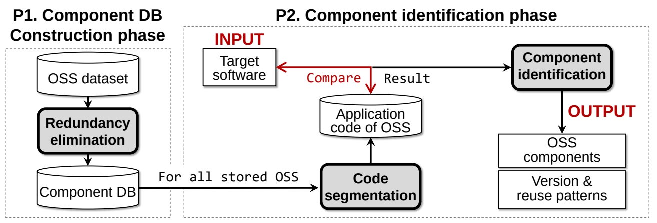


Fig. 2: High-level overview of the workflow of CENTRIS.

identify components regardless of whether all or only parts of the OSS code base are reused in the target software. In addition, although the concept of CENTRIS is applicable to any granularity of component units, we focus on the *function* units for the mechanism design and evaluation. As the term "OSS reuse" refers to utilizing all or some OSS functionalities [7], [36], [37], we determined that function units are more appropriate for detecting various OSS reuse patterns compared to other units (the benefits of function units have been discussed in previous studies [6], [7], [27], [38]). In light of this, CENTRIS extracts functions from all versions of the OSS in our dataset using a function parser (see Section IV), and performs lightweight text preprocessing to normalize the function by removing comments, tabs, linefeed, and whitespaces, which are easy to change but do not affect program semantics.

B. Component DB construction phase (P1)

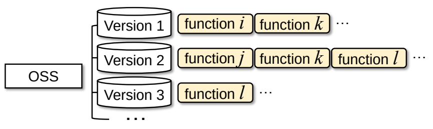
In this phase, we process the OSS projects to generate the component DB. However, we observed that simply storing all functions from all versions of every OSS makes the component identification phase extremely inefficient.

Redundancy elimination. We thus focus on the characteristics of OSS: the entire source code of an OSS is not newly developed each time the OSS is updated, and thus some parts common to different versions are redundantly compared with the target software when identifying OSS components. This characteristic gives the following intuition: if the functions common to multiple versions are only once compared with the target software, space and time complexity can be reduced.

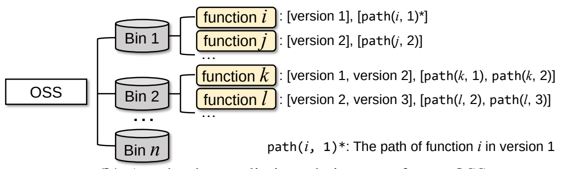
Let us define an OSS signature as a set of functions of the OSS, which will be stored in the component DB. The process for generating an OSS signature is as follows:

1) First, we extract all functions in all versions of an OSS.
2) Next, we create as many bins as the total number of versions in the OSS (denoted as n).
3) When a particular function appears in i different versions of the OSS, the function is stored in the i-th bin, along with the version information to which this function belongs, and the path information within each version.

Note that all the functions have undergone text preprocessing in accordance with our design assumptions. In addition, we apply a Locality Sensitive Hash (LSH) to the functions when storing them, which has native support for measuring the similarity between two hashes. The generated n bins of an OSS become the signature of the OSS (see Figure 3b).



(a) A naively generated OSS signature.


OSS
Bin 1
function i : [version 1], [path(i, 1)*]
function j : [version 2], [path(j, 2)]
Bin 2
function k : [version 1, version 2], [path(k, 1), path(k, 2)]
function l : [version 2, version 3], [path(l, 2), path(l, 3)]
Bin n
path(i, 1)*: The path of function i in version 1
…

(b) A redundancy eliminated signature for an OSS.

Fig. 3: Illustration of OSS signatures. We generate signatures for each OSS in the manner shown in (b), thereby reducing space complexity.

If we naively generate a signature by mapping the function to the version it belongs to (see Figure 3a), a function that exists in i different versions would be compared i times with the target software. However, our method of storing redundant functions only once in the corresponding bin reduces such unnecessary comparisons; the quantitative efficiency of redundancy elimination is described in Section V-C.

Another advantage is that even if the OSS is constantly updated, the number of functions newly added to the component DB is not large enough to impair scalability. In addition, because there are no functions excluded from indexing, if we designed an appropriate identification algorithm, the accuracy and, specifically, recall would not be impaired. By generating signatures for all OSS and storing them, the component DB is constructed.

C. *Component identification phase (P2)*

In this phase, CENTRIS identifies the reused OSS components in the target software.

Common functions. We first define the notion of common functions between two software projects. Each LSH algorithm provides its own *comparison method* and *cutoff* value [39]. Using the comparison method, we can measure the *distance* for each function pair between the two software projects, which indicates the syntactic difference between the two input functions. Hence, we define the relation between two functions (f_1, f_2) based on the *distance* and *cutoff* as follows:

LSH-based function relation decision:
- If $\big(distance(f_1, f_2) = 0\big)$: f_1 and f_2 are **identical**;
- If $\big(0 < distance(f_1, f_2) \leq cutoff\big)$: f_1 and f_2 are **similar**;
- If $\big(distance(f_1, f_2) > cutoff\big)$: f_1 and f_2 are **different**.

The similar and identical function pairs between the two software projects are determined as the common functions (the LSH algorithm is specified in Section IV).

Key concepts for precise identification. To identify modified components, we employ similarity threshold-based loose matching, *i.e.*, to check whether the code similarity between the target software and the OSS is greater than the predefined threshold. However, as previously mentioned, a simple threshold-based identification method suffers from a large number of false alarms.

False alarms may occur when (i) an OSS is nested or (ii) only the *borrowed code* of the OSS is included in the target software. Consequently, we present two concepts to reduce false alarms and precisely identify OSS components.

- **Prime OSS.** This refers to an OSS not containing any third-party software. If there is a number of common functions between a prime OSS and the target software, the prime OSS can be considered the correct component, because it violates condition (i) for false alarms.
- **Code segmentation.** If we only consider the *application code* of an OSS in component identification, no false alarms occur owing to the third-party software because this does not satisfy the false alarm condition (ii).

