Scalably Detecting Third-party Android Libraries with Two-stage Bloom Filtering

Jianjun Huang, Bo Xue, Jiasheng Jiang, Wei You, Bin Liang, Jingzheng Wu, and Yanjun Wu

Abstract—Third-party library (TPL) detection is important for Android app security analysis nowadays. Unfortunately, the existing techniques often suffer from poor scalability. In some situations, the detection time cost is even unacceptable. Although a few existing methods run relatively fast, they cannot provide enough effectiveness, especially for non-structure-preserving obfuscated apps, e.g., repackaged and flattened. In this paper, we treat TPLs detection as a set inclusion problem to effectively and efficiently analyze obfuscated apps, and develop a scalable two-stage detection approach, LIBLOOM. Specifically, the package and class signatures are encoded into two levels of Bloom filters respectively. At the first stage, the package filters are used to identify a limited number of candidate TPLs via set overlapping measurement to avoid unnecessary class-level set analysis. Subsequently, with the class filters, a similarity score is computed between the query app and each candidate to detect the integrated TPLs, and a novel entropy-based metric is presented to specially handle the repackaged and flattened apps. We have evaluated LIBLOOM on some large-scale benchmarks involving tens of thousands of TPL instances. The experiment results demonstrate that LIBLOOM outperforms state-of-the-art tools in both effectiveness and efficiency. Especially, the proposed two-stage method can run about ten times faster than the straightforward class-level analysis on flattened apps, and without loss of accuracy.

Index Terms—Third-party Library, Non-structure-preserving Obfuscation, Set Inclusion, Bloom Filter, Entropy.

1 INTRODUCTION

STUDIES have shown that most of Android apps contain third-party libraries (TPLs) [1] and the resulting security problems have attracted lots of concerns [2], [3]. Particularly, an app may passively introduce security flaws when it integrates a vulnerable TPL. Detecting TPLs in Android apps has become an important task [4], [5]. In practice, the analyst needs to identify the TPLs integrated in the app of interest and their specific versions to effectively check whether there is a reference to the vulnerable versions.

In the past few years, researchers have proposed various techniques to detect TPLs in Android apps and the signature-based similarity measurement has proven to be effective. Unfortunately, the existing techniques often suffer from poor detection efficiency. A recent study [6] has reported that some state-of-the-art approaches are not suitable for large-scale TPL detection. For example, given a data set of about 2,000 distinct library versions, LibID [7] requires nearly one day per app and LibPecker [8] takes more than five hours on average. Even worse, the time cost will be totally unacceptable when the analyst faces a batch of apps, e.g., emerging apps in an application market, and that the target TPL database often consists of tens of thousands of library instances, i.e., TPLs with different versions.

There exist some relatively fast TPL detection methods, but they cannot satisfy the demand of effectiveness. For example, Ma et al. [9] present LibRadar, a learning-based method, to check whether a given app potentially integrates TPLs with a pretrained model about the TPL signatures. LibRadar lacks the capability of telling the specific version of a used TPL or even sometimes cannot identify its name. In addition, the model needs to be frequently retrained to cover new TPLs. Backes et al. [10] propose LibScout, which relies on only structure-based hashes to measure the similarity between app and library classes. However, LibScout cannot effectively handle sophisticated code obfuscations, e.g., unused method removal, class repackaging and package flattening. We term such techniques the *non-structure-preserving obfuscation*, indicating they will modify the structures of classes or packages. It is worth noting that, modifying the structures is a common obfuscation technique that has been supported by popular obfuscators such as ProGuard [11], Allatori [12] and DashO [13]. As a result, LibScout cannot provide satisfiable detection performance for obfuscated apps. As shown in [7], the detection rate of LibScout never exceeds 15% when the non-structure-preserving obfuscations are enabled using ProGuard. A recent study by Zhan et al. [14] also points out that most existing tools are not resilient to those sophisticated code obfuscations.

An important question arises naturally as *how to accurately and scalably detect TPLs in Android apps*. In this paper, we abstract the TPL detection into a set inclusion problem to effectively handle varying obfuscations, and propose a two-stage retrieval technique to accelerate the problem solving.

Considering non-structure-preserving obfuscations have been applied popularly, the elements of a class in the code can be represented as a set of signatures. For a given app A, if there is a class lc in a TPL L whose signature set is the

- *J. Huang, B. Xue, J. Jiang, W. You and B. Liang are with the School of Information, Renmin University of China, Beijing 100872, China; and also with Key Laboratory of DEKE (Renmin University of China), MOE, China. (e-mail: {hjj, xuebo2020, jjscool, youwei, liangb}@ruc.edu.cn).*
- *J. Wu and Y. Wu are with the Institute of Software, Chinese Academy of Sciences, Beijing 100190, China. (e-mail: {jingzheng08, yanjun}@iscas.ac.cn).*
- *Bin Liang (liangb@ruc.edu.cn) is the corresponding author.*

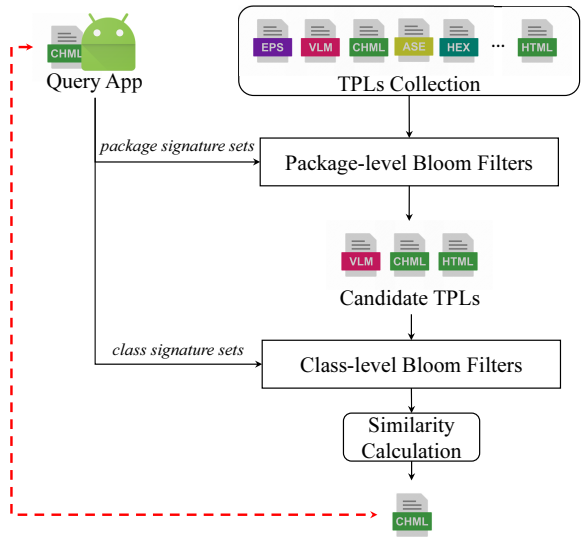


Fig. 1. Two-stage retrieval with Bloom filters.

subset of that of an A's class, we will label that A may refer to lc. And if most classes of L are labelled being referred to in A, we believe A integrates L. By employing such a robust statistical analysis, we can get an accurate detection for obfuscated apps.

Solving the set inclusion problem is still a time-consuming task when facing millions of classes. Thanks to the nature of set operations, the task can easily speed up via a high-level set overlapping measurement. In fact, the code base in Android apps and libraries is arranged in a hierarchical structure, i.e., app/library $\rightarrow$ packages $\rightarrow$ classes. If L are integrated in A, the related packages of L should be included in A or approximately included in A when being applied inter-class obfuscation, e.g., repackaging. Obviously, we can first filter the candidate integrated TPLs from the target collection via a package-level signature set analysis. Namely, we can identify a candidate TPL for an app A by checking whether the signature sets of its packages have a large overlapping similarity with their counterparts in A. Compared with a straightforward class-level analysis, the two-stage schema can guarantee the scalability. Our experiment shows that we can drop the number of candidate TPLs from tens of thousands to dozens.

To get further acceleration and save storage space, we leverage Bloom filter [15] to implement a TPL detection approach, LIBLOOM. As illustrated in Figure 1, the signature sets of packages and classes are encoded into two kinds of Bloom filters, each compactly representing a set using a bitmap. The signature set of a class consists of the fuzzy signatures of its methods, fields and hierarchy; and the signature set of a package combines all those of its inclusive classes. During detection, the package-level Bloom filters are first employed to measure the overlapping similarity between the packages of the query app and all target TPLs, and to identify the candidate TPLs. On this basis, strict subset queries occur on the class-level Bloom filters, and a similarity score is computed between the query app and a specific candidate TPL instance. For repackaged and flattened apps, we also present a novel entropy-based metric to identify the apps and get a reasonable similarity score for them purposefully. If the score is beyond a threshold, the TPL and its version are reported.

We evaluate LIBLOOM on three benchmarks, one of which contains more than ten thousand TPL instances. LIBLOOM has been demonstrated to outperform state-of-the-art tools regarding the effectiveness (1.6 times better against non-structure-preserving obfuscations and at least 20% better for commercial off-the-shelf (COTS) apps) and the efficiency (at least twice as fast). Furthermore, compared to the straightforward class-level analysis, LIBLOOM runs about ten times faster without loss of accuracy on flattened apps.

This paper makes the following contributions.

- We abstract the TPL detection to a set inclusion query problem, which leverages robust statistical analysis to handle sophisticated code obfuscations and can achieve high detection accuracy.
- We propose a two-stage retrieval technique and leverage Bloom filters to realize the approach named LIBLOOM, which can dramatically accelerate the TPL detection and guarantee the scalability. Besides, we present an entropy-based metric to identify potential repackaging and flattening.
- We implement a prototype of LIBLOOM and evaluate it on some large-scale benchmarks. The experiment results demonstrate that LIBLOOM outperforms state-of-the-art tools over effectiveness and efficiency.
- We build a special benchmark by applying three popular obfuscators and different non-structure-preserving obfuscations on open-source apps. It can be leveraged by other researchers to verify their approaches against those sophisticated obfuscations. It is available at https://github.com/iser-mobile/LIBLOOM.

