LibScan: Towards More Precise Third-Party Library Identification for Android Applications

Yafei Wu and Cong Sun, *State Key Lab of ISN, School of Cyber Engineering, Xidian University, China;* Dongrui Zeng, *Palo Alto Networks, Inc., Santa Clara, CA, USA;* Gang Tan, *The Pennsylvania State University, University Park, PA, USA;* Siqi Ma, *University of New South Wales, Australia;* Peicheng Wang, *State Key Lab of ISN, School of Cyber Engineering, Xidian University, China*

https://www.usenix.org/conference/usenixsecurity23/presentation/wu-yafei

This paper is included in the Proceedings of the
32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

LibScan: Towards More Precise Third-Party Library Identification for Android Applications

Yafei Wu[1], Cong Sun[1*], Dongrui Zeng[2], Gang Tan[3], Siqi Ma[4], Peicheng Wang[1]
[1]*State Key Lab of ISN, School of Cyber Engineering, Xidian University, China*
[2]*Palo Alto Networks, Inc., Santa Clara, CA, USA*
[3]*The Pennsylvania State University, University Park, PA, USA*
[4]*University of New South Wales, Australia*
[1]*suncong@xidian.edu.cn*, [2]*dzeng@paloaltonetworks.com*, [3]*gtan@psu.edu*, [4]*siqi.ma@unsw.edu.au*

Abstract

Android apps pervasively use third-party libraries (TPL) to reuse functionalities and improve development efficiency. The insufficient knowledge of the TPL internal exposes the developers and users to severe threats of security vulnerabilities. To mitigate such threats, people have proposed diversified approaches to identifying vulnerable or even malicious TPLs. However, the rich features of different modern obfuscators, including advanced repackaging, dead code removal, and control-flow randomization, have significantly impeded the precise detection of the TPLs. In this work, we propose a general-purpose TPL detection approach, LibScan. We first fingerprint code features to build the potential class correspondence relations between the app and TPL classes. Then, we use the method-opcode similarity and call-chain-opcode similarity to improve the accuracy of detected class correspondences. Moreover, we design early-stop criteria and reuse intermediate results to improve the efficiency of LibScan. In experiments, the evaluation with ground truths demonstrated the effectiveness of LibScan and its detection steps. We also applied LibScan to detect vulnerable TPLs in the top Google Play apps and large-scale wild apps, which shows the efficiency and scalability of our approach, as well as the potential of our approach as an auxiliary tool that helps malware detection.

1 Introduction

Third-party libraries (TPL) serve as indispensable components for modern mobile applications. For example, advertising, social networking, location services, and in-app payment add-ons are pervasive to provide helpful functionality in popular real-world apps. The evolution and maturity of various third-party libraries have drastically alleviated the development efforts for the apps. However, thorough knowledge of the inner details of TPLs is beyond app developers' abilities without the source code of TPLs. The diverse requirements, flexible usages, and indeterminate maintenance status of such external libraries usually introduce many vulnerabilities to the apps and threaten end-user privacy [29, 31].

[*]Corresponding Author

Various security evaluations and mitigations towards the threats caused by third-party libraries have been proposed. For example, analyzing the security-related updatability of the libraries, patching the app smoothly with non-intrusive library updates, or validating the runtime library updates can remedy the flaws caused by outdated libraries [26, 28, 37]. Isolating the TPLs from the apps is also a kind of popular mitigation that can be conducted at different granularities. The TPLs can be run as a new process [49, 59], a separate sandboxing app [38, 51], or be shared locally between apps and dynamically loaded in app executions [41]. Enforcing in-app privilege separations with new permission mechanisms can keep the apps' privileges from TPLs [35, 48]. Other works provide integrity verification to the TPLs [36], test the potential vulnerability of TPLs inside apps beyond the reach of GUI-based testing [23], or validate the security of the in-app payment process provided by the third-party payment components [55].

Many of the above protections rely on precise third-party library detection as a prerequisite. Nevertheless, due to the wide use of advanced obfuscations, repackaging, code shrinking, and optimization techniques over the released apps [24, 63], TPL detection confronts significant difficulty. Typically, popular code obfuscators or compilers (e.g., ProGuard [32], DashO [47], Allatori [39], and the R8 compiler [19] of Android Studio) can obfuscate the host apps and the TPLs, resulting in a vague boundary between them. The TPLs' interdependencies usually get changed by the app's integration due to development management or obfuscation.

The similarity-based TPL detections are the dominant approaches in recent studies. However, as we observe, the state-of-the-art similarity-based detections, e.g., [22, 54, 56, 58, 60], still have some defects in dealing with advanced obfuscations, repackaging, or optimizations. Specifically, LibScout [22] and LibPecker [60] depend on the package hierarchy information to decide if the TPLs' package matches the app's specific package; thus, both approaches may lose precision when the app is repackaged. Besides, the modern Android app obfuscator can not only eliminate the useless TPLs' classes and methods from the app, but also insert code and spurious control-flow edges into the used TPLs' methods. Such control-flow randomization will cause the offset variation of the original code relative to the beginning of basic blocks or sliding windows. Therefore, LibID [58] and ATVHunter [56] that hash the basic blocks inside the methods as units become significantly affected by such obfuscation. Beyond the effectiveness problems of these approaches to advanced obfuscations, detection efficiency is another issue. The similarity-based approaches tend to require app and TPL classes to be pairwise matched to decide which part of the app is matchable to the TPL. If treating methods as nodes and call relations as edges, the TPL detection can approximate an NP-complete subgraph isomorphism problem, and a costly solution is required. Due to our experiences, LibID [58] and LibPecker [60] are both time-consuming. LibID uses the Binary Integer Programming models to maximize the dependency matching count efficiently. However, when the app classes and TPL classes are numerous, pairwise dependency matching is still costly.

It is often challenging to enhance the anti-obfuscation ability of detection techniques while maintaining low costs. Therefore, in this work, we propose LibScan, an efficient and accurate similarity-based TPL detection approach for Android apps. In the first step, we extract a set of code features to generate a fingerprint for both app classes and TPL classes, based on which we build overapproximated class correspondence relations between app and TPL classes. If most classes of a TPL have corresponding classes in an app, LibScan determines that the app uses the TPL. Thus, the accuracy of class correspondence relations is critical. Given that the class correspondence relations generated in the first step are overapproximated, in the next steps, we determine the method-opcode similarity and call-chain-opcode similarity between classes to remove false class correspondence relations, which is resilient to the advanced obfuscation of repackaging and control-flow randomization. To accelerate the detection, we measure a confidence score at each detection step for the existence of TPL in the app. A score lower than a configurable threshold indicates an absence of TPL in the app, and if it remains above the threshold in all detection steps, LibScan reports the existence of the TPL. We summarize our contributions as follows:

1. We propose a 3-step TPL detection approach. The first step builds signature-based class correspondences that indicate the matching potentials between app and TPL classes. The second and third steps determine a method-opcode similarity and a call-chain-opcode similarity to remove false class correspondences to improve the overall detection accuracy. In experiment, the effect of each detection step was evaluated.
2. We compare our implementation with the state-of-the-art TPL detection approaches [22, 54, 58, 60] using ground truths over the apps built and obfuscated with ProGuard, DashO, Allatori, or the D8 and R8 compilers. We also compare LibScan with ATVHunter's public detection results [56]. We further investigate LibScan's effectiveness against different obfuscation levels of DashO and the D8/R8 compiler. The results demonstrate that LibScan outperforms other approaches in effectiveness on most obfuscation levels.
3. We investigated the existence of 205 vulnerable TPL versions in the 1,000 most popular apps on Google Play. The results show that our approach is more efficient than LibID, LibPecker, and Orlis. Besides, the scalability evaluation on 100,000 real-world apps to detect the 205 TPL versions indicates that 23 out of the 205 TPL versions are reused 3,949 times in 3,664 apps. We find suspicious recent usage of some TPLs confirmed to be vulnerable long ago. Thus, we conducted a proof-of-concept experiment to demonstrate that the existence of vulnerable TPLs can be used for malware detection.

2 Related Work

Apart from the early whitelist-based package name detections [25, 31], the TPL detection techniques can be mainly classified into three categories: clustering-based detections [42, 44, 50, 53, 61], learning-based detections [43, 45], and similarity-based detections [22, 30, 52, 54, 56, 58, 60]. Our approach falls into similarity-based detections. Besides, specific approaches have been proposed to identify native libraries [21] or predict useful TPLs for apps based on collaborative filtering [34].

Similarity-based TPL Detection. LibScout [22] uses class hierarchy to profile the features of TPLs. The library signatures are derived using a fixed-depth Merkle tree flattening the package layer and collecting the non-obfuscated partial method signatures. This tree-based signature cannot capture the cross-package code relocating. LibDetect [30] uses bytecode, labeled control-flow transfer, non-obfuscated supertypes, and (fuzzy) structural-preserving representation to filter the library code of apps hierarchically. Orlis [9, 54] uses the fuzzy method signature as used by [22], constructs a textual call graph for the method, and measures similarity based on a similarity digest-based score. LibPecker [60] constructs strict class signatures from the class dependencies. Its adaptive class matching conducts a fuzzy weighted similarity matching between library and app classes in case the similarity is above the adaptive threshold. LibPecker is relatively sensitive to package flattening or class repackaging because its package matching relies on the package hierarchy information. PANDroid [52] incorporates structural and content information of TPLs to depict the TPL signatures and investigate mutations by ProGuard to identify stable invariants during the mutation. LibID [58] profiles obfuscation-resilient features and proposes a multi-phase matching. First, LibID matches candidate library classes with app classes when each basic block of an app class can find an identical basic block in a specific library class. Then the dependency matching finds the truly matched pairs from the candidates and uses a binary integer programming model to maximize the number of matched class pairs. In this phase, uniqueness and hierarchy constraints are used for scalability, and several constraints are addressed for accuracy. The library matching confirms the matched library version by considering the proportion of matched classes in the matched app package. ATVHunter [56] proposes a two-phase detection approach to detect TPLs and their versions. The coarse-grained phase attaches each basic block in a method with a unique serial number. It converts the intra-procedural CFG from the adjacency lists into a method signature based on the assigned serial numbers. The fine-grained phase of ATVHunter uses fuzzy hashing over the per-sliding-window opcodes to avoid the localized feature change caused by obfuscation triggering a significant difference to the final fingerprint. However, control-flow randomization may add redundant opcodes, changing the offset of the original opcodes relative to the sliding windows. In such a case, the app methods may be missed in the method matching. In contrast, LibScan's method-opcode similarity determination focuses on the set-based inclusion relation of per-method opcodes. Our call-chain-opcode similarity determination addresses the set-based inclusion relation of the call-chain opcodes. Such treatments are resilient to redundant opcodes and control-flow randomization compared with ATVHunter.

3 Scope of LibScan

New code obfuscation techniques emerge when existing obfuscation techniques can be deobfuscated. Thus, it is impractical to develop a deobfuscator that can support any kind of obfuscation technique. The same challenge also applies to the design of an obfuscation-resistant TPL detection technique. Therefore, before we elaborate the technical details of LibScan, we first define the scope of obfuscation techniques and obfuscation tools that LibScan can support.

LibScan is a code-similarity based TPL detection tool motivated by the gap of prior work's capability in addressing the obfuscation techniques implemented by three popular obfuscators, including ProGuard [32], DashO [47], and Allatori [39]. In Table 1, we list all known obfuscation techniques summarized by previous papers [22, 33, 54, 56, 57] and mark the ones supported by the three obfuscation tools. LibScan is designed to overcome the obfuscation techniques implemented by the three obfuscators. Moreover, in experiment, we demonstrated LibScan's efficacy against the R8 compiler [19] of the recent Android Studio, even though R8 was not considered during the design process of LibScan.