Accordingly, our component identification process comprises the following three steps (S1 to S3):

S1) Detecting the prime OSS in the component DB;

S2) Extracting application code from all OSS projects;

S3) Identifying components within the target software.

The above steps are conducted after extracting all functions of the target software and then applying the text preprocessing and LSH algorithm to the extracted functions.

*1) **Detecting the prime OSS in the component DB:*** Let S be the OSS to be checked as to whether it contains any third-party software. To determine whether S is the prime OSS, we first detect common functions between S and each OSS (denoted as X) in the component DB. If there is an OSS project having one or more common functions with S, the relation between S and X can be determined as belonging to one of the following four categories ($\mathbf{R_1}$ to $\mathbf{R_4}$, see Table II).

TABLE II: Possible relations between $\boldsymbol{S}$ and $\boldsymbol{X}$

Type	Description
$\mathbf{R_1}$.	S and X share widely utilized codes (*e.g.*, hash function).
$\mathbf{R_2}$.	S and X simultaneously reuse some other OSS projects.
$\mathbf{R_3}$.	S reuses X.
$\mathbf{R_4}$.	X reuses S

Among these relations, we are interested in $\mathbf{R_2}$ and $\mathbf{R_3}$, which imply that S contains at least one third-party software; conversely, when S and every X are related by $\mathbf{R_1}$ or $\mathbf{R_4}$, we can determine that S is the prime OSS.

In fact, $\mathbf{R_1}$ contrasts with the other three relations because there are few common functions between S and X. Therefore, the main challenge in determining whether S is the prime OSS is to differentiate $\mathbf{R_4}$ from $\mathbf{R_2}$ and $\mathbf{R_3}$.

Subsequently, we focus on when a common function between S and X first appeared in each OSS; we refer to this as the *birth time* of the function. Suppose that X reuses S (*i.e.*, $\mathbf{R_4}$); then, the birth time of a particular reused function f in S would be earlier than that in X.

Based on the above idea, we calculate the similarity score (ϕ) between S and X as follows (let $birth(f, S)$ be the birth time of f in S):

$$\phi(S, X) = \frac{|G|}{|X|},$$

where $G = \{f \mid \big(f \in (S \cap X)\big) \wedge \big(birth(f, X) \leq birth(f, S)\big)\}$

As shown in the above equation, we measure the similarity score by considering only the common functions that appeared earlier in X than S for identifying that X exhibits the R_2 and R_3 relations with S. As there are several ways to obtain the birth time of a function in an OSS, *e.g.*, code generation time, we utilize the information we already have. Within a bin of an OSS signature, the function hash values and version information to which the functions belong to are recorded. Therefore, we assign the release date of the earliest version among all recorded versions of a function as the birth time of the function in the OSS. In addition, widely used generic code, *e.g.*, hash functions or error-handling routines, can exist in both S and X (R_1), and thus, we use θ as a threshold.

Finally, we determine that X belongs to the R_2 or R_3 relation if X satisfies the following condition:

$$\phi(S, X) \geq \theta \qquad (1)$$

One might consider that X could reuse a third-party software (denoted as R) at a time later than S. In this case, because the functions in R have earlier birth times in S than those in X, the functions would not affect the measurement of $\phi(S, X)$. Therefore, even though S and X contain common third-party software, Equation (1) may not be satisfied. However, this case has no effect on determining whether S is the prime OSS. Obviously, S contains the R code base, and even if $\phi(S, X)$ does not satisfy Equation (1), $\phi(S, R)$ will be greater than θ; and thus, S is not the prime OSS, which is the correct answer.

Consequently, if there is no X that satisfies Equation (1), we determine that S is the prime OSS.

$$S = \begin{cases} \text{Prime OSS} & \text{if } \forall X.\big(\phi(S, X) < \theta\big) \\ \text{Non-prime OSS} & \text{if } \exists X.\big(\phi(S, X) \geq \theta\big) \end{cases}$$

Otherwise, we consider every X that satisfies Equation (1) as possible members of S, and store them; this information will be utilized for the code segmentation.

2) ***Extracting application code:*** In this step, we extract the application code through code segmentation for every OSS in the component DB. As a prime OSS does not have any borrowed code, we only focus on non-prime OSS projects.

Let S be the OSS of interest (*i.e.*, the signature). One way to locate the application code of S (S_A) is to remove the borrowed code (S_B) from S (*i.e.*, $S_A = S \setminus S_B$). However, detecting the OSS that belongs to S_B leads to a paradox: the CENTRIS methodology for identifying OSS components from the target software requires the same methodology for identifying the components of an OSS.

Fortunately, we do not need to exactly identify the sub-components of S. Instead, we use the possible members of S (denoted as P), which were obtained from the previous step.

Algorithm 1: The high-level algorithm for the code segmentation

```
Input: $S$    // The OSS that will be segmented
Input: $DB$   // The component DB
Output: $S_A$ // The application code of the $S$
procedure CODESEGMENTATION($S, DB$)
    $S_A \leftarrow \emptyset$
    $isPrime, members \leftarrow$ CHECKPRIME($S, DB$)
    if $\neg(isPrime)$    // $S$ has borrowed code parts
     then
        for $P \in members$ do
            $S = (S \setminus P)$    // Set minus operation
    $S_A = S$
    return $S_A$
procedure CHECKPRIME($S, DB$)
    $isPrime \leftarrow$True
    $members \leftarrow \emptyset$
    for $X \in DB$ do
        $G \leftarrow \emptyset$
        for $f \in (S \cap X)$ do
            if $birth(f, X) \leq birth(f, S)$ then
                $G.add(f)$
        $\phi(S, X) = (|G|/|X|)$
                                  // Similarity measurement
        if $\phi(S, X) \geq \theta$ then
            $isPrime \leftarrow$False
            $members.add(X)$
    return $isPrime, members$
```

As P is a possible member of S, it would be reused in S (*i.e.*, R_2) or it reuses a common third-party software with S (*i.e.*, R_3). P has no code that might belong to the application code of S; this is because only the code of an OSS that exhibits the R_4 relation with S can belong to the application code of S. In other words, the common functions between S and P are exactly included in the borrowed code of S, as mentioned in our definition (see Section II-A).