2 BACKGROUND

Obfuscators and non-structure-preserving obfuscations. We have investigated three popular obfuscators, ProGuard [11], Allatori [12] and DashO [13], which are commonly used to obfuscate open-source Android apps in existing studies such as Orlis [17], LibID [7] and Zhan et al. [6], [18] and explicitly support the non-structure-preserving obfuscations of interest, i.e., code shrinking, repackaging and flattening. We take the TPL *glide-4.11.0* in the open-source app Wikipedia [16] as an example. Figure 2a shows part of the packages and classes in the original library. Identifier renaming (Figure 2b) is supported by all three obfuscators and we discuss below their support of the following sophisticated obfuscations.

Code shrinking. Obfuscators like ProGuard and DashO can remove unused methods and classes, even fields, by default during obfuscation. Note that, Allatori supports very weak code shrinking that can modify the class structure. Allatori users can only choose to remove the `toString` methods in classes to prevent potential information leak.

Repackaging. Renamed classes are moved into a single given package (i.e., *destination package*). In ProGuard and DashO, the default repackaging policy keeps the renamed classes whose package is not completely renamed but moves those with even partially renamed packages. Allatori support two kinds of repackaging and we denote them as

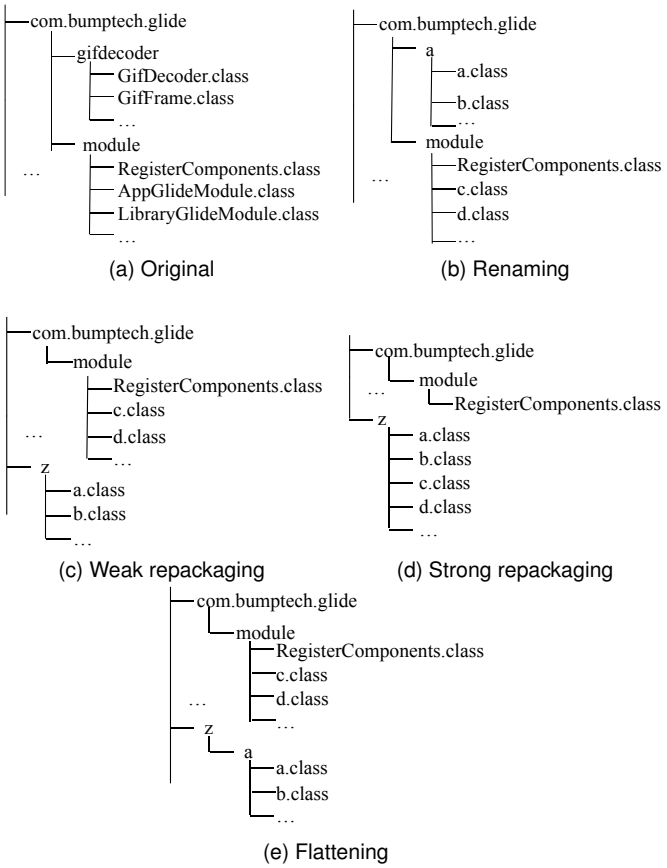


Fig. 2. Obfuscation examples. [*swapped with the next figure*]

weak and *strong*. Weak repackaging moves only the renamed classes whose package does not contain any unrenamed classes (Figure 2c) while strong repackaging moves any renamed classes no matter if their neighbors are unrenamed (Figure 2d).

Flattening. Behaving similar with repackaging, flattening moves obfuscated packages ("*a*" in Figure 2e) to a single given package ("*z*") and thus flattens the package hierarchy. Allatori does not support flattening and generally, repackaging and flattening are exclusive in other obfuscators.

Bloom filter. Bloom filter [19] was devised to test the set element membership with space efficiency. Using a bitmap of size M to compactly represent a set, a Bloom filter sets k random bits with *one* in the bitmap for each element. k is the number of independent hash functions. Each hash function generates an integer for an element and the integer indicates an index in the bitmap, which is set to one. One important property of Bloom filter based set element query is that there may be false positives but no false negative. Namely, if an element is in a set, the query to the corresponding Bloom filter must succeed.

Take Figure 3 as an example, which shows a set $\mathbb{S}$ (Figure 3b) and the hashes for each element (Figure 3c). Using $k = 3$ and $M = 256$ in the example, the corresponding Bloom filter $BF_{\mathbb{S}}$ is shown in Figure 3d.

Nowadays, Bloom filter is also widely used for subset queries [20]. Formally, given two Bloom filters, if $\forall i, BF_1[i] \leq BF_2[i]$, we claim that the set for BF_i holds a subset of that for BF_2. Goel et al. [20] have demonstrated the effectiveness and efficiency of subset queries using Bloom filter. Take Figure 3 as an example again. We remove two elements (boxed in Figure 3b) from $\mathbb{S}$ to get $\mathbb{S}'$ and make a Bloom filter $BF_{\mathbb{S}'}$ that keeps the same as the previous one but zeros the red bits. Those bits are corresponding to the removed elements. Obviously, the relation of $BF_{\mathbb{S}'}$ and $BF_{\mathbb{S}}$ satisfies the condition and $\mathbb{S}' \subseteq \mathbb{S}$.

TABLE 1
The detection result of the vulnerable *retrofit-2.4.0* in Apod MD with the state-of-the-art tools and the time cost (in seconds) in a large-scale TPL data set.

Obfuscation	LibScout	Orlis	LibPecker	LibID
Code shrinking	×(82s)	×(168s)	√(19054s)	√(1963s)
Repackaging	×(79s)	×(162s)	×(20013s)	√(2136s)
Flattening	×(83s)	×(172s)	×(19535s)	√(3125s)

3 Motivating Example

An important downstream task of the TPL detection is to identify vulnerable TPLs to reveal the potential risk of an app. However, besides protecting the code security, the non-structure-preserving obfuscation can hide the flawed TPLs, which hinders the state-of-the-art detectors from detecting TPLs. For instance, we seek an open source app Apod MD [21] from F-Droid [22], which integrates a vulnerable *retrofit-2.4.0* (CVE-2018-1000850 [23]). We build the app and obfuscate it with different techniques, i.e., code shrinking, repackaging and flattening respectively. The detection is carried out on a data set with 10k+ TPL instances, to verify if the four state-of-the-art detectors can effectively and efficiently report *retrofit-2.4.0* under different obfuscations with ProGuard.

Table 1 shows the result. LibScout [10], Orlis [17] and LibPecker [8] fail to detect the vulnerable TPL from the repackaged and flattened apps. Though LibID [7] can detect the TPL in all three scenarios, its low efficiency prevents the scalability, which is also illustrated in [6]. In a large-scale scenario of TPL detection, we believe that effectiveness and scalability are equally important.

4 Approach

We abstract the TPL detection as a set inclusion problem and propose a two-stage retrieval method to detect TPLs in Android apps effectively and efficiently. Our approach, LIBLOOM, extracts the signature sets from packages and classes and encodes them into two types of Bloom filters. As illustrated in Figure 1, package-level Bloom filters are leveraged to compute the package overlapping similarity to identify candidate TPLs. For candidate packages, LIBLOOM further performs subset queries over the class-level Bloom filters and computes a similarity score between the query app and a candidate TPL to indicate the likelihood that the TPL instance is integrated in the app. Note that, code shrinking will be handled by the abstraction of set inclusion, but two other obfuscations, i.e., class repackaging and package flattening, pose a challenge to accurately label the library packages and classes refer to their counterparts. To address the challenge, we employ entropy-based heuristics to identify potential repackaged and flattened apps, and

```
extends AtomicReference
implements CompletableObserver
        Disposable
long serialVersionID
EmptyCompletableObserver()
void dispose()
boolean isDisposed()
boolean hasCustomOnError()
void onComplete()
void onError(Throwable)
void onSubscribe(Disposable)
```

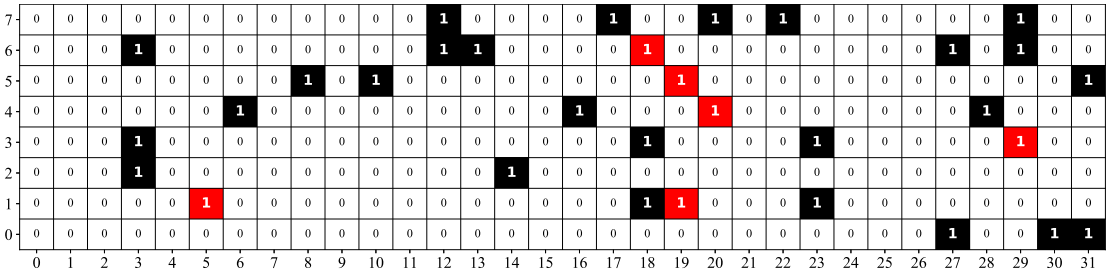
(a) Original class structure.

```
extends:Ljava/util/concurrent/atomic/AtomicReference;
implements:X1;
implements:X2;
J1
<init>()V;1
(X;)V;1
(X;)Z;1
(X;)Z;2
(X;)V;2
(X;Ljava/lang/Throwable;)V;1
(X;X;)V;1
```

(b) Class signature set with removed signatures in red boxes.

```
246, 31, 204
134, 156, 78
27, 244, 205
125, 37, 51
55, 236, 241
191, 114, 219
99, 144, 67
148, 179, 210
221, 170, 119
195, 253, 67
30, 168, 50
```

(c) Hashes.