At the granularity of obfuscation techniques, LibScan is effective in resisting the ones marked with (*) in Table 1. First, LibScan relies on code features irrelevant to the package hierarchy, identifiers, or string constant, making LibScan resilient to repackaging, identifier renaming, and string encryption. Furthermore, LibScan is resilient to control-flow randomization, because LibScan uses a set-based inclusion relation to compare opcodes, being able to ignore most of the reordering, duplication, and trivial insertions of opcodes. LibScan also has built-in thresholds to tolerate dead code removals and code addition. Besides, since LibScan analyzes code only, LibScan has inherent resistance to non-code-based obfuscations, e.g., Manifest transformation and data realignment mentioned in [33].

Table 1: Obfuscation techniques of android obfuscators (LibScan is robust against techniques marked with (*))

	Allatori	DashO	ProGuard
identifier renaming(*)	✓	✓	✓
code addition(*)	✓	✓	✓
dead code removal(*)	✓	✓	✓
package flattening/repackaging(*)	✓	✓	✓
string encryption(*)	✓	✓	–
control-flow randomization(*)	✓	✓	–
Manifest transformation (*)	–	–	–
data alignment (*)	–	–	–
app-level Dex encryption	–	–	–
virtualization-based protection	–	–	–
Java reflection	–	–	–
method inlining	–	–	–

However, LibScan is sensitive to the rest of the obfuscation techniques listed in Table 1. In detail, method inlining could undermine LibScan's method-opcode similarity determination. Java reflections can evade LibScan's call-chain-opcode similarity determination, which is demonstrated by our evaluation of LibScan against R8-optimized apps. Proxy classes wrapping existing methods may alter the fingerprinting code features used by LibScan's signature-based class correspondence detection. Techniques generating code at runtime, e.g., app-level Dex encryption [57] and visualization-based protection [22, 62], are addressed neither by our approach nor by most other similarity-based TPL detectors. In Section 6.3, we will discuss potential mitigation against some of these techniques.

4 Design of LibScan

4.1 System Workflow

The workflow of LibScan is presented in Figure 1. LibScan takes as input an APK app with a third-party library (TPL), and outputs a verdict for the existence of the TPL in the app. LibScan first decompiles the app and TPL into Java classes. LibScan then compares each app class with each TPL class and gives a set of pairwise class correspondence relations. Based on the class correspondence relations, LibScan computes a confidence score for the existence of the TPL in the app. If the score is lower than a configurable threshold in any of LibScan's detection steps, LibScan detects the TPL's

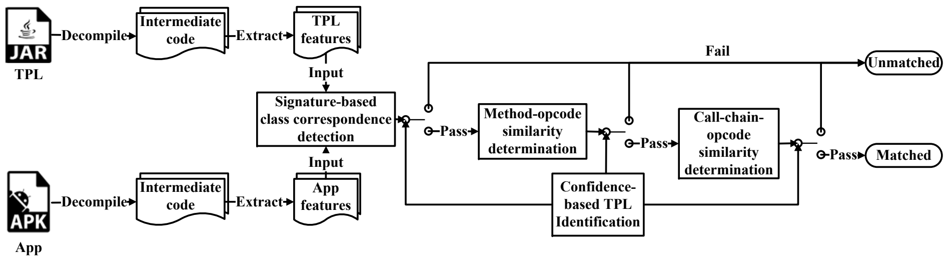
TPL
Decompile
Intermediate code
Extract
TPL features
Input
Signature-based class correspondence detection
Input
App
Decompile
Intermediate code
Extract
App features
Pass
Method-opcode similarity determination
Pass
Call-chain-opcode similarity determination
Pass
Matched
Fail
Unmatched
Confidence-based TPL Identification

Figure 1: Workflow of LibScan

absence in the app. If the score remains above the threshold in all the detection steps, LibScan decides that the TPL exists in the app.

However, the determination of pairwise class correspondence is non-trivial, given that we aim to accommodate obfuscated code. Thus, LibScan takes three steps to give the final determination. Before the 3-step procedure, LibScan first extracts the necessary information for every step from both the app and the TPL, including a set of fingerprinting code features and the method opcodes of each app and TPL class. In the first step, LibScan constructs a signature for both the app class and the TPL class, based on the code features. Such a signature is not supposed to change even if a class is obfuscated. Therefore, if two signatures are different, it indicates that the two classes are most likely different. Otherwise, the two classes could have the correspondence relation. We call this step *signature-based class correspondence detection*. Then, the second step compares the method opcodes in the two classes and calculates a similarity score for them. If the similarity score is above a predetermined threshold, LibScan decides that the two classes are method-wise similar. We call this step *method-opcode similarity determination*. To further improve the accuracy of the class correspondence between the app class and TPL class, the final step takes into account the method call chains in the method similarity comparison. We call this step *call-chain-opcode similarity determination*.

Next, we provide the details of the 3-step approach to determining the class correspondence relations in Section 4.2, 4.3, and 4.4. Then, we explain how to use class correspondence relations to compute the confidence score of TPL existence (Section 4.5) with key optimizations that increases the efficiency of LibScan (Section 4.6). Note that the term "set" used in the following descriptions refers to the mathematical definition of set, where elements in a set are deduplicated.

Table 2: Class-Level Features for Signature

Class Feature Type	`smali` keyword [5]
∃ Ordinary class	class
∃ Interface	interface
∃ Abstract class	abstract
∃ Enumerate	enum
∃ Static inner class	static
∃ Non-object parent	super

Table 3: Field-Level Features for Signature

Field Feature Type	`smali` keyword	static?
No field	–	N/A
∃Object type	Ljava/lang/Object	Yes/No
∃String type	Ljava/lang/String	Yes/No
∃¬Object ∧ ¬String Java standard ref type	Ljava/	Yes/No
∃Java primitive type	B, S, I, J, F, D, Z, C	Yes/No
∃Non-standard ref type	–	Yes/No
∃Java standard ref array type	[Ljava/	Yes/No
∃Java primitive array type	[B, [S, [I, [J, [F, [D, [Z, [C	Yes/No
∃Non-standard array type	–	Yes/No

Table 4: Method-Level Features for Signature

Method Return Type	`smali` keyword	static?	Parameters			
			∃Java std ref type	∃Java primitive type	∃array type	∃Non-std ref type
∃Object type	Ljava/lang/Object	Yes/No	Yes/No	Yes/No	Yes/No	Yes/No
∃String type	Ljava/lang/String					
∃¬Object ∧ ¬String Java standard ref type	Ljava/					
∃Java primitive type	B, S, I, J, F, D, Z, C					
∃Non-standard ref type	–					
∃Java standard ref array type	[Ljava/					
∃Java primitive array type	[B, [S, [I, [J, [F, [D, [Z, [C					
∃Non-standard array type	–					
∃void	V					

4.2 Signature-Based Class Correspondence Detection

Within the scope of LibScan defined in Section 3, we found some code features that may persist during obfuscation. For example, keyword `class` and primitive types used by the TPL code are not altered in order to preserve the original functionality. Such code features form a class fingerprint, which is used in the first step to perform a basic class correspondence detection. If two classes share the same fingerprint, they proceed to the next steps; otherwise, the class correspondence determination stops for this pair of classes.

We divide the fingerprinting code features into three categories: class-level features, field-level features, and method-level features (listed in Table 2, Table 3, and Table 4). The class-level features consist of 6 class/interface types. The field-level features have 22 types of `smali` keywords, each of which can be decorated with or without a `static` qualifier. In total, there are 44 (22×2) features. Additionally, we also create a feature to represent the case where no fields are found in the class. The method-level features are defined by three orthogonal feature dimensions: 23 return types in terms of `smali` keywords $\times$ 2 static/non-static method qualifiers $\times$ 16 (2^4) parameter combinations, totaling 736 features.

The code feature extraction procedure in Figure 1 searches keywords in the decompiled `smali` code of the app and the TPL to decide if each feature is satisfied. As a result, each feature is abstracted as a Boolean variable, and combining all features into a vector forms a signature for both app classes and TPL classes. Then, LibScan performs a pairwise signature matching between app classes and TPL classes. If the signature of an app class is identical to the signature of a TPL class, this class pair is qualified for the next steps of class correspondence detection.

4.3 Method-Opcode Similarity Determination

The signature-based class correspondence detection may correspond multiple app classes to one TPL class as well as multiple TPL classes to one app class, indicating that there can be false class correspondences. Therefore, in the second step, LibScan compares the method opcodes of two classes to determine if their correspondence is false. LibScan computes a similarity score for every pair of app class and TPL class. If the score is below a predetermined threshold, a false class correspondence is detected. As a result, this step removes false class correspondences and reduces the number of app classes that are corresponded to one TPL class.

Typical control-flow randomization inserts a small number of redundant opcodes into the TPL methods used by the app, while the dead code removal eliminates unnecessary methods, fields, and classes. In general, to guarantee the functionality of obfuscated TPL methods, TPL obfuscators are conservative in removing opcodes in the TPL methods that are necessary for the app. Based on this observation, the TPL method opcodes are likely to appear in the app if the TPL method is used. Therefore, LibScan uses the existence of TPL-method opcodes in the app methods to detect the class correspondence between an app class and a TPL class.

For an app class c and a TPL class l, let M_c and M_l be the set of methods of c and l, respectively. For an app method $m \in M_c$, of which the set of opcodes is represented by S_m, and a TPL method $n \in M_l$, of which the set of opcodes is represented by S_n, we define that m is matched to n, represented by $mMatch(m : M_c, n : M_l)$, if and only if 1) their method descriptor features defined in Table 4 are consistent and 2) the hash values of both methods' opcodes are identical or $S_n \subseteq S_m$. By definition, every Dalvik instruction [46] contains two parts: opcodes and operands. Note that we use only opcodes for method matching.

Theoretically, each TPL method should be matched by at most one app method. Thus, if multiple app methods in one app class are matched to one TPL method, LibScan selects the best-matched app method as the final match, which is the one with minimal opcode difference compared to the TPL method. Suppose the TPL method is n, the set of matched app methods is defined as $MM_c = \{m \in M_c \mid mMatch(m,n)\}$, and the best-matched app method is defined as:

$$b(M_c,n) = \begin{cases} argmin_{x \in MM_c} \mid S_x - S_n \mid, \text{ if } MM_c \neq \emptyset \\ \text{None, otherwise} \end{cases}$$

Moreover, we write $B_{c,l}$ for the set of all the method-opcode best-matched app methods in the app class c against the TPL class l, which is defined as $B_{c,l} = \{m \mid \exists n \in M_l, m = b(M_c,n)\}$.

Then, we define a *method-opcode similarity score* for a pair of classes (c,l) as follows:

$$MOSS(c,l) = \frac{\sum_{m \in B_{c,l}} \text{msize}(m)}{\sum_{m \in M_c} \text{msize}(m)},$$

where $\text{msize}(m)$ gives the number of opcodes in method m. A high method-opcode similarity score indicates that the proportion of best-matched app methods to the TPL class methods dominate the app methods of an app class in size. Therefore, we configure LibScan with a similarity-score threshold, θ_1, to determine whether two classes are method-wise similar. In other words, only if $MOSS(c,l) \geq \theta_1$, the class correspondence relation between c and l is preserved.

Note that the definition of *MOSS* has inherently dealt with many-to-many candidate method matching. $b(M_c,n)$ can find at most one app method that best matches the TPL method n. Suppose more than one app method can best match n, e.g., $S_{m_1} = \{op_1, op_2\}$ and $S_{m_2} = \{op_1, op_3\}$ best match $S_n = \{op_1\}$. We would include either m_1 or m_2 into $B_{c,l}$, because a TPL method can only match one app method within the scope of the app. On the contrary, one app method can match multiple TPL methods, e.g., S_{m_1} also matches $S_{n_2} = \{op_2\}$. In this case, m_1 can be considered as the combination of n and n_2 possibly introduced by obfuscation. Hence, m_1 should be counted only once in $B_{c,l}$.

