Identifying Open-Source License Violation and 1-day Security Risk at Large Scale

Ruian Duan*
Georgia Institute of Technology

Ashish Bijlani*
Georgia Institute of Technology

Meng Xu
Georgia Institute of Technology

Taesoo Kim
Georgia Institute of Technology

Wenke Lee
Georgia Institute of Technology

ABSTRACT

With millions of apps available to users, the mobile app market is rapidly becoming very crowded. Given the intense competition, the time to market is a critical factor for the success and profitability of an app. In order to shorten the development cycle, developers often focus their efforts on the unique features and workflows of their apps and rely on third-party Open Source Software (OSS) for the common features. Unfortunately, despite their benefits, careless use of OSS can introduce significant legal and security risks, which if ignored can not only jeopardize security and privacy of end users, but can also cause app developers high financial loss. However, tracking OSS components, their versions, and interdependencies can be very tedious and error-prone, particularly if an OSS is imported with little to no knowledge of its provenance.

We therefore propose OSSPolice, a scalable and fully-automated tool for mobile app developers to quickly analyze their apps and identify free software license violations as well as usage of known vulnerable versions of OSS. OSSPolice introduces a novel hierarchical indexing scheme to achieve both high scalability and accuracy, and is capable of efficiently comparing similarities of app binaries against a database of hundreds of thousands of OSS sources (billions of lines of code). We populated OSSPolice with 60K C/C++ and 77K Java OSS sources and analyzed 1.6M free Google Play Store apps. Our results show that 1) over 40K apps potentially violate GPL/AGPL licensing terms, and 2) over 100K of apps use known vulnerable versions of OSS. Further analysis shows that developers violate GPL/AGPL licensing terms due to lack of alternatives, and use vulnerable versions of OSS despite efforts from companies like Google to improve app security. OSSPolice is available on GitHub.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; *Digital rights management*; • **Software and its engineering → Software libraries and repositories**;

KEYWORDS

Application Security; License Violation; Code Clone Detection

1 INTRODUCTION

The mobile app market is rapidly becoming crowded. According to AppBrain, there are 2.6 million apps on Google Play Store alone [5]. To stand out in such a crowded field, developers build unique features and functions for their apps, but more importantly, they try to bring their apps to the market as fast as possible for the first-mover advantage and the subsequent network effect. A common development practice is to use open-source software (OSS) for the necessary but "common" components so that developers can focus on the unique features and workflows. With the emergence of public source code hosting services such as GitHub [34] and Bitbucket [6], using OSS for faster app development has never been easier. As of October 2016, GitHub [34] reported hosting over 46 million source repositories (repos), making it the largest source hosting service in the world.

Despite their benefits, OSS must be used with care. Based on our study, two common issues that arise from the careless use of OSS are software license violations and security risks.

License violations. The use of OSS code in apps can lead to complex license compliance issues. OSS are released under a variety of licenses, ranging from the highly permissive BSD and MIT licenses to the highly restrictive ones: General Public License (GPL), and Affero General Public License (AGPL). Use of OSS implicitly bounds the developer to the associated licensing terms, which are protected under the copyright laws. Consequently, failure to comply with those terms could have legal ramifications. For example, Cisco and VMWare were involved in legal disputes for failing to comply with the licensing terms of Linux kernel [69, 85].

Security risks. OSS may also contain exploitable vulnerabilities. For instance, recently reported vulnerabilities in Facebook and Dropbox SDKs [2, 53] could be exploited to hijack users' Facebook accounts and link their devices to attacker-controlled Dropbox

*Co-first authors with equal contribution.

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134048

accounts, respectively. Vulnerabilities found in OSS are typically patched in subsequent releases while apps using old, unpatched versions can put end users' security and privacy in jeopardy.

To obviate such issues, app developers must diligently manage all OSS components in their apps. In particular, developers not only need to track all OSS components being used and regularly update them with security fixes, but also comply with the license policies and best practices in all OSS components and follow license changes across versions.

However, manually managing multiple OSS components, their versions, and interdependencies can quickly become very tedious and error-prone, particularly if an OSS is imported with little to no knowledge of its provenance. Moreover, license engineering and compliance require both legal as well as technical expertise, which given the diversity of software licenses, can prove costly and time-consuming. Consequently, while some developers may ignore the need for managing OSS to avoid additional overheads, others may fail to correctly manage them due to ignorance or lack of tools and expertise, thereby inadvertently introducing security risks and license violations.

We have developed OSSPolice, a scalable and fully-automated tool to quickly analyze app binaries to identify potential software license violations and usage of known vulnerable OSS versions. OSSPolice uses software similarity comparison to detect OSS reuse in app binaries. Specifically, it extracts inherent characteristic features (a.k.a. software birthmarks [37]) from the target app binary and efficiently compares them against a database of features extracted from hundreds of thousands of OSS sources in order to accurately detect OSS versions being used. In the event that the correct version is missing from our database, or if two versions have no distinct features, in line with our findings, the closest version of OSS is detected.

Based on the detected usage of OSS versions, the ones containing known software security vulnerabilities or under restrictive free software licensing terms are reported. OSSPolice polls the Common Vulnerabilities and Exposures (CVE) database to track OSS versions affected with security vulnerabilities. We also include vulnerabilities found by Google's App Security Improvement program (ASIP) [41]. In this work, we only track OSS usage under GPL and AGPL licenses due to their wide usage and highly restrictive terms (e.g. require derivatives works to open-source) and flag detected cases as potential violations if app sources are not found. It is worth noting that OSSPolice focuses solely on the technical aspects of license compliance, not the legal issues. Although OSSPolice does perform extra validation before reporting an app, such as checking whether its source code is publicly available on the developer website or popular code hosting webservices (e.g., GitHub), raising legal claims is not a goal of OSSPolice.

The current prototype of OSSPolice has been designed to work with Android apps due to its popularity and market dominance. Nevertheless, the techniques used can also easily be applied to iOS, Windows, and Linux apps. OSSPolice can analyze both types of Android binaries: C/C++ native libraries and Java Dalvik executables (dex).

A number of code reuse detection approaches have been proposed, but each presents its own set of limitations when applied to our problem setting. For instance, whereas some assume availability of app source code [8, 9, 45, 49, 51], others either support only a subset of languages (C [39], Java [7, 10, 82]) or use computationally expensive birthmark features to address software theft [23, 59, 65, 66, 90, 91], known bugs [28, 30] and malware detection [25, 93, 96]. In contrast, the goal of OSSPolice is not to detect deliberate repackaging, software theft, or malware; rather it is a tool for developers to quickly identify inadvertent license violations and vulnerable OSS usage in their apps. To this end, we assume that app binaries have not been tampered with in any specific way to evade OSS reuse detection. Based on this assumption, we trade accuracy in the face of code transformations to gain performance and scalability in the design space. We use syntactical features, such as string literals and exported functions when matching native libraries against OSS sources. This is because these features are easy to extract and preserved even across stripped libraries. However, since Java code in Android apps is commonly obfuscated with identifier renaming, OSSPolice has been designed to be resilient to such simple code transformations. To match dex files against Java OSS, we rely on string constants and proven obfuscation-resilient features, such as normalized classes [7] and function centroids [22] as features.

OSSPolice maintains an indexing database of features extracted from OSS sources for efficient lookup during software similarity detection. One approach to build such a database, as adopted by BAT [39], is to create a direct (inverted) mapping of features to the target OSS. However, this approach fails to consider large code duplication across OSS sources [62] and, hence, suffers from low detection accuracy and poor scalability (§3.4.1). Indexing multiple versions of OSS further adds to the problem. OSSPolice, therefore, uses a novel *hierarchical indexing scheme* that taps into the structured layout (i.e., a tree of files and directories) of OSS sources to apply multiple heuristics for improving both, the scalability and the detection accuracy of the system (§3.4.3).

Our experiments show that OSSPolice is capable of efficiently searching through hundreds of thousands of source repos (billions of lines of code). We evaluated the accuracy of OSSPolice using open-source Android apps on FDroid [29] with manually labeled ground truth. OSSPolice achieves a recall of 82% and a precision of 87% when detecting C/C++ OSS usage and a recall of 89% and a precision of 92% when detecting Java OSS usage, which outperforms both BAT [39] and LibScout [7]. For version pinpointing, OSSPolice is capable of detecting 65% more OSS versions than LibScout [7].

In summary, we contribute as follows:

- We identify the challenges in accurately comparing an app binary against hundreds of thousands of OSS source code repos and propose a novel hierarchical indexing scheme to achieve both the accuracy and scalability goals.
- We present the design and implementation of OSSPolice, a scalable and fully-automated system for OSS presence detection in Android apps, and further use the presence information to identify potential license violations and usage of vulnerable OSS versions in Android apps.
- We apply OSSPolice to analyze over 1.6 million free Android apps from Google Play Store and compare their similarity

to 60K C/C++ and 77K Java OSS versions. To the best of our knowledge, this is the first large-scale study to do so. We present our findings, highlighting over 40K cases of potential GPL/AGPL violations and over 100K apps using vulnerable OSS versions (§6).
- We conduct further analysis on the detected results and find that developers violate GPL/AGPL licensing terms due to lack of choice, and use vulnerable OSS versions despite efforts from companies like Google to improve app security.

2 RELATED WORK

Previous efforts related to OSSPolice can be categorized into the two lines of work.

Software similarity detection. Software similarity detection techniques compare one software to another to measure their similarity. Various such techniques have been studied and applied in across domains. However, none of those are suitable for our problem setting, i.e. comparing Java dex files as well as *fused* C/C++ libraries in Android apps against hundreds of thousands of source code repos §3.4.1.

Code clone detection. One such technique is code clone detection that identifies the reuse of code fragments across source repos. It was historically used to improve software maintainability [8, 9, 11, 27, 47–49, 52], but has also been studied to detect software theft (or plagiarism) [1, 59, 65, 66, 74] and cloned bugs [32, 45, 57]. These methods assume the availability of app source code. OSSPolice, on the other hand, detects OSS code reuse in app binaries since their sources may not be available.

Java/Dalvik bytecode clone detection. Some works have studied similarity detection in Java bytecode code. Baker and Manber [10] Tamada et al. [82] proposed birthmarking to detect software theft.

Techniques to detect app cloning have also been studied to identify malicious and pirated apps. They computed similarity between apps using code-based similarity techniques [22, 38, 96] or by extracting semantic features from program dependency graphs [25, 26]. Other approaches have also studied third-party library detection on Android, ranging from naïve package name based [16, 36] whitelisting, to code clustering [26, 56, 61, 88] and machine learning [67] based approaches. In particular, WuKong [88] automatically identify third-party library uses with no prior knowledge with code clustering techniques, LibRadar [61] extended it by generating a unique profile for each cluster identified, and LibD [56] further adopted feature hashing algorithm to achieve scalability. However, these approaches are either not scalable or rely on the assumption that the third-party code is used by many apps without modification, which might not always hold true [79].

In contrast, LibScout [7] considered unused code removal and proposed a different feature: normalized class, as a summary of actual class to detect third-party libraries with obfuscation resiliency. However, LibScout [7] doesn't scale to a large number of OSS, because they iterate over all the third-party libraries to find matches for candidate apps.

