Precise and Efficient Patch Presence Test for Android Applications against Code Obfuscation

Zifan Xie[*][†]
Huazhong University of Science and Technology, China
xzff@hust.edu.cn

Ming Wen[*][†][‡]
Huazhong University of Science and Technology, China
mwenaa@hust.edu.cn

Haoxiang Jia[*][†]
Huazhong University of Science and Technology, China
haoxiangjia@hust.edu.cn

Xiaochen Guo[*][†]
Huazhong University of Science and Technology, China
xiaochenguo@hust.edu.cn

Xiaotong Huang[†]
Huazhong University of Science and Technology, China
zoyy@hust.edu.cn

Deqing Zou[*][†]
Huazhong University of Science and Technology, China
deqingzou@hust.edu.cn

Hai Jin[*][§]
Huazhong University of Science and Technology, China
hjin@hust.edu.cn

ABSTRACT

Third-party libraries (TPLs) are widely utilized by Android developers to implement new apps. Unfortunately, TPLs are often suffering from various vulnerabilities, which could be exploited by attackers to cause catastrophic consequences for app users. Therefore, testing whether a vulnerability has been patched in target apps is crucial. However, existing techniques are unable to effectively test patch presence for obfuscated apps while obfuscation is pervasive in practice. To address the new challenges introduced by code obfuscation, this study presents PHunter, which is a system that captures obfuscation-resilient semantic features of patch-related methods to identify the presence of the patch in target apps. Specifically, PHunter utilizes coarse-grained features to locate patch-related methods, and compares the fine-grained semantic similarity to determine whether the code has been patched. Extensive evaluations on 94 CVEs and 200 apps show that PHunter can outperform state-of-the-art tools, achieving an average accuracy of 97.1% with high efficiency and low false positive rates. Besides, PHunter is able to be resilient to different obfuscation strategies. More importantly, PHunter is useful in eliminating the false alarms generated by existing TPL detection tools. In particular, it can help reduce up to 25.2% of the false alarms with an accuracy of 95.3%.

CCS CONCEPTS

• **Software and its engineering** → *Software libraries and repositories*; • **Security and privacy** → **Software security engineering**.

KEYWORDS

Android Security, Patch Presence Test, Library Detection

ACM Reference Format:
Zifan Xie, Ming Wen, Haoxiang Jia, Xiaochen Guo, Xiaotong Huang, Deqing Zou, and Hai Jin. 2023. Precise and Efficient Patch Presence Test for Android Applications against Code Obfuscation. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598061

[*]Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Huazhong University of Science and Technology (HUST)
[†]School of Cyber Science and Engineering, HUST.
[‡]Corresponding author
[§]School of Computer Science and Technology, HUST.

1 INTRODUCTION

Android Applications (Apps) have dominated the market share of apps for smartphones nowadays. In particular, the number of available apps has recently reached 2.89 million in the Google Play Store [26]. One major reason that contributed to the massive success of Android is its open-source ecosystem, which has attracted active contributions from many developers. Consequently, there are massive emerged third-party libraries (TPLs) with all kinds of functionalities, which can be further utilized by developers to facilitate the development of new Android apps. TPLs account for an average of more than 60% of the code in Android apps [35]. Many apps even rely on over 20 distinct TPLs [21, 40, 46].

Unfortunately, TPLs often suffer from various vulnerabilities, and 74.95% of vulnerable TPLs are widely utilized by apps or other TPLs [44]. Consequently, the extensive usages of TPLs enable attackers to exploit TPLs' vulnerabilities, thus might cause severe consequences for app users [10, 32, 33]. For instance, the recently spotted vulnerabilities in *Apache Log4j2* [20], have affected more than 35,000 Java packages, amounting for over 8% of the Maven Central Repository [25] (i.e., the most important Java package repository). Until 17th December 2021, nearly 5,000 affected artifacts have been fixed. However, over 30,000 affected artifacts still have no fixing patches, many of which depend on other artifacts to be patched (transitive dependencies), thus blocking prompt remediation [30]. Due to the pervasiveness of such critical vulnerabilities in TPLs and their extensive usages in developing apps, it is crucial to understand, especially for App vendors (e.g., Google Play) and those users who have high demands of security (e.g., governments, security service providers), the absence/presence of vulnerable libraries or code in apps, thus protecting end-users from known threats.

Driven by such practical needs, plenty of **library version detection** approaches have been proposed to automatically detect TPLs and the corresponding versions contained in an app recently [11, 19, 21, 29, 44, 48, 49]. For instance, LibPecker [49] takes class dependencies as code features, encodes dependency graphs as a set of fuzzy class signatures and then calculates the Jaccard similarity between the library and application signatures to detect library with versions. ATVHunter proposed a two-phase detection method, leveraging features based on control-flow graphs and `opcode` in basic blocks to identify specific TPL versions via matching a pre-constructed TPL database [44]. Despite the promising performance achieved, the effectiveness and usefulness of such existing approaches are greatly compromised in real scenarios, in which apps are often ***obfuscated***. In particular, a recent study reveals that 24.9% of the apps are obfuscated before releasing to the market among 1.7 million free Android apps from Google Play [39]. Such a ratio rises up to 50.0% for popular apps with over 10 million downloads [39].

The limitations of existing approaches in such real scenarios mainly come from two aspects. First, they cannot detect the enclosed TPLs with versions precisely once the app has been obfuscated. Although some approaches [19, 21, 49] claim to resist certain obfuscation strategies, they usually report many false positives at the version-level for obfuscated code [44]. Second, their generated results are very coarse-grained. Particularly, if they identify a TPL with the version information in the app, they will query the NVD [34] and report alarms if the version lies in the affected version range of an existing CVE. However, such alarms might be incorrect since the library utilized by the app can actually be secure (e.g., the related vulnerable code has been patched or the vulnerable code has been removed during code obfuscation via the strategy of *removing unused code*). Therefore, it motivates the researches of ***patch presence test*** to spot such false alarms, which works at a finer granularity and aims to check whether the specific patch for a known CVE exists in the target artifact [7, 15, 47]. The behind intuition is to compare the similarity between the pre-/post- patch reference and the target artifact concerning their semantics. Unfortunately, we found that existing patch presence test approaches are also ineffective once the code has been obfuscated.

In this study, we aim to test the presence of patches of specific CVEs in real Android apps, which are often obfuscated. However, code obfuscation brings significant challenges for this task since the code features between the obfuscated code and the original one can be exceptionally different (see Listing 1a for a concrete example). For instance, identifiers, including methods and variable names cannot be matched due to identifier renaming strategy. Besides, specific obfuscation strategies will also induce plenty of redundant code via adding substantial local variables, constraints and redundant function calls (e.g., see `Integer.parseInt()` in Listing 1c at Line 14 for a concrete example). Consequently, existing approaches that rely on the line-to-line mapping between the reference patch and the target artifact (e.g., BScout [7]) are infeasible. Actually, existing approaches cannot locate the patch related methods in the first step since they rely on identifiers while such semantics have been changed to meaningless letters (e.g., "`abc`"), let alone to examine the presence of patches precisely.

To tackle the above challenges, we present PHunter in this paper. The core novelty and insight of PHunter is to extract ***obfuscation-resilient features*** at both the coarse-grained and fine-grained levels to compare patches' semantics, thus capturing their presences. In the coarse-grained step, PHunter identifies patch-related methods in obfuscated apps through fuzzy comparisons, in which only the type information is utilized to represent the signature of a method or a field. After obtaining a set of candidate patch-related methods, PHunter performs fine-grained semantic comparisons at the program path level. The core insight is that if all the paths affected by the reference patch also exist in the target obfuscated code with the same or similar semantics, it is likely that the vulnerability patch is present. PHunter utilizes the information of predicates, method invocations and critical variables along the path to summarize its semantics (i.e., *path summary*). Unfortunately, without matched identifiers, it is hard to compare semantic similarities precisely. Therefore, PHunter proposes to re-express local variables and predicates as expressions consisting of only constants, parameters, fields, function calls, etc via tracing the *def-use* chains of the concerned variables. Besides, it also leverages non-obfuscable information (e.g., the constructor's name) to help recover certain obfuscated variable names aiming to further enhance the precision (i.e., *type recovery*). Finally, PHunter performs semantic similarity comparisons among the extracted paths to report patch presence.

To evaluate the effectiveness and efficiency of PHunter, we compiled 200 apps that involve 31 common TPLs with 94 known CVEs, and each app involves at least one vulnerable TPL. Then we collected the corresponding affected versions for each TPL by querying NVD [34] as the oracle. We obfuscated these apps with Proguard [24], DashO [8], Allatori [1], Obfuscapk [2] to evaluate the obfuscation-resilient capabilities of PHunter. Finally, we compare it with four baselines, including LibScout [4], LibPecker [49], LibID [48], BinXray [43] and ATVHunter [44]. Our results show that PHunter achieves the optimum performance even for unobfuscated apps with an accuracy of 97.1%. As for apps obfuscated by Proguard, DashO, Allatori and Obfuscapk, the improvements *w.r.t.* accuracy are 0.8%, 21.0%, 8.4% and 5.8%, respectively. More importantly, our evaluation also demonstrates that the major components employed in this study can contribute significantly to the promising performance of PHunter. Besides, the results also show that PHunter is efficient, and can complete the entire analysis process within four minutes while achieving a high degree of accuracy for most cases.