4.4 Call-Chain-Opcode Similarity Determination

Given that two methods may look similar in terms of opcodes but behave in different ways, using method-opcode similarity alone could still lead to false class correspondences. Therefore, we use call chains originating from a method to abstract the functional behaviors of the method, based on which LibScan decides the final class correspondences that will be used for TPL detection. The decisions are made by comparing the opcodes in the call chains within the app class and the TPL class. We call this step call-chain-opcode similarity determination. At a high level, the difference between method-opcode similarity and call-chain-opcode similarity is where LibScan collects the set of opcodes for similarity comparison. For method-opcode similarity, LibScan considers the opcodes in the method only. In contrast, LibScan further considers opcodes in the subsequently invoked methods, which incorporates more information and is more precise.

In detail, for each TPL method $n \in M_l$, LibScan takes the best-matched app method $m = b(M_c, n)$ and n as the respective entry method and uses depth-first traversal on the program call graphs bounded by a configurable maximum depth to generate a respective set of call chains, represented by CC^m and CC^n. Note that standard library calls are skipped because such calls may disturb the sensitivity of the call-chain-opcode similarity determination. Each call chain is represented by a set of class methods, and each class method is represented as a set of opcodes. Hence, we define the call-chain opcode set for a method x as

$$CS^x = \{op \mid \exists cc \in CC^x,\ (m \in cc \wedge op \in m)\}.$$

If $CS^n \subseteq CS^m$, we say m and n have the *call-chain-opcode inclusion* relation. If two methods do not have the call-chain-opcode inclusion relation, LibScan determines that the two methods are no longer matched. Thus, we use a predicate $Match(c,l)$ to denote that a class pair (c,l) has the final class correspondence, which holds if and only if:

$$\forall m \in M_c, n \in M_l, (m = b(M_c, n) \implies CS^n \subseteq CS^m).$$

4.5 TPL Detection

With the detected class correspondence relations between the app and the TPL, LibScan computes a confidence score for the existence of TPL in the app. The intuition is to compute the coverage rate of the TPL. Next, we give the formal definition of the confidence score.

We write X for the set of app classes of an app and Y for the set of TPL classes of a TPL. As defined in Section 4.4, we use the predicate $Match(x : X, y : Y)$ to represent the class correspondence detection result, which determines for a class pair (x, y), where x is from the app and y is from the TPL, whether they have the class correspondence relation. Thus, the set of TPL classes that have correspondence in the app can be represented as $M(X,Y) = \{y \in Y \mid \exists x \in X,\ Match(x,y)\}$. Now, we can formally define the confidence score $\text{Confidence}(X,Y)$ as follows:

$$\text{Confidence}(X,Y) = \frac{\sum_{y \in M(X,Y)} \text{csize}(y)}{\sum_{y \in Y} \text{csize}(y)},$$

where $\text{csize}(y)$ gives the number of opcodes in a class. To yield the final verdict on the existence of TPL in the app, we configure LibScan with a confidence-score threshold, θ_2. If $\text{Confidence}(X,Y) \geq \theta_2$, LibScan determines that the TPL exists in the app.

4.6 Efficiency Optimization

While the accuracy is increasing for the three steps of class correspondence determination, the cost of the three steps is also increasing. If every pair of classes went through all three steps, LibScan would be unscalable. A key observation for improving efficiency is that the intermediate class correspondence at each step can be used to early stop the TPL detection. For example, suppose the signature-based detection step does not find enough TPL classes in an app to go beyond the confidence score threshold. In that case, we can stop the detection for the TPL right away since later steps can only reduce the confidence score of the app and the TPL. In Section 5, we demonstrate the efficiency improvement with experiments. For brevity, we elaborate on the details of this design by presenting the TPL detection algorithm in Appendix A.

We depict the feature extraction of the app and TPL in Figure 1 as a prior step before the three steps of deriving class correspondences. However, these features are reusable when determining the confidence score of different app-TPL pairs. We use a cache mechanism for the features to manage queries of cached features before extracting new features from apps or TPLs. The cached features include the signature-related features (Section 4.2), the opcodes and method invocation relations of the apps and TPLs for the similarity determinations (Section 4.3 and 4.4). Due to the relatively high cost, the call chain construction is postponed to the call-chain-opcode similarity determination instead of being cached. The cache mechanism is suitable for batch-job feature extraction on a large number of apps and TPLs; thus, LibScan can be used in app stores' vetting.

5 Evaluation

This section evaluates the effectiveness, efficiency, and scalability of LibScan, as well as our efforts in using our TPL detection to help malware detection. We first explain the threshold tuning process, after which we compare the effectiveness of LibScan with other approaches and evaluate LibScan's effectiveness against different obfuscation levels on the ground-truth benchmarks. Then, we evaluate the efficiency of LibScan. To demonstrate the necessity of each detection step, we compare LibScan's complete detection with the deployments that disable specific steps on effectiveness and efficiency. Moreover, we conducted a large-scale experiment to detect vulnerable TPL versions in real-world apps. We discovered an emerging phenomenon of reusing vulnerable TPL versions and were motivated to apply our TPL detection to malware detection. As a result, we found some undetected apps by clustering similar apps with the same vulnerable TPLs to propagate verdicts.

5.1 Experimental Settings and Datasets

Our experiments are conducted on a laptop with AMD Ryzen 7 5800H with Radeon Graphics@3.2GHz×8 CPU, 16GB RAM, Linux 5.4.0-42-generic kernel (Ubuntu 18.04 64-bit). We develop LibScan with Python 3.7. LibScan decompiles the apps with `Androguard 3.4.0` [6] to extract the features used by our approach. For the TPLs, we use `dex2jar 2.0` [16] to convert the Jar file into a Dex file, then decompile the Dex file with `Androguard` to extract the features.

To evaluate our approach, we designed four app datasets $AS_1 \sim AS_4$ and three related TPL datasets $LS_1 \sim LS_3$. The app datasets AS_1 and AS_2 have ground truths regarding the app's TPL versions. AS_1 consists of 1,049 apps, including 785 apps released by Orlis [9,54] and 264 released by ATVHunter [15]. AS_2 contains 204 apps built by us. In detail, we collected 51 apps' source code from the 17 categories of [3]; we built each app with the D8 compiler [17] and three different obfuscation levels of the R8 compiler [19] of Android Studio v3.5.1. AS_3 consists of the 1,000 most popular apps from Google Play's 35 categories. AS_4 consists of 100,000 apps released from Jan. 2012 to Dec. 2021, downloaded from AndroZoo [7, 20]. For each year, we randomly selected 10,000 apps. All the apps are strictly deduplicated.

Each app dataset will be evaluated against a TPL dataset. TPL dataset LS_1 consists of 452 TPLs released by Orlis. The ground-truth TPL existence between LS_1 and AS_1 is provided by Orlis. The TPL dataset LS_2 consists of 109 TPLs used by AS_2. The ground-truth TPL existence between LS_2 and AS_2 is acquired by checking their source code. The TPL dataset LS_3 consists of 192 vulnerable TPL versions released by ATVHunter [12] and 13 popular vulnerable TPL versions collected from CVEs (details in Appendix B). In all, each of the 205 vulnerable TPLs in LS_3 has at least one CVE. We obtained the 205 TPL versions in LS_3 from the Maven repository [2] and used them to work on AS_3 and AS_4.

To perform a more in-depth evaluation, we categorize the apps from AS_1 and AS_2 according to the compilers and obfuscation tools used to build the apps. Moreover, there are different obfuscation levels for each obfuscator. Thus, we further differentiate the apps built from the same obfuscator by their obfuscation levels. As a result, the categorization of apps in AS_1 and AS_2 is summarized in Table 5. Of the 785 Orlis benchmark apps in AS_1, 225 are not obfuscated, and the rest are obfuscated by DashO, ProGuard, or Allatori. ATVHunter [15] takes 88 of the 225 non-obfuscated apps and obfuscates these apps with three DashO options, i.e., control-flow randomization (cfr), package flattening+identifier renaming (pf-ir), and dead code removal (dcr). Besides, 181 of the Orlis benchmark apps are obfuscated with DashO on all three options enabled (cfr-pf-ir-dcr). For the 204 apps in AS_2, we first built 51 non-obfuscated apps with D8. Then we configured R8 with three obfuscation levels to build the rest 51×3 apps. Note that code shrinking (shrink) and shrinking+optimization (shrink-opt) are standard obfuscation levels of R8. The third level uses Orlis's ProGuard policy, enhancing the code shrinking level with crafted repackaging and renaming rules (shrink-orlis).

Table 5: Ground-Truth App Datasets Categorization

App Dataset	Category	#apps	#Tuning	Release Source
AS_1	Non-obfs	225	22	Orlis [9]
	Allatori	210	22	
	DashO-cfr	88	9	ATVHunter [15]
	DashO-pf-ir	88	9	
	DashO-dcr	88	9	
	DashO-cfr-pf-ir-dcr	181	22	Orlis [9]
	ProGuard	169	17	
AS_2	D8-non-obfs	51	5	build from source code
	R8-shrink	51	5	
	R8-shrink-opt	51	5	
	R8-shrink-orlis	51	5	

The ground-truth datasets AS_1 and AS_2 were used to evaluate LibScan's effectiveness (Section 5.3). Because AS_1's DashO-obfuscated apps and AS_2 were built on different obfuscation levels, we used these apps to evaluate LibScan's efficacy against different obfuscation levels (Section 5.3.3). We used AS_1 and AS_3 to demonstrate the detection efficiency of LibScan (Section 5.4). To further investigate the scalability of our approach, we applied LibScan to AS_4 to detect the vulnerable TPL versions in LS_3 (Section 5.5). Due to the lack of ground truths, AS_3 and AS_4 cannot be used to measure effectiveness. Only a few ground-truth apps (#Tuning in Table 5) were used in the threshold tuning (Section 5.2).

Our evaluations are designed to answer the following research questions.

- RQ1. How can we establish the optimal thresholds θ_1 in Section 4.3 and θ_2 in Section 4.5?
- RQ2. How can each step of LibScan contribute to LibScan's overall effectiveness?
- RQ3. Can LibScan reach higher precision, recall, and F1 score compared with the state-of-the-art TPL detectors?
- RQ4. How does LibScan perform on different obfuscation levels?
- RQ5. What is LibScan's efficiency compared with the state-of-the-art TPL detectors? How each step of LibScan contributes to its efficiency?
- RQ6. Can LibScan effectively detect potential vulnerable TPLs from large-scale real-world apps? Can the detection results be used to facilitate malware detection?

5.2 Thresholds Tuning (RQ1)

We divide each ground-truth app dataset in Table 5 into a validation set used in tuning the threshold θ_1 and θ_2, and a test set to evaluate the effectiveness of LibScan. For flexibility, we manually set a proportion for each app category in Table 5 (9.5%-12.0%). The tuning procedure produces the validation set by randomly selecting a certain number of apps (#Tuning in Table 5) from each category according to the proportion. As a result, 110 apps in AS_1 and 20 in AS_2 are selected for thresholds tuning, and the rest ground-truth apps are for the evaluations. To guarantee the apps used in the evaluation of Section 5.3.3 have the same ground truths on different obfuscation levels, we ensure the 9×3 DashO-obfuscated apps and 5×4 D8/R8-built apps are the same apps at different obfuscation levels.