Binary clone detection. Various approaches have been proposed to measure the similarity of two binaries [28, 30, 33, 60, 73, 90, 98]. OSSPolice, however, does not assume that the OSS binaries can be built from sources or obtained.

There are also approaches proposed to detect OSS code reuse in binaries [39, 50]. [50] computes signatures of functions present in both source and binary using the size of arguments and local variables, then employs k-gram method to perform similarity analysis. Similarly, Binary Analysis Tool (BAT) [39] extracts strings in binary files and compares them with information extracted from OSS source repos to perform similarity measurement analysis. However, both of them have not been designed to scale to the amount of repos OSSPolice faces. Moreover, they suffer from low detection accuracy due to inability to handle internal code cloning across OSS sources §3.4.1.

Commercial services. A number of commercial services, such as Black Duck Software's Protex [15], OpenLogic [72], Protecode [81], and Antelink [3] are also available that assist enterprises in managing OSS license compliance and identifying security risks. However, they scan source code to detect OSS code clones by comparing against their own database of OSS sources.

Third-party component security. [68] presented a threat scenario that target `WebView` apps and [63] further found that 28% of apps that uses embedded web browsers have at least one vulnerability, either due to use of insecure code or careless mistakes. [21, 89, 97] vetted the assumptions and implementations for authentication protocols in third-party SDKs and found that three popular SDKs are vulnerable. They further verified that many apps that relies on these SDKs are vulnerable too.

While similar in the final goal, these works actively test whether an app violates the specified protocols/procedures while OSSPolice only passively test whether an app is vulnerable by inferring from the presence of vulnerable versions of OSS components. [13] is also a passive approach, however, given a specific vulnerable version of OSS component, it uses dynamic driving to trigger the buggy code while OSSPolice is purely static.

Given the wide spread of vulnerable third-party components in mobile apps, researchers have also proposed various mechanisms to isolate untrusted third-party code from the code originated from app developers. [80] isolated components in native code; [76, 94] isolated operation of ad libraries from the rest of the app; [71, 92] provided ways to achieve access control on untrusted code. These works are orthogonal to OSSPolice and can be used as remedy actions for vulnerable OSS components that cannot be easily fixed by updating to the latest version.

3 DESIGN

3.1 Goals and Assumptions

We envision OSSPolice as a webservice (or a standalone tool) for mobile app developers that quickly compares their apps against a database of hundreds of thousands of OSS sources in view of identifying free software license violations as well as known vulnerable OSS being used.

Nonetheless, detection of software license violation entails both legal and technical aspects. OSSPolice, focuses solely on the latter;

Languages	Features	OP	BAT	LS	C
C/C++	String literal	✓	✓	NA	NA
	Exported function	✓	✓		
	Control-flow graph	✗	✗		
Java	String constant	✓	✓	✗	✗
	Function name	✗	✓	✗	✗
	Normalized class	✓	✗	✓	✗
	Function Centroid	✓	✗	✗	✓
	Control-flow graph	✗	✗	✗	✗

Table 1: Comparison of OSSPolice (OP) with state-of-the-art binary clone detection systems, including BAT [39], LibScout (LS) [7], and Centroid (C) [22]

its goal is to only collect statistical evidence suggesting a license violation, not draw any legal conclusions. Similarly, OSSPolice is not a system to discover new or existing security vulnerabilities. Its goal is to only highlight the reuse of known vulnerable OSS versions in apps, not to find or provide a concrete proof for vulnerabilities. We provide detailed reasoning for these design choices in §7.

OSSPolice assumes that the violations have been caused inadvertently and do not constitute of deliberate software theft or piracy. Therefore, it assumes that app binaries have not been tampered with to defeat code reuse detection.

To this end, we set the following specific goals:

- Accurate detection of OSS versions being used in app binaries,
- Collection of evidence suggesting license violations and presence of known vulnerable OSS versions,
- Efficient use of hardware resources, and
- Scalability to search against hundreds of thousands of OSS sources (billions of lines of code).

3.2 Apps vs OSS

Android apps mainly contain two kinds of binaries: dalvik executable (dex) files and native libraries. OSSPolice separately analyzes each binary type in an Android app and compares it against OSS sources to detect specific versions being used.

Native Libraries. Native libraries are built directly for machine architecture, such as ARM and x86 from C/C++ sources and loaded on demand at runtime. App developers use native libraries in Android apps for various reasons, such as code reuse, better performance, or cross-platform development. One way to detect OSS reuse in an app native library is to first build a native library from subject OSS sources, which can then be compared with the target app library leveraging existing binary similarity measurement techniques [23, 73, 98]. However, this approach suffers from the following limitations. First, it implies automating the build of OSS sources in order to be scalable, which is nontrivial if not impossible. OSS written in low-level languages, such as C/C++ demand specialized build environment, including all dependencies, build tools, and target-specific configuration. For example, native libraries present in Android apps must be built using Android Native Development Kit (NDK) toolchain. Consequently, automatically building a binary from C/C++ OSS sources is not a one-step procedure; instead one must follow complex build instructions to create the required build environment. However, such specific build instructions may not be available from the OSS developer as a part of the sources. Second, even if we are able to successfully build OSS sources, the generated OSS library may differ significantly from the target app library because of different compilation flags (e.g., optimizations) or mismatching system configuration. For instance, system configuration headers created during compilation time that capture the type (e.g., architecture data types, etc.) of the host system would be different on disparate systems. To avoid such pitfalls, we directly compare app native libraries to OSS sources.

Java Dex Files. Compared to native libraries, Android dex files are built from Java sources and executed under a sandboxed Java Virtual Machine runtime. Being amenable to reverse engineering, dex files are commonly obfuscated to hide proprietary details. In fact, the official Android development IDE, Android Studio [35] is shipped with a built-in obfuscation tool, called ProGuard [54], that removes unused code and renames classes, including any fields and functions with semantically obscure names to hide proprietary implementation details. For example, package name com.google.android is renamed to a.g.c. OSSPolice is designed to be resilient against common obfuscation techniques, such as identifier renaming and control-flow randomization for analyzing Java dex binaries. Although app developers can also adopt advanced code obfuscation methods, such as string or class encryption and reflection-based API hiding, we found such cases to be rare in our dataset, possibly because such mechanisms incur high runtime overhead.

3.3 Feature Selection

OSSPolice employs software similarity comparison to detect OSS reuse. Specifically, when analyzing mobile app binaries, OSSPolice uses software birthmarks [37] to compare their similarity to OSS sources to accurately detect usage of OSS versions. A software birthmark is a set of inherent features of a software that can be used to identify it. In other words, if software X and Y have the same or statistically similar birthmarks, then they, with high probability, are copies of each other.

Selecting birthmarks (a.k.a. features) entails balancing performance, scalability, and accuracy of software similarity detection; depending upon the design goals, appropriate trade offs can be made. For example, syntactic features, such as string literals are easy to extract and are preserved in the binary, but can also be obfuscated (e.g., string encryption) to defeat detection. Simple syntactic features are not reliable when applied to the problem of malware clone detection and app repackaging detection. Past works targeting such adversarial problems have, therefore, often employed program dependency graph or dynamic analysis to defeat advanced evasion techniques [17, 46, 90, 93, 95]. However, such semantic features are not only difficult to extract correctly, but also consume overwhelmingly high amount of CPU and memory resources, limiting system scalability.

OSSPolice is neither a tool to find malware in apps nor does it aim to detect deliberate software theft or piracy. We, therefore, trade accuracy against code transformations to gain performance and scalability in the design space. In particular, we assume that app binaries have not been tampered with to evade OSS detection

and rely on simple syntactical features, such as string literals and functions for the purposes of comparing Android native binaries against C/C++ OSS sources. Table 1 shows the list of all features used by OSSPolice. The reasons for selecting them are many-fold. Besides being easy to extract, we found these features to be stable against code refactoring, precise enough to distinguish between different OSS versions, and preserved (ASCII readable) even across stripped libraries. During our analysis of 1.6 million free Google Play Store Android apps, consisting of 271K native ARM libraries (98.9% stripped) and we found that 85% of native libraries have more than 50 features (strings and functions) preserved. We further found that for most native libraries, the number of functions increases linearly as the library size grows, which indicates that most of the apps do not strip or hide functions in native libraries. In fact, there are only 11.6% libraries that are larger than 40KB in size, but have less than 50 visible functions. Finally, these features have been widely used and proven effective in various binary clone detection schemes [39, 50].

Similar syntactical features are used by OSSPolice to match app dex files against Java OSS sources, namely string constants and class signatures. However, to be resilient against common obfuscation techniques, such as identifier renaming, we normalize classes before producing their signatures in a way that they lose all user-defined (custom) details, but retain their interactions with the common framework API. Normalized classes have been proven to survive ProGuard obfuscation process [7]. The signatures are derived in two steps. First, all functions in a class are normalized by removing everything except their argument list and return types and further replacing non-framework types with a placeholder. Next, the resulting normalized functions are sorted and hashed to get their class signature. However, our analysis revealed that while string constants and normalized class signatures can detect OSS reuse in Java dex files, they are too weak to accurately detect Java OSS versions. Thus, we also use function centroids [22] for additional entropy. Centroid of a function is generated through a deterministic traversal of its intra-procedural graph. It captures the control flow characteristics of a function and generates its signature as a three dimensional data point, representing basic block index, outgoing degree, and loop depth. Computing and comparing function centroids are, however, computationally expensive tasks. Therefore, we defer them until the later phase of similarity detection and use only to pinpoint OSS versions (§4).

To determine how unique these features are across OSS versions, we also analyzed cross-version uniqueness of these features for OSS collected by `OSSCollector` in §4, which contains 3K C++ and 5K Java software, totaling to 60K C/C++ and 77K Java versions, respectively. We find that 83% of C/C++ and 41% of Java OSS versions can be uniquely identified using the aforementioned features.

3.4 Similarly Detection

Given sets of features from app binaries (denoted by *BIN*) and OSS sources (denoted by *OSS*), a typical software similarity detection scheme is to compare the two feature sets and compute a ratio-based ($\frac{|OSS \cap BIN|}{|OSS|}$ or $\frac{|BIN \cap OSS|}{|BIN|}$) similarity score to detect OSS usage. However, designing a large-scale similarity measurement system to accurately detect OSS reuse in app binaries presents its own set of challenges.

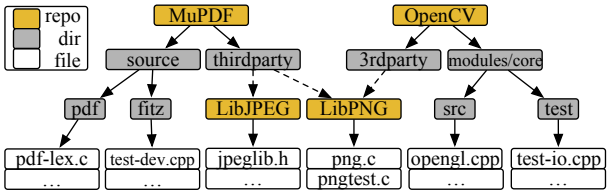


Figure 1: Real-world examples illustrating third-party code clones across OSS source repos. Various node types are highlighted using different colors.

3.4.1 ***Challenges.*** Here we first identify all the challenges we faced and follow up with the mechanisms we introduced for addressing them.