The precise results achieved by PHunter are very useful, in particular, in eliminating the *false alarms* generated by TPL detection approaches. Existing approaches usually identify the vulnerable libraries in apps, and then generate a warning for each vulnerability contained in the app to report that the app might be threatened [11, 29, 44, 48, 49]. However, such results might be imprecise as aforementioned. Given such generated warnings, PHunter can make further in-depth analyses for each vulnerability to investigate whether the vulnerable methods exist in the app or whether the vulnerable code has been patched. A controlled experiment shows that existing approaches generate a high false alarm ratio ranging from 3.7% to 25.2%, while PHunter can successfully identify such false positives with a high accuracy of 95.3%. We further performed a large-scale field study on the top 10,000 popular apps on Google Play, and found that PHunter can help spot around 15.5% of the false alarms, thus reducing manual efforts of security analysts.

To summarize, we make the following major contributions:

- **Originality.** We are the first to target testing the presence of vulnerability patches in obfuscated Android applications.
- **Approach.** We proposed an approach based on coarse-grained fuzzy comparisons and fine-grained semantic comparisons at the path level to test patch presence of known vulnerabilities in obfuscated apps. The designed obfuscation-resilient semantic features are the key novelty that enables our approach to resist code obfuscation.
- **Evaluation.** We implemented our approach as a prototype, named PHunter. Extensive evaluation shows that it can test patch presence precisely and efficiently, which also outperforms the state-of-the-art significantly. More importantly, extensive experiments also demonstrate that PHunter is useful in eliminating false alarms of existing approaches.
- **Artifact.** We open-sourced our approach and released our collected dataset of vulnerability TPLs, including the patches and the corresponding apps, thus facilitating future researches. All the artifacts can be accessed at:

  https://github.com/CGCL-codes/PHunter

2 BACKGROUND AND MOTIVATION

2.1 App Obfuscation

Code obfuscation is widely used by app developers to prevent reverse engineering, making it harder for attackers to decompile and exploit vulnerabilities through existing analysis tools. Nearly 24.92% of 1.7 million free Android apps on Google Play are obfuscated before release [39]. In this study, we use four popular obfuscators: Proguard [24], DashO [8], Allatori [1], and Obfuscapk [2]. Proguard, integrated with Android Studio, shrinks bytecode and obfuscates class, field, and method names, and is widely used by developers [23]. DashO and Allatori are mature commercial products offering strong code protection and have been extensively studied [5, 9, 13, 37, 45]. Obfuscapk [2], an open-source obfuscator from academia, supports advanced features like control-flow obfuscation. Table 1 summarizes the obfuscation strategies used by these four obfuscators, which can be easily configured with various options. Detailed explanations of each strategy can be found on our website [38] due to page limitations.

2.2 Existing Approaches

Detecting vulnerable TPLs and checking whether the vulnerable code has been patched have aroused huge research attention recently [4, 7, 15, 41, 44, 47–49], which are summarized as follows:

Table 1: Supported Obfuscation Strategies for Obfuscators

Obfuscation strategy	Proguard	DashO	Allatori	Obfuscapk
Code Shrinking	✓	✓	✗	✗
Package Flattening	✓	✓	✓	✗
Identifier Renaming	✓	✓	✓	✓
Control Flow Obfuscation	✗	✓	✓	✓
String Encryption	✗	✓	✓	✓

✓:support; ✗: not support

Library Version Detection. Intuitively, suppose a tool can accurately identify the version of the TPLs used in an app, one can directly query NVD [34] whether the version of the TPL is vulnerable. Driven by this, plenty of tools have been developed to pinpoint TPLs, along with the specific versions, in apps. For instance, LibScout [4] is a similarity-based library identification tool, which generates fuzzy signatures for each method by using placeholder `X` to replace non-JDK identifiers to obtain method signatures. LibPecker [49] takes class dependencies as code features. Instead of matching the dependency graphs, it encodes the graph as a set of fuzzy class signatures and calculates the Jaccard similarity between the library and application signatures. Although LibScout and LibPecker are resistant to identifier renaming, they cannot resist code shrinking and package flattening since they rely on the package hierarchy. LibID [48] overcomes several limitations of the previous studies. Specifically, it constructs the CFG, utilizes finer granularity features at the basic block level, and combines class dependency to identify TPLs. However, LibID assumes that most obfuscators will not affect the internal package hierarchy structures. This strong assumption directly compromised its effectiveness, thus making it ineffective to detect obfuscated TPLs.

Patch Presence Test. Patch Presence Test aims to infer whether a binary contains the patch for a specific vulnerability. BScout [7] utilizes the debug information (e.g., variable names or line numbers) contained in the Java bytecode to generate line-to-line mappings between the pre-patch/post-patch source code and the target bytecode. After generating the mappings, BScout archives patch presence test by calculating the similarity between the target binary and both the pre-patch and post-patch reference. PDiff [15] is another advanced approach that performs patch presence test for downstream Linux kernels. It then enumerates the method paths, extracts their semantics, and determines the presence of patches based on path similarity. BinXray [43] generates patch signatures by diffing pre-patch/post-patch functions via matching basic blocks. Finally, the patch signatures are used to match the target functions to determine whether they have been patched or not. Unfortunately, the above tools fail to resist various obfuscation strategies. For instance, for the most basic strategy of identifier renaming, all the debug information will be eliminated, and thus the mappings used by BScout cannot be constructed. Moreover, the strategy of control flow obfuscation can further break code structures and make it challenging for PDiff and BinXray to capture the patch semantics. Therefore, more effective and obfuscation-resistant semantic information needs to be extracted to resist code obfuscation.

2.3 Challenges and Insights

It is challenging and time-consuming to analyze whether the patched code exists in the targeted artifacts if they have been obfuscated.

```
1    public void parseCentralDirectoryFormat(
2                  byte[] data, int offset,int length) {
3        // ...
4        if (this.rcount > 0L) {
5            this.hashAlg=HA.getAlgorithm(getValue(data,offset+12));
6            this.hashSize=getValue(data, offset + 14);
7    -        for (int i = 0; i < this.rcount; i++){
8    +        for (long i = 0; i < this.rcount; i++){
9                //...
```

(a) Patch Snippet of CVE-2018-1324

```
1    public void c(byte[] bArr, int i, int i2) {
2        // ...
3        if (this.u1 > 0) {
4            this.v1 = t00.b.a(g20.a(bArr, i + 12));
5            this.w1 = g20.a(bArr, i + 14);
6            for (long j = 0; j < this.u1; j++){
7                // ...
```

(b) Patch Snippet of CVE-2018-1324 Obfuscated by Proguard.

```
1    public void c(byte[] bArr, int p1, int p2) {
2        int j;
3        long l1;
4        String str = "9";
5    //Omit 40 lines of code, including 6 if-else statements
6        if (this.u1 > 0L) {
7            if (j != 0) {
8                j = sf1.a(bArr, p1 + 12);
9                str = "0"
10           } else {
11               j = sf1.a(bArr, p1 + 8);
12               str = "1"
13           }
14           if (Integer.parseInt(str) == 0)
15               this.v1 = qc1.b.a(j);
16               this.w1 = sf1.a(bArr, p1 + 14);
17               for (long j2 = 0; j2 < this.u1; j2++){
18                   //...
19       }
```

(c) Patch Snippet of CVE-2018-1324 Obfuscated by DashO.

Listing 1: Patch of CVE-2018-1324 and the Obfuscated Code Snippets

Listing 1a shows the patch for CVE-2018-1324 [6], a vulnerability in *Apache Commons Compress*, which has been widely utilized by Android applications. The patch snippet modified a single line, which replaces `int` with `long`. The root cause of this vulnerability is that `this.rcount` is a `long` type field. Therefore, attackers can craft specialized Zip archives to cause an infinite loop when `this.rcount` exceeds the max value of `Integer`, which can be exploited to mount a denial of service attacks [3]. The code obfuscated by Proguard and DashO is shown in Listing 1b and Listing 1c respectively. The main challenges are summarized as follows.

First, the identifier names are unmatched. As shown in Listing 1b, the names of method and variable have been changed (e.g., `parseCentralDirectoryFormat` was changed to `c`), and the mappings between those variables cannot be easily constructed. Consequently, it is hard to identify the patch-related methods since the code semantics cannot be directly compared, let alone the patched code snippets (i.e., line 6 in Listing 1b). Therefore, we are motivated to leverage coarse-grained features to identify patch-related methods in a fuzzy way. Besides, we further represent local variables as expressions consisting of only constants, parameters, fields, and function calls via tracing its *def-use* chains to resist code obfuscation (see Section 3.2.2). Second, obfuscation will introduce extra code semantics. For instance, the obfuscators often construct dummy copies of basic blocks (e.g., lines 11-12 in Listing 1c is a dummy copy of lines 8-9) or inject redundant and unnecessary code elements (e.g., function calls such as `Integer.parseInt()`). Consequently, coarse-grained semantic comparisons (e.g., at the method level) are incapable of capturing the vulnerability related semantics precisely. Therefore, it motivates us to make precise comparisons at the program path level, thus enabling us to examine whether the target artifact's paths consume the patch related ones while ignoring those extraneous codes introduced by obfuscation (see Section 3.3). Lastly, code obfuscation may introduce the path explosion problem. For instance, if we unroll the loop once, the obfuscated code as shown in Listing 1c contains over 200 different paths, compared to only 4 in the original code. Actually, Listing 1a only shows a very simple patch. As the original patch's complexity increases, the number of paths in the obfuscated method will grow exponentially. Therefore, we propose to trim infeasible paths to further boost the analysis efficiency (see Section 3.2.3).