Therefore, we can obtain the application code of S by removing all functions of the possible members of S from the function set of S. The high-level algorithm for code segmentation is shown in Algorithm 1. Consequently, every OSS project in the component DB remains in a state wherein it is (i) detected as the prime or (ii) the application code is extracted (only for the non-prime OSS projects).

3) ***Identifying components:*** The next step is identifying the OSS components of the target software. Let T be the target software and S be the OSS in the component DB.

To identify whether S is the correct component of T, we measure the code similarity score between T and the application code of S. If S is the prime OSS, the application code (S_A) is the same as the entire S. The similarity score (Φ) is calculated as follows:

$$\Phi(T, S_A) = \frac{|T \cap S_A|}{|S_A|}$$

There may be a possibility that widely used or generic code exists in both T and S, as in the case of R_1; thus, we again employ the threshold θ as a filter. Finally, we determine that S is the correct component when $\boldsymbol{\Phi(T, S_A) \geq \theta}$. Once this process has been applied to all OSS in the component DB, we can get a set of OSS components of the target software.

Why Centris is accurate. First, as Centris does not rely on structural information in the identification phase, we can identify components regardless of structural change. Next, irrespective of whether OSS is nested, if the ratio of application code of each OSS is reused greater than θ, it can be identified as a correct component. Lastly, the code segmentation of Centris not only reduces false positives, but also helps to identify heavily modified components. Let consider the Velocypack component of ArangoDB, introduced in Section II-B; only 3.6% of Velocypack code base were reused in ArangoDB. In fact, Velocypack included another OSS (GoogleTest), and the ratio of the reused application code of Velocypack was measured as 12%. Highlighting the reuse patterns of only the application code of an OSS makes the similarity score between the target software and the OSS higher if the OSS is the correct component, and lower when the OSS is a false positive (*i.e.*, close to 0%). Using this distinct similarity score difference, Centris can precisely identify modified components with low false positives.

Version identification. To identify the reused version of each component, we focus on the reused functions of the OSS component. In the modified reuse, the functions of multiple versions could be simultaneously reused in the target software. Therefore, we assign a *weight* to each reused function. Specifically, we utilize a weighting algorithm that satisfies the condition that a larger weight is assigned to functions belonging to fewer versions. TF-IDF [40] suffices, where Term Frequency (TF) refers to the frequency of a function appearing in a particular version and Inverse Document Frequency (IDF) refers to the inverse of the number of versions containing this function. The IDF that satisfies the condition we set is utilized as the main weight function, and we use the "Boolean Frequency" as the TF; *i.e.*, we assign 1 to the TF of all functions.

Let n be the total number of versions of an OSS, and $V(f)$ be the versions to which a particular function f belongs. The weight function W is defined as $W(f) = \log\left(n/|V(f)|\right)$. Note that the $|V(f)|$ value of f that belongs to the i-th bin is i, by the definition of our signature generation. Accordingly, we loop through all the reused functions of the OSS component and add the weight of each function to the score of the versions to which it belongs. After scoring all functions, we identify the utilized version with the highest score.

Reuse pattern analysis. We then analyze the reuse pattern of the detected components. First, to identify code changes occurring during OSS reuse, we utilize the *distance* measured using the comparison method of the LSH algorithm (as explained at the beginning of P2) for each function pair between the OSS component and the target software. We determine whether the function is reused (*distance* = 0), not reused (*distance* $>$ *cutoff*), or reused with code changes ($0 <$ *distance* $\leq$ *cutoff*). Next, to measure the structural changes, we analyze the path differences between the reused functions and original functions. We split each function path using "/" (slash), traverse each path backward starting from the filename, and compare each path level. The criterion for comparison is the path of the original function. If any directory or file name is different, we determine that the structure has been modified.

Finally, according to the definition in Section II-A, if all functions of the OSS are reused without any modification, we refer to it as *exact reuse*. If there are unused functions, we refer to it as *partial reuse*. If the structure is changed while reusing, *structure-changed reuse* occurs. If any code is modified while reusing the OSS, we refer to it as *code-changed reuse*.

IV. Implementation of Centris

Centris comprises three modules: an *OSS collector*, a *preprocessor*, and a *component detector*. The OSS collector gathers the source code of popular OSS projects. The preprocessor stores the OSS signatures generated through redundancy elimination, and then extracts the application code of the OSS through code segmentation. The OSS collector and preprocessor need to be executed only once. Thereafter, the component detector performs the actual component identification on the target software. Centris is implemented in approximately 1,500 lines of Python code. The source code of Centris will be publicly available at the time of publication.

Initial dataset. Many programming languages provide dependency information (*e.g.*, Gemfile in Ruby). However, C and C++, which are two of the most popular languages (combined rank 3 in GitHub). do not provide dependency information despite the need. Although Centris is not restricted to a particular language, we demonstrate Centris targeting C/C++ software to prove its efficiency without any prior knowledge of dependency. We targeted GitHub, which has the largest number of repositories among version control systems [41]. Finally, we collected all repositories having more than 100 stars. The OSS collector of Centris collected 10,241 repositories including Linux kernel, OpenSSL, Tensorflow, among others (as of April. 2020). When we extracted all versions (*e.g.*, tags) from the repositories, we obtained 229,326 versions; the total lines were 80,388,355,577. This dataset is significantly larger than those used in previous approaches (*e.g.*, 2.5 billion C/C++ LoC database [7]).