Internal code clones. A known advantage of using OSS is code reuse. OSS developers frequently reuse third-party OSS sources to leverage existing functionality. Reused code is often cloned and maintained internally, as a part of the OSS development sources (e.g., to allow easy customizations, to ensure compatibility, etc.). We refer to such nested third-party OSS clones as *internal code clones*. Internal code cloning results in high code duplication across OSS sources [62]. Therefore, a naïve database of OSS sources for similarity search will not only impose high hardware requirements, thereby hurting the system scalability, but also cause OSSPolice to report false positive matches against the internal third-party code clones. To understand why, let us look at source layouts of two popular C/C++ OSS sources, namely `MuPDF` and `OpenCV` as depicted in Figure 1. Both the repos contain code clones of `LibPNG` as a part of their source trees. Consequently, when trying to match features from `LibPNG` binary against `LibPNG,MuPDF,OpenCV` sources, all three of them will be reported as matches, although `LibPNG` is the only true positive match. Such false positives can result into incorrect license violations if the true and the reported matched repos are under different software licenses.

Partial OSS Builds. App developers may also choose to include only partial functionality from an OSS. For example, sources that are specific to one machine architecture (e.g., say x86) will not be compiled into a binary targeted for a different architecture (e.g., arm). Many C/C++ OSS sources provide `configure` options to selectively enable/disable architecture-specific functionality. Similarly, some OSS sources may also contain source files and directories that are not compiled into the target binary, such as `examples` and `testsuite`. While such unused sources could potentially be identified by analyzing build scripts (e.g., `gradle`, `Makefile`, etc.), there exists a number of build automation tools that will have to be supported by OSSPolice in order to correctly parse the build scripts and filter out unused parts; yet, the process may remain error-prone. Moreover, commonly used app shrinking tools, such as ProGuard analyze Java dex bytecode and remove unused classes, fields, and methods. While the binary remains functionally equivalent in such cases, number of features preserved from source to binary may, however, decrease significantly. We call these binaries *partially built binaries*. When comparing features from such a binary (*BIN*) with features (*OSS*) from the corresponding OSS sources, the matching ratio ($\frac{|BIN \cap OSS|}{|OSS|}$) can be arbitrarily low even if all the elements

from *BIN* are found in *OSS*. In fact, the more number of unused features are detected, the lower is the matching score, indicating a false negative match.

Fused app binaries. During the app build process, multiple binaries from disparate OSS sources could be tightly coupled together to generate a single app binary. For example, all Java class files in Android app, including any imported OSS jars are compiled into a single dex bytecode file (`classes.dex`). Similarly, multiple native libraries built from various C/C++ OSS sources, could be statically linked into a single shared library, thus blurring the boundaries between them. In such multi-binary files, features across multiple OSS components are effectively fused into a superset. We refer to them as *fused binaries*. As such, in the example depicted by Figure 1, `MuPDF` binary will also contain features from `LibJPEG`. As a result, when matching fused feature set (*BIN*) against a set of features (*OSS*) from a single OSS, the matching ratio ($\frac{|BIN \cap OSS|}{|BIN|}$) will be arbitrarily low even though *BIN* includes all the elements of *OSS*. In fact, the more number of disparate binaries are fused together, the lower is the matching score, resulting into false negatives.

3.4.2 ***Mechanisms.*** For efficient and scalable lookup during similarity comparison, OSSPolice maintains an indexing database of features extracted from OSS sources. An intuitive approach to indexing OSS sources is to consider each OSS as a document and its features as words, and create a direct (inverted) mapping of features to the target OSS (document). Figure 2a depicts the layout of such an indexing database. BAT [39] uses a similar scheme to maintain a database of features (string literals) extracted from OSS sources. However, this approach assumes that each OSS (document) is unique, and fails to consider large code duplication across OSS sources due to internal code cloning (§3.4.1). Consequently, such a naïve indexing scheme not only causes high false positives matches against internally cloned third-party OSS sources, but also imposes high storage requirements and does not scale as number of OSS to be indexed grows. Indexing multiple versions of OSS to enable version pinpointing further adds to the problem of code duplication.

We address the aforementioned challenges by tapping into the structurally rich tree-like layout of OSS sources. We will use the OSS source repo layouts in Figure 1 throughout this section for illustration purposes. The key observation that we make is that OSS developers typically follow the best practices of software development to improve collaboration and allow faster development. Hence, OSS sources are well organized in a modular and hierarchical fashion for easy maintainability. For instance, source files (e.g., a C/C++ or Java class file) typically encapsulates related functions. Directory (dir) nodes at each level of the source tree cluster all related child files and dirs together. Referring to our example layout in Figure 1, we can see that `src` and `source` dirs in `OpenCV` and `MuPDF`, respectively group all related source files and dirs under them. Similarly, internal code clones of third-party OSS (e.g., `LibPNG` and `LibJPEG`) are maintained in separate dirs (`thirdparty` and `3rdparty`, respectively). We utilize this property to perform ratio-based feature matching against each file or dir node (i.e., $\frac{|BIN \cap NODE|}{|NODE|}$) along the OSS source tree hierarchy as opposed to matching against the entire OSS repo (i.e., $\frac{|BIN \cap OSS|}{|OSS|}$), which may result in low accuracy in case of partial OSS reuse (§3.4.1). Specifically, if the ratio-based feature matching reports a high score against a node n (e.g., `LibPNG`) at a particular level l in the OSS source tree hierarchy, but reports a low aggregated score when matched against one of n's parent nodes p (e.g., `OpenCV`) at level $> l$, then we only report a match against node n (i.e., `LibPNG`), but not against the parent p (i.e., `OpenCV`) or any siblings at the same level. In this example, the matched OSS path reported by OSSPolice would be `OpenCV/LibPNG`.

To detect internal clones and filter out spurious matches against them, we apply multiple additional heuristics that leverage the modularized layout of OSS sources. During indexing we visit each dir node n in OSS sources and check for the presence of common software development files, such as `LICENSE` or `COPYING` (OSS licensing terms), `CREDITS` (acknowledgements), and `CHANGELOG` (software change history). These files are typically placed in the top-level source dir of OSS project repos. C/C++ OSS sources also typically host build automation scripts (e.g., `configure` and `autogen` in top-level source dirs. As such, cloned third-party OSS sources are likely to retain these files, which can be used to identify internal OSS clones. However, since some OSS sources may not be well organized, we further leverage the large code duplication across OSS sources resulting from OSS reuse to identify such internal clones. The observation we make is that due to OSS reuse, dir nodes (n) of commonly reused OSS sources will have multiple parents p in our database in contrast to unique OSS source dirs (e.g., `MuPDF/source/pdf`). This helps us identify all popular OSS clones in our database. All identified clones are further annotated so that they can be filtered out during matching phase in order to minimize false positives (see matching rules in §3.4.4).

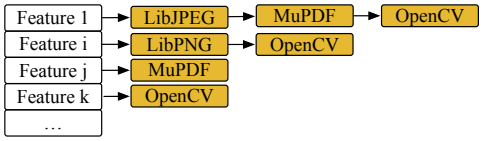
3.4.3 ***Hierarchical Indexing.*** We devise a novel *hierarchical indexing* scheme that retains the structured hierarchical layout of OSS sources (depicted in Algorithm 1). Specifically, instead of creating a direct mapping of features to the target OSS (i.e. the top-level dir in the OSS source tree), we map features to their immediate parent nodes (i.e., files and middle-level dirs). Figure 2 shows the layout of our indexing database constructed from OSS sources in Figure 1. We use this figure to walk through the steps to index an OSS. We populate an OSS in our indexing database, by separately processing each node (feature, file, or dir) in its source tree in a bottom-up fashion, starting from the leaf nodes that represent features (e.g., strings, functions, etc.). In order to retain the structured layout of OSS sources, we treat identifiers of parent nodes (i.e., files and dirs) as features, which are further indexed for efficient lookup. We refer to them as *hierarchical features*. At each level l of OSS source hierarchy, for a given node n, we create two types of mappings for each feature f under it: inverted mapping of f to n (immediate parent at level l) and straight mapping of n to f. Given a feature, the first mapping allows us to quickly find its matching parents, whereas we use the latter to perform ratio-based similarity detection. Our hierarchical indexing scheme efficiently captures uniqueness of features at each level of hierarchy. For example, after indexing we can know that features in `LibPNG` are contained in source dir `LibPNG`, which in turn is contained in multiple nodes, such as `3rdparty` in `OpenCV` and `thirdparty` in MuPDF.

We take advantage of internal OSS clones, to perform code deduplication for efficient use of hardware resources during indexing.

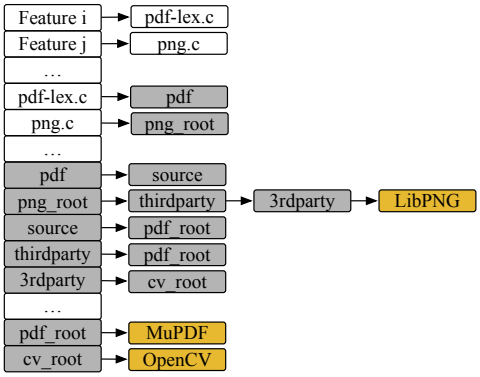
Algorithm 1 Pseudo code for hierarchical indexing algorithm

```
1:  procedure IndexRepo(repoRoot, repoInfo)
2:      file2Features ← ∅
3:      for file ∈ repoRoot do
4:          file2Features[file] ← ClangParser(file)
5:      path2Id ← ∅
6:      dir2Features ← ∅
7:      dir2Children ← ∅
8:      for (file, features) ∈ file2Features do
9:          path2Id[file] ← Simhash(features)
10:         for feat ∈ features do
11:             UpdateIndexDB(MD5(feat), path2Id[file])
12:         UpdateVersionDB(features, repoInfo)
13:         child ← file
14:         while child ≠ repoRoot do
15:             parent ← parentof(child)
16:             dir2Features[parent].add(features)
17:             dir2Children[parent].add(child)
18:             child ← parent
19:     for (dir, features) ∈ dir2Features do
20:         path2Id[dir] ← Simhash(features)
21:     IndexDir(repoRoot, dir2Children, path2Id)
22:     AddMappingToDB(path2Id[repoRoot], repoInfo)
23: procedure IndexDir(dir, dir2Children, path2Id)
24:     children ← dir2Children[dir]
25:     for child ∈ children do
26:         if ¬IsIndexed(child) then
27:             IndexDir(child, dir2Children, path2Id)
28:         else
29:             UpdateIndexDB(path2Id[child], path2Id[dir])
30: procedure UpdateIndexDB(f, n)
31:     parents′ ← GetParentsFromDB(f)
32:     if sizeof(parents′) ≥ T_{N_p} then
33:         continue
34:     if ∀n′ ∈ parents′ : H(n, n′) ≥ D then
35:         AddMappingToDB(f, n)
36:         AddMappingToDB(n, f)
```

To do so, we assign content-based identifiers to all the nodes in the source tree. We use 128-bit `md5` hash to generate such identifiers for features (leaf) nodes and use Simhash [19] algorithm to assign identifiers of parent (non-leaf) nodes, derived from the identifiers of all features (leaf nodes) under them. Simhash is a Locality Sensitive Hashing (LSH) algorithm that takes a high dimensional feature set and maps them to a fixed size hash. Hamming distance between hash values reveals cosine similarity between the original feature set. Since the Hamming distance between different identifiers reflects their similarity, before inserting a new mapping from feature f to parent n, we lookup whether f is already mapped to a similar parent node n' with Hamming distance less than a particular threshold D (i.e. $H(n, n') < D$). If such a parent node n' already exists, then we simply skip populating our indexing table with mappings for n'. Note that if n happens to be a large middle-level dir node, containing several source files and dirs within it (e.g., `thirdparty/LibPNG`) and



(a) Inverted flat indexing table mapping features to parent OSS.