Because the value of θ_1 and θ_2 have a coefficient impact on the metrics of our evaluation, we conduct a grid search over different choices of θ_1 and θ_2 to establish the optimal value of θ_1 and θ_2. The metric we use in the grid search is the F1 score. Therefore, before introducing the grid search results, we present the metrics used in the effectiveness evaluations. The TPL *version-level* standard metrics used in the effectiveness evaluations are as follows:

$$Precision = \frac{TP}{TP+FP} \qquad Recall = \frac{TP}{TP+FN}$$

$$F1\text{-}score = \frac{2 \times Recall \times Precision}{Recall + Precision}$$

The *true positive* (TP) refers to the case that the app contains a TPL version, and the tool detects the existence of this TPL version. The *false positive* (FP) means the tool reports some TPL version that does not exist in the app. The *false negative* (FN) represents that a TPL version contained in an app is reported absent by the tool.

We have tried to unify the hyperparameters of LibScan and the related TPL detection tools on AS_1 and AS_2. However, due to R8 and D8's new feature on app compilation and optimization, the hyperparameters of all the detection tools have to be tuned respectively on AS_1 and AS_2. Specifically, LibScan needs lower θ_1 to tolerate more redundant code insertion and lower θ_2 to tolerate more code removal on the D8/R8-built apps. Since detecting whether an APK was built with R8, D8, or the historic compiler DX is straightforward using the R8 retrace tool [14, 18], retuning hyperparameters for D8/R8 built apps is reasonable.

Because θ_1 is only for the method-opcode similarity determination and θ_2's earliest use is at the end of the signature-based detection, we reserve the results prior to the earliest confidence-based TPL detection and iteratively perform the opcode similarity determination steps on different (θ_1,θ_2) instances to accelerate the thresholds tuning procedure. This procedure takes 21∼22 hours on the 110 apps in AS_1 to generate the grid search results in Table 6. We observe that the peak F1 score is reached at (θ_1, θ_2)=(0.7, 0.85) for AS_1 when the step length is 0.05. The generalization ability of the optimal θ_1 and θ_2 supports LibScan's high effectiveness on the 939 apps in AS_1, efficiency on dataset AS_1 and AS_3, and scalability on dataset AS_4. Using similar procedure, we tune (θ_1, θ_2)=(0.2, 0.35) for the D8-built non-obfuscated apps and (θ_1, θ_2)=(0.15, 0.1) on the three obfuscation levels of R8. More hyperparameter tuning results of related tools are in Appendix C.

Table 6: Grid Search on F1-scores to Establish Optimal Threshold θ_1 and θ_2

		θ_1						
		0.65	0.7	0.75	0.8	0.85	0.9	0.95
θ_2	0.5	0.891	0.894	0.894	0.895	0.894	0.885	0.880
	0.55	0.893	0.896	0.896	0.894	0.948	0.939	0.947
	0.6	0.894	0.897	0.897	0.894	0.947	0.938	0.944
	0.65	0.894	0.897	0.952	0.949	0.947	0.938	0.942
	0.7	0.901	0.904	0.958	0.955	0.953	0.944	0.942
	0.75	0.899	0.902	0.958	0.955	0.952	0.956	0.933
	0.8	0.954	0.956	0.956	0.953	0.950	0.952	0.910
	0.85	0.964	**0.967**	0.966	0.965	0.961	0.932	0.883
	0.9	0.944	0.947	0.939	0.921	0.912	0.882	0.805
	0.95	0.838	0.832	0.814	0.811	0.808	0.776	0.743

5.3 Effectiveness Evaluation

Following the traditional effectiveness comparisons of the related works, we evaluate the effectiveness of LibScan at both the library and the version level. The library-level metrics measure different tools' ability to detect specific TPL in the app. At the library level, *true positive* stands for the case the TPL existed in the app detected by the tool. We use TP_0 to represent this library-level true positive and distinguish it from the version-level true positive defined in Section 5.2. The *library-level false positive* (FP_0) means the tool-reported TPL does not exist in the app. The *library-level false negative* (FN_0) means the TPL in the app is reported absent by the tool. Based on the library level TP_0/FP_0/FN_0, we define the library-level metrics $Precision_0/Recall_0/F1\text{-}score_0$. The library-level metrics can be different from the version-level metrics. A typical example is that some tool reports the detection of `okhttp 4.9.0` in an app, but the TPL version is indeed `okhttp 4.9.1`. In such a case, we reach a library-level true positive but a version-level false negative. For each detection tool and a specific ground-truth dataset with ground-truth positive GTP, we have $TP+FN=TP_0+FN_0=GTP$ and $TP+FP=TP_0+FP_0$ due to the relation of library-level and version-level detection. Also, we have $TP_0 \geq TP$, $FP_0 \leq FP$, and $FN_0 \leq FN$.

The effectiveness evaluation is threefold. First, we demonstrate the necessity of LibScan's each step by investigating the metrics of LibScan's different step combinations. Then we measure and compare the standard metrics of different tools at both library and version levels on the app dataset AS_1. Finally, we evaluate LibScan's effectiveness against different obfuscation levels of DashO and R8.

5.3.1 Stepwise Contribution to Effectiveness (RQ2)

LibScan's different detection steps exclude different amounts of app classes. For example, signature-based detection can abandon over 99.6% of app classes on dataset AS_3, as shown in Section 5.4. However, the app classes abandoned at different steps show different disturbance effects to the TPL detection. We demonstrate that the two opcode similarity de-

Table 7: Effectiveness Comparison of Different Tools on 939 apps of Dataset AS_1 (5,956 Ground-Truth TPL Existences)

	Library-level						Version-level					
Tool	TP_0	FP_0	FN_0	$Precision_0$	$Recall_0$	$F1_0$	TP	FP	FN	Precision	Recall	F1
LibID-S	2,209	1,358	3,747	0.6193	0.3709	0.4639	2,192	1,375	3,764	0.6145	0.3680	0.4604
LibID-A	2,098	622	3,858	0.7713	0.3522	0.4836	2,091	629	3,865	0.7688	0.3511	0.4820
LibPecker	4,563	1,798	1,393	0.7173	0.7661	0.7409	4,243	2,118	1,713	0.6670	0.7124	0.6890
Orlis	1,507	**45**	4,449	**0.9710**	0.2530	0.4014	730	822	5,226	0.4704	0.1226	0.1945
LibScout	2,679	314	3,277	0.8951	0.4498	0.5987	2,664	**329**	3,292	0.8901	0.4473	0.5954
$LibScan^{I}$	5,872	2,211	84	0.7265	0.9859	0.8365	5,846	2,237	110	0.7232	0.9815	0.8328
$LibScan^{I+II}$	5,812	1,199	144	0.8290	0.9758	0.8964	5,685	1,326	271	0.8109	0.9545	0.8768
LibScan	**5,741**	326	**215**	0.9463	**0.9639**	**0.9550**	**5,659**	408	**297**	**0.9328**	**0.9501**	**0.9414**

termination steps, even abandoning fewer app classes, are indispensable to LibScan's detection. In Table 7, $LibScan^{I}$ is a setup of LibScan with only signature-based class correspondence detection. $LibScan^{I+II}$ is a setup with both signature-based detection and method-opcode similarity determination. Adding the opcode similarity determinations as LibScan's detection steps generally causes a minor increase in the false negatives but gains a significant decrease in the false positives. Although signature-based class correspondence detection serves as a competitive detection by itself, the opcode similarity determinations are indispensable in outperforming other approaches, especially on precision and F1 score.

5.3.2 Effectiveness Comparison with State-of-the-art TPL Detectors (RQ3)

We compare the effectiveness of LibScan with LibID [58], LibPecker [60], Orlis [54], and LibScout [22]. Here we only compare the complete deployment of LibScan. From Table 7, we know that on the 5,956 ground-truth TPL existences, LibScan generally outperforms other approaches on both library and version levels. Orlis reports a higher precision at the library level but has relatively lower recall. Indeed, the similarity digests of Orlis cause low recalls on both obfuscated and non-obfuscated apps. The fewer false positives of Orlis and LibScout than LibScan are at the cost of their high false negatives. Although reporting low recalls in several cases, Orlis and LibScout are very competitive on precision. This advantage makes them suitable for the scenario when high precision is preferred, e.g., constructing exploits relying on specific TPL. Among these detection tools, only LibPecker and LibScan can effectively restrict the false negatives on obfuscated TPLs. More standard metrics comparisons decomposed against different obfuscators are in Appendix E.

We have not obtained the implementation of ATVHunter [56]; thus, we cannot compare LibScan with ATVHunter on the complete ground-truth app datasets. However, ATVHunter has released ground truths [11] on a subset of AS_1: the 552 ground-truth apps in [11] are a subset of the 785 Orlis benchmark apps, including 138 non-obfuscated apps and 138×3 apps obfuscated with DashO, ProGuard, and Allatori, respectively. ATVHunter detects the ground-truth TPL existence list [10] in these apps. As reported in [11], the 552 apps contain 3,124 ground-truth TPL existences, i.e., TP+FN=3,124, and ATVHunter reports 2,558 true positives on these apps. Therefore, the Recall of ATVHunter is 81.88%. We detect the same TPL existence list on the 552 apps with LibScan. LibScan finds 3,046 true positives and reports 94 false positives and 78 false negatives. The recall reaches 97.50%. The false positives of ATVHunter on the ground truths are unavailable; thus, the precision is incomparable, but we can confirm that our F1 score is better than ATVHunter on the apps.

5.3.3 LibScan's Sensitivity on Obfuscation Levels (RQ4)

We use the 79×4=316 DashO-obfuscated apps in AS_1 and 46×3=138 R8-obfuscated apps in AS_2 to evaluate LibScan's effectiveness against different obfuscation levels of DashO and the R8 compiler, respectively. Of the 316 DashO-obfuscated apps, the 79 apps on the cfr-pf-ir-dcr level are from the 181 Orlis benchmark apps in AS_1 to ensure the 316 apps are indeed 79 apps on 4 different obfuscation levels with the same ground truths. For comparison, the results on the corresponding non-obfuscated apps are also reported.

Table 8 presents the effectiveness of TPL detectors on different DashO obfuscation levels (more details in Table 17). LibScout, Orlis, and LibID reach fewer false positives on these obfuscation levels at the cost of high false negatives, while LibPecker has moderate false positives and false negatives. LibScan reports more false negatives on the dead code removal cases than on other obfuscation levels. However, this sensitivity is remarkable due to the overall high recall. It does not mean LibScan has worse performance than other tools for resisting dead code removal. LibScout and LibID take particular consideration for the precision on obfuscated apps, introducing more false positives on non-obfuscated apps. Interestingly, LibScan, LibScout, and LibID have a better overall effectiveness (F1 score) against the most rigorous level, i.e., cfr-pf-ir-dcr, than against several looser obfuscation levels, indicating the combination of obfuscation techniques may suppress the disturbing effect of some specific technique to these detectors.