(d) Bloom filter with $k = 3$ and $M = 256$. Bits in red boxes are reset to zero for the shrinked class, corresponding to the removed field and method.

Fig. 3. An example showing the structure, signature set and the corresponding Bloom filter of `EmptyCompletableObserver` in *rxjava-3.0.4* integrated in app Wikipedia [16].

compute the similarity for them in a suitable granularity respectively.

In the below sections, we will discuss the details of LIBLOOM.

4.1 Structure-based Signature Set

In this paper, we aim to extract the signatures of classes and inspect if a library class can refer to an app class based on their signatures. When enough library classes can be mapped to app classes, we can claim that the library instance is integrated to the app. Such signature-based TPL detection has proven to be effective [10]. To preserve the efficiency and effectiveness of the detection, we build a signature set for each class and perform set inclusion analysis between classes to measure their relation. Specifically, a signature set consists of the following three kinds of structure-based signatures.

- *Methods.* Methods substantially illustrate the function of a class and should be naturally considered. To avoid the side effect of identifier renaming, we use the fuzzy method descriptors, i.e., keeping any framework type name as is and replacing non-framework types with *X*. Meanwhile, the primitive data types are replaced with one-character identifiers by the underlying analysis framework WALA [24], i.e., *void*, *boolean*, *int*, *short*, *long*, *float* and *double* are denoted by ***V***, ***Z***, ***I***, ***S***, ***J***, ***F*** and ***D***, respectively. Different from existing approaches like LibScout [10], we keep the method names for those methods that will never be renamed, e.g., the constructor method `<init>`. Note that, we also append a sequence number to each fuzzy descriptor, to differentiate the methods with the same fuzzy descriptors. Similar trick is used for the following two types of signatures.
- *Fields.* Fields denote the data processed by a class and can be a significant indicator to differentiate classes. We reserve the field types and eliminate variable names to generate the field signatures. We include the numbered fuzzy types of the fields in the related signature sets.
- *Hierarchy.* Inheriting a base class or implementing a specific interface points out the hierarchical position of a class in a bunch of function-dependent classes. We prepend *extends* or *implements* to the fuzzy type of the base class or interface, respectively, to make up the hierarchy signatures.

The signatures in a class form its signature set and all signature sets of the inclusive classes in a package constitute the package's signature set. Figure 3b shows an example of a signature set associated with the class structure in Figure 3a.

Class signature set is the basic analysis unit in our work. The obfuscation technique with the most significant impact on the class signature set is code shrinking, which can remove methods or fields from classes. In other words, a shrunk TPL class owns a subset of signatures compared with the original one. When we represent each class as a signature set, it is naturally to leverage set inclusion to deal with code shrinking. Take as an example a class that comes from *rxjava-3.0.4* in Wikipedia [16] with ProGuard as the obfuscator. The original class structure is shown in Figure 3a and the signature set is in Figure 3b. When the library is compiled into the app, one field and one method is removed (corresponding to the boxed line). As a consequence, the shrunk signature set with nine signatures is a strict subset of the original signature set (11 signatures). By this means, the affected classes by code shrinking can be effectively matched to their origin. In this study, we utilize Bloom filter to efficiently solve the set inclusion problem and perform TPL detection.

4.2 Bloom Filter Construction

We build two kinds of Bloom filters. A class-level Bloom filter BF_c of size M_c is constructed for a class c by applying

k independent hash functions to every signature element in c's signature set. Each hash operation generates an integer as a bit position in BF_c. The position is set to *one* in the Bloom filter. For each package, we similarly build a Bloom filter BF_p of size M_p.

We use *murmur3_128* in Guava [25] to simulate the k independent hash functions [26]. The optimal number of hash functions and the size of Bloom filter are computed by Eq. 1 and Eq. 2 [15].

$$k = -\frac{\ln fpp}{\ln 2} \tag{1}$$

$$M = \frac{k \cdot n}{\ln 2} \tag{2}$$

in which, the false positive probability fpp indicates how likely an element not belonging to a set is reported to be in the set and n is the maximum degree of the signature set. Figure 3d presents the class-level Bloom filter corresponding to the signature set in Figure 3b, with $k = 3$ and $M_c = 256$. We will show in Section 5.2 how to empirically determine the proper k, M_c and M_p for LIBLOOM.

4.3 Package-level Overlapping Measurement

The first stage of LIBLOOM focuses on screening candidate TPL packages via package-level set overlapping measurement, to accelerate the TPL detection especially in large-scale scenarios. Given two Bloom filters, BF_{lp} and BF_{ap}, respectively for a library package lp and an app package ap, the overlapping is measured as Eq. 3.

$$overlap\_ratio(lp, ap) = \frac{BF_{lp} \cdot BF_{ap}}{\min\left(|BF_{lp}|, |BF_{ap}|\right)} \tag{3}$$

where "$\cdot$" means the dot production of two Bloom filters and "$|BF|$" counts the number of *one*s in the Bloom filter. To cover the obfuscated packages in which a large number of classes have been removed from their library counterparts, we compute the overlapping ratio of the overlapped elements to the smaller signature set.

When the ratio reaches a threshold, we label $<lp, ap>$ as a candidate package pair for further class-level analysis. Otherwise, the pair is considered unmatched and thus discarded. If the candidate packages of a TPL instance cannot potentially indicate the use of that library in the query app, the TPL instance is thrown away. In practice, an app can integrate very few libraries and most TPLs are unrelated and can be excluded by the ratio. By this means, we can effectively reduce the number of packages for the subsequent analysis and the number of TPLs in large-scale detection, and boost the efficiency.

Obviously, the threshold can affect the efficiency and detection rate. A small threshold can hardly eliminate unmatched package pairs or TPL instances. In other words, most packages are left to class-level analysis and nearly all TPLs are kept as candidates, making little contribution to detection efficiency. A too large threshold can emit more accurate package pairs and candidate TPLs to dramatically accelerate the detection. However, many packages may be missed, e.g., the ones with partially repackaged classes, resulting in a low detection rate. To trade off the efficiency and detection rate, we conduct a pilot study to determine a proper overlapping threshold, 0.8 as shown in Section 5.2.3.

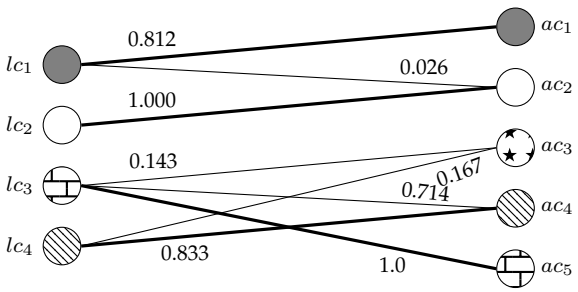


Fig. 4. Partial pairwise class inclusion scores for some classes in the library *gson* used in the app Wikipedia [16]. Circles with the same patten are the library class (left) and corresponding app class (right).

4.4 Class-level Subset Query

Strict subset queries are applied to the class-level Bloom filters. Pairwise query takes place between any library class lc and app class ac for each candidate package pair $<lp, ap>$. Given two Bloom filters BF_{lc} and BF_{ac}, we say ac holds a subset of lc's signature set if Eq. 4 satisfies.

$$BF_{ac} \,\&\, BF_{lc} = BF_{ac} \tag{4}$$

Note that, in the subset query scenario, even if we have done the best to distinguish uncorrelated classes with multifactorial structure-based signatures, it is still possible for multiple app classes matching one library class and one app class containing the subset of more than one library class. Figure 4 presents a pairwise matching result among four library classes and five obfuscated classes. A link means the subset query succeeds between the obfuscated app class and the linked library class.

Obviously, the query result is not suitable to directly determine the number of potentially matched classes for app-library similarity computation. We aim to further obtain a confident one-to-one mapping and design a class-level inclusion score for the candidate class pair $<lc, ac>$ as Eq. 5.

$$class\_inclusion\_score(lc, ac) = \frac{|BF_{ac}|}{|BF_{lc}|} \tag{5}$$

Consequently, if an obfuscated class has nothing (or very little) removed from its original version, the inclusion score is definitely (or approaches) *one*. A very small score typically means heavy code shrinking during obfuscation or a small app class is indeed unrelated to the mapped library class. In Figure 4, for example, lc_1 has a score of only 0.026 with ac_2, while the score reaches 0.812 between lc_1 and its counterpart ac_1.

Finally, according to the score, we perform a maximum matching on the classes to generate a confident mapping between classes in lp and ap. The Kuhn-Munkres algorithm [27] is employed, which can effectively exclude the unmatched query results and produce a satisfied mapping, as the bold lines in Figure 4 shows.

4.5 Similarity Computation & TPL Detection

On the basis of the class-level set analysis, we compute a similarity score between the query app and a TPL instance. A TPL instance can be claimed to be integrated into a query app if their similarity score reaches a given threshold. Furthermore, we employ entropy-based heuristics to compute the similarity score specially for potential repackaging and flattening.