Parser and LSH algorithm. To extract functions from software, we employed universal Ctags [42], an accurate and fast open-source regular expression-based parser. Next, among the various LSH algorithms [43]–[45], we selected the TLSH, as it is known to incur fewer false positives, and have a reasonable hashing and comparison speed as well as low influence of the input size [39], [45]. Its comparison algorithm, `diffxlen`, returns the quantified *distance* between the two TLSH hashes provided as inputs. In the context of Centris, functions that undergo modification after reuse fall into this category. We set the cutoff value to 30, referring to [45].

V. Evaluation and Findings

In this section, we evaluate Centris. Section V-A investigates how accurately Centris can identify OSS reuse in practice. Section V-B compares Centris with DéjàVu, motivating the need for code segmentation. In Section V-C, we evaluate the scalability of Centris and the efficacy of redundancy elimination. Section V-D reports our findings on OSS reuse patterns. Finally, we discuss a potential use case of Centris, software vulnerability management, in Section V-E. We evaluated Centris on a machine with Ubuntu 16.04, Intel Xeon Processor @ 2.40 GHz, 32GB RAM, and 6TB HDD.

A. *Accuracy of* Centris

Methodology. We conducted a cross-comparison experiment on our dataset of 10,241 repositories. To do so, we first selected representative versions (*i.e.*, the version with the most functions) of each OSS. As the reused components are mostly similar across different versions in one OSS, we decided to identify the components only for the representative version for each OSS, and measure the detection accuracy. To evaluate the accuracy of Centris, we used five metrics: true positives (TP), false positives (FP), false negatives (FN), precision $\left(\frac{\#TP}{\#TP + \#FP}\right)$, and recall $\left(\frac{\#TP}{\#TP + \#FN}\right)$.

Validation methods. Since C/C++ software has no standard format for recording OSS components, manual validation of the results is indispensable. To reduce the burden and improve the reliability of the validation process, we utilized the following three factors to automatically verify a portion of the detection results:

- **Paths:** The file paths of the reused functions (for stricter validation, we only consider the case when the name of the detected component is included in the reused function path);
- **Header files:** The header file configured with the OSS name;
- **Metadata files:** One of the `README`, `LICENSE`, and `COPYING` files in the top-level directory of the OSS ([46], [47]).

If one of the above factors of the detected OSS is intact in the target software, we determine that the detected OSS is the correct component of the target software. As an example of path-based validation, "`inflate.c`" of Zlib is reused in the path of "`src/../zlib-1.2.11/inflate.c`" in MongoDB. As examples of other validation methods, Redis reuses Lua while containing "`Lua.h`," and Libjpeg is reused in ReactOS where the `README` file of Libjpeg is contained in ReactOS with the path of "`dll/3rdparty/libjpeg/README`."

When a false alarm occurs, neither the name of the falsely detected component is included in the reused function path, nor the main header file and metadata files of the detected component are reused in the target software. These validation methods are only used to verify the results detected by Centris. Obviously, finding the correct answers is a more complex problem than verifying the obtained answers, and an issue arises when identifying components by relying sorely

* measured by only using the three automated validation methods.

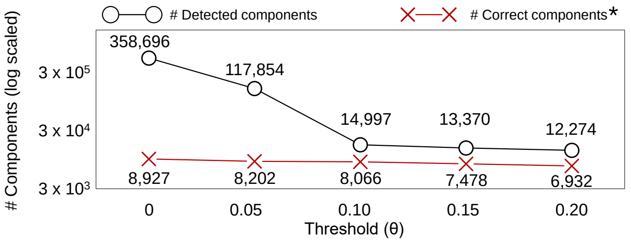


Fig. 4: Experimental results for measuring efficiency of θ.

on these validation methods: the target software can implicitly reuse an OSS without utilizing both the header files and the metadata files of the OSS. Thus, these validation methods do not negate the need for Centris.

Parameter setup. We then selected a suitable θ value to mitigate false alarms due to widely utilized code (see Section III-C). To select θ, we evaluated each cross-comparison result using predefined validation methods while setting θ to 0, 0.05, 0.1, 0.15, and 0.2. The results are depicted in Figure 4. Notably, the proportion of correct components in the detected components drops significantly when θ is less than 0.1. On the contrary, if θ is greater than 0.1, the proportion of correct components in the detected components increases slightly; however, the number of correct components decreases. The overall result implies that a widely utilized code is often shared among different OSS projects and accounts for only a small portion of each OSS project (generally less than 10%). For our experiment, we set θ as 0.1 to balance recall and precision.

Accuracy measurement. From the cross-comparison result, we observed that 4,434 (44%) out of 10,241 OSS projects were reusing at least one other OSS; a total of 14,997 components were detected. As it is challenging to identify literally every component in the target software, we cannot easily measure false negatives. Hence, we only considered false negatives that occurred when the application code of an OSS is reused less than the θ ratio and thus Centris fails to identify it, which can be measured by subtracting the number of correct components when θ is 0.1 from that when θ is zero (see Figure 4).

Among the cross-comparison results, we successfully validated 8,066 results (54%) using three predefined validation methods, and the remaining 6,931 detection results were manually analyzed. The manual validation was performed by two people and took two weeks. We manually viewed the paths, header files, and metadata files, as well as the reused source code and comments within the source code to determine whether the identified OSS is the correct component. The accuracy measurement results are presented in Table III.

Although most of the detected OSS components were reused with modification (95%), Centris achieved **91% precision** and **94% recall**. Although Centris precisely identified reused components in most cases, it reported several false results. We observed that false positives were mainly caused when the target software and the detected component only shared the third-party software that was not included

TABLE III: Accuracy of CENTRIS component identification results.