Table 9 presents the effectiveness of TPL detectors on D8's compilation and R8's obfuscation levels (more details in Table 18). The opcodes of D8-built app are optimized to a certain degree, affecting LibScan's precision. However, D8 reserves the package structure of TPL; thus, LibScout and LibPecker, which use the package hierarchy information,

Table 8: Effectiveness Comparison of Detection Tools to Different DashO Obfuscation Levels (PR=Precision, RC=Recall)

Detection Level	Obfuscation Level	LibScan			LibScout			Orlis			LibPecker			LibID-A		
		PR_0	RC_0	$F1_0$	PR_0	RC_0	$F1_0$	PR_0	RC_0	$F1_0$	PR_0	RC_0	$F1_0$	PR_0	RC_0	$F1_0$
Library-level	Non-obfustated	0.984	1.000	**0.992**	0.918	0.875	0.896	1.000	0.346	0.515	0.759	0.984	0.857	0.796	0.774	0.785
	DashO-cfr	0.984	0.982	**0.983**	1.000	0.180	0.305	1.000	0.337	0.504	0.723	0.790	0.755	1.000	0.048	0.093
	DashO-pf-ir	0.986	0.984	**0.985**	1.000	0.169	0.289	1.000	0.346	0.515	0.708	0.723	0.715	1.000	0.002	0.005
	DashO-dcr	0.997	0.873	**0.931**	1.000	0.169	0.289	1.000	0.321	0.486	0.708	0.718	0.713	1.000	0.055	0.105
	DashO-cfr-pf-ir-dcr	0.986	0.977	**0.981**	0.936	0.203	0.334	1.000	0.212	0.350	0.527	0.337	0.411	0.870	0.046	0.088
		PR	RC	F1	PR	RC	F1	PR	RC	F1	PR	RC	F1	PR	RC	F1
Version-level	Non-obfustated	0.984	1.000	**0.992**	0.918	0.875	0.896	0.507	0.176	0.261	0.747	0.968	0.843	0.796	0.774	0.785
	DashO-cfr	0.954	0.952	**0.953**	1.000	0.180	0.305	0.493	0.166	0.249	0.641	0.700	0.669	1.000	0.048	0.093
	DashO-pf-ir	0.958	0.956	**0.957**	1.000	0.169	0.289	0.507	0.176	0.261	0.622	0.635	0.629	1.000	0.002	0.005
	DashO-dcr	0.963	0.843	**0.899**	1.000	0.169	0.289	0.482	0.155	0.234	0.622	0.630	0.626	1.000	0.055	0.105
	DashO-cfr-pf-ir-dcr	0.956	0.947	**0.951**	0.936	0.203	0.334	0.446	0.095	0.156	0.516	0.330	0.403	0.870	0.046	0.088

Table 9: Effectiveness Comparison of Detection Tools to Different D8/R8 Obfuscation Levels (PR=Precision, RC=Recall)

Detection Level	Obfuscation Level	LibScan			LibScout			Orlis			LibPecker			LibID-A		
		PR_0	RC_0	$F1_0$	PR_0	RC_0	$F1_0$	PR_0	RC_0	$F1_0$	PR_0	RC_0	$F1_0$	PR_0	RC_0	$F1_0$
Library-level	D8-non-obfs	0.783	0.981	0.871	0.818	0.969	**0.887**	0.579	0.500	0.536	0.786	0.975	0.871	0.821	0.821	0.821
	R8-shrink	0.904	0.580	**0.707**	0.389	0.272	0.320	0.632	0.457	0.530	0.754	0.568	0.648	0.704	0.352	0.469
	R8-shrink-orlis	0.903	0.574	**0.702**	0.488	0.130	0.205	0.630	0.463	0.534	0.739	0.506	0.601	0.585	0.235	0.335
	R8-shrink-opt	1.000	0.080	0.149	0.258	0.105	**0.149**	0.545	0.037	0.069	0.917	0.068	0.126	1.000	0.068	0.127
		PR	RC	F1	PR	RC	F1	PR	RC	F1	PR	RC	F1	PR	RC	F1
Version-level	D8-non-obfs	0.719	0.901	0.800	0.818	0.969	**0.887**	0.336	0.290	0.311	0.716	0.889	0.793	0.753	0.753	0.753
	R8-shrink	0.808	0.519	**0.632**	0.372	0.259	0.305	0.342	0.247	0.287	0.467	0.352	0.401	0.679	0.340	0.453
	R8-shrink-orlis	0.796	0.506	**0.619**	0.488	0.130	0.205	0.361	0.265	0.306	0.441	0.302	0.359	0.569	0.228	0.326
	R8-shrink-opt	0.769	0.062	0.114	0.197	0.080	0.114	0.273	0.019	0.035	0.917	0.068	0.126	1.000	0.068	**0.127**

have good effectiveness on D8-built apps. When the code shrinking is introduced without R8's optimization, i.e., on the levels R8-shrink and R8-shrink-orlis in Table 9, the code shrinking raises the false negatives of LibScan, but LibScan can still outperform other detectors. The crafted repackaging and renaming rules of Orlis's ProGuard policy have a minor effect on disturbing LibScan's detection. R8's optimization over the code shrinking, i.e., R8-shrink-opt in Table 9, significantly reduces the effectiveness of all the TPL detectors. We infer the reason is that R8's optimization intensively removes dead code and uses both method inlining and reflections, as discussed in Section 6.3.

Table 10: Per-App Detection Efficiency of Different Tools on AS_3

	LibID-S(s)	LibPecker(s)	Orlis(s)	LibScout(s)	LibScan(s)
`Q1`	47.52	498.23	51.34	3.40	35.12
`mean`	956.69	797.00	135.66	5.45	45.99
`median`	151.88	741.01	110.21	5.04	44.10
`Q3`	654.63	1036.98	219.62	7.14	57.61

Table 11: Per-App Efficiency Benefit from Different LibScan Detection Steps

	T_1(s)	T_2(s)	T_3(s)	T_4(s)	T_5(s)	T_{total}(s)
LibScan^{III}	29.07	–	–	780.20	10.54	819.81
LibScan^{II+III}	29.07	–	480.10	0.01	10.02	519.20
LibScan	29.07	6.14	0.01	0.01	10.76	45.99

5.4 Efficiency Evaluation (RQ5)

We compare the detection time cost of LibScan with other approaches and evaluate the memory cost of LibScan. We deploy the tools to the environment in Section 5.1. For each app in AS_1, we record the time cost to detect the 452 TPL versions in LS_1. For the popular Google Play apps in AS_3, we record the time cost to detect the 205 vulnerable TPL versions in LS_3. We also record the memory costs of LibScan in this procedure. We use four metrics (`Q1`, `mean`, `median`, `Q3`) to evaluate the time cost. `Q1`, `median`, and `Q3` are the three quartiles that divide the sorted per-app detection times. `Mean` stands for the average detection time. The per-app time costs on AS_3 are presented in Table 10. We observed that on AS_3, LibID, LibPecker, and Orlis conduct more feature matching or digest computation on all the app classes, thus being more time-consuming. LibScout is the most efficient approach because LibScout uses package structures in advance before the class and method matchings, which significantly shortens the detection time but causes LibScout affected by the repackaging of the apps. The average memory cost of LibScan on each app of AS_3 is 4.88 GB. (More results on AS_1 are in Appendix D.)

It is essential to evaluate the necessity of each detection step of LibScan and the potential efficiency loss when disabling one or several of these steps. We observed that on AS_3, LibScan's signature-based detection excludes 99.615% of the app classes as unmatched for TPL class; the method-opcode similarity determination excludes 0.002% of app classes; the subsequent call-chain-opcode similarity determination excludes 0.232% of app classes. Section 5.3.1 has demonstrated the necessity of opcode similarity determinations for effectiveness. Here we investigate the performance benefit of applying signature-based detection before the costly opcode similarity

Table 12: Categories of the 3,949 Existences of the Vulnerable TPL Versions

Vul TPL	#exist	Version	Version #exist
retrofit	1,255	2.2.0	518
		2.3.0	524
		2.4.0	213
nifi-web-content-access	987	1.1.1	987
commons-io	679	2.4	625
		2.5	49
		2.6	5
simple-xml	378	2.7.1	378
okhttp	198	3.10.0	198
httpclient	168	4.3	168
log4j	130	1.2.17	130
fastjson	50	1.2.7	43
		1.2.22	5
		1.2.24	2
bcprov-jdk15	33	1.46	33
commons-collections	31	3.2.1	31
xstream	16	1.4.6	16
jackson-databind	10	2.9.4	6
		2.9.7	4
library	6	1.7.3	6
FileDownloader	6	1.7.3	6
groovy	1	2.4.3	1
bcprov-ext-debug-jdk15on	1	1.57	1

determinations as well as the confidence-score based early stop. Consequently, we deploy two variants of LibScan, i.e., LibScan^{III} and LibScan^{II+III}. LibScan^{III} is a setup that enables the call-chain-opcode similarity determination while disabling the first two steps. LibScan^{II+III} deploys the two opcode similarity determination steps while disabling the signature-based detection. The monotonic exclusion on the class correspondences of the three steps makes these customizations flexible. We compare the average detection time of different setups on dataset AS_3 and decompose the detection time by step. In Table 11, T_1 is the time for feature extraction and caching. T_2 is the time for signature-based class correspondence detection. T_3 and T_4 are the time for method- and call-chain-opcode similarity determination, respectively. T_5 is a cost of the incomplete analysis of some libraries caused by library dependency (depicted in Section 6.1). Table 11 shows that disabling the earlier steps will expose more app-TPL class correspondence decisions to the time-consuming later steps. In summary, LibScan's multi-step detection benefits the detection effectiveness (Section 5.3.1) and efficiency (Section 5.4).

5.5 Large-Scale Vulnerable TPL Detection (RQ6)

This section conducts a large-scale TPL detection on 100,000 real-world apps with LibScan to demonstrate its scalability in detecting real-world vulnerable TPLs. On app dataset AS_4, we detect the 205 vulnerable TPL versions in LS_3. We find that 3,664 apps in app dataset AS_4 are detected to use at least one of the vulnerable TPL versions. The vulnerable TPL versions have 3,949 existences, and after deduplication, 23 out of 205 TPL versions are found in the apps. The results validate our approach over the long-term and large-scale wild apps. We further investigate the categories of the 3,949 existences of the TPL versions, as presented in Table 12. Due to the absence of ground truths, we consulted two independent app developers to manually confirm the correctness of 5% of the vulnerable TPL detections in the big categories (#*exist* > 100). Of the 190 TPL detections, 186 were manually confirmed, and four were unknown. We did not try to confirm the TPL versions manually, since the potential obfuscation made the task impractical by human effort.

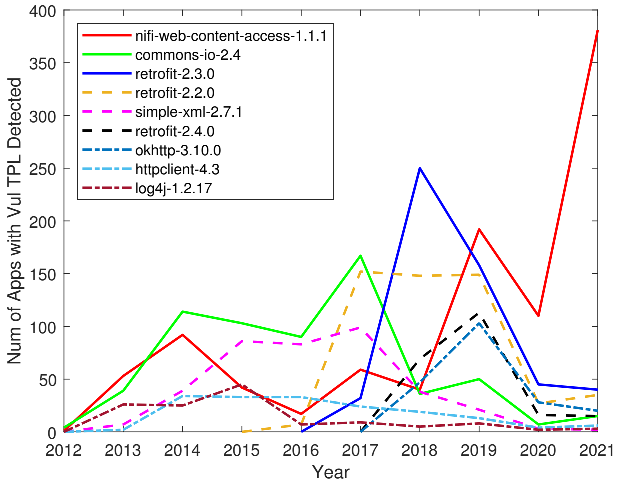

Figure 2: Annual Existences of Top Vulnerable TPL Versions

We figure out how the top detected TPL versions are distributed in the apps released yearly. Following this evidence, the developers may predict and mitigate the potential TPL exploits more effectively. The distributions of the top 9 TPL versions are shown in Figure 2. We choose 10,000 apps for each year; thus, these usage distributions also reflect the prospective usage frequencies. We notice that several TPL versions confirmed to be vulnerable long time ago, e.g., `nifi-web-content-access-1.1.1` and `retrofit-2.3.0/2.2.0`, are still used in a considerable number of recent apps. Especially the usage of `nifi-web-content-access-1.1.1` even increases in 2021. Such frequent usage of the old TPLs also inspires us to investigate the app maliciousness potentially correlated with the vulnerabilities of TPLs.