(b) Inverted hierarchical indexing table mapping features to files, files to dirs, and dirs to parent repo. Colored boxes highlight repos and their root dirs.

Figure 2: Example illustrating OSS reuse and how hierarchical indexing take its advantage to reduce storage consumption.

is similar to an existing node (i.e., `3rdparty/LibPNG` in our database, then our content-based deduplication design achieves significant storage savings. Additionally, some features can be very popular. For instance, commonly occurring function names, such as *main* or *test*. Such features do not contribute to the uniqueness of an OSS. Worse yet, their long list of parent mappings (f to n) wastes storage space and increases search time. Therefore, we put a threshold on the maximum number of parent nodes for each child node (T_{N_p}).

Additionally, to enable accurate version pinpointing, we track unique features across OSS versions for each OSS in the indexing phase. This is separately maintained using two lists ($List_{overall}$ contains all features ever appeared in an OSS and $List_{unique}$ records unique features in each version) because with the benefit of deduplication based on similarity in the indexing phase, we also lose track of the uniqueness among similar nodes.

3.4.4 ***Hierarchical Matching.*** Our matching algorithm (depicted in Algorithm 2) leverages the OSS layout information preserved in indexing table for improving the accuracy of ratio-based similarity detection and filtering out duplicate OSS sources. In order to do so, we use a TF-IDF metric that assigns a higher score to the unique part of each parent node (files and dirs) and penalizes the non-unique part.

$$NormScore(p) = \frac{\sum_1^n f_{c_i} \times \log \frac{N_p}{1+Rc_i}}{\sum_1^n F_{c_i} \times \log \frac{N_p}{1+Rc_i}} \quad (1)$$

TF-IDF based metric. Let c denote child nodes, p denote parent nodes in the hierarchical indexing structure and N_p denote the total number of parent type nodes in the database. Let f_{c_i}, F_{c_i} and

Algorithm 2 Pseudo code for hierarchical matching algorithm

```
1:  procedure MatchBinary(binary)
2:      features ← ElfParser(binary)
3:      repos ← ∅
4:      while sizeof(features) > 0 do
5:          parents ← GetParentsFromDB(features)
6:          for p ∈ parents do
7:              if IsMatchingRepo(p) then
8:                  repos.add(p)
9:          p2Children ← ∅
10:         for p ∈ parents do
11:             p2Children[p] ← GetChildrenFromDB(p)
12:         p2NormScore ← ∅
13:         p2CumScore ← ∅
14:         for (p, children) ∈ p2Children do
15:             p2NormScore[p] ← NormScore(p, children)
16:             p2CumScore[p] ← CumScore(p, children)
17:         features ← ∅
18:         for p ∈ parents do
19:             if ¬MatchingRules(p) then
20:                 continue
21:             ns ← p2NormScore[p]
22:             cs ← p2CumScore[p]
23:             if ns ≥ T_NormScore ∧ cs ≥ T_CumScore then
24:                 features.add(p)
25:     versions ← SearchVersionDB(repos, features)
26:     return repos, versions
```

R_{c_i} denote number of matching features, number of total features and number of matching parent nodes (references) of the i-th child node, respectively. We then define log $\frac{N_p}{1+Rc_i}$ as IDF of the i-th child, measuring its importance to the parent node. Finally, we weigh each child using their IDF and define the weighted matching ratio as *NormScore* in Equation 1.

When matching against the indexing table, we first query features to get files, then query files to get dirs, and so on. After every round of query, we use *NormScore* to assign higher weights to unique parts of a parent node and filter these parent nodes for next round of query based on *NormScore*. With this normalization score, when we search binary of LibPNG, we can achieve a close to 1.0 score, but when we move up from LibPNG to `3rdparty` in OpenCV, the score significantly drops, and we can conclude a matching of LibPNG. Additionally, we also track total number of matched features, denoted as *CumScore*, to complement *NormScore*, since the latter only tracks matched ratio, whereas the former shows matched count. With the rich information extracted in indexing phase and the defined metrics, we apply the following **matching rules** to filter out false positives:

- Skip dirs that have license, since they are likely to be third-party OSS Clones.
- Skip source files that matches low ratio of functions or header files that matches low ratio of features, since they are likely to be tests, examples or unused code (e.g. partial builds).

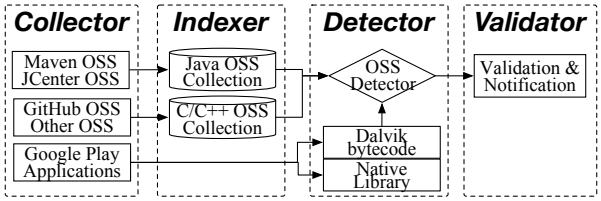


Figure 3: OSSPolice architecture and workflow

- Skip popular files/dirs by checking whether they are much more popular than the siblings, where popularity refers to number of matching parent nodes for each node (R_{c_i}).

Based on the detected OSS, we then compare the features from the app binary with the unique features across OSS versions to identify the matched OSS version. However, in practice, we find that unique features may cross match. For example, version string "2.0.0" from OkHTTP may match the version "2.0.0" of MoPub, while the actual matched version of MoPub is "3.0.0". To address this issue, we leverage co-location information preserved in the binary and indexing table (bi-directional mapping between n and f), and considers a unique feature as *valid* if all the other features in the same file/class also matches.

4 ARCHITECTURE

OSSPolice is written mostly in Python. This allows us to reuse existing production-quality tools within the language ecosystem. In particular, we use Celery [18] job scheduler for distributing work to multiple servers, Scrapy [75] for efficient crawling of OSS repos, and Redis distributed key-value cluster [70] for storing and querying indexing result.

Figure 3 depicts OSSPolice workflow. It consists of four modules, namely OSSCollector, Indexer, Detector, and Validator. Each module has an extensible plugin-based design to incorporate additional functionality as need. Here we briefly describe the function of each module.

OSSCollector. Our OSSCollector module is responsible for crawling multiple OSS hosting web services and downloading source repos or Java artifacts. We use Scrapy [75] web crawling framework. OSSCollector currently can only collect OSS from popular C/C++ source code hosting webservices, such as GitHub [34] and commonly used centralized webservices for distributing Java bytecode (artifacts), such as Maven [83] and JCenter [14]. However, due to an extensible design of OSSCollector, support for other hosting services, such as Bitbucket [6], SourceForge [78], and Sonatype [77] can be easily added.

When a new repo is discovered, OSSCollector first collects its metadata, such as software name, unique repo identifier, repo size, its popularity, programming languages used, number of lines of code, and details of available release versions (e.g., version identifier, software license, date created, etc.). Collected metadata is passed through additional filters to evaluate if an OSS repo should be downloaded for indexing. Based on the metadata filters, OSSCollector either skips the repo or downloads it and notifies Indexer to start processing it. Our current prototype deploys filters based on

three constraints: OSS popularity, license type, and vulnerability score.

We use Fossology[31], an open-source tool from HP, to extract and identify software licenses of OSS repos by examining license-like files in root directory of GitHub repos and project description file, namely *pom.xml* for Java artifacts. OSSCollector currently works only with GPL/AGPL OSS sources.

OSSCollector also collects vulnerability information for each OSS by transforming the software names into Common Platform Enumeration (CPE) format [20] and querying cve-search [58] to get a detailed list of all related Common Vulnerabilities and Exposures (CVE) vulnerabilities, including CVE id, its description, Common Vulnerability Scoring System (CVSS) score, affected versions, etc. OSSCollector further filters out CVEs based on their CVSS score and only retains CVEs with CVSS score higher than 4.0, which we refer to as **Severe CVEs**. This is done to limit the focus of this study to only detecting the use of OSS versions that are affected with critical vulnerabilities.

OSSCollector only downloads software that is either popular or is being used by at least one FDroid [29] app, which makes our evaluation dataset (described in §5). Each GitHub repo is attributed with *stargazers count* and *fork count*, indicating approximated number of users interested in it and number of times its copy has been created, respectively. We use these attributes to determine popularity of a GitHub repo. In particular, we downloaded Github repos with more than 100 stargazers to form our C/C++ OSS Collection ($OSS_{C/C++}$), which consisted of 3,119 repos and 60,450 OSS versions. Popularity information, however, is not available from Maven and is available only for a few Java artifacts from JCenter in the form of total number of downloads. This is because JCenter OSS developers may optionally choose to hide the download statistics[1]. Therefore, while compiling a list of popular Java software, we included additional sources, such as MvnRepository [64] AppBrain [4], and Android Studio [35]. We narrowed down to Java software artifacts that received more than 5K downloads, resulting in our Java OSS Collection (OSS_{Java}) with 4,777 artifacts and 77,308 artifact versions.

Of $OSS_{C/C++}$, 896 repos were GPL/AGPL-licensed and 347 were vulnerable with 5,611 severe CVEs, whereas of OSS_{Java}, 110 repos were GPL/AGPL-licensed and 83 were vulnerable with 452 severe CVE ids. The two datasets were used for evaluating the OSSPolice as well as reporting findings on Google Play Store apps.

AppCollector. It is responsible for crawling appstores and downloading app packages (`apks`) and their metadata, such as developer information, download count, and app description. Our current prototype only supports Google Play Store and borrows techniques from PlayDrone [87]. We used AppCollector to download 1.6M free Android mobile apps from Google Play Store in Dec, 2016.

Indexer. It extracts birthmark features from C/C++ source and Java jar/aar files in OSS repos to create an indexing database for efficient lookup. For feature extraction from C/C++ OSS, we use a Clang-based fuzzy parser to parses all source files (including headers). At first, we used a regular expression-based feature extractor. However, it failed to correctly report features in many cases. For instance, it failed to correctly extract strings or functions wrapped in a preprocessing macro.

Our parser retrieves string literals and function names from C/C++ source files. Additionally, it also extracts parameter types, class names, and namespaces for functions while parsing C++ source files since they are preserved in native libraries. Since parsing OSS files may fail due to missing configuration files and external dependencies, we designed the parser to infer the semantic context and insert dummy identifiers for missing data types. Further, we skip function bodies to speed up the parsing process as we use only function names and their arguments. To preserve the hierarchical layout of repos for content deduplication, we separately index source and header files. As a result, we are also able to easily skip common strings and functions defined in standard framework and system include files that tend to dilute matching results because of their popularity across several source repos. However, we do enable all `#include` directives, to resolve data types defined in header files and correctly identify function names and string literals that are wrapped in preprocessing macros, but are referenced in source files. Conditional preprocessing directives, such as `#if` and `#else` branch directives could also be skipped because of default config options. We, therefore, process the code within such directives separately, each forming a conditional group of extracted features Sometimes developers may comment out a certain piece of code within `#if 0` or `#elif 0`, which may be erroneous; we detect and skip such cases. We also skip non-Android and non-arm OS- and arch-specific macros.

For feature extraction from Java OSS, Indexer uses a Soot-based parser[55] for both source code and bytecode, which gives us the flexibility to support various kinds of inputs: jar, dex, apk, and source code. Indexer extracts features described in §3.3, including string constants, normalized classes, and centroids.