3 APPROACH

This study presents PHunter to address the above challenges, and Figure 1 shows the overview. PHunter takes the pre-patch/post-patch TPL (i.e., the "`.jar`" or "`.aar`" compiled from the source code before/after applying the patch respectively), the patch file, and the target app as inputs. It then automatically identifies whether the patch is present in the target app. To equip PHunter with the capability of obfuscation resilience, the key insight is to extract obfuscation-resilient semantics to compare the app's semantics with the patch/post-patch TPL. Our proposed obfuscation-resilient semantics mainly include *coarse-grained fuzzy signatures considering the type information* and *fine-grained path summaries with re-expressed predicates and variables*. PHunter is composed of three steps: *Locating Candidate Methods*, *Path Extraction & Trimming* and *Path Summary & Comparison*, which are described as follows.

3.1 Locating Candidate Methods

PHunter takes the following three steps to locate the patch-related method in the first step, which is crucial for patch presence test.

3.1.1 Coarse-Grained Feature Extraction. By parsing the patch source file, we can easily locate the patch-related methods in TPLs. However, the obfuscators may rename all user-defined identifiers in the target app. Thus, identifier-irrelevant features need to be extracted to locate the patch-related methods in the obfuscated app. Inspired by previous works [48, 49], PHunter constructs the call graph for the app and collects coarse-grained features, which include:

Method Fuzzy Signature: PHunter uses a method's return type and parameter types to represent its signature inspired by LibScout [4]. For user-defined classes, PHunter uses a placeholder `X` to replace it. For instance, for a method with the prototype of "`void fun(int p0, MyClass[] p1)`", its extracted fuzzy signature is "`void,int,X[]`".

Callers and Callees: PHunter integrates the callee and caller information, similarly in a fuzzy way, of a method as its fuzzy signature. Through the call graph, it records all the callees and callers in two sets named S_{callee} and S_{caller}, respectively.

Accessed Field: PHunter records the type for each accessed field. Similar to the method's fuzzy signature, it uses a placeholder `X` to replace user-defined classes. For example, for a field with prototypes

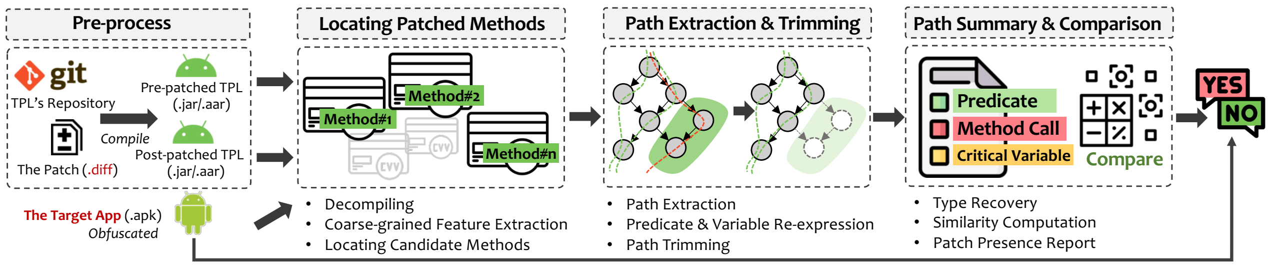


Figure 1: Overview of PHunter

of "`MyClass f2`", it has the fuzzy signatures of "`X`". All the accessed fields in the method are denoted as a set S_{field}.

3.1.2 Locating Patch-Related Methods. PHunter aims to locate methods in the app that correspond to those patch-related methods in the TPL as candidate methods. Specially, given a patch-related method in TPL with the fuzzy signature, and the corresponding features S^{tpl}_{callee}, S^{tpl}_{caller} and S^{tpl}_{field}, it traverses all the methods in the app to compute its similarity with the TPL as follows:

$$Simcoarse = \frac{\left|S^{tpl}_{callee} \cap S^{app}_{callee}\right| + \left|S^{tpl}_{caller} \cap S^{app}_{caller}\right| + \left|S^{tpl}_{field} \cap S^{app}_{field}\right|}{\left|S^{tpl}_{callee}\right| + \left|S^{tpl}_{caller}\right| + \left|S^{tpl}_{field}\right|} \quad (1)$$

The intuition behind is such coarse-grained features (callees, callers, and fields accessed) can reflect important semantics of a method. Therefore, all the methods with the same fuzzy signature as the patch-related method and whose similarity computed by Equation 1 exceeds a predefined $Threshold_{coarse}$ are selected as the candidate methods. We performed a preliminary study to select such a threshold, which is described in Section 5.1.

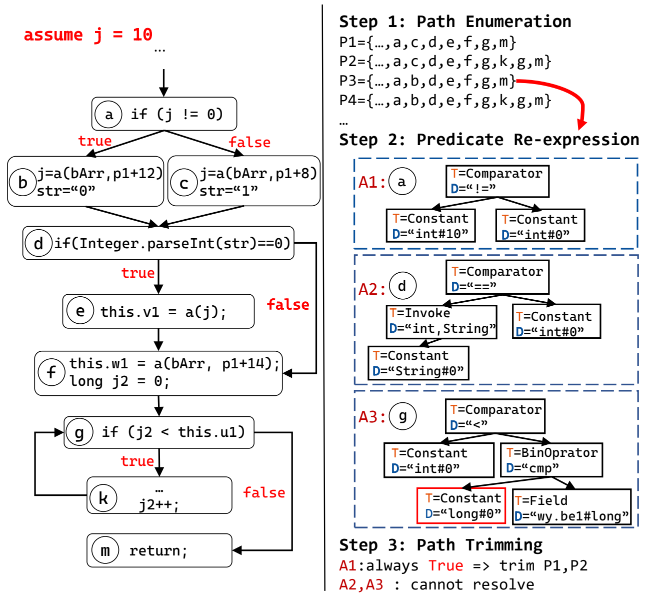
Note that if none of the candidate methods exist, we will terminate the whole process and report the patch is not present. There are three main reasons: (1) the app uses a lower version of the TPL, and thus the patch-related method has not yet been introduced; (2) the patch-related method originally exists in the TPL but has been removed due to different reasons such as Code Shrinking; or (3) the app does not use the TPL.

3.2 Path Extraction & Trimming

Given a candidate method, PHunter further extracts fine-grained information on path constraints, function calls, and critical variables (fields and array type variables) to summarize the method's semantics. The semantics must be obfuscation-resilient, meaning we cannot rely on any identifiers, line numbers, etc. Figure 2 shows how PHunter works for Listing 1c (lines 17-27). The CFG of the code snippet and partial results of each step are presented in Figure 2(a) and Figure 2(b), respectively. Details are as follows:

3.2.1 Path Extraction. PHunter treats the CFG as a directed cyclic graph and adopts a depth-first search algorithm to traverse all paths of the method. For example, there exists a feasible path `P3={..,a,b,d,e,f,g,m}` which passes through three predicates at nodes ⓐ, ⓓ and ⓖ and ends with a return statement at node ⓜ. Note that due to the presence of loops, we restrict each edge (`jumps` between basic blocks) to pass at most once in a path following existing studies [15, 27].

3.2.2 Variable & Predicate Re-Expression. Catering to the needs of capturing obfuscation-resilient semantic features, we devise a



(a) CFG for target (b) Path Extraction

Figure 2: Path Extraction, Predicate Re-expression and Path Trimming for the Example in Listing 1c (lines 7-18)

novel Tree Structure to represent variables and predicates based on the following insights. First, instead of using the traditional source-level or AST representation of expressions with identifier names (e.g., `if(a ⩾ 0)`), we need to abstract away certain detailed information (e.g., identifier names), thus obtaining obfuscation-resilient capability. Second, all the local variables and predicates can be eventually evaluated as expressions consisting of only constant, parameters, fields, and function calls via tracing the *def-use* chains of the concerned variables. Therefore, for all the expressions defined in the Backus-Naur Form (BNF) of Jimple [14, 31], we group them into several types of nodes as shown in Table 2, and each node contains two attributes: ExprType and Data. ExprType denotes the expression type while Data records certain obfuscation resilient information, such as the type of the concerned variables or specific operators. Leaf nodes include Constant, Parameter, Field, and those that cannot be further represented. Accordingly, non-leaf nodes can be broken down into leaf nodes. For example, for `compare` expressions (e.g., `x > y`), PHunter uses `Comparator` as their ExprType with the operator symbol (i.e., >) as their Data. Such nodes contain two children, representing the left and right operands of the operator, respectively. For the leaf node of type Field, instead of recording the specific field name, its Data records the DeclaringClass with the declared types as "`MyClass#Long`".

Table 2: Definition for NodeType, Data of the Tree Structure

ExprType	Data	Example
Comparator	Comparator symbols.	">", "⩾", "==", "!=", etc
BinOperator	Binary operators.	"+", "-", "*", "/", etc.
UnaryOperator	Unary operators	"lengthof", "-"(negative sign)
InstanceOf	Specific class name	Suppose a statement is "v instanceof MyClass", and its Data is "MyClass"
Invoke	The return type and parameter types	Suppose the framework function "startsWith (String a)" is called, its Data is "boolean,String"
Array	The base type of the array.	An array with the long type, its Data form is "long[]"
Constant★	Constant types with values.	A long type constant with value 15 is represented as "long#15"
Class★	Name of the class.	"MyClass"
Parameter★	Index of the target method's parameters.	Suppose the prototype of the target method is "public void fun(MyClass p1, int p2)", then *p1* is "1", *p2* is "2"
Field★	DeclaringClass with the declared types.	A long type field declared in the MyClass.java, its Data form is "MyClass#long"
CaughtException★	An unique string for CaughtException	"@caughtexception"

★ denotes leaf nodes.