4.5.1 Similarity computation

In practice, the Android code in both apps and libraries is organized in packages. Naturally, based on the class-level inclusion analysis result and taking the package as the measurement unit, we compute a similarity score between the query app and a TPL instance.

In general, package names are often renamed and we have the following observations. First, the package hierarchy levels are often not changed. In other words, a three-level library package *a.b.c* is more likely to be renamed to *o.p.q* instead of a two-level package *x.y*. Second, the sub-packages keep the hierarchy relation with their parent packages in obfuscated apps. In the above example, *a.b.c.d* can refer to *o.p.q.r*, instead of *o.p.m.r*.

Based on the observations, as done in [10], we generate an optimal association $\mathcal{A}$ among packages in the query app and TPL instance. We then count the number of max-matched classes in these packages and compute a ratio of it to the number of classes in the TPL instance, as in Eq. 6. The ratio is treated as the similarity score between the query app and the library instance.

$$sim\_score(lib, app) = \frac{\sum\limits_{(lp,ap)\in\mathcal{A}} |\text{max\_match}(lp, ap)|}{\#classes\ in\ lib} \quad (6)$$

where max_match outputs a set of mapped classes between lp and ap by the algorithm.

4.5.2 Similarity for repackaging and flattening

Though the above hierarchy-based approach is effectively for a large number of obfuscated apps, some TPLs may be missed to be reported, i.e., the similarity score is lower than 0.6. Such cases often involve class repackaging or package flattening, which can dramatically modify the package hierarchy and make obfuscated app packages and their original versions owning different hierarchy levels or unmatched parent/sub-package relations, as shown in Figure 2.

A straightforward and effective solution is to identify the destination package of repackaging or flattening and design special similarity computation methods accordingly. We notice that, repackaging and flattening moves classes or packages from different sources to a single parent package, and inevitably mixes diverse data operation logics together. Naturally, the uncertainty of the referred types associated with the destination package obviously increases. In contrast, the original packages, organized with correlated classes in general, usually posses more cohesive type references and relatively low uncertainty. To this end, we can measure the uncertainty of referred types in an app package and employ *entropy* [28] to distinguish the two kinds of packages from normal packages.

Our observation has also shown that the package with the largest entropy is most probably the destination package of repackaging or flattening. To further differentiate the two exclusive obfuscations, we propose two kinds of entropy computation based on their organization characteristics.

Repackaging-oriented entropy. Repackaging produces a large single package with many direct classes that come from different packages. To differentiate the packages from other normal packages, we need to compute the entropy for each package based on the direct classes. Given a package and its inclusive classes $P^r = \{c_1, c_2, \ldots, c_n\}$, we collect the app-defined types referred to in methods, fields and class hierarchy to form a type reference sequence of a class c_i, i.e., $c_i = \{t_1^i, \ldots, t_x^i\}$. All such sequences are united to form a set of types, say $RS^r = \{t_1, \ldots, t_y\}$. The entropy is computed as Eq. 7.

$$H_r = \sum_{t\in RS^r} p(t)\log\frac{1}{p(t)},\ \text{where}\ p(t) = \frac{\sum\limits_{c_i}\text{count}(t\in c_i)}{\sum\limits_{c_i}|c_i|} \quad (7)$$

Flattening-oriented entropy. Different from repackaging, the destination package of flattening involves many sub-packages that originally reside in different parent packages. The sub-packages, after being flattened, will not contain any further sub-packages. We formally denote a package and its satisfied sub-packages as $P^f = \{sp_1, sp_2, \ldots, sp_n\}$ and $\forall i, sp_i = \{c_{i1}, c_{i2}, \ldots, c_{im}\}$. Besides, $c_{ij} = \{t_1^{ij}, \ldots, t_z^{ij}\}$. The merged type set is represented by RS^f and we compute the entropy via Eq. 8.

$$H_f = \sum_{t\in RS^f} p(t)\log\frac{1}{p(t)},\ \text{where}\ p(t) = \frac{\sum\limits_{sp_i}\sum\limits_{c_{ij}}\text{count}(t\in c_{ij})}{\sum\limits_{sp_i}\sum\limits_{c_{ij}}|c_{ij}|} \quad (8)$$

By computing and ranking the entropy for the app packages, we obtain two kinds of largest entropy and the corresponding packages, denoted by $(\mathbb{H}_r, ap_r)$ and $(\mathbb{H}_f, ap_f)$, respectively. Empirically, we can effectively identify the destination package of repackaged apps ($\mathbb{H}_r > \mathbb{H}_f$) or flattened apps ($\mathbb{H}_r < \mathbb{H}_f$).

In addition, to reduce the influence of popular libraries, we exclude the supporting packages with prefixes "*android*", "*androidx*", "*kotlin*" and "*kotlinx*" during the entropy computation. We also exclude the packages with extremely long names, which are usually not generated by the two kinds of obfuscations.

Below we describe how to compute the similarity score for the identified repackaged or flattened apps. We break the association for every $<lp, ap>$ in $\mathcal{A}$ unless lp and ap have the same unobfuscated package name. Package level similarity scores are computed for each $lp \in lib$ and ap_r (for repackaged apps) or between lp and the satisfied candidate sub-packages in ap_f (for flattened apps). Therefore, we can build an association $\mathcal{R}$ between library packages and ap_r or $\mathcal{F}$ for potentially flattened packages. To simplify the detection, we assume that one library package can at most

TABLE 2
The data sets used in the evaluation.

	Open-source	Closed-source [30]	Large-scale
#Apps	100 × 7	221	2,552
#Distinct TPLs	349	59	515
#TPL Instances	551	2,144	11,648

refer to one app package with the maximum similarity. As a result, the similarity score for repackaged or flattened apps is computed by Eq. 9, in which $\mathcal{C}$ can be either $\mathcal{R}$ or $\mathcal{F}$, respectively, for one kind of identified obfuscation.

$$sim\_score(lib, app, \mathcal{C}) = \frac{\sum \max\left(|\max\_match(lp, ap)|, \ |\max\_match(lp, ap)|\right)}{(lp,ap)\in\mathcal{C} \qquad lp \overset{name}{=} ap} \Big/ \#classes\ in\ lib \tag{9}$$

4.5.3 TPL detection

Given a query app and a TPL instance, LIBLOOM first computes a package hierarchy-based app/library similarity score using Eq. 6. If the score reaches a predefined threshold, the TPL instance is reported to be used in the app. We choose 0.6 as the threshold, the same as in [10], which means that only when at least 60% of the TPL classes can refer to app classes, can we determine the use of TPL instance in the app.

If the similarity is lower than 0.6, we apply entropy-based heuristics to identify potential repackaging or flattening. The corresponding similarity is computed via Eq. 9 and the TPL instance will be reported if the similarity satisfies the threshold requirement.

5 EVALUATION

In this section, we will evaluate LIBLOOM and compare it with state-of-the-art detection tools over the following two aspects.

- **Effectiveness.** We evaluate LIBLOOM and other tools on two data sets to assess the detection effectiveness of LIBLOOM. Besides, we also measure the effectiveness of the proposed entropy-based metric.
- **Efficiency/Scalability.** We compare the detection time of LIBLOOM with the other tools and particularly run it on a large-scale data set to show the scalability of our two-stage retrieval method.

We perform a comparative analysis with four state-of-the-art TPL detectors, LibID [29], LibPecker [8], Orlis [17] and LibScout [10], which can report the specific versions of integrated TPLs. The similarity threshold for the tools, if applicable, is fixed to 0.6 as done in LibScout [10]. All experiments are carried out on a Ubuntu 20.04 machine with Intel Xeon Scalable Silver 4110 CPU and 208GB memory.

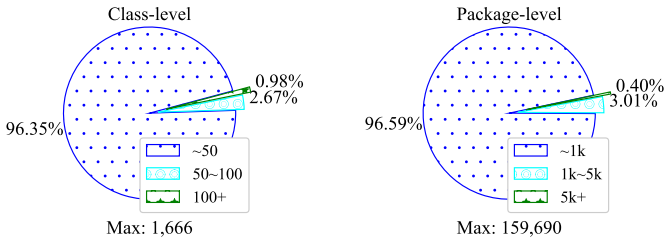


Fig. 5. Statistics of the signature sets for the 70 apps.

5.1 Benchmarks

We have collected three benchmarks for evaluation. Table 2 shows their statistics.

First, measuring the effectiveness requires the ground truth. To this end, we build a special benchmark mainly for detectors to evaluate their detection effectiveness against non-structure-preserving obfuscations. One hundred *open-source* apps are downloaded from the F-Droid repository [22]. For each app, we build seven APKs with its default obfuscation policy by ProGuard (code shrinking is enabled) and six repackaging/flattening policies (plus code shrinking if applicable) using three obfuscators. According to the apps' build configurations, we collect their dependent TPL instances from repositories like Maven [31]. The total number of distinct TPLs is 349 and the instances accumulate to 551. In addition, we reuse the benchmark constructed by Zhan et al. [6]. It includes 221 *closed-source* Android apps and only 59 TPLs with totally 2144 versions.