Even if the existence of vulnerable TPLs alone is insufficient for confirming the malicousness of an app, we believe vulnerable TPLs can be used as a feature to help improve malware detection. Hence, we did a proof-of-concept experiment to demonstrate this direction. We propose to use the vulnerable TPL existence to improve the accuracy of fuzzy-hash similarity based clustering, which can be used for malware detection. In detail, we put apps into one cluster if two conditions are satisfied: 1) every pair of apps in the cluster have two Dex files, one from each app, which have an ssdeep [8]

Table 13: AV Vendor Mark Updates of VirusTotal on Different CooTek App Clusters

Cluster ID	0				0’								1		2	3		
#Vendor reported	27	1	25	25	15	0	0	0	0	0	0	0	7	10	28	1	1	1
#Vendor reanalyzed	30	27	26	26	28	24	21	19	19	18	18	17	10	10	20	8	8	8

similarity score >95%; 2) the apps use the same vulnerable TPLs. As a result, each cluster shares high code similarity and depends on the same vulnerable TPLs. Then, we label apps as benign or malicious according to VirusTotal [1]. We deem an app malicious as long as there are at least 10 vendors on VirusTotal detecting the app as malware; otherwise, we say it is *VT-undetected*. Then, if a cluster contains one malware, we propagate the malware verdict to the whole cluster. In this process, requiring apps sharing the same vulnerable TPLs increases the confidence of propagating the malware verdict to VT-undetected apps.

Since our detection approach is to propagate malware verdicts to VT-undetected samples, our detection gives a superset of what VirusTotal can detect. Therefore, we cannot rely on VirusTotal to perform a systematic efficacy evaluation of our detection approach. Instead, we decide to do manual checking for some outstanding cases. During our manual checking, we found an interesting methodology to validate our detection result. That is, anti-virus vendors may flip their verdicts for apps within a period of time after the first time they saw the app. However, VirusTotal triggers re-analysis for an app only on user demand. Thus, the vendor verdicts on VirusTotal could be outdated. Thanks to this phenomenon, we had a chance to use outdated verdicts to predict up-to-date verdicts. Specifically, we do verdict propagation based on outdated verdicts and we validate our detection results using the verdicts after we request for re-analyzing the apps.

We conducted clustering-based detection on the 3,664 apps detected to use the vulnerable TPLs in LS_3. We label the apps using VirusTotal’s reports for these apps, with their report generation dates between May 2019 and Sep. 2022. We focused on the clusters whose apps have contradicting verdicts and found an interesting cluster of four apps. In this cluster, two *Graffiti* v6.3.22.2019, one *Sparkling Purple Heart* v6.3.22.2019, and one *Fingerprint Style* v6.3.26.2019 are clustered to have highly similar Dex files and use the vulnerable TPL `retrofit-2.4.0` (Cluster 0 in Table 13). They are all provided by the same app-vendor CooTek. 27 vendors of VirusTotal reported one *Graffiti* malicious. Only one vendor of VirusTotal reported the other *Graffiti*. 25 vendors of VirusTotal reported maliciousness for the other two apps. Based on our verdict propagation strategy, we predicted the one-vendor-reported *Graffiti* as malicious. Then, we demanded VirusTotal to reanalyze the four apps in Feb. 2023. We found the AV vendor marks were updated from (27, 1, 25, 25) to (30, 27, 26, 26), which validated our detection result.

To further understand the contribution of the vulnerable TPL existence in our proof-of-concept approach, we collected around 70 apps from Koodous [40] released by CooTek to extend our experiment. We re-clustered these CooTek apps and collected their vendor marks (both outdated and up-to-date versions) on VirusTotal. Generated clusters are shown in Table 13. First, 8 more apps were added into the Cluster 0 to form Cluster 0+0’, in which our verdict propagation can predict the correct maliciousness for the extra 7 apps. Similarly, our approach works for Cluster 1. Then, we discharged the requirement of sharing the same vulnerable TPLs during our clustering. As a result, we observed differences in the clustering result: a *Graffiti boy* v6.3.23.2019 app (marked as Cluster 2) was merged into Cluster 0+0’; and three more apps (i.e., represented as Cluster 3) were added into Cluster 1. However, the decreasing vendor marks of the *Graffiti boy* app may indicate that it could be a false positive of vendors; and apps in Cluster 3 are not considered as malware by our definition. Thus, if we did not involve the existence of vulnerable TPL in the clustering, our verdict propagation would detect all apps in Cluster 3 as well as the *Graffiti boy* app, resulting in 3 false positives and 1 potential false positive. In all, we conclude that the existence of vulnerable TPL is a prominent auxiliary feature for malware detection.

6 Discussions

6.1 TPL Dependencies Identification

A TPL may be developed based on other TPLs. For example, the HTTP library `okhttp3` [13] is developed based on the stream library `okio` [4]. When the app uses `okhttp3`, it will also import the related `okio` library into the app. The usage of `Maven` and `Gradle` makes dependency management easy.

Our call-chain-opcode similarity determination relies on the opcode inclusion relation between the app and TPL call chains. The effectiveness of this fine-grained analysis significantly depends on the cohesiveness of the TPLs in the app. Usually, a TPL method will only call the methods in the current library. When the TPL is imported into the app, this characteristic remains in most cases. However, when a TPL depends on other TPLs, such characteristics will be broken after these TPLs are imported into the app. Specifically, when analyzing the library Jar file, a part of the call chain in the depended TPLs will be missed. On the other hand, when analyzing the in-app TPL method, because the depended TPLs are also imported, the call chain in the app will be complete.

We cannot confirm in the app if some callee method belongs to a depended TPL. Therefore, we try to recover the partial call chains missed in the TPL code. Specifically, when we analyze the class correspondences of an app and the TPL l_1, the features extracted from l_1 may indicate that method m_{l_1}

calls a method m_{callee}, and m_{callee} is neither a standard API nor an app method. In this case, we infer m_{callee} as a potential method in another library depended by l_1. We use m_{callee}'s fully qualified name to search to decide m_{callee}'s existence in other TPLs. If m_{callee} is in the TPL l_2, we merge the partial call chain of the method m_{callee} in l_2 into the call chain of l_1. Then, the call-chain-opcode similarity determination in Section 4.4 can be conducted. In our implementation, if the depended TPL l_2 has not been analyzed and there is no idle working process available to analyze l_2, LibScan will drop the incomplete analysis of l_1 and start analyzing l_2. In other words, the working process for l_1 is preempted. Therefore the time cost T_5 in Table 11 comes into being.

6.2 Optimizations on Performance

Although the multi-step detection procedure of LibScan has reached a reasonable balance between detection effectiveness and efficiency, we believe LibScan's performance can be further improved from the following aspects. First, feature extraction and caching have considerable overhead, as illustrated in Table 11. The dimensions of the boolean vector of the features can be reduced when considering the application domain of the TPL. For example, some string-operation TPL may neither have primitive-array nor non-standard-array return type/parameters. Therefore, the feature extraction can be accelerated by preparing a shorter feature vector of domain-specific TPLs and apps. However, LibScan is a general-purpose TPL detection approach, and we prefer to address more general code features. Second, the highest overhead comes from deciding the call-chain-opcode inclusion relation between the call chains of TPL and the app. To reduce the complexity of the opcodes inclusion decision, we may truncate the call chains with a smaller length bound. The truncation can increase the number of methods in the call-chain-matched TPL method set and change the optimal thresholds.

6.3 Threats to Validity

The method- and call-chain-opcode similarity determinations rely on the opcode inclusion relation to catch the similarity of TPL and app. This approach has a high resistance to repackaging. Also, it is intuitively natural to resist the cases when obfuscators insert redundant code or conduct control-flow randomization in the used TPL methods.

Table 8 shows that LibScan is more sensitive to dead code removal. In such cases, particular opcode types would be removed to make the set-based opcode inclusion decision difficult. The threshold θ_2 can determine LibScan's sensitivity on dead code removal. LibScan has good effectiveness if only a small percentage of the TPL code is removed. Removing a significant portion of the TPL code would cause a low confidence score for detecting a TPL, thus leading to false negatives. However, if the majority of a vulnerable TPL is removed, its vulnerability could also be removed. Hence, such false negatives may not cause severe consequences. Considering the cases where the remaining minority TPL code contains vulnerability, we admit that library-level detection is insufficient; we need class/method-level detection. Although our approach has class/method-level features, we cannot ensure a precise class-level or method-level detection with LibScan.

Our call-chain-opcode similarity determination is sensitive to the Java reflection mechanism. Several large-scale studies [27, 33] have mentioned using reflection techniques for obfuscation. Reflection is a transformation that converts direct method invocations into reflective calls using reflection APIs. This transformation can modify the call chain or remove a part of the call chain from the original call graph; thus, it can evade our call-chain-opcode similarity determination that relies on the correlation between the direct method calls.

When different TPL versions have only minor code differences, LibScan may be ineffective in distinguishing them. For example, `com.android.support.support-v13.23.1.1` and `com.android.support.support-v13.23.1.0` caused LibScan to raise a false negative by reporting the incorrect version. However, this should be a common issue for most code-based TPL detectors. Besides, a TPL may be developed based on another popular TPL and only introduce a small amount of new code. This situation causes the high similarity of the two TPLs; thus, LibScan may build more class correspondences and raise a false alarm.

LibScan could also be sensitive to traditional optimization techniques, e.g., method inlining and method wrapping. Method inlining affects LibScan's method-opcode similarity determination, while wrapping TPL methods into proxy class affects the class- and field-level features of the signature-based detection. We can disable the signature-based detection or method-opcode similarity determination to support detecting these obfuscations, given that all the steps have similar input-output interfaces in the work pipeline. However, disabling these steps would sacrifice efficiency, which is an effectiveness versus performance tradeoff for LibScan.

7 Conclusion

We proposed LibScan, an accurate third-party library detection approach for Android apps. LibScan fingerprints code features to initialize class correspondence relations prior to the method- and call-chain-opcode similarity determinations that can reduce the false class correspondences of the overall detection. The confidence score decisions are applied on the monotone-decreasing number of class correspondences of each detection step to ensure the early stop and efficiency of the detection. Taking proper threshold settings, LibScan outperforms most of the state-of-the-art TPL detections on effectiveness and efficiency and has superior scalability. The temporal analysis of large-scale apps demonstrates LibScan's ability to predict more threatening TPLs and more security analysis indicates that LibScan can be used as an auxiliary tool to help malware detection.

Acknowledgments

Yafei Wu, Cong Sun, and Peicheng Wang were supported by the National Natural Science Foundation of China (Key Program 62232013, 62272366) and the Key Research and Development Program of Shaanxi (2023-YBGY-371).

Availability

The implementation of LibScan has been made publicly available at https://github.com/wyf295/LibScan.

References

[1] VirusTotal. https://www.virustotal.com/gui/home/upload.

[2] MVNRepository, 2006-2022. https://mvnrepository.com.

[3] F-Droid, 2010-2022. https://f-droid.org/en/packages/.

[4] Okio, 2013. https://square.github.io/okio/.

[5] Types, methods and fields of smali, 2015. https://github.com/JesusFreke/smali/wiki/TypesMethodsAndFields.

[6] Androguard, 2016. https://github.com/androguard/androguard.

[7] AndroZoo, 2016. https://androzoo.uni.lu/.

[8] ssdeep, 2017. https://ssdeep-project.github.io/ssdeep/index.html.

[9] Orlis benchmarks, 2018. https://github.com/presto-osu/orlis-orcis/tree/master/orlis/open_source_benchmarks.

[10] ATVHunter Ground-Truth TPLs, 2019. https://drive.google.com/drive/folders/1IeA1dTLVEt3IAL5nUNzICWB4GMKeoI02.

[11] ATVHunter-Results, 2019. https://docs.google.com/spreadsheets/d/1CDwNZdRh8ZZUAmt9emApgqnNQL6JAhDQ/edit#gid=1800752455.

[12] ATVHunter Vulnerable TPLs, 2019. https://docs.google.com/spreadsheets/d/1SmxtXc28qdlvpBo7ZHGXSpGT2I59FwiPPMSX4LAMY6g/edit#gid=0.

[13] Okhttp3, 2019. https://square.github.io/okhttp/.