Based on such designs, along a specific path, PHunter constructs the tree structures for all the variables and predicates. Specially, it first extracts all the assign statements and predicates along the path (i.e., denoted as list $Intr$), and also creates a map to record the tree structures (i.e., $\mathcal{M}$) for each variable. It then iterates $Intr$ one by one. When encountering an assignment, PHunter constructs a tree for the assigned variable and record it in $\mathcal{M}$. In particular, if the right side of the assignment involves other local variables, it iteratively replaces these variables with the corresponding tree structures via checking with $\mathcal{M}$ until they can all be appropriately represented by leaf nodes. When encountering a predicate, PHunter constructs a tree structure based on its expression type as shown Table 2, and it also replaces all local variables with the corresponding trees stored in $\mathcal{M}$. Figure 2(b) shows the three predicates involved in path `P3` at nodes ⓐ, ⓓ and ⓖ and the corresponding created tree structures. We denote them as `A1`, `A2` and `A3` in order. For instance, for the third predicate `A3`, Soot [28] converts the predicate "`j2 < this.u1`" to "`i0 < 0`", where the int-type temporary variable `i0` is assigned to "`j2 cmp this.u1`" (`cmp` is the opcode of Jimple). PHunter represents local variable `j2` by the tree node with ExprType as "`Constant`" and Data as "`long#0`" (as marked it in red). However, in the pre-patch version, the Data of the node is "`int#0`", which reflects the difference between the pre-patch and post-patch versions, and such difference can be captured by the Tree Editing Distance Algorithm [22] based on our devised tree representation (see Section 3.3.2). Without such fine-grained obfuscation-resilient semantics, the subtle differences cannot be captured.

3.2.3 Path Trimming. As all variables and predicates will be re-expressed along a specific path, and the value of certain predicates can be evaluated. For those predicates in which all leaf nodes are with the ExprType of Constant, PHunter will try to resolve them to remove infeasible paths. Such cases are pervasive since the obfuscators often utilize simple strategies to introduce redundant constraints. As a result, many redundant predicates only involve constants. The path trimming strategy will help alleviate the path explosion problem. For example, for the first predicate `A1`, since `10 != 0` is always evaluated to be `true`, we trim the path that goes through `false` (i.e., from node ⓐ to ⓒ). Consequently, `P1` and `P2` can be removed.

3.3 Path Summary & Comparison

PHunter utilizes the summaries of a set of paths to represent a method's semantics. It ensures that the same path can obtain similar path summaries before and after obfuscation. Specifically, we use *constraints*, *function calls* and *critical variables* along a specific path as its fine-grained summaries, which are introduced as follows:

Path Constraints. The path constraints are recorded orderly as a list of predicates, and we represent each predicate as a tree. As aforementioned, each tree node contains two attributes: ExprType and Data, which captures obfuscation-resilient features.

Function Calls. For each function call, we record its fuzzy signature (the return type and the argument type) as a list.

Critical Variables. Inspired by existing works [15, 41], we record the types of field and array variables sequentially as a list. For each field, we record its `declaringClass` name and type (e.g., `ob.be1#long`). For the array variable, we record its type (e.g., `int[]`).

3.3.1 Type Recovery. We observe that there exist non-obfuscable information, which can be leveraged to help recover partial obfuscated names. Specifically, the constructor's name `<init>` (constructor's names in binary are compiled as `<init>`) of each class will not be obfuscated. Therefore, we can map the types in the constructor arguments before and after obfuscation, thus recovering certain identifiers. For example, there is a patch-related method in TPL that contains a constructor in its `declaringClass` with the prototype `<init>(Myclass1 p1)`. Suppose there is an obfuscated candidate method whose `declaringClass` also contains a constructor with the prototype `<init>(ob.a b1,ob.b P2)`. If the similarity of these two constructors exceeds the $Threshold_{coarse}$, we can infer that `Myclass1` is renamed as `ob.a`.

Then, we utilize this map to try to recover obfuscated identifiers in the path semantics, including function return types, function parameters types, field's `declaringClass` names and field's types. If an obfuscated identifier cannot be recovered, PHunter will replace it with a placeholder `X`.

3.3.2 Similarity Computation. We consider the average similarity of path constraints, function calls and critical variables as the similarity metric for path summaries.

Similarity of Path Constraints. Constraints are represented as tree structures. We utilize Tree Edit Distance (TED) [22] to obtain a minimal-cost sequence of node edit operations required to transform one tree into another. To achieve such a goal, we need to define the edit operations costs, including "insert", "delete" and "modify". Our insight is straightforward. If two nodes match exactly, no edit is required, and thus the cost is 0. If two nodes do not match, either the ExprType or Data, we need one full edit to make them the same, and thus the cost is 1. Otherwise, if two nodes fuzzily match, it only requires half of the operation to make them the same (i.e., the edit cost is thus 0.5). For insert and delete operations, the cost is 1.0. In particular, given two predicates A_1 and A_2, we calculate their similarity as follows:

$$sim(A_1, A_2) = 1/(1 + TED(A_1, A_2)) \quad (2)$$

To compute the similarity between two lists L_1=$\{A_1, A_2, \ldots, A_m\}$ and L_2=$\{A'_1, A'_2, \ldots, A'_n\}$, PHunter's intuition is to identify the optimized ordered matches between the elements in L_1 and L_2 to maximize the similarity. In particular, PHunter implements a Dynamic Programming as shown in Algorithm 1 to achieve the goal, in which $matrix[i, j]$ records the maximum similarity between the two sub-lists $[A_0,..., A_i]$ and $[A'_0,..., A'_j]$. The final similarity $sim(L_1, L_2)$ is computed as as $matrix[m, n]/m$.

Algorithm 1: Similarity Maximization

```
input  : two lists of elements, $L_1 = \{A_1, A_2, \ldots, A_m\}$ and
         $L_2 = \{A'_1, A'_2, \ldots, A'_n\}$ // assume that $L_2$ is shorter than $L_1$
output : similarity of the lists, $sim(L_1, L_2)$
matrix ← $\phi$ // initialize matrix to 0
for $i = 1$ to $m$ do
    for $j = i$ to $n - m + i$ do
        $matrix[i, j] = max(matrix[i-1, j-1] + sim(A_i, A'_j),$
                            $matrix[i-1, j], matrix[i, j-1])$
    end
end
$sim(L_1, L_2) = matrix[m, n]/m$
return $sim(L_1, L_2)$
```

Similarity of Function Calls. PHunter also calculates the similarity of two function call lists using Algorithm 1, which requires the definition of the similarity between individual elements (i.e., function signatures). Similar to our previous design, if two signatures match exactly (the signatures do not contain placeholder `X`), the similarity is 1. If they are fuzzily matched (one of the signatures contains placeholder `X`), the similarity is 0.5 and 0 otherwise.

Similarity of Critical Variables. We calculate the similarity of two critical variable lists similarly as function calls.

Similarity of Methods. As aforementioned, we represent a method as a set of fine-grained path semantics. Given two methods m_{tpl}=$\{p_1, p_2, \ldots, p_m\}$ from TPL and m_{app}=$\{p'_1, p'_2, \ldots, p'_n\}$ from app, where p_i or p'_i denotes a path semantic. PHunter formally define their similarity as follows:

$$sim(m_{tpl}, m_{app}) = \sum_{i=1}^{m} \sum_{j=1}^{n} m_{ij} \times sim_{path}(p_i, p'_j) / |m_{tpl}| \qquad (3)$$

where m_{ij}=1 if p_i matches p'_j and 0 otherwise. $sim_{path}(p_i, p'_j)$ measures the similarity between two paths. Since the elements are unordered, PHunter utilizes the Hungarian Algorithm [16] to search for the optimum matrix m to maximize Equation 3.

3.3.3 Patch Presence Report. PHunter examines the presence of a patch based on the methods' semantic similarities. Specifically, for a patch-related method m_{pre} in the pre-patch TPL, we calculate the method similarity between it and all the candidate methods in the target app, and we select the one with the highest similarity $sim(m_{pre}, m_{app})$ in the app as the matched one. Then, the similarity $sim(PRE, APP)$ between the pre-patch TPL and the target app is calculated as the average of the similarity between all patch-related methods and the matched candidate. Similarly, PHunter also computes the similarity $sim(POST, APP)$ between the post-patch TPL and app. If $sim(POST, APP)$ or $sim(PRE, APP)$ exceed a pre-defined $Threshold_{fine}$, PHunter then determines the patch result according to formula 4. We utilize such a threshold since it is less likely the patch exists if the similarity is too low. We performed a study to select appropriate thresholds (see Section 5.1). Specifically, we consider the target app contains the patch if $sim(POST, APP)$> $sim(PRE, APP)$, and vice versa. Otherwise, PHunter reports the patch does not exist.

$$result = \begin{cases} patched, & sim(POST, APP) > sim(PRE, APP) \\ unpatched, & sim(POST, APP) \leqslant sim(PRE, APP) \end{cases} \qquad (4)$$

4 EXPERIMENT SETUP

4.1 Dataset Construction

In order to validate the effectiveness of PHunter, we need to prepare a dataset of Android apps that utilize vulnerable TPLs containing known CVEs with patches available. Since no publicly available benchmark dataset can serve such a purpose, we created a dataset as follows. First, we crawled 4,561 open-source apps from F-Droid [12], a repository for open-source Android apps. By parsing the Gradle build files, we recognize all the libraries used by each app and then collect the reported vulnerabilities and the corresponding affected versions for each library by querying NVD [34]. However, identifying the corresponding patch for a CVE is a challenging task [36]. Finally, we collected 94 CVEs that affect 31 distinct common libraries after spending huge manual efforts (all these 94 CVEs are displayed on our project website). Second, following the existing work [44], we randomly selected 200 apps from F-Droid that utilized these concerned libraries (we ensure that each app uses at least one TPL that contains CVE) and compiled them without obfuscation. Third, to evaluate the capabilities of PHunter regarding different obfuscators, each of the 200 apps in the dataset is obfuscated using four obfuscators (namely Proguard, DashO, Allatori and Obfuscapk) with **all** obfuscation strategies enabled.