Second, though we have already used an existing closed-source benchmark, we want to particularly evaluate the efficiency and scalability with much more apps and TPLs. To this end, we randomly collect 2,552 apps from the Xiaomi App Store [32] in China and download 11,648 instances of 515 popular TPLs from Maven to form a *large-scale* benchmark. To the best of our knowledge, it has the largest available TPL data set among related studies.

5.2 Parameter Tuning

Parameter tuning is inevitable to trade off between accuracy and efficiency or between false positives and false negatives [10], [8], [7]. To avoid the unacceptable performance resulting from intuitive parameter settings, we perform a comprehensive study to tune the parameters.

Specifically, other than the parameters that are directly employed from existing tools, e.g., 0.6 from [10], in our work, there are four particular parameters, i.e., the number of hash functions (k) to build Bloom filters, the sizes of class-level Bloom filters (M_c) and package-level filters (M_p), and the overlapping threshold. To reduce the impact of package overlapping measurement, we implement a tool named Libloom-p to tune the first two parameters. Libloom-p implements the straightforward class-level set analysis, without considering the package filtering. After k and M_c are determined, we use LIBLOOM to tune the other two parameters.

In this section, we randomly select ten apps (each with seven variants in total) from the open-source benchmark and measure the effect of different parameter values. We show the statistics of the signature sets in Figure 5. Among

TABLE 3
Preliminary experiment with different false positive probabilities on the 70 apps, with $n = 1,666$.

	fpp = 0.1	fpp = 0.01	fpp = 0.001
k	3	7	10
Precision/Recall/F1	69.40% / 66.36% / 66.24%		
Avg. Time (seconds)	103.4	205.7	258.7

the 70 apps, the class-level signature set contains at most 1,666 signatures and the maximum number of the package-level signatures in a set reaches 159,690. Despite the extremely large ones, 96.35% of the class-level signature sets have no more than 50 signatures and 96.59% of the package-level sets own at most 1,000 signatures.

5.2.1 Number of hash functions

According to Eq. 1, the number of hash functions, k, depends on the false positive probability fpp. Empirically, we select three kinds of sufficiently low probability, as shown in Table 3 and choose the maximum number of the class-level signatures, i.e., $n = 1,666$, to build class-level Bloom filters. We apply Libloom-p to the 70 apps. The result is presented in Table 3. All parameter settings produce the same detection result but a smaller k means smaller time overhead. The reason of the same detection result for different *fpp* is that *fpp* indicates how likely an element is falsely identified in a set via a Bloom filter-based query. In the context of the subset query, the probability that a set is falsely identified to be a subset of another set is much lower than the proposed *fpp*s. In addition, a TPL is reported when a large portion of the library classes are matched, which further reduces the impact of a few class mismatches. Therefore, *fpp* shows little impact in precision/recall/F1. We choose $k = 3$ for all subsequent experiments.

5.2.2 Class filter size

With $k = 3$, we tune the size of class-level Bloom filters. Note that, though we can use the maximum number for M_c, the huge size of Bloom filters can increase the time cost of set inclusion analysis, when most signature sets are actually small, according to Figure 5. We aim to find an appropriate size of Bloom filters, which achieves the best efficiency but does not affect the effectiveness.

We use Libloom-p again to determine M_c. Based on the breakdown of the class-level signature sets, we conduct experiments for three different n, i.e., 50, 100 and 1,666. The result is shown in Figure 6. When $n = 50$, the average time cost is reduced by 4.8% $\sim$ 71.5% with a negligible drop of F1 (0.2%). We consider it reasonable to trade the accuracy loss off with the apparent acceleration. Consequently, $n = 50$ and $M_c = 256$, by rounding up to a multiple of eight, are used afterwards.

5.2.3 Package filter size and overlapping threshold

Fixing $k = 3$ and $M_c = 256$, we combine different sizes of package signature sets and the overlapping thresholds and apply LIBLOOM to the 70 apps. According to Figure 5, we choose three n, i.e., 1k, 5k and 159,690. In addition, the overlapping threshold ranges from 0.5 to 1.0.

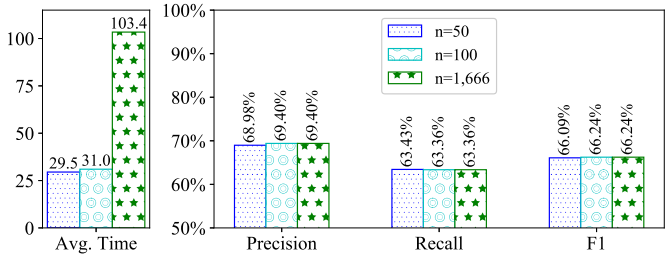


Fig. 6. Experiment result for the 70 apps when $k = 3$ using Libloom-p.

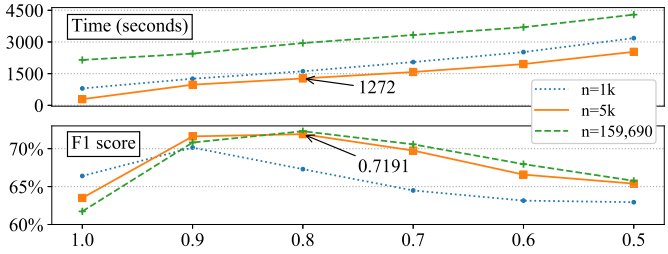


Fig. 7. Preliminary results for the 70 apps with different combinations of n and overlapping threshold (x-axis).

The results of time cost and detection performance (F1 scores) are described in Figure 7. From the upper figure, we are surprised to see that n = 5k, other than 1k, makes the fastest detection speed under each threshold setting. In the lower figure, the F1 score on the solid line (n = 5k) reaches the peak when the threshold is 0.8. For all subsequent experiments, we use M_p = 21,640 (corresponding to n = 5k) and the overlapping threshold as 0.8. Other combinations can also be selected, relying on specific purposes, e.g., (5k, 1.0) for extremely high efficiency (290 seconds) but lower effectiveness.

5.3 Effectiveness

We use two benchmarks to evaluate the effectiveness of TPL detection. To demonstrate the effectiveness of the entropy-based metric for identifying the destination packages of repackaged and flattened apps, we also carry out a further study.

5.3.1 Detecting open-source benchmark

To assess the detection ability against non-structure-preserving obfuscations, we evaluate LIBLOOM, Libloom-p and four state-of-the-art detectors on the specially built open-source benchmark. Table 4 shows the results. There are some notable findings in Table 4. Libloom-p generally contributes slightly higher recall than LIBLOOM but lower precision because some packages are kept as candidates without the overlapping measurement. LibID gets a higher recall for ProGuard (rpkg[1]) data set but the precision is much lower than others. Orlis achieves the best precision (100%) over all tools for DashO (flt) but it recognizes the fewest TPLs (0.25%) in the data set. In addition, Allatori with strong

1. We use *rpkg* for *repackaging* and *flt* for *flattening*.

TABLE 4
Detection result on the open-source benchmark and comparison with other tools.

		LIBLOOM	Libloom-p	LibID	LibPecker	Orlis	LibScout
PGD	P	97.30%	75.63%	35.20%	57.41%	61.75%	67.29%
	R	49.02%	49.75%	45.65%	28.19%	11.91%	15.06%
	F1	65.19%	60.02%	39.75%	37.81%	19.97%	24.61%
PGR	P	91.59%	59.81%	14.83%	45.96%	60.50%	89.08%
	R	36.29%	39.40%	39.39%	16.34%	10.41%	7.98%
	F1	51.98%	47.51%	21.55%	24.11%	17.76%	14.65%
PGF	P	88.42%	54.35%	29.33%	48.83%	60.53%	85.16%
	R	45.73%	45.96%	34.37%	18.03%	10.99%	9.12%
	F1	60.28%	49.80%	31.65%	26.34%	18.60%	16.48%
ALW	P	91.04%	79.50%	16.67%	49.79%	72.04%	48.79%
	R	89.75%	90.67%	83.39%	42.94%	13.22%	10.55%
	F1	90.39%	84.72%	27.79%	46.11%	22.34%	17.35%
ALS	P	73.06%	59.20%	11.33%	0.00%	-	0.00%
	R	90.73%	91.31%	27.57%	0.00%	0.00%	0.00%
	F1	80.94%	71.83%	16.06%	-	-	-
DOR	P	87.62%	58.00%	12.96%	42.09%	-	26.67%
	R	33.64%	35.86%	0.45%	11.96%	0.00%	0.52%
	F1	48.62%	44.32%	0.87%	18.63%	-	1.02%
DOF	P	90.37%	49.79%	14.71%	45.21%	100.0%	22.22%
	R	46.02%	46.35%	0.47%	12.46%	0.25%	0.65%
	F1	60.98%	48.01%	0.91%	19.54%	0.50%	1.26%
Avg. F1		65.48%	58.03%	19.80%	24.65%	11.31%	10.70%

PGD: ProGuard (default); PGR: ProGuard (rpkg); PGF: ProGuard (flt); ALW: Allatori (weak); ALS: Allatori (strong); DOR: DashO (rpkg); DOF: DashO (flt). P: precision; R: recall; "-": non-applicable, e.g., no report for <Orlis, DashO (rpkg)>.

repackaging and DashO with repackaging/flattening make the obfuscated apps difficult to analyze. Some tools even cannot emit any useful result in certain cases.