[14] r8.jar, 2019-2023. https://maven.google.com/web/index.html#com.android.tools:r8.

[15] ATVHunter, 2020. https://sites.google.com/view/atvhunter/home.

[16] dex2jar, 2021. https://github.com/pxb1988/dex2jar.

[17] D8, 2022. https://developer.android.google.cn/studio/command-line/d8.

[18] R8 retrace, 2023. https://developer.android.com/studio/command-line/retrace.

[19] Shrink, obfuscate, and optimize your app, 2023. https://developer.android.com/studio/build/shrink-code.

[20] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: collecting millions of android apps for the research community. In *MSR'16*, pages 468–471, New York, NY, USA, 2016. ACM.

[21] Sumaya Almanee, Arda Ünal, Mathias Payer, and Joshua Garcia. Too quiet in the library: An empirical study of security updates in android apps' native code. In *ICSE'21*, pages 1347–1359. IEEE, 2021.

[22] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *CCS'16*, pages 356–367. ACM, 2016.

[23] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. Brahmastra: Driving apps to test the security of third-party components. In *23rd USENIX Security Symposium*, pages 1021–1036. USENIX Association, 2014.

[24] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin T. Vechev. Statistical deobfuscation of android applications. In *CCS'16*, pages 343–355. ACM, 2016.

[25] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *ICSE '14*, pages 175–186. ACM, 2014.

[26] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In *CCS'17*, pages 2187–2200. ACM, 2017.

[27] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. Understanding android obfuscation techniques: A large-scale investigation in the wild. In *SecureComm'18*, volume 254 of *LNICST*, pages 172–192. Springer, 2018.

[28] Yue Duan, Lian Gao, Jie Hu, and Heng Yin. Automatic generation of non-intrusive updates for third-party libraries in android applications. In *RAID'19*, pages 277–292. USENIX Association, 2019.

[29] William Enck, Damien Octeau, Patrick D. McDaniel, and Swarat Chaudhuri. A study of android application security. In *20th USENIX Security Symposium*. USENIX Association, 2011.

[30] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. CodeMatch: obfuscation won't conceal your repackaged app. In *ESEC/FSE'17*, pages 638–648. ACM, 2017.

[31] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *WiSec'12*, pages 101–112. ACM, 2012.

[32] Guardsquare. ProGuard, 2016-2022. https://www.guardsquare.com/proguard.

[33] Mahmoud Hammad, Joshua Garcia, and Sam Malek. A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products. In *ICSE'18*, pages 421–431. ACM, 2018.

[34] Qiang He, Bo Li, Feifei Chen, John C. Grundy, Xin Xia, and Yun Yang. Diversified third-party library prediction for mobile app development. *IEEE Trans. Software Eng.*, 48(2):150–165, 2022.

[35] Fu-Hau Hsu, Nien-Chi Liu, Yanling Hwang, Che-Hao Liu, Chuan-Sheng Wang, and Chang-Yi Chen. DPC: A dynamic permission control mechanism for android third-party libraries. *IEEE Trans. Dependable Secur. Comput.*, 18(4):1751–1761, 2021.

[36] Wenhui Hu, Damien Octeau, Patrick D. McDaniel, and Peng Liu. Duet: library integrity verification for android applications. In *WiSec'14*, pages 141–152. ACM, 2014.

[37] Jie Huang, Nataniel P. Borges Jr., Sven Bugiel, and Michael Backes. Up-To-Crash: evaluating third-party library updatability on android. In *EuroS&P'19*, pages 15–30. IEEE, 2019.

[38] Jie Huang, Oliver Schranz, Sven Bugiel, and Michael Backes. The ART of app compartmentalization: Compiler-based library privilege separation on stock android. In *CCS'17*, pages 1037–1049. ACM, 2017.

[39] Smardec Inc. Allatori, 2021. http://www.allatori.com/.

[40] MI21 Malware Intelligent. Koodous, 2015-2022. https://koodous.com/.

[41] Bodong Li, Yuanyuan Zhang, Juanru Li, Runhan Feng, and Dawu Gu. APPCOMMUNE: automated third-party libraries de-duplicating and updating for android apps. In *SANER'19*, pages 344–354. IEEE, 2019.

[42] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. LibD: scalable and precise third-party library detection in android markets. In *ICSE'17*, pages 335–346. IEEE / ACM, 2017.

[43] Bin Liu, Hongxia Jin, and Ramesh Govindan. Efficient privilege de-escalation for ad libraries in mobile apps. In *MobiSys'15*, pages 89–103. ACM, 2015.

[44] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. LibRadar: fast and accurate detection of third-party libraries in android apps. In *ICSE'16*, pages 653–656. ACM, 2016.

[45] Annamalai Narayanan, Lihui Chen, and Chee Keong Chan. AdDetect: automated detection of android ad libraries using semantic analysis. In *IEEE ISSNIP*, pages 1–6. IEEE, 2014.

[46] Gabor Paller. Dalvik Opcodes. https://quantiti.github.io/dalvik-opcodes/.

[47] PreEmptive. DashO, 2021. https://www.preemptive.com/products/dasho/.

[48] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. FLEXDROID: enforcing in-app privilege separation in android. In *NDSS'16*. The Internet Society, 2016.

[49] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. AdSplit: separating smartphone advertising from applications. In *21th USENIX Security Symposium*, pages 553–567. USENIX Association, 2012.

[50] Charlie Soh, Hee Beng Kuan Tan, Yauhen Leanidavich Arnatovich, Annamalai Narayanan, and Lipo Wang. LibSift: automated detection of third-party libraries in android applications. In *APSEC'16*, pages 41–48. IEEE Computer Society, 2016.

[51] Mengtao Sun and Gang Tan. NativeGuard: protecting android applications from third-party native libraries. In *WiSec'14*, pages 165–176. ACM, 2014.

[52] Zhushou Tang, Minhui Xue, Guozhu Meng, Chengguo Ying, Yugeng Liu, Jianan He, Haojin Zhu, and Yang Liu.

Securing android applications via edge assistant third-party library detection. *Comput. Secur.*, 80:257–272, 2019.

[53] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. WuKong: a scalable and accurate two-phase approach to android app clone detection. In *ISSTA'15*, pages 71–82. ACM, 2015.

[54] Yan Wang, Haowei Wu, Hailong Zhang, and Atanas Rountev. ORLIS: obfuscation-resilient library detection for android. In *MOBILESoft@ICSE 2018*, pages 13–23. ACM, 2018.

[55] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Hui Liu, Qing Wang, Yueheng Zhang, and Dawu Gu. Show me the money! finding flawed implementations of third-party in-app payment in android apps. In *NDSS'17*. The Internet Society, 2017.

[56] Xian Zhan, Lingling Fan, Sen Chen, Feng Wu, Tianming Liu, Xiapu Luo, and Yang Liu. ATVHUNTER: reliable version detection of third-party libraries for vulnerability identification in android applications. In *ICSE'21*, pages 1695–1707. IEEE, 2021.

[57] Xian Zhan, Lingling Fan, Tianming Liu, Sen Chen, Li Li, Haoyu Wang, Yifei Xu, Xiapu Luo, and Yang Liu. Automated third-party library detection for android applications: Are we there yet? In *ASE'20*, pages 919–930. IEEE, 2020.

[58] Jiexin Zhang, Alastair R. Beresford, and Stephan A. Kollmann. LibID: reliable identification of obfuscated third-party android libraries. In *ISSTA'19*, pages 55–65. ACM, 2019.

[59] Xiao Zhang, Amit Ahlawat, and Wenliang Du. AFrame: isolating advertisements from mobile applications in android. In *ACSAC '13*, pages 9–18. ACM, 2013.

[60] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. Detecting third-party libraries in android applications with high precision and recall. In *SANER'18*, pages 141–152. IEEE Computer Society, 2018.

[61] Zicheng Zhang, Wenrui Diao, Chengyu Hu, Shanqing Guo, Chaoshun Zuo, and Li Li. An empirical study of potentially malicious third-party libraries in android apps. In *WiSec'20*, pages 144–154. ACM, 2020.

[62] Wu Zhou, Zhi Wang, Yajin Zhou, and Xuxian Jiang. DIVILAR: diversifying intermediate language for anti-repackaging on android platform. In *CODASPY'14*, pages 199–210. ACM, 2014.

[63] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *CODASPY'12*, pages 317–326. ACM, 2012.

A TPL Detection Algorithm

We know from Section 4.5 and 4.6 that there are different $Match(X,Y)$ predicates at different steps of LibScan. Therefore, these predicates decide different TPL class sets for $M(X,Y)$ in Section 4.5, e.g., M_1 for the signature-based class correspondence detection, M_2 for the method-opcode similarity determination, and M_3 for the call-chain-opcode similarity determination. Algorithm 1 presents the TPL detection algorithm used at each determination step, as demonstrated in Section 4.6.

Algorithm 1: TPL Detection Algorithm

```
input : App $X$, TPL $Y$, and the TPL class set
        $\mathcal{M} = M_1, M_2$, or $M_3$
1  $conf \leftarrow 0$;
2  $O_Y \leftarrow \sum_{c \in Y} \text{csize}(c)$;
3  $matchedOps \leftarrow 0$;
4  forall $c \in \mathcal{M}(X,Y)$ do
5      $matchedOps \leftarrow matchedOps + \text{csize}(c)$;
6  $conf \leftarrow matchedOps / O_Y$;
7  if $conf < \theta_2$ then
8      return false;
9  else if $\mathcal{M} = M_1$ or $M_2$ then
10     return next step trigger
11 else                                  /* $\mathcal{M} = M_3$ */
12     return true;
```

B CVEs of 13 Vulnerable TPL Versions in LS_3

Table 14 presents the 13 popular vulnerable TPL versions of LS_3 mentioned in Section 5.1 and the related CVE information.

Table 14: CVEs w.r.t. 13 Popular Vulnerable TPLs in LS_3

TPL version	CVE No.
FileDownloader-1.7.3	CVE-2018-11248
log4j-core-2.12.1	CVE-2021-45105
log4j-core-2.12.3	CVE-2021-44832
log4j-core-2.14.0	CVE-2021-44832
retrofit-2.2.0	CVE-2018-1000850
retrofit-2.3.0	CVE-2018-1000850
retrofit-2.4.0	CVE-2018-1000850
commons-io-2.4	CVE-2021-29425
commons-io-2.5	CVE-2021-29425
commons-io-2.6	CVE-2021-29425
fastjson-1.2.22	CVE-2017-18349
fastjson-1.2.23	CVE-2017-18349
fastjson-1.2.24	CVE-2017-18349

C Configurations of Related Approaches

To make the effectiveness comparisons fair, we also use the hyperparameter-tuning apps in datasets AS_1 and AS_2 to establish the optimal hyperparameters for the related tools, i.e., LibID [58], LibPecker [60], Orlis [54], and LibScout [22].

LibScout. *min_partial_matching _score* takes the threshold of 0.8 for the apps in AS_1, 0.9 for the D8-built apps (D8-non-obfs in Table 5), and 0.1 for R8-obfuscated apps (R8-shrink, R8-shrink-orlis, and R8-shrink-opt in Table 5).

Orlis. Orlis provides the decision of class-level correspondences. To compare Orlis with LibScan on the library level, we apply LibScan's library-level decision (Section 4.5) on Orlis. On AS_1, the threshold for the digest functions other than TLSH takes the default value of 0. For the D8-built apps in AS_2, this threshold takes 50. For the R8-obfuscated apps in AS_2, this threshold takes the default value of 0.

LibPecker. The *library similarity threshold* takes 0.7, and the *package similarity threshold* takes 0.5 for the apps in AS_1. On the D8-built apps, the *library similarity threshold* takes 0.9, and the *package similarity threshold* takes 0.5. On the R8-obfuscated apps, the *library similarity threshold* takes 0.5, and the *package similarity threshold* takes 0.1.