As a result, our dataset includes five sets of apps: a set of 200 non-obfuscated apps, and four sets of apps (200 × 4) obfuscated by the four obfuscators. Eventually, we construct 909 App-CVE pairs (one app may involve multiple vulnerable TPLs), and 56.8% of which have been patched while the others have not. To determine appropriate thresholds, we randomly selected 30 unobfuscated apps (concerning 145 App-CVE pairs) and marked them as $dataset_{param}$. All the remaining apps and pairs will be used for further evaluation, which are marked as $dataset_1$. Our approach also takes the pre-/post-patch TPLs as input. Therefore, for all the 94 CVEs, we manually compiled the pre-patch/post-patch reference from the source code before and after applying the patch.

To investigate the effectiveness of PHunter with respect to its capability in resisting different obfuscation options, we randomly selected 50 unobfuscated apps from $dataset_1$ and used DashO to obfuscate these apps with **different** obfuscation options individually (as shown in Table 1). This is a time-consuming process. Finally, we obtain one group (50 apps) of the original apps and four groups (50 × 4) of the obfuscated apps (marked as $dataset_2$). Compiling Android apps with obfuscation is time-consuming, and it took us about 150 hours to compile all of the apps and the TPLs.

4.2 Baseline Selection

We first compare PHunter with existing state-of-the-art open-source **TPL detection** tools, including LibScout [4], LibPecker [49], LibID [48] and ATVHunter [44]. These tools can specify the TPL versions used by the app in our dataset. Although they are not designed to solve the patch presence test problem effectively, they can be easily adapted since it is possible to find out whether the library is affected by a vulnerability by querying the NVD database [34]. Among these tools, since ATVHunter is not publicly available due to commercial issues, we re-implemented it based on the descriptions in the paper (denoted as ATVHunter*). We attempted to reproduce the results using the original dataset and compared them with the results as reported in the paper [44]. Our re-implementation yielded consistent results and it outperforms other library detection tools as reported in the original paper, which affirms its reliability. The existing **patch presence test** tools (e.g., BScout [7], PDiff [15] and BinXray [43]) are designed for unobfuscated artifacts and none of them are specifically designed for Android apps. We choose BinXray as our baseline since it is advanced and open-sourced, and we carefully adapted it to work for Android apps (see our website [38] for more details). Besides, BinXray assumes the target functions are either vulnerable or patched in its original design. Therefore, we have equipped BinXray with our coarse-grained semantics to locate patch-related methods. On the contrary, BScout and PDiff are not open-sourced and rely on code features that vary for different obfuscation strategies. For example, PDiff utilizes the name of callees to construct its path constraints and invocations to compare path similarities. BScout relies on the debug information to generate a line-to-line mapping between the Java source code and bytecode instructions. In summary, we select LibScout, LibPecker, LibID, ATVHunter* and BinXray as our baselines.

4.3 Research Questions

We aim to answer the following research questions:

RQ1: How is the effectiveness of PHunter? In this RQ, we investigated the performance of PHunter on $dataset_1$. We first evaluate whether PHunter can identify the correct patch-related method. We use the mapping file (contains the mapping between obfuscated names and the original names) as the oracle to determine whether a candidate is the correct patch related method and employ method-level accuracy as the metrics. We then evaluate (3) the overall effectiveness of the patch presence test and employed ***Accuracy*** ($\frac{TP+TN}{TP+TN+FP+FN}$) and ***FPR*** ($\frac{FP}{FP+TN}$) as metrics following the existing work [7]. In particular, if PHunter reports that it identifies a CVE patch in an app that indeed contains the patch, we consider it as a *true positive* (TP); If PHunter reports that it finds a CVE patch in an app in which the patch actually does not exist, we consider it as a *false positive* (FP); For those cases in which the vulnerability is not patched and PHunter reports there is no patch, we consider it as a *true negative* (TN); Otherwise, it is considered as a *false negative* (FN). For library version detection tools, we set the threshold corresponding to the original paper. Since they may report multiple versions of a library for an app at a time, following previous studies [7, 48], we choose the library version(s) with the highest similarity score as the matched version. If the matched version shares the same patch status as the used TPL, we consider the result is true positive or true negative. If the tool does not report any version of the library, we consider there is one false negative.

RQ2: How can PHunter resist different obfuscation options? We further investigate the effectiveness of PHunter with respect to its capability in resisting different obfuscation options. Therefore, we run PHunter on $dataset_2$ and then check the testing results. We further compared PHunter with our selected baselines with respect to the metric of Accuracy.

RQ3: How do the major components of PHunter contribute to its performance? Three major components that contribute to PHunter's promising performance are the devised *type recovery* strategies, *path summaries* and *method similarity computation.* In this RQ, we aim to investigate the contributions of these components separately. In particular, to investigate type recovery, we simply disable it in PHunter (i.e., denoted as $\mathbf{PHunter_r}$) and then investigate its performance on $dataset_1$. The experimental setting is the same as RQ1. As for path summaries, since we are the first to propose such obfuscation-resilient path summaries, there is no existing work for direct comparisons. To evaluate it, we adapt the strategies adopted by BinXray [43] (originally designed to compare path similarities via utilizing the edit distance of two sequential instructions) to replace our devised path summaries in PHunter (i.e., denoted as $\mathbf{PHunter_a}$). In particular, we extract the instructions for each path and utilize the formula "$1/(1+Edit(path_1, path_2))$" (similar to Equation 2) to measure the similarity between two paths, where Edit means the Levenshtein distance [18]. After utilizing such a strategy to summarize paths, we then evaluate PHunter's performance. Our proposed fine-grained features essentially equip PHunter with the obfuscation-resilient method similarity computation. To evaluate its effectiveness, we replace our devised features with that as adopted by ATVHunter to compute method similarities, and then evaluate PHunter's performance (i.e., denoted as $\mathbf{PHunter_m}$). Specifically, given a method, ATVHunter assigns a unique sequence ID (starting from 0) to each node of CFG. It prioritizes the child node (subsequent node of a branch node) with more outgoing edges and statements and assigns each node a sequence ID accordingly. Then, ATVHunter concatenates all opcode sequences of each node according to the sequence ID and calculates method similarity using the fuzzy hash Algorithm [50].

RQ4: How is the efficiency of PHunter? We measured the test time of PHunter on $dataset_1$. Specifically, we record the main steps contributing to the tool's overhead, including decompiling the input binary (one apk and pre-/post-patch TPLs), locating method candidates, and computing fine-grained similarity.

5 EXPERIMENT RESULTS

We implemented PHunter as a tool with 11k lines of Java code based on Soot [28]. We first performed a preliminary study to choose appropriate thresholds for PHunter as shown in Section 5.1. We then evaluate PHunter with respect to general effectiveness (RQ1), the capability of code obfuscation-resilience (RQ2), the dissection of performance (RQ3) and efficiency (RQ4).

5.1 Threshold Tuning

To avoid bias, we utilize $dataset_{param}$ to determine appropriate thresholds for PHunter and the parameters of selected baselines. We leverage the same strategy to tune all the tools for fair comparisons. Note that the apps in $dataset_{param}$ do not overlap with those used in the evaluation.

Our approach needs to set two thresholds, $Threshold_{coarse}$ to ensure the correct patch-related method is included in the candidate set, and $Threshold_{fine}$ to ensure that PHunter can correctly

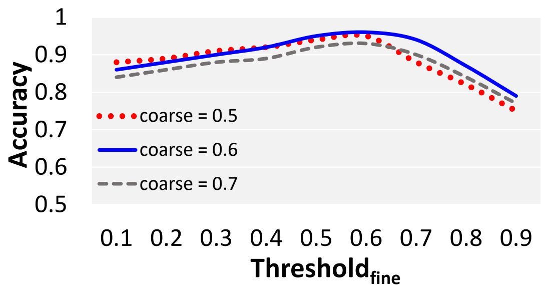


Figure 3: Similarity Threshold Selection

identify the patch-related methods. Following the existing studies [19, 42], we employ *Grid Search* [17] to determine the thresholds empirically via optimizing the accuracy. Specifically, we first set the two thresholds to 0 and gradually adjust them with the step of 0.1 until the upper bound of 1. We then select those values that can achieve the highest accuracy. Figure 3 shows parts of the thresholds tuning results. Specifically, when we set $Threshold_{coarse}$ to 0.5, 0.6 and 0.7, respectively, we can observe how the accuracy varies with $Threshold_{fine}$. We can see that the three curves share a similar pattern. They increase until the highest accuracy is achieved when $Threshold_{fine}$=0.6 and then the curve decreases. Accordingly, we choose $Threshold_{coarse}$=0.6 and $Threshold_{fine}$=0.6 aiming to ahiceve the highest accuracy.

For the library detection tools, LibScout needs to set one threshold as the minimal similarly to match the reference TPLs with the target apk. LibPecker needs to set three thresholds for method-, package- and TPL-level matching. LibID needs to set two thresholds for TPL-level matching and balancing between the detection of TPLs and false positives. ATVHunter* also needs to set two thresholds for method- and TPL-level matching. To tune thresholds for these tools and adapt them to the problem of patch presence test, we adopt the same procedure as discussed above to identify the thresholds that achieve the highest accuracy at the patch-level. The fine-tuned thresholds were then selected for further evaluation.