Detector. It first extracts the same types of features (§3.3) as the Indexer from mobile app binaries. We write a custom Python module around pyelftools [12], to extract strings and exported function names from native libraries, and use the same Soot-based parser to extract string constants, normalized classes and centroids. Detector then queries extracted features against the indexing table built by Indexer to find out a list of matched OSS versions. Detector selectively report these OSS version usage to Validator based on their license and vulnerability annotations. In particular, Detector reports usage of GPL/AGPL-licensed OSS as potential license violations, and usage of OSS version annotated with at least one Severe CVE as vulnerable usage.

Validator. It performs different checks based on the detected OSS versions. In the GPL/AGPL-violation scenario, it uses developer's information from Google Play Store, searches through app description and the developer's website for source code hosting links (e.g. GitHub). If found, it compares the similarity of app binary with the hosted source code to determine if the hosted code matches indeed is a match. If the Validator fails to find hosting links or if

[1]Developers may distribute multiple Java software and expose their download statistics selectively on JCenter. For example, apache owns both commons-vfs2 and commons-compress, but only chooses to disclose the download count for the former (13,478) and hides it for the latter although both of them are popular.

the similarity match fails, it reports the app as a potential violator of GPL/AGPL licensing terms. In case of vulnerable OSS detection, Validator simply retains the OSS versions that matched with unique features, and presents vulnerability details, such as OSS version and CVE ids to the user. If no unique features matched, it simply ranks the detected OSS versions based on their TF-IDF score. However, fine-grained function-level features (e.g., intra-procedural graph) can be extracted from both OSS sources and app binaries to increase the accuracy of version pinpointing at the cost of higher consumption of system resources (CPU, memory, etc.) and increased search time. We leave this for future work.

5 EVALUATION

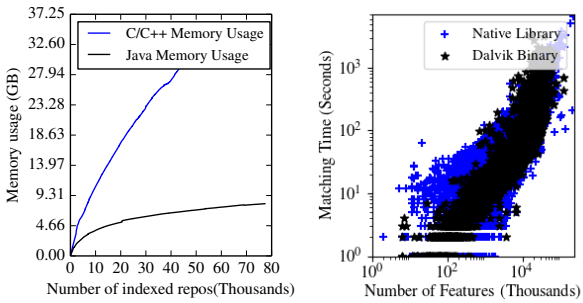
In this section, we first present the performance and scalability evaluation results of OSSPolice to show that it can efficiently match millions of app binaries against hundreds of thousands of OSS source repos. We then follow up with the accuracy analysis of OSSPolice using FDroid [29] open-sourced apps (for ground truth) to demonstrate that it can accurately detect OSS versions being used even in the presence of internal code clones, partially built binaries, and fused binaries (§3.4.1). For comparative analysis, we also report accuracy of BAT [39] and LibScout [7] since they are the state-of-the-art tools for OSS reuse detection, closest to ours.

5.1 Performance and Scalability

We deployed OSSPolice on ten servers, each with 16-core Intel Xeon CPU E5-2673 v3 @ 2.40GHz, 56GB memory, and 4TB drives.

Indexing. To evaluate the scalability of OSSPolice, we indexed a total of 137,758 OSS repos (60,450 C/C++ and 77,308 Java). We short-listed them because of their high popularity (§4). While indexing, we empirically set Simhash distance threshold (D) to 5 (§3.4.3) and number of maximum parent nodes (T_{N_p}) for each child node to 2,000 (§3.4). We present the change in memory consumption with the number of indexed repos in Figure 4a. As the figure demonstrates, the memory consumption grows sub-linearly due to content deduplication, suggesting that OSSPolice can easily be scaled to index more OSS repos.

At the end of indexing, OSSPolice processed 13 million C/C++ source files and 31 million Java classes, which amounts to more than 2 billion lines of C/C++ source code and 500 million lines of Java bytecode instructions, respectively. The total number of entries (keys) in the Redis database reached around 44 million and 9 million and the database grew to 30GB and 9GB for C/C++ and Java OSS, respectively. The number of entries created for C/C++ indexing table was higher than Java because C/C++ repos are generally larger in size and include auxiliary sources, such as tests, examples, and third party code, whereas Java bytecode files do not contain such auxiliary sources. On average, extracting all types of features described in Table 1 and indexing a source repo take 1,000 and 40 seconds for C/C++ and Java OSS, respectively. For C/C++ OSS, the majority of indexing time is spent in parsing source files for feature extraction. This is because the current implementation of our Clang parser is single-threaded and not optimized to include precompiled headers. Thus, it recompiles common headers for every source



(a) Memory consumption. **(b) OSS detection time.**

Figure 4: OSSPolice indexing and detection scalability. (a) shows memory consumption of indexing database over time and (b) shows how number of features in an app affects the detection time.

file. We expect that parallelizing the parser and adding support for precompiled headers will substantially improve its performance. However, we leave that for future work. In comparison, indexing time of Java OSS first increases and then remains stable because the majority of indexing time is spent on content deduplication, where number of similarity comparisons first grows with number of indexing nodes, but later reaches the limit of maximum parent nodes T_{N_p}. Our Soot-based [55] feature extractor is fast because it is multi-threaded and works directly on the precompiled jar packages.

Detection Time. A typical phenomenon in similarity detection schemes is that as the app grows bigger and more complex, the time taken to detect its similarity can increase exponentially, making these schemes unsuitable for handling large and complex apps. To test whether this limitation applies to OSSPolice, we randomly sampled 10,000 Android apps from Google Play Store dataset and queried them against our OSS database. Figure 4b shows the relationship between time taken by OSSPolice to analyze them for OSS reuse and number of features found in the selected app binaries (representative of app complexity). As seen from the plot, there is a linear relationship between the number features and the detection time; 80% of Dalvik binary and native library detection queries finish within 100 and 200 seconds, respectively, thus making OSSPolice suitable for analyzing apps at Google Play Store scale.

5.2 Accuracy

In order to evaluate the accuracy of OSSPolice in detecting OSS binary clones in Android apps, one needs a labeled mapping of apps to OSS usage for ground truth. However, no such dataset is publicly available from previous works. Randomly selecting binaries from actual dataset and labeling them for ground truth may include obfuscated and stripped binaries, rendering the labeling process error-prone. We, therefore, decided to use FDroid apps since their source code and binaries are both publicly available. FDroid hosted a total of 4469 apps at the time of collection (Feb, 2017). Of those, 579 apps contained at least one native library. We labeled C/C++ OSS by manually analyzing the source code and subsequently validating their presence in app binaries by collecting informative strings and function names. For instance, `LibPNG` sources were confirmed by cross-checking whether the function names in the app binaries began with prefix `png_`. Java OSS labels were generated by parsing the

OSS Labels	# Uses	OSSPolice				BAT [39]			
		P(%)	R(%)	VM	VP'(%)	P(%)	R(%)	VM	VP'(%)
$FDroid_{C/C++}$	295	82	87	55/67	82	75	61		
OSS Labels	# Uses	OSSPolice				LibScout[7]			
		P(%)	R(%)	VM	VP'(%)	P(%)	R(%)	VM	VP'(%)
$FDroid_{Java}$	7,055	89	92	478/520	92	92	71	295/320	92

P, R, VM, VP' refers to Precision, Recall, Version Match Results and Version Precision for OSS with unique feature/profile matches.

Table 2: Accuracy of OSSPolice and comparison with LibScout and BAT.

app build scripts, such as Maven *pom.xml* and Gradle *gradle.build* files that list app build dependencies. However, the specified build dependencies may further depend on more libraries, making the labels incomplete. For example, MoPub package is known to contain string *mopub-intent-ad-report*. Therefore, we validated the labels by checking package names and strings in the jars.

We labeled a total of 295 C/C++ OSS uses (56 distinct), denoted as $FDroid_{C/C++}$ and 7,055 Java OSS uses (279 distinct), denoted as $FDroid_{Java}$. We then queried FDroid app binaries against our indexing database from §5.1, and adjusted thresholds representing matched ratio ($T_{NormScore}$) for NormScore in Equation 1 and feature count ($T_{CumScore}$) for number of features matched to find a sweet spot between precision and recall. Our results indicate that OSSPolice achieves a precison of 82% and a recall of 87% when $T_{NormScore}$ = 0.5 and $T_{CumScore}$ = 50 for C/C++ OSS detection. Similarly, OSSPolice reported a precision of 89% and a recall of 92% when $T_{NormScore}$ = 0.7 and $T_{CumScore}$ = 100 for Java OSS detection. In cases where the target OSS is detected correctly and there were unique features matched, which amounted to 67 C/C++ and 520 Java OSS usage [2], OSSPolice achieved 82% and 92% version detection accuracy, respectively.

We inspected the results reported by OSSPolice and found that the main cause of false positives is the failure to correctly detect and filter out internal code clones, which may happen if the target OSS sources are not well organized (i.e., dirs containing code clones lack license and other common top-level software development files §3.4.2) and the cloned OSS is not popular in our database (i.e., it is cloned by only a few OSS repos, resulting in a small number of parent nodes). We found that false negatives in OSSPolice are reported only if partial functionality from an OSS is reused with too few features intact.

Comparative Analysis. Here we present a comparison of OSS detection accuracy results with that of BAT [39] and LibScout [7]. To do so, we first used BAT to generate a database of OSS in $FDroid_{C/C++}$ and LibScout to build library profiles for OSS in $FDroid_{Java}$. We queried FDroid apps binaries against BAT and LibScout databases. The results are shown in Table 2. Compared to BAT, OSSPolice reported more C/C++ OSS at a higher precision. Since BAT does not detect OSS versions, we only report version detection accuracy of OSSPolice in Table 2. To understand why OSSPolice outperforms BAT, we conducted further analysis and found that partially built libraries and internal code clones (§3.4.1) were the main causes for false negatives and false positives, respectively. Partially built libraries contain minimum part of OSS and have few features, making the matching score in BAT lower than the threshold. For example, all 41 uses of `JPEG` library were missed due to low number of features. Internal code clones cause BAT to match complex repos while only the reused OSS is present. For example, all 13 reported uses of `FreeType` also included 5 matches against `MuPDF` because `FreeType` is internally cloned by `MuPDF`, resulting in false positives.

Similar to LibScout, OSSPolice achieves comparable OSS precision (P) and version precision (VP'), but reports more number of OSS being used (R) and can detect more OSS versions (VM). We investigated the differences between OSSPolice and LibScout results and found that the main cause for false negatives of both system is unused code removal (§3.4.1). Nonetheless, OSSPolice outperformed LibScout. It is, however, worth noting that while LibScout uses only normalized classes to identified Java software reuse, we use two types of features, namely strings and normalized classes. Thus, compared to LibScout, OSSPolice works with a larger set of features, which is more indicative of OSS uses. For version pinpointing, LibScout reports OSS versions for both complete and incomplete profile matches. The versions returned in incomplete profile matches were mostly inaccurate and unfit for comparison. Hence, we only focus on results for complete profile matches (VM) in Table 2. OSSPolice pinpoints more OSS versions for two reasons: (1) OSSPolice extracts more features and can track uniqueness of more OSS versions. For example, some versions of Facebook and OkHttp can only be distinguished using version strings. (2) Version pinpointing in LibScout cannot handle unused code removal because no unique profile, which is defined as hash of Java package tree, will match in this case, since the package tree changes due of code removal. OSSPolice reports some false positives in version pinpointing as a result of cross matching of unique features (i.e. app with PrettyTime and Joda-time binaries may falsely report PrettyTime version using features from Joda-time).