LIBLOOM, on the other hand, outperforms all other tools on the TPL detection against non-structure-preserving obfuscations. It obtains the highest F1 scores for all scenarios, at least 1.6 times higher than state-of-the-art tools. For the apps obfuscated by Allatori, LIBLOOM achieves remarkable results because Allatori does not heavily shrink the code and our entropy-based heuristics can effectively identify the destination packages (see Section 5.3.3 for more details), making class-level analysis more accurate. Facing heavy code shrinking by ProGuard and DashO, LIBLOOM produces satisfied results as well.

To further verify if the same thresholds can work on the other open-source benchmarks, we conduct a detection on the one released with Orlis [17], which contains 162×3 obfuscated apps by using ProGuard, Allatori and DashO. LIBLOOM achieves F1 scores of 90.31%, 89.46% and 82.75%, respectively for the three different obfuscators. As a comparison, the corresponding highest F1 scores achieved by the other tools are 79.53% (LibScout), 76.13% (LibPecker) and 41.40% (LibScout). This experiment has demonstrated the chosen thresholds also work well on other open-source benchmarks.

5.3.2 Detecting closed-source benchmark

LIBLOOM is evaluated on the closed-source benchmark and we directly reuse the data in [6] for the other tools. The detection result is presented in Figure 8. Though LIBLOOM has slightly lower precision than LibPecker and LibScout, it gains the highest detection rate and F1 score. In fact, none

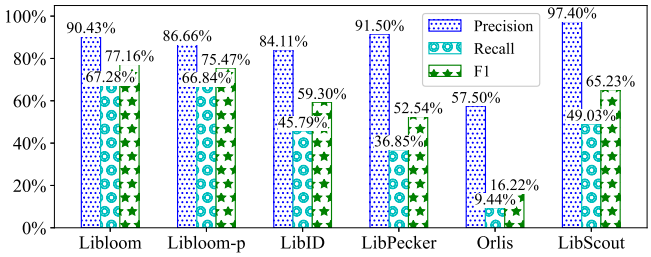


Fig. 8. Detection result on the closed-source benchmark.

TABLE 5
Result for 30 repackaged/flattened apps in the closed-source benchmark.

	LIBLOOM	LibPecker	LibScout
Precision	87.69%	90.38%	100.0%
Recall	34.20%	19.09%	23.48%
F1	49.21%	31.52%	38.03%

of the other tools can detect more than half of the TPLs used by the apps, but LIBLOOM achieves a recall of 67.28%, contributing 37.2% more TPLs than LibScout and seven times of Orlis. The evaluation also demonstrates LIBLOOM does not lose accuracy for detecting arbitrary COTS apps compared to Libloom-p.

In practice, most real-world Android apps with sophisticated obfuscations are closed-source and the adopted obfuscation techniques are unknown. To further demonstrate LIBLOOM's ability, we manually identify 30 such apps from the closed-source benchmark. They are obfuscated with repackaging or flattening in addition to code shrinking which is generally a default option in popular obfuscators. We compare LIBLOOM with the other four tools but LibID and Orlis fail to run on the apps. From the result in Table 5, we can see that even for the real-world obfuscated apps, LIBLOOM outperforms LibPecker and LibScout by detecting more TPLs and achieves the highest F1 score.

5.3.3 Identifying repackaging/flattening

In this section, we measure the effectiveness of the entropy-based metric for distinguishing the obfuscation types and destination packages for repackaged or flattened apps. The numbers of identified packages in the six repackaging/flattening-enabled data sets are used in the measurement. Since each kind of obfuscation has exact

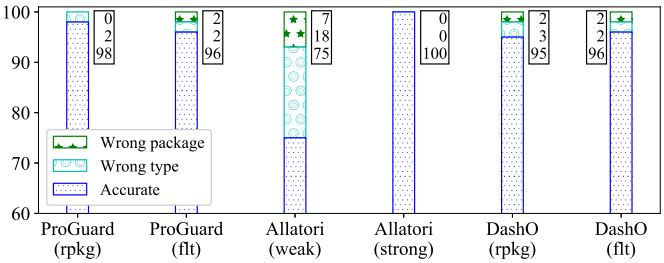


Fig. 9. The result of identified destination packages of repackaging/flattening.

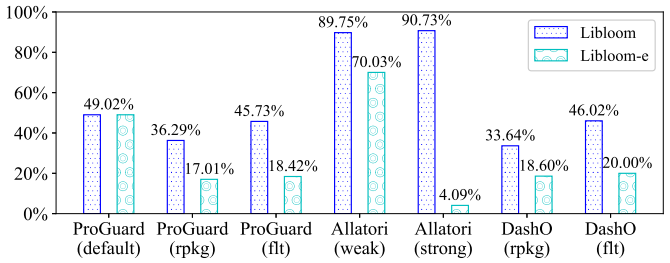


Fig. 10. Comparison of detection rate between LIBLOOM and Libloom-e.

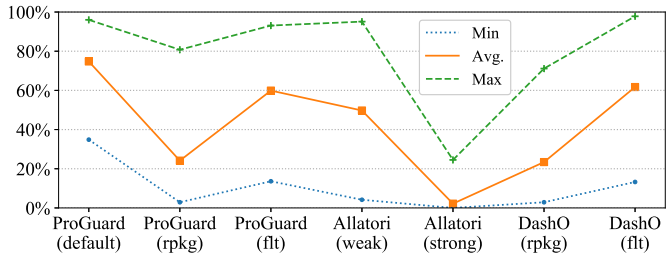


Fig. 11. The effect of package filtering on the seven data sets in open-source benchmark.

TABLE 6
Total detection time (in seconds if not specified) on the open-source benchmark.

	LIBLOOM	Libloom-p	LibID	LibPecker	Orlis	LibScout
PGD	99	676	217h	66h	42h	611
PGR	637	1591	24h	67h	16h	612
PGF	140	1508	233h	66h	25h	610
ALW	405	1080	185h	70h	32.7h	539
ALS	4124	5430	75h	69h (Recall:0%)	290 (Recall:0%)	548 (Recall:0%)
DOR	690	1745	2037	41h	1717 (Recall:0%)	837
DOF	146	1644	132h	10h	1684	988

100 apps, we can easily determine the effectiveness of the entropy-based metric via Figure 9. We split the identification results into three categories, i.e., *accurate*, *wrong-type* and *wrong-package*. The second means to report a repackaged app as flattened, or vice versa. The last indicates a correct obfuscation type report but the destination package is incorrectly recognized. The false identifications are caused by sophisticated packages that originally contain lots of classes or sub-packages. The resultant large entropy can mislead our tool. However, LIBLOOM accurately identifies 93.3% of the destination packages. Even though only 75 are accurately identified on Allatori(weak), there is very limited influence on the detection effectiveness because the weak repackaging breaks the hierarchy of only a small quantity of classes. As a result, many TPLs are detected, as shown in Table 4.

To further show the advantage of identifying repackaging/flattening correctly, we implement Libloom-e, a tool without handling repackaging/flattening at all, and compare it with LIBLOOM. Figure 10 depicts the comparison. Except for ProGuard (default), LIBLOOM reports 28% $\sim$ 2,100% more TPLs, illustrating the effectiveness of our method.

5.4 Efficiency/Scalability

In this section, we measure the efficiency and scalability of LIBLOOM. We will compare LIBLOOM with state-of-the-art tools and Libloom-p. Especially, we will show the dramatic acceleration resulting from the package filtering.

5.4.1 Efficiency on heavily obfuscated apps

The time cost for the open-source benchmark is shown in Table 6. In general, LIBLOOM and LibScout show better performance than the other tools. Especially, LIBLOOM has an extremely fast detection speed for ProGuard (default), and two flattening-enabled data sets ($<$ 150 seconds). We can see that, repackaging often has a noticeable impact on LIBLOOM, but it still achieves comparable (and sometimes the best) performance in three scenarios. LIBLOOM takes about 1.15 hours to analyze the Allatori (strong) data set while Orlis and LibScout require less than ten minutes. However, Orlis and LibScout report nothing on this data set (see Table 4), making the fast detection absolutely meaningless. Excluding Allatori (strong), LIBLOOM achieves a speed at 3.53 seconds per app, about twice as fast as LibScout (7 seconds/app).

Compared with the straightforward class-level analysis approach, Libloom-p, the two-stage filtering of LIBLOOM shows a tenfold improvement on the flattened apps, i.e., ProGuard (flt) and DashO (flt), and nearly seven times as fast on ProGuard (default). Besides, on ProGuard (rpkg), Allatori (weak) and DashO (rpkg), LIBLOOM requires only 40% of the time needed by Libloom-p. Figure 11 can explain the time cost of LIBLOOM compared to Libloom-p, which shows the min/max/average percentages of TPLs that are excluded by package filtering. Some apps have more than 95% of TPLs filtered out before class-level analysis and some have quite a few (less than 5%). On average, more than 60% of TPLs are excluded in three cases, resulting in the high speed. Three types of repackaging show lower exclusion rate because a huge destination package may overlap many irrelevant TPL package. Allatori (strong) makes the situation even worse, leaving about 98% of the instances to class-level analysis and thus significantly slowing down the detection.