(For 14,997 cases)	Validation result				
	#TP	**#FP**	**#FN**	**Precision**	**Recall**
Automated validation results					
Paths (V_P)	3,685	N/A	N/A	N/A	N/A
Header files (V_H)	3,286	N/A	N/A	N/A	N/A
Metadata files (V_M)	4,175	N/A	N/A	N/A	N/A
Combined all validation methods ($V_P \cup V_H \cup V_M$)					
	8,066	N/A	N/A	N/A	N/A
Manual validation	5,510	1,421	861	0.80	0.86
Total	**13,576**	**1,421**	**861**	**0.91**	**0.94**

*According to our definition, all the results verified by the validation methods are TP, thus, the remaining columns are filled with N/A.

in our component DB. Hence, the application code of OSS projects was not properly obtained, resulting in false alarms. In addition, if the reuse ratio of the application code of the OSS was less than θ, or the reused component was not included in the component DB, CENTRIS failed to detect the correct OSS components, *i.e.*, false negatives occurred. However, simply decreasing θ for reducing false negatives can impair precision. Expanding current component DB such as collecting more OSS projects from various sources would be an efficient solution to reduce false results, even so, we believe that the method of minimizing false alarms through the proposed code segmentation works efficiently, and the selected θ simultaneously maintains a good balance in terms of precision and recall.

Version identification accuracy. Some components are not managed by the versioning system, and further the target software often does not reuse files or codes containing version information of a reused component. Subsequently, we decided to measure version identification accuracy for the three most widely reused OSS projects in our results: GoogleTest, Lua, and Zlib. Their version information is relatively well-defined compared to that of other OSS while still providing a sufficient pool to measure the accuracy. In our cross-comparison experiment, these three OSS projects were reused a total of 682 times. Approximately half of the reuses provided the utilized version using related files: "zlib.h" in Zlib, README in Lua, and CHANGES in GoogleTest. When these files were not reused in the target software, the version information was manually analyzed (*e.g.*, using the commit log). The version identification result is presented in Table IV. Partial reuse mainly occurred in Lua, code changes mostly appeared in Zlib, and structural changes primarily arose in GoogleTest.

CENTRIS succeeded in identifying the utilized version information with 91.5% precision. We failed to identify the accurate version in some modified reuse cases, especially when the functions from different versions (in extreme cases, more than 10 versions) of an OSS were mixed in the target software, *i.e.*, code-changed reuse. In such cases, we determined that not only is identifying the correct version a challenge, but also that the version identification is meaningless. Therefore, we concluded that identifying the OSS reuse and the most similar version would be sufficient for the code-changed reuses.

TABLE IV: Version identification accuracy of CENTRIS.

Reuse patterns		**#TP**	**#FP**	**Precision**
Exact reuse	E	115	0	100%
Modified reuse	P	112	3	97%
	SC	25	0	100%
	P & CC	185	29	86%
	P & SC & CC	187	26	88%
Total		624	58	**91.5%**

E: Exact reuse, P: Partial reuse, SC: Structure-Changed reuse, CC: Code-Changed reuse

B. In-depth comparison with DéjàVu

Tool selection. We reviewed several related approaches published since 2010; however, most SCA approaches are only applicable to identifying components in Android applications or software binaries ([7], [11]–[15]). For example, OSSPolice [7] is open to the public but its targets are Android applications. Moreover, as the parser for the C/C++ library is not open source, it would be difficult to apply their algorithm to our experiment. Therefore, we decided to compare CENTRIS with DéjàVu, a similar approach in terms of technology and purpose [29]. DéjàVu is based on the code clone detection technique (*i.e.*, SourcererCC [27]), and it aims to analyze the software dependencies among GitHub repositories by detecting project-level clones. Thus, we concluded that the detection results of DéjàVu can be compared with those of CENTRIS.

Methodology. Currently, the DéjàVu software is not publicly available; only the detection results previously obtained using the dataset (*i.e.*, GitHub C/C++ repositories in 2016) are provided[2]. We thus attempted to examine the component identification results of the common datasets between CENTRIS and DéjàVu, and compare them. In particular, DéjàVu determined the existence of a dependency based on the code similarity score between the two software projects. We set the similarity threshold to 50%, 80%, and 100% in DéjàVu (refer to [29]) and analyzed the number of correct target software and OSS component pairs from their results where the similarity score exceeded the selected threshold. CENTRIS employs θ as 10%. To demonstrate the efficiency of code segmentation, we provide both component identification results when code segmentation is turned on and off.

Comparison results. Among our cross-comparison results, four of the top 50 software projects (*i.e.*, ArangoDB, Crown, Cocos2dx-classical, and Splayer) with the maximum OSS reuse were observed to be part of the DéjàVu datasets as well. We decided to compare the OSS component detection results between CENTRIS and DéjàVu for these four software projects; the results are listed in Table V.

DéjàVu failed to identify many modified components. In fact, most identified components for the selected software were reused with modifications (see Table VI). As DéjàVu could not identify components when the reused code ratio was less than the selected threshold, the results showed low recall values (*i.e.*, at most 40%). Moreover, although DéjàVu aimed to detect project-level clones, its mechanism did not include

[2]http://mondego.ics.uci.edu/projects/dejavu/

TABLE V: Component identification results of DéjàVu and CENTRIS.

	CENTRIS (with *cs*)			CENTRIS (without *cs*)			DéjàVu (classified by the threshold)								
							50%			80%			100%		
Software	#T	#FP	#FN	#T	#FP	#FN	#T	#FP	#FN	#T	#FP	#FN	#T	#FP	#FN
ArangoDB	29	2	0	29	450	0	11	411	18	8	236	21	7	0	22
Crown	23	0	0	23	750	0	9	171	14	6	23	17	3	0	20
Cocos2dx	19	2	0	19	231	0	8	52	11	2	6	17	1	0	18
Splayer	16	1	0	16	275	0	7	236	9	6	27	10	3	0	13
Total	**87**	**5**	**0**	87	1,706	0	35	870	52	22	292	65	14	0	73
Precision	**0.95**			**0.05**			**0.04**			**0.07**			**1.0**		
Recall	**1.0**			**1.0**			**0.40**			**0.25**			**0.16**		

cs: code segmentation; #T: the number of true positives;
#FP: the number of false positives; #FN: the number of false negatives.