5.2 RQ1: General Effectiveness

Identifying Patch-Related Methods. We first need to pinpoint the correct patch-related method in the app. Our evaluation results show that PHunter achieves the method-level accuracy of 96.3%, 94.0%, 95.2%, 93.5% and 95.9% for the unobfuscated apps, apps obfuscated by Proguard, DashO, Allatori and Obfuscapk, respectively. Through further analysis, we found that the major reason for the missed cases is *function evolution*. Particularly, after a CVE patch is applied to the TPL, the method may be further modified or even deleted during subsequent software evolution. Therefore, the control flow and code features (e.g., function calls) of the patch-related method may be drastically changed. In this case, even if the app is not obfuscated, PHunter cannot locate the patch-related methods in the app, resulting in false negatives. Nevertheless, we also observed that the existing baselines cannot detect such cases neither.

Overall Results. Table 3 shows the comparison between PHunter and other baselines in terms of patch presence test *w.r.t.* various metrics. For the test on unobfuscated apps, PHunter accurately identifies the patch for 97.1% of the App-CVE pairs and reports few false negatives (2.1%). For the test on apps obfuscated by Proguard, DashO, Allatori and Obfuscapk, PHunter can achieve an accuracy of 92.8%, 93.3%, 87.3%, 93.1% with a false positive rate of 7.0%, 1.7%, 5.5% and 6.2% respectively. To summarize, our tool achieves the optimum results among all the techniques. On average, the accuracy of PHunter outperforms the best baselines by 2.2% (97.1%-95.9%) for unobfuscated apps. As for the apps obfuscated by Proguard, DashO, Allatori and Obfuscapk, the improvements *w.r.t.* accuracy are 0.8% (92.8%-92.0%), 21.0% (93.3%-72.3%), 8.4% (87.3%-78.9%) and 5.8% (93.1%-87.3%), respectively. As we have explained that *function evolution* can cast significant impacts on pinpointing correct methods. Such issues can eventually affect the final patch presence test results, which might lead to incorrect results.

Finding 1*. PHunter is effective in testing the presence of patches for vulnerabilities in obfuscated apps (i.e., the achieved accuracy is up to 93.3%). It also outperforms SOTA baselines significantly.*

The promising results achieved by PHunter as reflected by the above results is attributed to the fact that PHunter focus on extracting the features that are obfuscation-resilient, including both coarse-grained and fine-grained features. In contrast, existing state-of-the-art library version detection tools utilize too coarse-grained code features to identify patches or rely on the package structure as supplementary information (see Section 2.2 for more details). For instance, ATVHunter* achieves the best results among existing library detection tools. However, for the apps that are obfuscated by DashO, Allatori and Obfuscapk, its accuracy is significantly compromised due to control flow obfuscation. This is because ATVHunter* relies on the *fuzzy hash* algorithm [50] to compute method similarities, which relies on the sequence of statements to generate the fingerprints. Nevertheless, the control flow obfuscation will change the code order, thus making it challenging for ATVHunter* to precisely capture the code semantics. For BinXray, it can achieve an accuracy of 95.4% for unobfuscated apps. However, for apps obfuscated by DashO, Allatori and Obfuscapk, the accuracy drops significantly to 60.0%, 49.7% and 52.8%, respectively. Such results are mainly due to the control flow obfuscation performed by the obfuscators. Specifically, BinXray relies on the hash values of the basic blocks to perform basic block matching and compare the similarity between functions. This design makes it less resilient to any changes in the basic block. Unfortunately, the control flow obfuscation will insert redundant variables, constraints and invocations, which results in significantly different hash values for basic blocks.

5.3 RQ2: Resilience to Each Obfuscation Option

We compare PHunter with other baselines on apps obfuscated by different obfuscation options. The results are presented in Table 4. It shows that PHunter achieves the optimum performance for all the obfuscation options compared to the baselines. In particular, PHunter achieves 100% accuracy for Identifier Renaming and Package Flattening. For the resilience to Control Flow Obfuscation (CFO) and Code Shrinking, PHunter decreases by about 2.1% and 3.3%, respectively, demonstrating the capability of PHunter towards common obfuscation options. In comparison, the accuracy of other TPL detection tools decreases to various degrees for different obfuscation options. Specifically, the accuracy of LibID is only 42.1% for Identifier Renaming. LibPecker can only correctly identify 69.4% of the patches with Package Flattening. As for the CFO, the accuracy of LibScout, LibID and ATVHunter* is reduced to 13.9%, 3.7% and

Table 3: Comparison on the Effectiveness of Patch Presence Test

Tools	Non-obfuscated				Proguard				DashO				Allatori				Obfuscapk			
	TP	TN	Acc.	FPR	TP	TN	Acc.	FPR	TP	TN	Acc.	FPR	TP	TN	Acc.	FPR	TP	TN	Acc.	FPR
LibScout	436	185	81.3%	8.3%	0	0	0.0%	-	0	0	0.0%	-	0	0	0.0%	-	302	204	55.5%	9.8%
LibPecker	479	229	92.7%	4.1%	259	307	62.3%	8.0%	236	273	56.0%	14.1%	375	263	70.2%	18.1%	494	252	82.1%	21.9%
LibID	427	170	78.1%	5.9%	198	221	46.1%	12.9%	4	21	2.8%	8.6%	47	165	23.3%	6.8%	0	0	0.0%	-
ATVHunter*	483	249	95.9%	2.5%	451	349	88.0%	9.9%	272	385	72.3%	14.2%	413	304	78.9%	10.6%	362	431	87.2%	6.3%
BinXray	477	252	95.4%	4.5%	480	356	92.0%	7.0%	129	416	60.0%	23.2%	106	346	49.7%	19.2%	151	329	52.8%	29.7%
PHunter	482	260	**97.1%**	2.1%	488	346	**92.8%**	7.0%	382	466	**93.3%**	1.7%	446	348	**87.3%**	5.5%	495	351	**93.1%**	6.2%

Table 4: Comparison on Common Obfuscation Options

Tools	Renaming	Repackage	CFO	Shrinking
LibScout	88.6%	23.2%	13.9%	15.3%
LibPecker	91.2%	69.4%	91.2%	64.4%
LibID	42.1%	45.5%	3.7%	19.0%
ATVHunter*	96.3%	87.6%	80.1%	94.2%
BinXray	95.2%	95.2%	56.2%	93.1%
PHunter	**100.0%**	**100.0%**	**97.9%**	**96.7%**

Renaming denotes Identifier Renaming. Repackage denotes Package Flattening. CFO denotes Control Flow Obfuscation. Shrinking denotes Code Shrinking.

Table 5: Contribution of the Major Components

Tools	Without Obfuscation	With Obfuscation			
		Proguard	DashO	Allatori	Obfuscapk
PHunter$_r$	95.2%	88.6%	89.7%	83.5%	92.0%
PHunter$_a$	96.1%	86.9%	72.8%	76.7%	85.8%
PHunter$_m$	95.9%	85.1%	67.3%	76.2%	86.9%
PHunter	**97.1%**	**92.8%**	**93.3%**	**87.3%**	**93.1%**

PHunter$_r$ denotes PHunter disabling type recovery. PHunter$_a$ denotes PHunter with path summaries from BinXray. PHunter$_m$ denotes PHunter utilizing the method similarity computation strategy designed by ATVHunter.

80.1%, indicating that these TPL detection tools are inadequately resistant to these common obfuscation options. BinXray achieves an accuracy of 95.2% for Identifier Renaming since it does not rely on identifiers. For CFO, as aforementioned, the redundant function calls and variables will affect the hash values of basic blocks. Therefore, the accuracy of BinXray drops to 56.2%.

***Finding 2**. PHunter can resist to various common obfuscation strategies effectively with a high accuracy of 96.7%, including advanced ones such as control-flow randomization and code shrinking.*

5.4 RQ3: Dissecting PHunter's Performance

Table 5 shows the results. In particular, PHunter$_a$ achieves an accuracy of 96.1% and 86.9% for unobfuscated apps and apps obfuscated by Proguard in which CFO is disabled. Such results are very promising since PHunter outperforms it by 2% to 3% respectively. However, when the CFO is enabled, as shown in the results of DashO, Allatori and Obfuscapk, the accuracy of PHunter$_a$ drops to 72.8%, 76.7% and 85.8%, respectively. In contrast, PHunter can significantly outperform them by 20.5% (93.3%-72.8%), 10.6% (87.3%-76.7%) and 7.3% (93.1%-85.8%). This is because the instructions sequences as path summaries (we adapted from BinXray) are changed drastically due to the redundant code from CFO.

PHunter$_m$ achieves an accuracy of 95.9% and 85.1% for unobfuscated apps and apps obfuscated by Proguard. However, the accuracy PHunter$_m$ is compromised when CFO is enabled. It achieves the accuracy of 67.3%, 76.2% and 86.9% for apps obfuscated by DashO, Allatori and Obfuscapk, respectively. After analyzing some of missed cases, we found that CFO affects the code order on which the fuzzy hash relies (ATVHunter utilizes fuzzy hash to calculate method similarity), thus compromising the accuracy. Such results demonstrate that our devised fine-grained path summaries can effectively counteract the effects introduced by CFO. For PHunter$_r$, it achieves an accuracy of 95.2%, 88.6%, 89.7%, 83.5% and 92.0% on unobfuscated apps and apps obfuscated by Proguard, DashO, Allatori and Obfuscapk, respectively. In practice, type recovery can help infer about 10.5 obfuscated identifiers in the patch-related method on average and improve the accuracy by about 1% to 4%, proving its effectiveness.