According to Section 5.2.3, the speed can be further boosted by trading off with the detection accuracy. We apply the overlapping threshold of 1.0 to Allatori (strong) and finish the analysis in only 672 seconds, an 83.7% drop. At the meantime, we get 79.85%, 30.97% and 44.36% for precision, recall and F1, respectively, better than the other tools for Allatori (strong).

5.4.2 Large-scale evaluation

We evaluate the scalability of LIBLOOM, using the closed-source and large-scale benchmarks. Due to inefficiency of LibID, LibPecker and Orlis (refer to [6] and Table 6), we compare LIBLOOM only with Libloom-p and LibScout.

LIBLOOM takes 2.44 hours to complete the detection on the closed-source benchmark. In contrast, Libloom-p and LibScout use 10.41 and 4.98 hours, respectively. On the large-scale benchmark, 50.6, 215.5 and 98.7 hours are required by LIBLOOM, Libloom-p and LibScout. Figure 12 (left) shows the average time cost. In general, LIBLOOM runs two

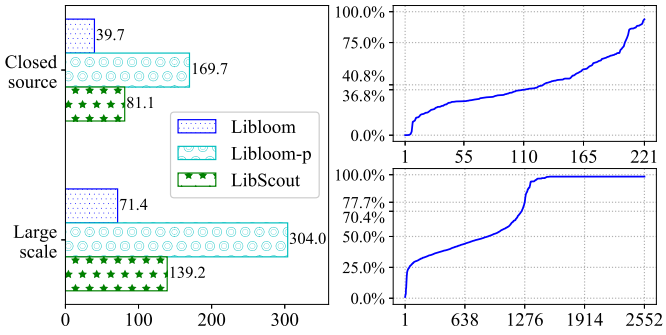


Fig. 12. Average time cost in seconds (left) and sorted TPL exclusion rates (right) on two benchmarks.

times as fast as LibScout and three times faster than Libloom-p.

We have also investigated the effect of package filtering and present the TPL exclusion rate for each app in Figure 12 (right). On the closed-source benchmark, LIBLOOM can only exclude less than 36.8% of the TPLs for half of the apps and the average is 40.8% (877/2114). On the large-scale benchmark, the median and average exclusion rates are 77.7% and 70.4% (8292/11648), respectively. The phenomenon is caused by the fact that the TPLs in the closed-source benchmark are all related to the apps and each TPL has around 40 instances, whereas the large-scale benchmark involves potentially unrelated TPLs, each with fewer instances. Such a difference makes the increment of the time cost nonlinear to that of the TPL collection.

5.5 Vulnerable TPL detection

We further evaluate LIBLOOM's effectiveness of detecting vulnerable TPLs on the large-scale data set.

First, we inspect the top 10 most popular TPLs identified in the APPs. There are four vulnerable libraries, i.e., *retrofit* (*from 2.0 to 2.4.0*, CVE-2018-1000850 [23]), *okhttp* (*2.x before 2.7.4, and 3.x before 3.1.2*, CVE-2016-2402 [33]), *gson* (*before 2.8.9*, CVE-2022-25647 [34]) and *fastjson* (*before 1.2.83*, CVE-2022-25845 [35]). In contrast, LibScout reports only *retrofit*, *okhttp* and *gson* in its top 10 TPLs.

Next, we count the number of apps that integrate the above vulnerable TPLs. The number of the apps reported by LIBLOOM with the vulnerable *retrofit*, *okhttp*, *gson* and *fastjson* are 392, 33, 997 and 514, higher than the number reported by LibScout (359, 33, 899 and 481). We manually check the result and find that LIBLOOM reports more true vulnerable apps than LibScout. Specifically, LIBLOOM discovers 28 apps with vulnerable *okhttp* while the number for LibScout is 26. For those including vulnerable *gson* and *fastjson*, LIBLOOM detects 98 and 33 more true positives, respectively.

Furthermore, we use LIBLOOM to detect the motivating examples in Section 3 on the large-scale TPL data set. The three obfuscated apps can be found to integrate the vulnerable *retrofit* in 27, 74 and 34 seconds, respectively, much less than that for LibID, the only tool that succeeds in detecting the vulnerable TPL in three apps.

5.6 Summary

The two-stage approach has been demonstrated to be effective and efficient, outperforming state-of-the-art TPL detection tools. Moreover, the entropy-based metric has proven to be effective on identifying the destination packages of repackaged and flattened apps. Further studies have also illustrated that our approach provides a scalable detection in large-scale scenarios.

6 DISCUSSION

LIBLOOM relies on only coarse-grained signatures (i.e., the class structures) to detect TPLs. Fine-grained signatures involve the code implementation and can be significantly influenced by heavy code obfuscations like string encryption and control flow randomization. Our approach is robust against those obfuscations that are posed to the code implementation and the experiments have demonstrated the effectiveness.

We make a trade-off between the accuracy and efficiency. While the speed is acceptable in our two-stage approach, it can be further improved by paralleling the detection or utilizing proper hardware (e.g., TCAM) to accelerate the subset queries [20] for industrial use.

Depending on counting the classes, LIBLOOM suffers from severe unused class removal, which can inevitably degrade the detection rate. Many state-of-the-art tools encounter the same problem. A possible solution would be to introduce multidimensional signatures as far as possible, identify appropriate code clusters and detect the integration of TPLs based on the clusters instead of the whole library code base.

LIBLOOM can report a specific version of a target TPL or a limited range of possible versions if multiple TPL instances have shown the same similarity with the one used in an app. Reporting the versions can help analysts identify if vulnerable TPL instances are integrated and we have shown the usefulness in Section 5.5.

Other than the investigated obfuscators that shrink code by default, we notice that a black-box obfuscator, Obfuscapk [36], *bloats* the apps by obfuscation. Obfuscapk can insert wrapper methods (*CallIndirection*), create abundant overloaded methods (*MethodOverload*) and add random fields (*FieldRename*, undocumented). These obfuscations modify the class structure and can threaten our approach that leverages set inclusion to purposefully deal with code shrinking. To examine how far class bloating can affect LIBLOOM's performance, we apply Obfuscapk to the unobfuscated open-source apps and run LIBLOOM on them. The apps are obfuscated with as many obfuscation options as possible. LIBLOOM achieves a precision of 79.66% and a recall of 71.86%, with a time cost of 315 seconds. Bloating-based obfuscation is beyond the scope of this work, but the result is acceptable. We find that those bloating operations affect only a small portion of classes and thus leave a majority of TPLs detected. We will explore an effective and scalable solution in the future to address the bloating-based obfuscation.

Note that, there are some techniques that may threaten the proposed detection though they are not widely adopted. First, the package name of a TPL can be designed the same

as the target app. In this case, the filtering step may exclude the TPL for further analysis due to a low package-level similarity. However, we can add the equivalent package name as a heuristic rule and directly apply class-level analysis, which can improve the detection rate in such a scenario. Second, the target app may be packed. To this end, unpacking techniques such as PackerGrind [37] can be first employed, before performing TPL detection with LIBLOOM. Third, new-generation structural attacks such as HRAT [38] may insert methods to classes. This kind of bloating-based technique is beyond the scope of this work. We leave it as the future work to detect TPLs in this case.

7 Related Work

In this section, we will briefly discuss related work in two aspects, set inclusion query related studies and third-party library detections.

Goel et al. [20] store Bloom filters in TCAM to solve the subset query problem with a single lookup, which can be utilized in TPL detection for acceleration. Charikar et al. [39] investigated new algorithms for subset query with Bloom filter. Zhang et al. [40] provide an analog Bloom filter for multi-bit simultaneous query in wireless network. Adle et al. [41] conduct research on measuring the similarity among sets, even if they are not in a relationship of containment. Our approach is inspired by these studies and we may leverage these ideas in the future to further improve our approach.

Detecting third-party libraries has been carried out over the years. Tang et al. [42] utilize the basic code features and function embeddings to filter the candidates and then leverage the call graph information to determine the TPLs in binary code. Kim et al. [43] create a whitelist of libraries in the apps, to further identify Android malware. Ma et al. [9] cluster millions of Android apps and extract the features for potential third-party libraries, with which they can detect TPLs without the libraries present. Li et al. [44] utilizes the internal code dependencies, i.e., inclusion, inheritance and call, to build the code features. Wang et al. [17] extract the call relations as the signatures of methods, classes and packages and leverage locality sensitive hash functions to measure the signature similarity. Glanz et al. [45] abstract the low level implementations to denote the code signatures, based on which they measure the similarity for library detection. Zhan et al. [46] leverage control flow graphs to roughly match possible libraries and then use basic block information to pinpoint the exact version. Zhang et al. [8] use the dependency information of class inheritance and field/method declaration to generate strict class signatures. Backes et al. [10] use the high level class structure information as strict signatures for library detection. Our approach abstracts the detection as a set inclusion problem and proposed package-level filtering to speed up the detection, which has shown the resilience against non-structure-preserving obfuscations and remarkable efficiency.