TABLE VI: OSS reuse patterns in the four software projects.

	Reuse patterns			
Software	Exact	Partial	Structure-Changed	Code-Changed
ArangoDB	7	19	4	11
Crown	3	20	3	16
Cocos2dx	1	17	2	14
Splayer	3	10	4	7
Total	**14**	**66**	**13**	**48**

*P, SC, and CC can occur simultaneously in the modified component.

a handling routine for false positives caused by nested OSS. Subsequently, DéjàVu reported many false positives, *i.e.*, it showed 4% and 7% precision when the threshold was selected as 50% and 80%, respectively. Even though DéjàVu showed 100% precision when the threshold was selected as 100%, it could not detect any partially reused components, as indicated by the fact that the recall was 16%.

In contrast, CENTRIS yielded substantially better accuracy than DéjàVu, *i.e.*, 95% precision and 100% recall when the code segmentation is applied. In the absence of code segmentation, CENTRIS reported numerous false positives (*i.e.*, 5% precision) with the same cause as DéjàVu: this implies that the method of using only the application code of OSS for matching through code segmentation can successfully filter out countless false positives. Lastly, OSS components that were identified only in DéjàVu and not identified in CENTRIS did not appear in the four software projects.

C. Speed and Scalability of CENTRIS

Efficacy of redundancy elimination. We can reduce space complexity by eliminating redundancies across OSS versions. The total number of functions in all versions of every OSS project that we collected is 2,205,896,465. After eliminating redundancies, we confirm that the number of non-redundant functions only accounts for 2.2% (49,330,494 functions) of the total functions, indicating that the size of the comparison space can be reduced by 45 times compared to all functions.

Speed. When we measured the preprocessing (extracting functions from the OSS, storing hashed functions, and generating the component DB) time of CENTRIS, on average, it took 1 min to preprocess 1 million LoC. Note that the OSS does not need to be preprocessed again after it undergoes initial preprocessing. In contrast, component identification occurs frequently. Hence, it is necessary to achieve fast speeds for practical use. When we compared 10,241 representative versions with the component DB, CENTRIS took less than

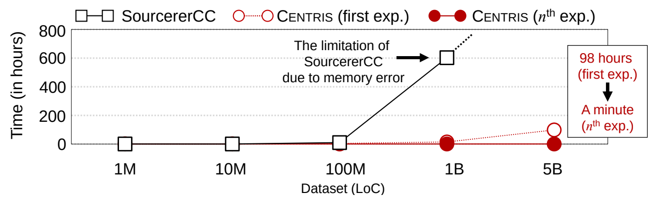


Fig. 5: Total time consumed on varying dataset sizes. CENTRIS exhibited tremendous time efficiency because of recycling of the preprocessed OSS projects in the dataset, whereas SourcererCC required approximately three weeks to process the matching using the 1 billion LoC dataset.

100 hours in total. This implies that CENTRIS takes less than a minute per target application on average, which is sufficiently fast for practical use.

Scalability. To evaluate the scalability of CENTRIS, we measured the time taken to compare the target software of 1 million LoC with different datasets ranging from 1 million to 5 billion LoC. We compared the performance of CENTRIS with that of the core algorithm of DéjàVu (SourcererCC [27]). Figure 5 depicts the results. In the first experiment, CENTRIS required 98 hours for preprocessing and matching. After the first experiment, because CENTRIS could recycle the preprocessed component DB, the required time was significantly reduced to less than a minute. SourcererCC needed three weeks for 1 billion LoC dataset, and when the dataset was increased in size, we could not measure the time consumed owing to memory errors in our evaluation environment; even if the experiment is performed with a larger memory, we expect the processing time to be significantly high.

D. Findings on OSS reuse patterns

From the cross-comparison result, we found that 4,434 (44%) OSS projects were reusing at least one other OSS. Surprisingly, the *modified reuses accounted for 95%* of the detected components. The distribution of detected reuse patterns and the average degree of modification are depicted in Figure 6. We summarize two key observations as follows.

Partial reuse accounts for 97% of all modified reuses. We observed that developers were mostly reusing only part of the OSS code base they needed. Mainly, functions deemed unnecessary for the target software to perform the desired operation, testing-purposed functions such as located in "`test/`" directory, and cross-platform support functions were excluded during an OSS reuse.

Code and structural changes also frequently occur. Among all modified reuses, 53% changed at least one original function, and 26% changed the original structure. We found that code changes occurred primarily due to developers' attempts to adapt the reused functions to their software (*e.g.*, change variable names), and to fix software vulnerabilities propagated from reused OSS. Moreover, we observed that the reused functions of an OSS are often merged in a single file rather than scattered in different structures. For example, Rebol software reused only 30% of Libjpeg while integrating all of the reused functions into "`src/core/u-jpg.c`."

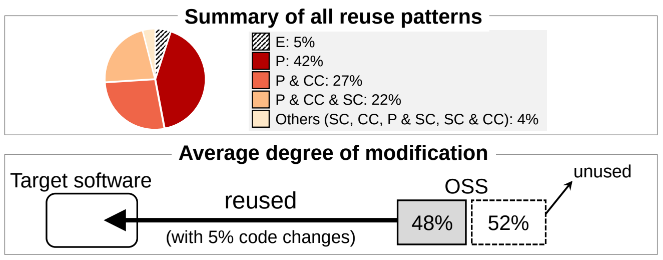


Fig. 6: Depiction of detected reuse patterns and averaged modification degree obtained from our experiment. Partial reuse appeared the most, and we found that developers reused only half of the OSS code base with 5% code changes on average.