6 FINDINGS

We used OSSPolice to conduct a large-scale OSS usage analysis in Google Play Store apps. This section presents our findings. In particular, we seek answers to the following questions.

- **OSS Usage.** What are some commonly used OSS? What are they used for? (§6.1)
- **OSS Licenses.** What are some commonly used software licenses for OSS? (§6.2).
- **License violations.** How many apps potentially violate OSS licensing terms? In general, what is the attitude of OSS developers towards violators? (§6.3).
- **Vulnerable OSS.** How commonly can one find vulnerable OSS versions in Android apps? How responsive app developers are to vulnerability disclosures? (§6.4).

Our dataset consists of 1.6 million free Android apps collected by crawling Google Play Store in December 2016. Our OSS database consisted of 3K popular C/C++ and 5K popular Java OSS.

[2] A large portion of labeled Java OSS were android support libraries (e.g. support-v4 and support-v13) whose versions are not distinguishable using features in OSSPolice.

Owner	Name	Type	License	# Uses
Square	OkHttp	Network	Apache	100,548
Facebook	Bolts Framework	Utils	BSD	97,350
Facebook	Facebook SDK	Social	FB Platform	85,742
Square	Picasso	Image	Apache	71,806
Apache	Http Components	Network	Apache	65,457
Sergey T.	Univ. Img Loader	Image	Apache	60,845
Square	Okio	Utils	Apache	56,997
Twitter4J	Twitter4J-Core	Social	BSD	54,045
Apache	Common Codec	Codec	Apache	46,530
SignPost	OAuth Library	Utils	Apache	43,647

Table 3: Top 10 detected Java OSS excluding Android and Google OSS.

Owner	Name	Type	License	# Uses
JPEG Group	JPEG	Codec	IJG	86,975
PNG Dev Group	LibPNG	Codec	LibPNG	78,117
Cocos2d	Cocos2d-X	Game	MIT	75,568
FreeType	FreeType	Font	FTL	65,109
OpenSSL	OpenSSL	Network	OpenSSL	50,489
OpenAL	OpenAL	Audio	LGPL	37,581
Libexpat	Expat	Codec	MIT/X	35,175
ArtifexSoftware	MuPDF	Viewer	GPL	34,055
LibTIFF	LibTIFF	Codec	BSD	33,721
Gailly and Adler	Zlib	Codec	Zlib	30,762

Table 4: Top 10 detected C/C++ OSS

Permission	License	Java	Native
Public	Public Domain	0.3%	1.9%
	WTFPL	0.1%	0.1%
Permissive	MIT	17.9%	28.5%
	BSD	5.7%	16.7%
	Apache	40.5%	7.0%
Weakly Protective	LGPL	4.2%	6.4%
Strong Protective	GPL	1.6%	30.8%
Network Protective	AGPL	0.1%	0.3%
-	Unclassified	27.2%	5.6%

Table 5: Software license distribution in Java- and native-based OSS

6.1 OSS Use in Mobile Apps

Table 3 and Table 4 list the top 10 detected usage of Java excluding Android and Google OSS (group id prefixed with com.android, com.google) and C/C++ OSS in Android apps, respectively. Our findings show that OSS usage distribution in Android apps is long-tailed; only a few OSS repos are very commonly used and a large number of OSS repos are used by only a few apps. Table 3 shows that various types of Java OSS are used, ranging from Utils to Social, while Table 4 shows that native OSS are mainly used for Codec and Game. In addition, we find that some high usage OSS is due to frequent indirect use. This means that the app developer will be building a library that he is not aware of the full dependency, and may lead him into legal issues or security harzards. For example, in Table 4, LibPNG is reused internally by Cocos2d.

6.2 Software Licenses in OSS

We first analyzed the popularity of different software licenses in Java- and native-based OSS projects. The license popularity result on 3K C/C++ and 5K Java OSS is shown in Table 5.

Consistent with previous research findings [86], the most popular software license for Java-based OSS is Apache license mostly due to the license choice of the Java programming platform and Android, which fall under this license. In comparison, most commonly used software licenses for C/C++ OSS are GPL and MIT. Therefore, Java-based OSS tend to be more permissive than C/C++ OSS.

Owner	Name	Type	# Uses
iText	iTextPDF	Codec	1,325
MySQL	Java Connector	Utils	396
greenDAO	Generator	Compiler	75
Proguard	Proguard	Compiler	27
Univ. of Waikato	Weka-Dev	Utils	15

Table 6: Top 5 most offended GPL/AGPL-licensed Java-based OSS projects.

Owner	Name	Type	# Uses
ArtifexSoftware	MuPDF	Codec	34,055
FFmpeg	FFmpeg†	Codec	4,326†
Teluu	PJSIP	Communication	2,113
VideoLan	VLC and X264	Codec	988
Belledonne Comm.	BZRTP	Communication	356

Table 7: Top 5 most offended GPL/AGPL-licensed native-based OSS projects. † shows only GPL uses of all FFmpeg, which can be either LGPL or GPL

6.3 License Violations

As discussed in §5.2, we believe that a similarity score of 0.5 or higher with more than 50 matching features would generate a very few false positives while detecting the presence of a C/C++ OSS component in an Android app. Similarly, a score of 0.7 or higher with more than 100 matching features would generate a very few false positives while detecting the presence of a Java OSS component. However, given that GPL/AGPL license violation is a strong claim that could result in severe legal consequences, we chose to be conservative and adjusted the similarity threshold for *NormScore* (§3.4.4) to 0.7 and *CumScore* (§3.4.4) to 200 for C/C++ OSS, and to 0.8 and 400 for Java OSS. Under these stricter conditions, OSSPolice detected around 40K apps using at least one GPL/AGPL-licensed C/C++-based OSS component while 2K apps using at least one GPL/AGPL-licensed Java-based OSS component. The *Validator* filtered out only 55 apps as there are clear indications that these apps are open-sourced, flagging most apps as potential violators of GPL/AGPL licensing terms. The most offended Java and C/C++ OSS projects under GPL/AGPL license are shown in Table 6 and Table 7, respectively.

Similar to the distribution of OSS usage per app, the distribution of OSS under GPL and AGPL licenses is long-tailed, with only a few OSS being used in many apps; whereas a large number of OSS see only one or two violating apps. In terms of GPL/AGPL-licensed OSS usage in apps, the maximum we saw is 1,325 iTextPDF for Java OSS and 34,055 MuPDF for C/C++ OSS, both are PDF related libraries. To understand why developers are using these libraries, we collect popular PDF libraries that support both rendering and editing over the Internet and found that most of them were either GPL/AGPL licensed or not free. In particular, the top two PDF libraries listed in [84], RadaeePDF SDK and PDFNet SDK both paid PDF rendering/editing engines. Therefore, our findings suggest that app developers use these iTextPDF and MuPDF due to lack of free alternatives.

OSS developer responses. We emailed a few corporate developers of the OSS victims (`MuPDF`, `PJSIP`, `FFmpeg`, `VideoLAN`, and `iText`), each with a list of apps that potentially violate their copyrights. The reason behind it is to filter out their legitimate customers because many of these companies use dual-license model for their software, under which the open-sourced variant (e.g., GPL license) requires any derivative work to be open-sourced, and, therefore, a

separate commercial license is needed for commercial use without open-sourcing. Developers of a derivative work can choose to open source their code under the same license or pay these companies to avoid source code disclosure. For instance, Dropbox and HP are licensees of MuPDF.

We received responses from these companies. PJSIP replied that they have Non-Disclosure Agreement (NDA) with their customers and cannot reveal their information. VideoLAN and FFmpeg both showed interest in the list, but FFmpeg developers mentioned that they lack resources to enforce license compliance. MuPDF requested our list and returned a filtered list of app developers that use their software, but are not their customers. In addition, MuPDF mentioned that even identifying legitimate customers is not straightforward because they sub-license MuPDF to Adobe and all Adobe licensees are also legally permitted to use MuPDF without open-sourcing. iText, however, did not reply to our email.

Awareness of OSS licensing issues. From the results reported by Validator, it is difficult to draw conclusions whether developers are violating OSS licensing terms, nor can we tell whether they are infringing intentionally or inadvertently because developers may display link to source code within their app or on random websites. We notice that GPL/AGPL requires that if one distributes derivative works of GPL/AGPL-licensed software, then they must provide the source code upon request. Therefore, for further insights, we randomly emailed 10 developers of the apps we found to have violated GPL/AGPL licensing terms and requested access to their source code. Unfortunately, at the time of writing, none of them provided their code. One of these developer, however, had claimed in the description on Google Play Store that their app is licensed under GPL: [3]

> *Weird Voice is based on CSipSimple and is licensed under GNU GPL v3. More information in the app.*

Nonetheless, when we emailed them for access to their code, the response received redirected us to a GitHub page of another app that they claimed to be "99%" similar and refused to release the sources of their own app. From these cases, we can see that people are not aware of the specific requirements of the GPL/AGPL license, and currently there is no appropriate way to enforce GPL/AGPL compliance.

6.4 Vulnerable OSS Versions

In order to report vulnerable OSS version usage results, we retain a subset of the detected results with at least one unique feature matched, which is shown to have reasonable precision in version pinpointing in §5.2. Since Google has launched an App Security Improvement program (ASIP) [41] to help developers improve the security of their apps by checking vulnerable code usage, we classify detected OSS versions as vulnerable based on ASIP description, if the OSS is also listed by ASIP (e.g., LibPNG). For an OSS not listed by ASIP, we classify its version as vulnerable if it is tagged with at least one **Severe CVE** (defined as CVSS score greater than 4.0 in §4). We present six C/C++ OSS and four Java OSS with most vulnerabilities in Table 8. Of those, LibPNG, OpenSSL and MoPub are also tracked by ASIP. As shown in Table 8, the number of apps that use vulnerable versions of LibPNG and OkHttp amounts to more than 40K and 39K, respectively. To understand their impact on users, we further break down these apps by the number of downloads. Our findings indicate that 20% of these apps have received over 10K downloads.

From **Language** column in Table 8, we can see that there are more vulnerable C/C++ OSS uses than Java, despite the fact that Java OSS are popular as in Table 3. This is because most Java OSS are not tagged with Server CVE ids.

Despite their measures towards security of apps, we found more than 40K, 27K, and 2K vulnerable uses of OSS that are tracked by ASIP, namely LibPNG, OpenSSL and MoPub, respectively.

 # ASIP Misses further shows number of apps that were only detected by OSSPolice as vulnerable, but were not tracked by ASIP. These numbers were obtained based on ASIP claim that Google Play Store would ban future app updates if the developers do not fix vulnerable OSS usage in their apps after the deadline, which was set as Sep 17, 2016 for LibPNG and Jul 11, 2016 for OpenSSL and MoPub. We assume that Google Play Store enforced the claimed policy and simply report the number of apps (downloaded in Dec, 2016) that were still flagged as vulnerable by OSSPolice and were updated after their respective ASIP deadlines. **# ASIP Misses** in Table 8 shows that ASIP missed at least 1,244 LibPNG and 4,919 OpenSSL cases compared to OSSPolice. For MoPub, no flagged apps were updated after the specified deadline. For further validation, we contacted Google by sending them a comprehensive list of vulnerable apps, including the ones missed by ASIP. Unfortunately, by the time of this writing we did not receive a response from them on it.