LibID. For LibID-S, the *probability_threshold_scalable* (Γ_1) takes 0.8, and the *shrink_threshold_scalable* (Γ_2) takes 0.4. For LibID-A, Γ_1 takes 0.9, and Γ_2 takes 0.8 for the apps in AS_1. For the D8-built apps, LibID-A's Γ_1 takes 0.85, and Γ_2 takes 0.9. For the R8-obfuscated apps, LibID-A's Γ_1 takes 0.4, and Γ_2 takes 0.4.

The other parameters of these related tools take their default values in our settings.

D More Results on Detection Efficiency of LibScan

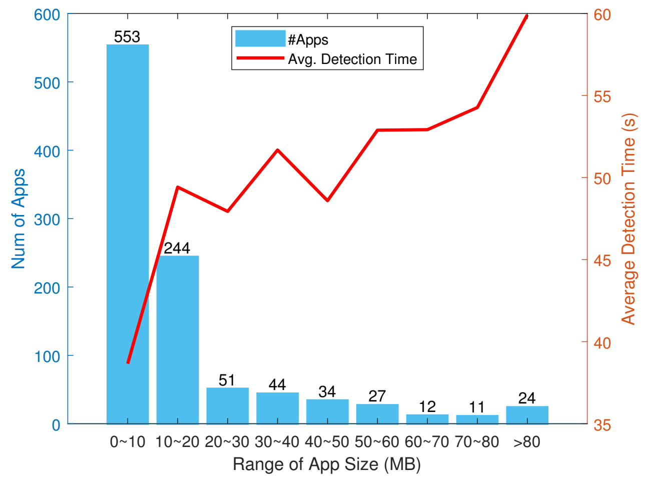

Figure 3: LibScan's Detection Time Costs on Apps with Different Sizes

Table 15: Per-App Detection Efficiency of Different Tools on AS_1

	LibID-S(s)	LibPecker(s)	Orlis(s)	LibScout(s)	LibScan(s)
Q1	10.08	250.29	39.66	1.17	22.08
mean	72.14	307.75	52.98	1.35	24.18
median	64.92	290.54	51.52	1.30	23.56
Q3	103.69	344.62	64.41	1.49	26.74

Fig. 3 presents a comprehensive report based on LibScan's evaluation results in Table 10. The size of the apps in app dataset AS_3 varies from 15.7 KB to 553 MB, with an average size of 17.6 MB. In Fig. 3, we show the average detection time on different app size ranges of AS_3, as well as the quantitative distribution of the number of apps in different size ranges. The detection time is not strictly positively correlated with the app size because the detection efficiency also depends on the number of potential TPLs and the size of methods (call graphs) with correspondences.

On the app dataset AS_1, the time costs of different detection tools are in Table 15, and the average memory cost of LibScan on AS_1 is 3.43 GB. Though most of the apps in AS_1 are intentionally obfuscated, they have smaller sizes (2.0 MB on average); thus, the overall detection cost is lower than the apps in AS_3 (Table 10).

E Effectiveness Comparisons Decomposed against Different Obfuscators or Obfuscation Levels

To elaborate on the effect of LibScan on different obfuscators, we classify the 939 apps in AS_1 into four categories: 1) the 203 non-obfuscated apps with 1,428 ground-truth TPL existences, 2) the 188 apps obfuscated by Allatori with 1,376 ground-truth TPL existences, 3) the 396 apps obfuscated by DashO with 2,330 ground-truth TPL existences, and 4) the 152 apps obfuscated by ProGuard with 822 ground-truth TPL existences. The results of different TPL detection tools and LibScan's step combinations on AS_1 against different obfuscators are presented in Table 16. We observe that LibID and LibScout report considerably low recall on the apps obfuscated by Allatori and DashO. The low recalls could be caused by the control-flow randomization over the Allatori-obfuscated and DashO-obfuscated apps. In contrast, ProGuard does not support control-flow randomization; thus, LibID and LibScout behave effectively on recall.

Table 17 presents the effectiveness of TPL detectors in true positives, false positives, and false negatives on different DashO obfuscation levels. Table 18 presents the TP/FP/FN of the TPL detectors on different D8/R8 obfuscation levels.

Table 16: Effectiveness Comparisons on AS_1's Ground-Truth Apps

App Category	#Ground-Truth TPL Existences	Tool	Library-level						Version-level					
			TP$_0$	FP$_0$	FN$_0$	Precision$_0$	Recall$_0$	F1$_0$	TP	FP	FN	Precision	Recall	F1
Non-Obfs	1,428	LibID-S	1,162	542	266	0.6819	0.8137	0.7420	1,157	547	271	0.6790	0.8102	0.7388
		LibID-A	1,161	302	267	0.7936	0.8130	0.8032	1,158	305	270	0.7915	0.8109	0.8011
		LibPecker	1,405	440	23	0.7615	0.9839	0.8585	1,358	487	70	0.7360	0.9510	0.8298
		Orlis	386	**19**	1,042	**0.9531**	0.2703	0.4212	195	210	1,233	0.4815	0.1366	0.2128
		LibScout	1,269	163	159	0.8862	0.8887	0.8874	1,260	172	168	0.8799	0.8824	0.8811
		LibScan^{I}	1,421	530	7	0.7283	0.9951	0.8411	1,414	537	14	0.7248	0.9902	0.8369
		LibScan^{I+II}	1,420	312	8	0.8199	0.9944	0.8987	1,412	320	16	0.8152	0.9888	0.8937
		LibScan	**1,417**	130	**11**	0.9160	**0.9923**	**0.9526**	**1,411**	**136**	**17**	**0.9121**	**0.9881**	**0.9486**
Allatori	1,376	LibID-S	280	355	1,096	0.4409	0.2035	0.2785	272	363	1,104	0.4283	0.1977	0.2705
		LibID-A	126	72	1,250	0.6364	0.0916	0.1601	125	73	1,251	0.6313	0.0908	0.1588
		LibPecker	982	350	394	0.7372	0.7137	0.7253	862	470	514	0.6471	0.6265	0.6366
		Orlis	222	**2**	1,154	**0.9911**	0.1613	0.2775	111	113	1,265	0.4955	0.0807	0.1388
		LibScout	192	6	1,184	0.9697	0.1395	0.2440	192	**6**	1,184	**0.9697**	0.1395	0.2440
		LibScan^{I}	1,374	431	2	0.7612	0.9985	0.8639	1,360	445	16	0.7535	0.9884	0.8551
		LibScan^{I+II}	1,322	227	54	0.8535	0.9608	0.9039	1,295	254	81	0.8360	0.9411	0.8855
		LibScan	**1,282**	51	**94**	0.9617	**0.9317**	**0.9465**	**1,274**	59	**102**	0.9557	**0.9259**	**0.9406**
DashO	2,330	LibID-S	138	50	2,192	0.7340	0.0592	0.1096	137	51	2,193	0.7287	0.0588	0.1088
		LibID-A	179	22	2,151	0.8905	0.0768	0.1414	179	22	2,151	0.8905	0.0768	0.1414
		LibPecker	1,371	680	959	0.6685	0.5884	0.6259	1,246	805	1,084	0.6075	0.5348	0.5688
		Orlis	627	**9**	1,703	**0.9858**	0.2691	0.4228	297	339	2,033	0.4670	0.1275	0.2003
		LibScout	497	18	1,833	0.9650	0.2133	0.3494	495	**20**	1,835	**0.9612**	0.2124	0.3480
		LibScan^{I}	2,261	860	69	0.7244	0.9704	0.8296	2,260	861	70	0.7241	0.9700	0.8292
		LibScan^{I+II}	2,254	436	76	0.8379	0.9674	0.8980	2,167	523	163	0.8056	0.9300	0.8633
		LibScan	**2,227**	63	**103**	0.9725	**0.9558**	**0.9641**	**2,164**	126	**166**	0.9450	**0.9288**	**0.9368**
ProGuard	822	LibID-S	629	411	193	0.6048	0.7652	0.6756	626	414	196	0.6019	0.7616	0.6724
		LibID-A	632	226	190	0.7366	0.7689	0.7524	629	229	193	0.7331	0.7652	0.7488
		LibPecker	805	328	17	0.7105	0.9793	0.8235	777	356	45	0.6858	0.9453	0.7949
		Orlis	272	**15**	550	**0.9477**	0.3309	0.4905	127	160	695	0.4425	0.1545	0.2290
		LibScout	721	127	101	0.8502	0.8771	0.8635	717	131	105	0.8455	0.8723	0.8587
		LibScan^{I}	816	390	6	0.6766	0.9927	0.8047	812	394	10	0.6733	0.9878	0.8008
		LibScan^{I+II}	816	224	6	0.7846	0.9927	0.8765	811	229	11	0.7798	0.9866	0.8711
		LibScan	**815**	82	**7**	0.9086	**0.9915**	**0.9482**	**810**	**87**	**12**	**0.9030**	**0.9854**	**0.9424**

Table 17: TPL Detection Tools Sensitivity to Different DashO Obfuscation Levels in TP/FP/FN

Detection Level	Obfuscation Level	LibScan			LibScout			Orlis			LibPecker			LibID-A		
		TP$_0$	FP$_0$	FN$_0$	TP$_0$	FP$_0$	FN$_0$	TP$_0$	FP$_0$	FN$_0$	TP$_0$	FP$_0$	FN$_0$	TP$_0$	FP$_0$	FN$_0$
Library-level	Non-obfustated	433	7	0	379	34	54	150	0	283	426	135	7	335	86	98
	DashO-cfr	425	7	8	78	0	355	146	0	287	342	131	91	21	0	412
	DashO-pf-ir	426	6	7	73	0	360	150	0	283	313	129	120	1	0	432
	DashO-dcr	378	1	55	73	0	360	139	0	294	311	128	122	24	0	409
	DashO-cfr-pf-ir-dcr	423	6	10	88	6	345	92	0	341	146	131	287	20	3	413
		TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
Version-level	Non-obfustated	433	7	0	379	34	54	76	74	357	419	142	14	335	86	98
	DashO-cfr	412	20	21	78	0	355	72	74	361	303	170	130	21	0	412
	DashO-pf-ir	414	18	19	73	0	360	76	74	357	275	167	158	1	0	432
	DashO-dcr	365	14	68	73	0	360	67	72	366	273	166	160	24	0	409
	DashO-cfr-pf-ir-dcr	410	19	23	88	6	345	41	51	392	143	134	290	20	3	413

Table 18: TPL Detection Tools Sensitivity to Different D8/R8 Obfuscation Levels in TP/FP/FN

Detection Level	Obfuscation Level	LibScan			LibScout			Orlis			LibPecker			LibID-A		
		TP$_0$	FP$_0$	FN$_0$	TP$_0$	FP$_0$	FN$_0$	TP$_0$	FP$_0$	FN$_0$	TP$_0$	FP$_0$	FN$_0$	TP$_0$	FP$_0$	FN$_0$
Library-level	D8-non-obfs	159	44	3	157	35	5	81	59	81	158	43	4	133	29	29
	R8-shrink	94	10	68	44	69	118	74	43	88	92	30	70	57	24	105
	R8-shrink-orlis	93	10	69	21	22	141	75	44	87	82	29	80	38	27	124
	R8-shrink-opt	13	0	149	17	49	145	6	5	156	11	1	151	11	0	151
		TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
Version-level	D8-non-obfs	146	57	16	157	35	5	47	93	115	144	57	18	122	40	40
	R8-shrink	84	20	78	42	71	120	40	77	122	57	65	105	55	26	107
	R8-shrink-orlis	82	21	80	21	22	141	43	76	119	49	62	113	37	28	125
	R8-shrink-opt	10	3	152	13	53	149	3	8	159	11	1	151	11	0	151