Table 6: Average Time Consumption

	Total	Pre-process	Coarse-grained	Fine-grained
No-Obfuscation	236.9s	212.1s	1.9s	7.5s
Proguard	163.8s	155.7s	1.8s	6.1s
DashO	203.5s	162.3s	2.2s	37.2s
Allatori	260.8s	231.7s	2.0s	27.1s
Obfuscapk	243.2s	226.1s	1.9s	15.2s

Pre-process denotes decompiling the binary. Coarse-grained denotes locating candidate methods. Fine-grained denotes fine-grained similarity comparison.

5.5 RQ4: Efficiency

Table 6 dissects the time consumption for PHunter on $dataset_1$. We can find that if the app has not been obfuscated, the average time for PHunter to detect a patch for an app is about 4 minutes, of which about 95% of the overhead comes from decompiling the input app. Coarse-grained and fine-grained analysis only takes 1.9s and 7.5s respectively, demonstrating that the tool's efficiency mainly depends on the decompiler tools. For testing apps obfuscated by Proguard, the time overhead is lower (163.8s per app) due to two main reasons: (1) some unused code has been removed by the obfuscator, which shrinks the binary size and (2) the patch-related method in the app is potentially removed, and thus the fine-grained similarity comparison is not performed. For testing apps obfuscated by DashO, Allatori and Obfuscapk, more time will be spent on fine-grained similarity computation (37.2s, 27.1s and 15.2s respectively) due to the increase in paths. However, it will take a significantly longer time, which are 151.2s, 105.8s and 56.4s, if the path trimming is not applied. In particular, we found that for the unobfuscated app, the average number of paths dropped from 24.1 to 8.2 after path trimming. For the app obfuscated by Control Flow Obfuscation, the average number of paths dropped from 246.5 to 94.6, which proves PHunter can effectively alleviate the path explosion problem.

***Finding 3**. PHunter is efficient and can finish the analysis within minutes, in which the pre-process takes the majority of the time.*

6 USEFULNESS OF PHUNTER

Existing TPL detection tools can identify the vulnerable libraries in apps, and then usually generate a warning (i.e., an App-CVE

Table 7: Detecting False Alarms of TPL Detection Tools

	Total Warnings	False Alarms (Ratio)	Accuracy
$dataset_1$(unobfuscated)	326	12 (3.7%)	100%
$dataset_1$(obfuscated)	575	145 (25.2%)	95.3%
RealApps	3,957	613 (15.5%)	-

pair) for each vulnerability contained in libraries to report that the app might be threatened [11, 19, 21, 29, 44, 48, 49]. However, existing approaches might report plenty of false alarms [44, 49]. A warning is a **false alarm** *if either the vulnerable method does not exist in the app or the vulnerable code has already been patched.* In such cases, the vulnerability actually would not threaten the app. Fortunately, PHunter can detect such false alarms since it can examine the vulnerable code to see whether it exists or has been patched. Therefore, we perform the following experiments to demonstrate such usefulness of PHunter.

Controlled Experiments. First, we perform a controlled experiment on $dataset_1$. In particular, we utilize all the apps in $dataset_1$ in two scenarios: *all the apps are not obfuscated* and *all the apps are obfuscated using Proguard.* Such two scenarios can reflect the best and worst cases compared to real-world apps since they usually contain obfuscated apps mixed with non-obfuscated ones. We then apply ATVHunter* (since it achieves the optimum performance as revealed in RQ1) to detect TPLs on the above datasets to see how many warnings it generates. Specifically, we utilize the collected 31 common TPLs of all versions and the 94 CVEs as the dataset for reference checking, which are introduced in Section 4.1. Finally, ATVHunter* generates 326 and 575 warnings for non-obfuscated apps and apps obfuscated by Proguard, respectively. We then apply PHunter to examine whether such warnings are false alarms. Since we know the ground truth of this dataset (i.e., whether the vulnerable methods exist or the vulnerable code has been patched), we can evaluate the accuracy of PHunter in spotting such false alarms.

Large-scale Field Study. We further perform a large-scale field study to evaluate the usefulness of PHunter in practice. In particular, we collect the top 10,000 popular apps from Google Play (marked as *RealApps*) ranked by the number of installations. Similarly, we leverage ATVHunter* to detect TPLs as well as the vulnerabilities enclosed in these apps. We also utilize the 31 common TPLs and 94 CVEs for reference checking. In total, ATVHunter* has detected that these apps used 9,721 times of the concerned 31 TPLs. For each detected TPL, if its version lies in the affected version range of a corresponding CVE, ATVHunter* generates a warning. Finally, ATVHunter* reports 3,957 warnings for all the 10,000 apps. We then apply PHunter to detect whether such warnings are false alarms.

Experimental Results. For each reported warning (i.e., APP-CVE pair), PHunter can analyze the vulnerability code to see if it indeed exists in the app, and then detect false alarms as previously defined. Table 7 shows the overall results on the warnings generated by ATVHunter*. For the non-obfuscated apps, since ATVHunter* can detect TPLs more precisely in this scenario, only 3.7% of them are false alarms, and 100% of them can be detected and eliminated by PHunter (i.e., Accuracy = 100%). For obfuscated apps, ATVHunter* is less effective and generates a high false alarms ratio of 25.2%. Fortunately, PHunter can precisely detect such false alarms with a high accuracy of 95.3%. For the real-world apps, since it is time-consuming to construct the ground truth via manual checking for all the 10,000 apps, we only report the false alarm ratio reported by ATVHunter*, which is 15.5%. Since PHunter is precise in identifying such false alarms (i.e., as evaluated on $dataset_1$), substantial false alarms can be eliminated by PHunter, thus alleviating the debugging efforts for developers.

***Finding 4**. PHunter is useful, and can detect the false alarms generated by existing TPL detection tools with a high accuracy of 95.3%. Actually, existing TPL detection tools can generate a high ratio of false alarms, ranging from 3.7% to 25.2%.*

7 DISCUSSION

Threats to Validity. The validity of this study suffers from two main threats. *1). Bias of obfuscation strategies.* In this study, we have considered common-used obfuscation strategies, but there also exists more complex strategies. Nevertheless, the obfuscators we tested, Proguard, is popular and about 88% of apps use it for obfuscation [37]. Another obfuscator, DashO, is also a powerful commercial software, which is used by over 5,000 companies. PHunter's ability to achieve high accuracy against these tools proves its obfuscated-resistant ability. *2). The implementation of BinXray and ATVHunter*.* BinXray [43] was originally designed for C/C++. Fortunately, we can re-use the main code of BinXray, which differs only in the pre-processing phase (i.e., extract binary instructions) while the other parts of code are directly re-uesd. The adapted BinXray achieves excellent results for unobfuscated apps (see Table 3), demonstrating the validity. For ATVHunter*, since its source code is not available and certain points are unclear [44], the implementation of ATVHunter* might be biased. However, our implementation yielded consistent results and it outperforms other library detection tools as reported in the original paper, which affirms its reliability. Therefore, such a threat is mitigated.

Limitation and Future Work. As aforementioned, after vulnerable patches are applied, some patch-related methods may be further modified or deleted in subsequent revisions, and PHunter is ineffective to handle such cases. In fact, we found that patch evolution is widespread. Limited by the difficulty of dataset collection, this study does not evaluate the ability of the PHunter to resist patch evolution in detail, while it is our important future work.

8 CONCLUSION

Ensuring app security is vital for vendors, while developers frequently obfuscate code upon release. Consequently, patch presence testing for obfuscated apps becomes essential. With this motivation, we introduce PHunter, designed to extract obfuscation-resistant features for identifying vulnerability patches. We evaluate PHunter on a large dataset, demonstrating its effectiveness against advanced code obfuscation strategies. To contribute to Android app security assurance, we have released our tool and data.

ACKNOWLEDGMENTS

We sincerely thank all anonymous reviewers for their valuable comments. This work was supported by the National Natural Science Foundation of China (Grant No. 62002125), the Young Elite Scientists Sponsorship Program by CAST (Grant No. 2021QNRC001) as well as the Hubei Province Key R&D Technology Special Innovation Project under Grant No.2021BAA032.

REFERENCES

[1] Allatori. 2022. https://allatori.com/. Accessed: 2022-10.

[2] Simone Aonzo, Gabriel Claudiu Georgiu, Luca Verderame, and Alessio Merlo. 2020. Obfuscapk: An open-source black-box obfuscation tool for Android apps. *SoftwareX* 11 (2020), 100403. https://doi.org/10.1016/j.softx.2020.100403

[3] Alessandro Armando, Alessio Merlo, Mauro Migliardi, and Luca Verderame. 2012. Would You Mind Forking This Process? A Denial of Service Attack on Android (and Some Countermeasures). In *Information Security and Privacy Research - 27th IFIP TC 11 Information Security and Privacy Conference, SEC 2012, Heraklion, Crete, Greece, June 4-6, 2012. Proceedings (IFIP Advances in Information and Communication Technology, Vol. 376)*, Dimitris Gritzalis, Steven Furnell, and Marianthi Theoharidou (Eds.). Springer, 13–24. https://doi.org/10.1007/978-3-642-30436-1_2

[4] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and its Security Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 356–367. https://doi.org/10.1145/2976749.2978333

[5] Salman Abdul Baset, Shih-Wei Li, Philippe Suter, and Omer Tripp. 2017. Identifying Android library dependencies in the presence of code obfuscation and minimization. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE Computer Society, 250–252. https://doi.org/10.1109/ICSE-C.2017.79

[6] CVE-2018-1324. 2022. https://nvd.nist.gov/vuln/detail/CVE-2018-1324. Accessed: 2022-10.