Our observation results suggest the need to detect heavily modified components (*i.e.*, only 48% of the OSS code base were reused on average), but the existing approaches did not consider this trend (*e.g.*, both DéjàVu and OSSPolice selected the lowest threshold as 50%), hence failed to identify many correct components. From this point of view, CENTRIS would be a better solution for the efficient SCA process, as it can precisely identify modified components.

E. Use case: software vulnerability management

One potential use case of CENTRIS is software vulnerability management, which reduces security issues by identifying newly found but unpatched vulnerabilities. Below, we discuss our experience of using CENTRIS in this regard.

By referring to the National Vulnerability Database (NVD), we can obtain the affected software and version information, *i.e.*, Common Platform Enumeration (CPE), for each reported vulnerability. We have extensively examined whether the names and versions of detected OSS components are included in the obtained CPE [7]. Consequently, we discovered that 572 OSS projects contain at least one other vulnerable OSS component. Among them, 27 OSS projects are still reusing the vulnerable OSS in their latest version.

For the cases of successfully reproducing the vulnerability, we have reported to the corresponding vendors. Of these, the most notable example related to modified reuse is Godot (32K GitHub stars). We found that the latest version of Godot was reusing vulnerable JPEG-compressor contains CVE-2017-0700 (CVSS 7.8). Godot was reusing only *one file* from JPEG-compressor (“`jpgd.cpp`”), which contains the exact vulnerable code. More seriously, this vulnerability could be reproduced by simply uploading a malicious image file to the Godot project. We reported this information on their repository’s issues; developers immediately patched the vulnerability (Jul. 2019). Likewise, we could successfully reproduce a vulnerability in Stepmania, Audacity (so far, reusing vulnerable Libvorbis), LibGDX (reusing vulnerable JPEG-compressor), and Redis (reusing vulnerable Lua); for all cases, we reported to the corresponding development and security teams, and confirmed that proper actions were taken, such as vulnerability patches. Even though developers reuse only a small part of an OSS, the vulnerability in that part opens up the attack surface. To address this, we can apply CENTRIS for more attentive vulnerability management as shown here.

F. Threats to validity

First, although our dataset is more expansive compared to those in previous approaches, the benchmark OSS projects utilized herein might not be representative. Second, to our knowledge, there are no approaches that directly attempt to identify modified components. Although we conducted an in-depth comparison with DéjàVu, our intention is not to deny the accuracy and performance of DéjàVu, but to demonstrate that our approach is much more efficient for the purpose of identifying modified components. Finally, there may be hidden components in a target software that both CENTRIS and DéjàVu failed to identify; as all OSS reuse statuses are not known, we cannot exactly measure the missed components, and these are the false negatives of CENTRIS.

VI. RELATED WORK

Code clone detection. Over the past decades, numerous techniques have been proposed to detect code clones [6], [16]–[35], and CENTRIS adopts a signature-based clone detection method [6]. However, as we demonstrated in this paper, using an existing clone detection technique as it is suffers from false alarms when identifying modified reuse of nested OSS.

Software composition analysis. Existing SCA techniques [7], [11]–[15] are not accurate enough to identify modified OSS reuse. Duan et al. [7] proposed OSSPolice to find third-party libraries of an Android application. They utilized constant features of obfuscation to extract the version information and determine if vulnerable versions were utilized. They minimized false alarms through hierarchical indexing and matching. Since their concern is more on accurately identifying third-party libraries at the binary level, thus, it differs from our concern to detecting modified components. Backes et al. [12] and Bhoraskar et al. [48] also do not consider detection of modified OSS reuse. CoPilot [15] analyzes security risks that arise from unmanaged OSS components. However, as it is based on dependency files, it can be applied only for languages in which dependencies are managed. To our knowledge and experience, commercial SCA tools such as Black Duck Hub [49] by Synopsys [50] and Antepedia [51] do not consider modified reuse and hence miss many reused components.

VII. CONCLUSION

Identifying OSS reuse is a pressing issue in modern software development practice, because unmanaged OSS components pose threats by increasing critical security and legal issues. In response, we presented CENTRIS, which departs significantly from existing techniques by enabling precise and scalable identification of reused OSS components even when they are heavily modified and nested. With the information provided by CENTRIS, developers can mitigate threats that arise from unmanaged OSS components, which not only increases the maintainability of the software, but also renders a safer development environment. CENTRIS will be open-sourced with a public web service and all experimental data and results in this paper will be publicly available to foster future research.

REFERENCES

[1] S. Koch, "Evolution of open source software systems–a large-scale investigation," in *Proceedings of the 1st International Conference on Open Source Systems*, 2005, pp. 148–153.

[2] A. Deshpande and D. Riehle, "The total growth of open source," in *IFIP International Conference on Open Source Systems*. Springer, 2008, pp. 197–209.

[3] *2018 open source security and risk analysis (OSSRA)*, Synopsys, 2018, https://www.blackducksoftware.com/about/news-events/releases/audits-show-open-source-risks.

[4] *The GitHub Blog - Thank you for 100 million repositories*, GitHub, 2018, https://github.blog/2018-11-08-100m-repos/.

[5] H. Li, H. Kwon, J. Kwon, and H. Lee, "CLORIFI: software vulnerability discovery using code clone verification," in *Concurrency and Computation: Practice and Experience*, vol. 28, no. 6. Wiley Online Library, 2016, pp. 1900–1917.

[6] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery," in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017, pp. 595–614.

[7] 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 (CCS)*. ACM, 2017, pp. 2169–2185.

[8] *Software Composition Analysis Explained*, WhiteSource, 2019, https://resources.whitesourcesoftware.com/blog-whitesource/software-composition-security-analysis.

[9] *Technology Insight for Software Composition Analysis*, Gartner, Inc., 2019.