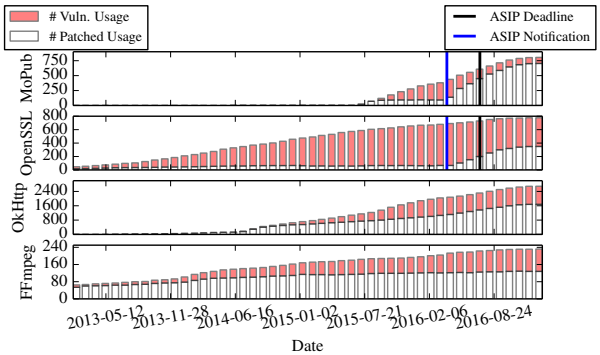


Figure 5: Selected 4 popular vulnerable libraries and longitudinal study of their usage by app developers.

Awareness of Vulnerable OSS uses. To understand how quickly and how frequently app developers adopt the patched OSS versions, and what makes them update their apps with these patched OSS versions, we conduct a longitudinal study of OSS usage by app developers. We selected top 10K apps from Google Play Store, downloaded their past versions. A total of 300K app versions were analyzed with OSSPolice to report all cases of vulnerable OSS usage. To get insight into the attitude of app developers towards vulnerable OSS usage, in particular, whether ASIP policy enforcement

[3] We found that app Voice changer calling (package com.weirdvoice) reuses PJSIP sources, which are licensed under GPL

Owner	Name	Language	# Vuln. Apps	% Vuln.	Latest Ver.	Vuln. Ver.	# Severe CVE	# ASIP Misses
PNG Dev Group	LibPNG	C/C++	40,902	52	1.6.28	1.0-1.0.65, 1.2-1.2.55, 1.3-1.4.18, 1.5-1.5.25	35, ASIP [44]	1,244
Square	OkHttp	Java	39,019	39	3.6.0	2.0-2.7.4 or 3.0-3.1.2	1	
Libexpat	Expat	C/C++	35,155	99	2.2.0	1.95.1-2.1.1	9	
LibTIFF	LibTIFF	C	27,117	80	4.0.7	3.4-4.0.3, 4.0.6	90	
OpenSSL	OpenSSL	C/C++	27,103	54	1.0.2k	1.0.1-1.0.1q, 1.0.2-1.0.2e	160, ASIP [43]	4,919
FreeType	FreeType	C/C++	21,762	34	2.7.1	<2.5.4	76	
FFmpeg	FFmpeg	C/C++	8,737	57	3.3	2.0-2.8.4	218	
MoPub	MoPub-SDK	Java	2,594	16	4.11.0	<4.4.0	0, ASIP [42]	0
Apache	Commons-Compress	Java	827	48	1.14	1.0-1.4	1	
Apache	Commons-Collections	Java	619	33	4.1	3.0-3.2.1, 4.0	1	

 # Severe CVE refers to CVEs that have more than 4.0 Common Vulnerability Scoring System (CVSS) score.
 # ASIP Misses refers to number of vulnerable apps updated after App Security Improvement Program's [41] deadline.

Table 8: Top 6 C/C++ and 4 Java vulnerable OSS.

can make them update their apps regularly, we selected two OSS (`OpenSSL` and `MoPub`) that were reported by ASIP and two (`FFmpeg` and `OkHttp`) that were only reported by OSSPolice as vulnerable and carried out a comparison. The results are shown in Figure 5. For `FFmpeg` and `OkHttp`, both patched and vulnerable usage increased over time. In comparison, usage of vulnerable versions of `OpenSSL` and `MoPub` kept increasing until ASIP notification date (i.e., when the app developers received emails from ASIP, apprising them of vulnerable OSS usage in their apps), but quickly drops after that. Such a pattern indicates that app developers slowly adopt patched OSS versions and even use old and vulnerable OSS in updated app versions or newly developed apps. Nevertheless, our findings suggest that ASIP can help developers identify security issues with their apps and force them to regularly update their apps to use patched OSS versions.

7 DISCUSSION

In this section, we discuss the limitations of OSSPolice, potential solutions, and future research directions.

License Compliance. OSSPolice focuses only on the technical aspects of license compliance engineering, such as OSS reuse detection, checking for a license copy in app installation package, and validating hosted source code. Therefore, only statistical evidence indicating potential license violation is reported to further help the app developers quickly identify true violations, but no concrete proof or legal conclusions are derived from the collected evidence. The reasons for this design choice are manyfold. First, several OSS are available under a dual license. Therefore, an app containing the OSS could be a case of legitimate use. Second, OSSPolice may fail to correctly detect source weblinks for an app because the current design only inspects app description and corresponding developer website for weblinks pointing to popular source code hosting services, such as GitHub and Bitbucket, as an indicator of open-sourcing. Furthermore, app developers may also choose to generate source code links dynamically in the app or simply host outside the checked open-source links.

App Obfuscation and Optimization. OSSPolice is designed to be resilient against simple and common app obfuscation techniques, such as identifier renaming in Java classes. However, advanced obfuscation may alter or even destroy features in app binaries. For example, string encryption will render all string constants in a binary ineffective for similarity comparison. Nevertheless, such techniques are generally used by malware writers to evade detection and are not common for benign apps because of their additional runtime overhead (e.g., encryption/decryption). However, should this become a problem, advanced obfuscation-resilient similarity detection mechanisms, such as data-dependence [25] or program-dependence [26] graph comparison can be used at the cost of higher consumption of system resources (CPU, memory, etc.) and search time.

To optimize their apps for size and faster loading, app developers may further remove unused OSS code or hide functions in native libraries, thereby reducing the size of the symbol table. OSSPolice may either fail to detect OSS in such libraries or report inaccurate results because of lack of enough syntactical features. While we found only 11.6% cases of such libraries (§3.3), we believe the system accuracy can be improved by augmenting with semantic features, such as control flow [28, 30] at the cost of increased detection time.

Version Pinpointing. It is possible that OSS source code might have very minimal changes across two releases. Given no unique features can be used to distinguish these versions, OSSPolice will return a sorted list of matched versions based on *NormScore* (§3.4.4). We believe that OSSPolice has achieved reasonable coverage because 83% of C/C++ and 41% of Java OSS versions can be uniquely identified using current features in OSSPolice. However, should this becomes an issue, OSSPolice can be plugged in to use fine-grained function-level features (e.g., intra-procedural graph) to further distinguish these versions.

More Programming Languages. OSSPolice currently supports only Java and C/C++-based OSS repos and app binaries because of their popularity. However, we are also aware that mobile apps nowadays use a more diverse set of programming languages. For example, apps built by PhoneGap [40] and Corona[24] tend to rely on many JavaScript and Lua libraries. We leave the support for these programming languages for future work.

8 CONCLUSION

In this paper, we presented OSSPolice, the first large-scale tool for mobile app developers to identify potential open-source license violations and 1-day security risks in their apps. OSSPolice taps into the structured and modularized layout of OSS sources and introduces hierarchical indexing scheme to achieve high efficiency and accuracy in comparing app binaries with hundreds of thousands of OSS sources (billions of lines of code). We applied OSSPolice to analyze 1.6M free Google Play Store apps and found that over

40K apps potentially violated GPL/AGPL licensing terms, and over 100K apps use known vulnerable versions of OSS. OSSPolice can also be deployed by app stores, such as Google Play Store to check and notify app developers of potential licensing issues and security risks in their apps and enforce policies. Source code of OSSPolice is available on GitHub (https://github.com/lingfennan/osspolice).

9 ACKNOWLEDGMENT

We thank the anonymous reviewers for their helpful feedback. This research was supported, in part, by the NSF under award CNS-0831300, CNS-1017265, DGE-1500084, CNS-1563848, SFS-1565523, CRI-1629851, and CNS-1704701, ONR under grant N00014-15-1-2162, N00014-16-1-2710, and N00014-17-1-2895, DARPA TC (No. DARPA FA8650-15-C-7556) and XD3 programs (No. DARPA HR0011-16-C-0059), and ETRI IITP/KEIT [B0101-17-0644].