[7] Jiarun Dai, Yuan Zhang, Zheyue Jiang, Yingtian Zhou, Junyan Chen, Xinyu Xing, Xiaohan Zhang, Xin Tan, Min Yang, and Zhemin Yang. 2020. BScout: Direct Whole Patch Presence Test for Java Executables. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 1147–1164. https://www.usenix.org/conference/usenixsecurity20/presentation/dai

[8] DashO. 2022. https://www.preemptive.com/products/dasho/. Accessed: 2022-10.

[9] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. 2018. Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild. In *Security and Privacy in Communication Networks - 14th International Conference, SecureComm 2018, Singapore, August 8-10, 2018, Proceedings, Part I (Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Vol. 254)*, Raheem Beyah, Bing Chang, Yingjiu Li, and Sencun Zhu (Eds.). Springer, 172–192. https://doi.org/10.1007/978-3-030-01701-9_10

[10] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021.* The Internet Society. https://www.ndss-symposium.org/ndss-paper/towards-measuring-supply-chain-attacks-on-package-managers-for-interpreted-languages/

[11] Ruian Duan, Ashish Bijlani, Yang Ji, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee. 2019. Automating Patching of Vulnerable Open-Source Software Versions in Application Binaries. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019.* The Internet Society. https://www.ndss-symposium.org/ndss-paper/automating-patching-of-vulnerable-open-source-software-versions-in-application-binaries/

[12] F-Droid: Free and Open Source Software. 2022. https://f-droid.org. Accessed: 2022-10.

[13] Mahmoud Hammad, Joshua Garcia, and Sam Malek. 2018. A large-scale empirical study on the effects of code obfuscations on Android apps and anti-malware products. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 421–431. https://doi.org/10.1145/3180155.3180228

[14] Soot Expr Interface. 2022. https://www.sable.mcgill.ca/soot/doc/soot/jimple/Expr.html. Accessed: 2022-10.

[15] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. 2020. PDiff: Semantic-based Patch Presence Testing for Downstream Kernels. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 1149–1163. https://doi.org/10.1145/3372297.3417240

[16] Harold W. Kuhn. 2010. The Hungarian Method for the Assignment Problem. In *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*, Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey (Eds.). Springer, 29–47. https://doi.org/10.1007/978-3-540-68279-0_2

[17] Steven M. LaValle, Michael S. Branicky, and Stephen R. Lindemann. 2004. On the Relationship between Classical Grid Search and Probabilistic Roadmaps. *Int. J. Robotics Res.* 23, 7-8 (2004), 673–692. https://doi.org/10.1177/0278364904045481

[18] Levenshtein_distance. 2022. https://en.wikipedia.org/wiki/Levenshtein_distance. Accessed: 2022-10.

[19] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. 2017. LibD: scalable and precise third-party library detection in android markets. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 335–346. https://doi.org/10.1109/ICSE.2017.38

[20] Apache Log4j2. 2022. https://github.com/apache/logging-log4j2. Accessed: 2022-10.

[21] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: fast and accurate detection of third-party libraries in Android apps. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 653–656. https://doi.org/10.1145/2889160.2889178

[22] Mateusz Pawlik and Nikolaus Augsten. 2016. Tree edit distance: Robust and memory-efficient. *Inf. Syst.* 56 (2016), 157–173. https://doi.org/10.1016/j.is.2015.08.004

[23] Yuxue Piao, Jin-hyuk Jung, and Jeong Hyun Yi. 2013. Structural and functional analyses of proguard obfuscation tool. *The Journal of Korean Institute of Communications and Information Sciences* 38, 8 (2013), 654–662.

[24] Proguard. 2022. https://www.guardsquare.com/proguard. Accessed: 2022-10.

[25] Maven Central repository. 2022. https://www.maven.org/. Accessed: 2022-10.

[26] Android Market Share. 2022. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. Accessed: 2022-10.

[27] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 693–706. https://doi.org/10.1145/3192366.3192418

[28] Soot. 2022. https://github.com/soot-oss/soot. Accessed: 2022-10.

[29] Zhushou Tang, Minhui Xue, Guozhu Meng, Chengguo Ying, Yugeng Liu, Jianan He, Haojin Zhu, and Yang Liu. 2019. Securing android applications via edge assistant third-party library detection. *Comput. Secur.* 80 (2019), 257–272. https://doi.org/10.1016/j.cose.2018.07.024

[30] understanding impact of apache log4j. 2022. https://security.googleblog.com/2021/12/understanding-impact-of-apache-log4j.html. Accessed: 2022-10.

[31] Raja Vallee-Rai and Laurie J Hendren. 1998. Jimple: Simplifying Java bytecode for analyses and transformations. (1998).

[32] Nikos Vasilakis, Achilles Benetopoulos, Shivam Handa, Alizee Schoen, Jiasi Shen, and Martin C. Rinard. 2021. Supply-Chain Vulnerability Elimination via Active Learning and Regeneration. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 1755–1770. https://doi.org/10.1145/3460120.3484736

[33] Nikos Vasilakis, Achilles Benetopoulos, Shivam Handa, Alizee Schoen, Jiasi Shen, and Martin C. Rinard. 2021. Supply-Chain Vulnerability Elimination via Active Learning and Regeneration. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 1755–1770. https://doi.org/10.1145/3460120.3484736

[34] National vulnerability database. 2022. https://nvd.nist.gov. Accessed: 2022-10.

[35] Haoyu Wang and Yao Guo. 2017. Understanding third-party libraries in mobile app analysis. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE Computer Society, 515–516. https://doi.org/10.1109/ICSE-C.2017.161

[36] Xinda Wang, Kun Sun, Archer L. Batcheller, and Sushil Jajodia. 2019. Detecting "0-Day" Vulnerability: An Empirical Study of Secret Security Patch in OSS. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019.* IEEE, 485–492. https://doi.org/10.1109/DSN.2019.00056

[37] Yan Wang and Atanas Rountev. 2017. Who Changed You? Obfuscator Identification for Android. In *4th IEEE/ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017.* IEEE, 154–164. https://doi.org/10.1109/MOBILESoft.2017.18

[38] Online website of PHunter. 2022. https://github.com/CGCL-codes/PHunter.

[39] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Bradley Reaves, Patrick Traynor, and Sascha Fahl. 2018. A Large Scale Investigation of Obfuscation Use in Google Play. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018.* ACM, 222–235. https://doi.org/10.1145/3274694.3274726

[40] Apps with most third-party libraries. 2022. http://privacygrade.org/third_party_libraries. Accessed: 2022-10.

[41] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. 2020. Precisely Characterizing Security Impact in a Flood of Patches via Symbolic Rule Comparison. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020.* The Internet Society. https://www.ndss-symposium.org/ndss-paper/precisely-characterizing-security-impact-in-a-flood-of-patches-via-symbolic-rule-comparison/

[42] Liang Xiao, Ruili Wang, Bin Dai, Yuqiang Fang, Daxue Liu, and Tao Wu. 2018. Hybrid conditional random field based camera-LIDAR fusion for road detection. *Inf. Sci.* 432 (2018), 543–558. https://doi.org/10.1016/j.ins.2017.04.048

[43] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. 2020. Patch based vulnerability matching for binary programs. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 376–387. https://doi.org/10.1145/3395363.3397361

[44] Xian Zhan, Lingling Fan, Sen Chen, Feng Wu, Tianming Liu, Xiapu Luo, and Yang Liu. 2021. ATVHUNTER: Reliable Version Detection of Third-Party Libraries for Vulnerability Identification in Android Applications. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1695–1707. https://doi.org/10.1109/ICSE43902.2021.00150

[45] Xian Zhan, Lingling Fan, Tianming Liu, Sen Chen, Li Li, Haoyu Wang, Yifei Xu, Xiapu Luo, and Yang Liu. 2020. Automated Third-Party Library Detection for Android Applications: Are We There Yet?. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 919–930. https://doi.org/10.1145/3324884.3416582

[46] Xian Zhan, Tianming Liu, Lingling Fan, Li Li, Sen Chen, Xiapu Luo, and Yang Liu. 2022. Research on Third-Party Libraries in Android Apps: A Taxonomy and Systematic Literature Review. *IEEE Trans. Software Eng.* 48, 10 (2022), 4181–4213. https://doi.org/10.1109/TSE.2021.3114381

[47] Hang Zhang and Zhiyun Qian. 2018. Precise and Accurate Patch Presence Test for Binaries. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 887–902. https://www.usenix.org/conference/usenixsecurity18/presentation/zhang-hang

[48] Jiexin Zhang, Alastair R. Beresford, and Stephan A. Kollmann. 2019. LibID: reliable identification of obfuscated third-party Android libraries. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 55–65. https://doi.org/10.1145/3293882.3330563

[49] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. 2018. Detecting third-party libraries in Android applications with high precision and recall. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd (Eds.). IEEE Computer Society, 141–152. https://doi.org/10.1109/SANER.2018.8330204

[50] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. 2012. Detecting repackaged smartphone applications in third-party android marketplaces. In *Second ACM Conference on Data and Application Security and Privacy, CODASPY 2012, San Antonio, TX, USA, February 7-9, 2012*, Elisa Bertino and Ravi S. Sandhu (Eds.). ACM, 317–326. https://doi.org/10.1145/2133601.2133640

Received 2023-02-16; accepted 2023-05-03