[10] A. S. Barb, C. J. Neill, R. S. Sangwan, and M. J. Piovoso, "A statistical study of the relevance of lines of code measures in software projects," in *Innovations in Systems and Software Engineering*, vol. 10, no. 4. Springer, 2014, pp. 243–260.

[11] 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.

[12] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 356–367.

[13] 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 *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 335–346.

[14] W. Tang, D. Chen, and P. Luo, "Bcfinder: A lightweight and platform-independent tool to find third-party components in binaries," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2018, pp. 288–297.

[15] *An open source management solution*, CoPilot, 2019, https://copilot.blackducksoftware.com/.

[16] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Reverse Engineering, Proceedings of 2nd Working Conference on*. IEEE, 1995, pp. 86–95.

[17] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings. International Conference on Software Maintenance*. IEEE, 1998, pp. 368–377.

[18] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *International static analysis symposium*. Springer, 2001, pp. 40–56.

[19] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," in *IEEE Transactions on Software Engineering*, vol. 28, no. 7. IEEE, 2002, pp. 654–670.

[20] G. Myles and C. Collberg, "Detecting software theft via whole program path birthmarks," in *International Conference on Information Security*. Springer, 2004, pp. 404–415.

[21] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code," in *OSDI*, vol. 4, no. 19, 2004, pp. 289–302.

[22] G. Myles and C. Collberg, "K-gram based software birthmarks," in *Proceedings of the 2005 ACM symposium on Applied computing*. ACM, 2005, pp. 314–318.

[23] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2007, pp. 96–105.

[24] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003, pp. 76–85.

[25] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," in *Queen's School of Computing TR*, vol. 541, no. 115, 2007, pp. 64–68.

[26] ——, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *2008 16th IEEE International Conference on Program Comprehension*. IEEE, 2008, pp. 172–181.

[27] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling code clone detection to big-code," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 1157–1168.

[28] Y. Semura, N. Yoshida, E. Choi, and K. Inoue, "CCFinderSW: Clone Detection Tool with Flexible Multilingual Tokenization," in *Asia-Pacific Software Engineering Conference (APSEC), 2017 24th*. IEEE, 2017, pp. 654–659.

[29] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "DéjàVu: a map of code duplicates on GitHub," in *Proceedings of the ACM on Programming Languages*, vol. 1, no. (OOPSLA). ACM, 2017, p. 84.

[30] M. A. Nishi and K. Damevski, "Scalable code clone detection and search based on adaptive prefix filtering," in *Journal of Systems and Software*, vol. 137. Elsevier, 2018, pp. 130–142.

[31] *A source code search engine*, Searchcode, 2019, http://searchcode.com/.

[32] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, "CCAligner: a token based large-gap clone detector," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, 2018, pp. 1066–1077.

[33] D. Luciv, D. Koznov, G. Chernishev, H. A. Basit, K. Romanovsky, and A. Terekhov, "Duplicate finder toolkit," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*. ACM, 2018, pp. 171–172.

[34] M. Gharehyazie, B. Ray, M. Keshani, M. S. Zavosht, A. Heydarnoori, and V. Filkov, "Cross-project code clones in GitHub," in *Empirical Software Engineering*. Springer, 2018, pp. 1–36.

[35] T. Vislavski, G. Rakic, N. Cardozo, and Z. Budimac, "LICCA: A tool for cross-language clone detection," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 512–516.

[36] C. W. Krueger, "Software reuse," in *ACM Computing Surveys (CSUR)*, vol. 24, no. 2. ACM, 1992, pp. 131–183.

[37] M. L. Griss, "Software reuse architecture, process, and organization for business success," in *Proceedings of the Eighth Israeli Conference on Computer Systems and Software Engineering*. IEEE, 1997, pp. 86–89.

[38] R. Duan, A. Bijlani, Y. Ji, O. Alrawi, Y. Xiong, M. Ike, B. Saltaformaggio, and W. Lee, "Automating Patching of Vulnerable Open-Source Software Versions in Application Binaries," in *In Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[39] A. Lee and T. Atkison, "A comparison of fuzzy hashes: evaluation, guidelines, and future suggestions," in *Proceedings of the SouthEast Conference*. ACM, 2017, pp. 18–25.

[40] G. Salton and M. J. McGill, *Introduction to modern information retrieval*. New York: McGraw - Hill Book Company, 1983.

[41] *Version Control Systems Popularity in 2016*, Rhodecode, 2016, https://rhodecode.com/insights/version-control-systems-2016.

[42] *Universal Ctags*, Ctags, 2019, https://github.com/universal-ctags/.

[43] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," in *Digital investigation*, vol. 3. Elsevier, 2006, pp. 91–97.

[44] V. Roussev, "Hashing and data fingerprinting in digital forensics," in *IEEE Security & Privacy*, vol. 7, no. 2. IEEE, 2009, pp. 49–55.

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

[46] G. M. Kapitsaki, N. D. Tselikas, and I. E. Foukarakis, "An insight into license tools for open source software systems," *Journal of Systems and Software*, vol. 102, pp. 72–87, 2015.

[47] S. Ikeda, A. Ihara, R. G. Kula, and K. Matsumoto, "An empirical study of readme contents for javascript packages," *IEICE TRANSACTIONS on Information and Systems*, vol. 102, no. 2, pp. 280–288, 2019.

[48] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall, "Brahmastra: Driving Apps to Test the Security of Third-Party Components," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, 2014, pp. 1021–1036.

[49] *A complete open source management solution by Synopsys*, Black Duck Hub, 2019, https://www.blackducksoftware.com/products/hub.

[50] *A comprehensive software analysis solution*, Synopsys, 2019.

[51] *A Software Artifacts Knowledge Base (the service is currently hold)*, Antepedia, 2019, http://www.antepedia.com/.