REFERENCES

[1] A. Aiken. 2017. Moss: a system for detecting software plagiarism. (2017). http://theory.stanford.edu/~aiken/moss/
[2] Devdatta Akhawe. 2015. Security bug resolved in the Dropbox SDKs for Android. (2015). https://blogs.dropbox.com/developers/2015/03/security-bug-resolved-in-the-dropbox-sdks-for-android/
[3] Antepedia. 2017. Antepedia, Software Artifacts Knowledge Base. (2017). http://www.antepedia.com
[4] AppBrain. 2016. Android library statistics. (2016). http://www.appbrain.com/stats/libraries
[5] AppBrain. 2017. Number of Android applications. (2017). https://www.appbrain.com/stats/free-and-paid-android-applications
[6] Atlassian, Inc. 2016. Code, Manage, Collaborate. (2016). https://bitbucket.org
[7] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and its Security Applications. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. Vienna, Austria.
[8] Brenda S. Baker. 1995. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proceedings of the 2nd Working Conference on Reverse Engineering (WCRE)*. Toronto, Ontario, Canada.
[9] Brenda S. Baker. 1997. Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance. *SIAM J. Comput.* 26, 5 (Oct. 1997), 1343–1362.
[10] Brenda S. Baker and Udi Manber. 1998. Deducing Similarities in Java Sources from Bytecodes. In *Proceedings of the 1998 USENIX Annual Technical Conference (ATC)*. New Orleans, Louisiana.
[11] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance (ICSM)*. Bethesda, Maryland, USA.
[12] Eli Bendersky. 2016. Pure-python library for parsing ELF and DWARF. (2016). https://github.com/eliben/pyelftools
[13] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. 2014. Brahmastra: Driving Apps to Test the Security of Third-Party Components. In *Proceedings of the 23rd USENIX Security Symposium (Security)*. San Diego, CA.
[14] Bintray.com. 2016. JCenter is the place to find and share popular Apache Maven packages. (2016). https://bintray.com/bintray/jcenter
[15] Black Duck Software, Inc. 2016. Black Duck Protex Automate Open Source Compliance. (2016). https://www.blackducksoftware.com/products/protex
[16] Theodore Book, Adam Pridgen, and Dan S. Wallach. 2013. Longitudinal Analysis of Android Ad Library Permissions. In *Proceedings of the IEEE CS Security and Privacy Workshops (SPW)*. San Francisco, CA.
[17] Martial Bourquin, Andy King, and Edward Robbins. 2013. BinSlayer: Accurate Comparison of Binary Executables. In *Proceedings of the 13th ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. Rome, Italy.
[18] CeleryProject. 2016. Celery: Distributed Task Queue. (2016). http://www.celeryproject.org
[19] Moses S Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC)*. MontrÃĺal, QuÃĺbec, Canada.
[20] Brant A Cheikes, David Waltermire, and Karen Scarfone. 2011. Common platform enumeration: Naming specification version 2.3. *NIST Interagency Report 7695, NIST-IR* 7695 (2011).
[21] Eric Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. 2014. OAuth Demystified for Mobile Application Developers. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*. Scottsdale, Arizona.
[22] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. Hyderabad, India.
[23] Seokwoo Choi, Heewan Park, Hyun-Il Lim, and Taisook Han. 2007. A Static Birthmark of Binary Executables Based on API Call Structure. In *Proceedings of the 12th Advances in Computer Science Conference: computer and network security*. Doha, Qatar, 2–16.
[24] Corona Labs. 2016. Cross-Platform Mobile App Development for iOS, Android. (2016). https://coronalabs.com
[25] Jonathan Crussell, Clint Gibler, and Hao Chen. 2012. Attack of the Clones: Detecting Cloned Applications on Android Markets. In *Proceedings of the 17th European Symposium on Research in Computer Security (ESORICS)*. Pisa, Italy.
[26] Jonathan Crussell, Clint Gibler, and Hao Chen. 2015. AnDarwin: Scalable Detection of Android Application Clones Based on Semantics. *IEEE Transactions on Mobile Computing* 14, 10 (2015), 2007–2019.
[27] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. 1999. A Language Independent Approach for Detecting Duplicated Code. In *Proceedings of the International Conference on Software Maintenance (ICSM)*. Oxford, England, UK.
[28] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovre: Efficient cross-architecture identification of bugs in binary code. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.
[29] F-Droid Limited and Contributors. 2016. F-Droid. (2016). https://f-droid.org
[30] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-based Bug Search for Firmware Images. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. Vienna, Austria.
[31] FOSSology Workgroup. 2016. Open Source License Compliance by Open Source Software. (2016). https://www.fossology.org/
[32] Mark Gabel, Junfeng Yang, Yuan Yu, Moises Goldszmidt, and Zhendong Su. 2010. Scalable and Systematic Detection of Buggy Inconsistencies in Source Code. In *Proceedings of the 20th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Reno/Tahoe, Nevada, USA.
[33] Debin Gao, Michael K. Reiter, and Dawn Song. 2008. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *Proceedings of the 10th International Conference on Information and Communications Security*. Birmingham, UK.
[34] GitHub, Inc. 2016. How people build software. (2016). https://github.com/features
[35] Google Inc. 2016. Android Studio, The Official IDE for Android. (2016). https://developer.android.com/studio/index.html
[36] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. 2012. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*. Budapest, Hungary.
[37] Derrick Grover. 1989. *The Protection of Computer Software—its Technology and Applications*. Cambridge University Press, New York, NY, USA. 119–150 pages.
[38] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. 2012. Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*. Heraklion, Crete, Greece.
[39] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. 2011. Finding Software License Violations Through Binary Code Clone Detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR)*. Honolulu, HI.
[40] Adobe Systems Inc. 2016. Build amazing mobile apps powered by open web tech. (2016). http://phonegap.com
[41] Google Inc. 2016. App Security Improvement Program. (2016). https://developer.android.com/google/play/asi.html
[42] Google Inc. 2016. How to address MoPub vulnerabilities in your apps. (2016). https://support.google.com/faqs/answer/6345928
[43] Google Inc. 2016. How to address OpenSSL vulnerabilities in your apps. (2016). https://support.google.com/faqs/answer/6376725
[44] Google Inc. 2016. How to fix apps containing Libpng Vulnerability. (2016). https://support.google.com/faqs/answer/7011127
[45] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: finding unpatched code clones in entire os distributions. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.
[46] Jiyong Jang, David Brumley, and Shobha Venkataraman. 2011. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*. Chicago, Illinois.

[47] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*. Minneapolis, MN.

[48] J. Howard Johnson. 1993. Identifying Redundancy in Source Code Using Fingerprints. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*. Toronto, Ontario, Canada, 171–183.

[49] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002).

[50] Dongjin Kim, Seong je Cho, Sangchul Han, Minkyu Park, and Ilsun You. 2014. Open Source Software Detection using Function-level Static Software Birthmark. *Journal of Internet Services and Information Security (JISIS)* 4, 4 (2014), 25–37.

[51] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*. San Jose, CA.

[52] Raghavan Komondoor and Susan Horwitz. 2001. Using Slicing to Identify Duplication in Source Code. In *Proceedings of the 8th International Symposium on Static Analysis*. Paris, France.

[53] Mohit Kumar. 2014. Facebook SDK vulnerability puts millions of smartphone users' accounts at risk. (2014). http://thehackernews.com/2014/07/facebook-sdk-vulnerability-puts.html

[54] Eric Lafortune. 2016. ProGuard. (2016). http://proguard.sourceforge.net/

[55] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Proceedings of the 2011 Cetus Users and Compiler Infrastructure Workshop*. Galveston Island, TX.

[56] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. 2017. Libd: Scalable and precise third-party library detection in Android markets. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. Buenos Aires, Argentina.

[57] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. San Francisco, CA.

[58] Jason Long. 2016. cve-search - a tool to perform local searches for known vulnerabilities. (2016). http://cve-search.github.io/cve-search/

[59] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. Hong Kong.

[60] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering* PP, 99 (2017).

[61] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. Austin, TX.

[62] Audris Mockus. 2007. Large-Scale Code Reuse in Open Source Software. In *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*. Minneapolis, MN.

[63] Patrick Mutchler, Adam Doupe, John Mitchell, and Chris Kruegeland Giovanni Vigna. 2015. A Large-Scale Study of Mobile Web App Security. In *Proceedings of the Mobile Security Technologies (MoST)*. San Jose, CA.

[64] MvnRepository. 2016. Maven Repository: Search/Browse/Explore. (2016). https://mvnrepository.com

[65] Ginger Myles and Christian Collberg. 2004. Detecting software theft via whole program path birthmarks. In *International Conference on Information Security*. Palo Alto, California.

[66] Ginger Myles and Christian Collberg. 2005. K-gram Based Software Birthmarks. In *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC)*. Santa Fe, New Mexico.

[67] Annamalai Narayanan, Lihui Chen, and Chee Keong Chan. 2014. Addetect: Automated detection of android ad libraries using semantic analysis. In *Proceedings of the 9th Intelligent Sensors, Sensor Networks and Information Processing*. Singapore, Singapore.

[68] Matthias Neugschwandtner, Martina Lindorfer, and Christian Platzer. 2013. A View To A Kill: WebView Exploitation. In *Proceedings of the 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*. Washington, D.C.

[69] Ryan Paul. 2009. Cisco settles FSF GPL lawsuit, appoints compliance officer. (2009). http://arstechnica.com/information-technology/2009/05/cisco-settles-fsf-gpl-lawsuit-appoints-compliance-officer

[70] RedisLabs. 2016. Redis Cluster Specification. (2016). http://redis.io/topics/cluster-spec

[71] Franziska Roesner and Tadayoshi Kohno. 2013. Securing Embedded User Interfaces: Android and Beyond. In *Proceedings of the 22th USENIX Security Symposium (Security)*. Washington, DC.

[72] Inc Rogue Wave Software. 2016. Solve open source issues with full-stack enterprise support. (2016). http://www.roguewave.com/products-services/open-source-support

[73] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. 2009. Detecting Code Clones in Binary Executables. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. Chicago, IL.

[74] David Schuler and Valentin Dallmeier. 2006. Detecting software theft with api call sequence sets. In *Workshop Software Reengineering (WSR 2006)*. Bad-Honnef, Germany.

[75] ScrapingHub. 2016. Scrapy, A Fast and Powerful Scraping and Web Crawling Framework. (2016). https://scrapy.org

[76] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. 2012. AdSplit: Separating Smartphone Advertising from Applications. In *Proceedings of the 21st USENIX Security Symposium (Security)*. Bellevue, WA.

[77] Inc Sonatype. 2016. Sonatype Releases. (2016). https://oss.sonatype.org/content/repositories/releases/

[78] SourceForge.net. 2016. Find, Create, and Publish Open Source software for free. (2016). https://sourceforge.net

[79] Android Studio. 2016. Shrink Your Code and Resources. (2016). https://developer.android.com/studio/build/shrink-code.html

[80] Mengtao Sun and Gang Tan. 2014. NativeGuard: Protecting Android Applications from Third-Party Native Libraries. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*. Oxford, UK.

[81] Synopsys. 2017. Software Composition Analysis - Protecode. (2017). https://www.synopsys.com/software-integrity/products/software-composition-analysis.html

[82] Haruaki Tamada, Masahide Nakamura, and Akito Monden. 2004. Design and evaluation of birthmarks for detecting theft of Java programs. In *Proceedings of the IASTED IASTED International Conference on Software Engineering*. Innsbruck, Austria.

[83] The Apache Software Foundation. 2016. Apache Maven Project. (2016). https://maven.apache.org/index.html

[84] ToughDev. 2015. Comparison of popular PDF libraries on iOS and Android. (2015). http://www.toughdev.com/content/2015/02/comparison-of-popular-pdf-libraries-on-ios-and-android/

[85] Steven Vaughan. 2015. VMware sued for failure to comply with Linux license. (2015). http://www.zdnet.com/article/vmware-sued-for-failure-to-comply-with-linuxs-license

[86] Christopher Vendome. 2015. A Large Scale Study of License Usage on GitHub. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. Florence, Italy.

[87] Nicolas Viennot, Edward Garcia, and Jason Nieh. 2014. A Measurement Study of Google Play. In *Proceedings of the 2014 ACM SIGMETRICS Conference*. Austin, TX.

[88] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. 2015. WuKong: A Scalable and Accurate Two-Phase Approach to Android App Clone Detection. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. Baltimore, MA.

[89] Rui Wang, Yuchen Zhou, Shuo Chen, Shaz Qadeer, David Evans, and Yuri Gurevich. 2013. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In *Proceedings of the 22th USENIX Security Symposium (Security)*. Washington, DC.

[90] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. 2009. Behavior Based Software Theft Detection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*. Chicago, IL.

[91] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. 2009. Detecting software theft via system call based birthmarks. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. Honolulu, Hawaii, USA.

[92] Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du. 2014. Compac: Enforce Component-level Access Control in Android. In *Proceedings of the 4th Annual ACM Conference on Data and Applications Security and Privacy (CODASPY)*. San Antonio, TX.

[93] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. 2014. ViewDroid: Towards Obfuscation-Resilient Mobile Application Repackaging Detection. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*. Oxford, UK.

[94] Xiao Zhang, Amit Ahlawat, and Wenliang Du. 2013. AFrame: Isolating Advertisements from Mobile Applications in Android. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. New Orleans, LA.

[95] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. 2013. Fast, Scalable Detection of âĂIJPiggybackedâĂİ Mobile Applications. In *Proceedings of the 3rd Annual ACM Conference on Data and Applications Security and Privacy (CODASPY)*. San Antonio, TX.

[96] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. 2012. Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *Proceedings of the 2nd Annual ACM Conference on Data and Applications Security and Privacy (CODASPY)*. San Antonio, TX.

[97] Yuchen Zhou and David Evans. 2014. SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities. In *Proceedings of the 23rd USENIX Security Symposium (Security)*. San Diego, CA.

[98] Zynamics. 2017. zynamics.com - BinDiff. (2017). https://www.zynamics.com/bindiff.html