8 Conclusion

In this paper, we propose a novel approach to detecting third-party libraries in Android apps with a two-stage Bloom filtering method. Our approach, LIBLOOM, extracts three class structure-based signatures and forms class-level and package-level signature sets, which are encoded into Bloom filters. Package-level Bloom filters are employed to measure the package overlapping so as to keep potential candidate packages and then TPLs for further analysis. By this means, a significant amount of TPLs will be excluded and the detection speed is dramatically improved. Class-level Bloom filters are then leveraged in set inclusion analysis between candidate package pairs. The success of a subset query between two classes indicates that the app class may refer to the library class. If sufficient library classes can be referred to by the classes in a query app, we claim the library of specific version is integrated into the app. To particularly handle sophisticated obfuscations like repackaging and flattening, we design an entropy-based metric to the identify potential obfuscation types and the corresponding destination package of repackaging/flattening. The experiments have demonstrated that LIBLOOM outperforms state-of-the-art detectors in both effectiveness and efficiency and is suitable for large-scale detection.

Acknowledgment

We thank the anonymous reviewers for their insightful comments. This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grants U1836209, 62272465, and 62002361, and in part by the CCF-Huawei Populus euphratica Innovation Research Funding.

References

[1] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," *SIGMETRICS Perform. Eval. Rev.*, vol. 42, no. 1, p. 221–233, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2637364.2592003

[2] S. Wisseman, "Third-party libraries are one of the most insecure parts of an application," https://techbeacon.com/security/third-party-libraries-are-one-most-insecure-parts-application, 2017.

[3] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. Liu, "An empirical study of usages, updates and risks of third-party libraries in java projects," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 35–45.

[4] X. Zhan, T. Zhang, and Y. Tang, "A comparative study of android repackaged apps detection techniques," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 321–331.

[5] L. Li, T. F. Bissyandé, J. Klein, and Y. Le Traon, "An investigation into the use of common libraries in android apps," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 403–414.

[6] X. Zhan, L. Fan, T. Liu, S. Chen, L. Li, H. Wang, Y. Xu, X. Luo, and Y. Liu, "Automated third-party library detection for android applications: Are we there yet?" in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 919–930.

[7] J. Zhang, A. R. Beresford, and S. A. Kollmann, "Libid: Reliable identification of obfuscated third-party android libraries," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 55–65. [Online]. Available: https://doi.org/10.1145/3293882.3330563

[8] Y. Zhang, J. Dai, X. Zhang, S. Huang, Z. Yang, M. Yang, and H. Chen, "Detecting third-party libraries in android applications with high precision and recall," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 141–152.

[9] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: Fast and accurate detection of third-party libraries in android apps," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 653–656. [Online]. Available: https://doi.org/10.1145/2889160.2889178

[10] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 356–367. [Online]. Available: https://doi.org/10.1145/2976749.2978333

[11] "ProGuard: Java obfuscator and Android app optimizer," https://www.guardsquare.com/proguard, 2021.

[12] "Allatori," https://www.allatori.com/, 2021.

[13] "Dasho," https://www.preemptive.com/products/dasho, 2021.

[14] X. Zhan, T. Liu, L. Fan, L. Li, S. Chen, X. Luo, and Y. Liu, "Research on third-party libraries in android apps: A taxonomy and systematic literature review," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[15] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, p. 422–426, Jul. 1970.

[16] "Wikipedia: Official feature-rich viewer of the free online encyclopedia," https://f-droid.org/en/packages/org.wikipedia/, 2021.

[17] Y. Wang, H. Wu, H. Zhang, and A. Rountev, "Orlis: Obfuscation-resilient library detection for android," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 13–23. [Online]. Available: https://doi.org/10.1145/3197231.3197248

[18] X. Zhan, T. Liu, Y. Liu, Y. Liu, L. Li, H. Wang, and X. Luo, "A systematic assessment on android third-party library detection tools," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[19] "Bloom filter," https://en.wikipedia.org/wiki/Bloom_filter, 2021.

[20] A. Goel and P. Gupta, "Small subset queries and bloom filters using ternary associative memories, with applications," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 143–154.

[21] "An open source project using a vulnerable TPL," https://f-droid.org/zh_Hans/packages/com.jvillalba.apod.md/, 2018.

[22] "F-Droid," https://f-droid.org/, 2021.

[23] "CVE-2018-1000850," https://nvd.nist.gov/vuln/detail/CVE-2018-1000850, 2018.

[24] "WALA: T.J. Watson Libraries for Analysis," http://wala.sourceforge.net, 2021.

[25] "Guava: Google Core Libraries for Java," https://github.com/google/guava, 2021.

[26] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: Building a better bloom filter," *Random Struct. Algorithms*, vol. 33, no. 2, p. 187–218, Sep. 2008.

[27] "Hungarian algorithm," https://en.wikipedia.org/wiki/Hungarian_algorithm, 2021.

[28] S. Ce, "The mathematical theory of communication." *M.D. computing : computers in medical practice*, vol. 14, no. 4, pp. 306–317, 1963.

[29] "LibID," https://github.com/MIchicho/LibID, 2021.

[30] "Benchmarks for TPLs detection," https://sites.google.com/view/libdetect/, 2020.

[31] "MvnRepository," https://mvnrepository.com/, 2021.

[32] "Xiaomi APP Store," https://app.mi.com/, 2021.

[33] "CVE-2016-2402," https://nvd.nist.gov/vuln/detail/CVE-2016-2402, 2016.

[34] "CVE-2022-25647," https://nvd.nist.gov/vuln/detail/CVE-2022-25647, 2022.

[35] "CVE-2022-25845," https://nvd.nist.gov/vuln/detail/CVE-2022-25845, 2022.

[36] S. Aonzo, G. C. Georgiu, L. Verderame, and A. Merlo, "Obfuscapk: An open-source black-box obfuscation tool for android apps," *SoftwareX*, vol. 11, p. 100403, 2020.

[37] L. Xue, H. Zhou, X. Luo, L. Yu, D. Wu, Y. Zhou, and X. Ma, "Packergrind: An adaptive unpacking system for android apps," *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 551–570, 2022.

[38] K. Zhao, H. Zhou, Y. Zhu, X. Zhan, K. Zhou, J. Li, L. Yu, W. Yuan, and X. Luo, "Structural attack against graph based android malware detection," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 3218–3235.

[39] M. Charikar, P. Indyk, and R. Panigrahy, "New algorithms for subset query, partial match, orthogonal range searching, and related problems," in *Automata, Languages and Programming*, P. Widmayer, S. Eidenbenz, F. Triguero, R. Morales, R. Conejo, and M. Hennessy, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 451–462.

[40] Z. Zhang, "Analog bloom filter and contention-free multi-bit simultaneous query for centralized wireless networks," *IEEE/ACM Trans. Netw.*, vol. 25, no. 5, p. 2916–2929, Oct. 2017.

[41] T. D. Ahle and J. B. T. Knudsen, "Subsets and supermajorities: Optimal hashing-based set similarity search," in *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, 2020, pp. 728–739.

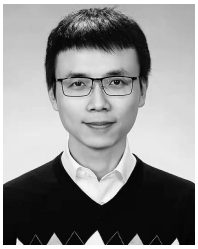
[42] W. Tang, Y. Wang, H. Zhang, S. Han, P. Luo, and D. Zhang, "Libdb: An effective and efficient framework for detecting third-party libraries in binaries," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 423–434.

[43] K. Kim, J. Lee, S. Lee, and J. Hong, "Whitelist for analyzing android malware," in *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, ser. RACS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 222–227. [Online]. Available: https://doi.org/10.1145/3129676.3129726

[44] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "Libd: Scalable and precise third-party library detection in android markets," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017, p. 335–346. [Online]. Available: https://doi.org/10.1109/ICSE.2017.38

[45] L. Glanz, S. Amann, M. Eichberg, M. Reif, B. Hermann, J. Lerch, and M. Mezini, "Codematch: Obfuscation won't conceal your repackaged app," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 638–648. [Online]. Available: https://doi.org/10.1145/3106237.3106305

[46] X. Zhan, L. Fan, S. Chen, F. We, T. Liu, X. Luo, and Y. Liu, "Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1695–1707.



Jianjun Huang received the Ph.D. degree in Computer Science from Purdue University. He is currently an assistant professor at School of Information, Renmin University of China. His research interests focus on program analysis, vulnerability detection, and mobile security.

Bo Xue is a Ph.D. student in Information Security from Renmin University of China. His research interests focus on moblie application security and code security analysis.

Jiasheng Jiang received the M.S. degree in Information Security from Renmin University of China. His research interests focus on blockchain security.

Wei You received the Ph.D. degree in Computer Science from School of Information, Renmin University of China. He is currently an associate professor at School of Information, Renmin University of China. His research interests focus on program analysis, mobile security and Web security.

Bin Liang received the Ph.D. degree in Computer Science from Institute of Software, Chinese Academy of Sciences. He is currently a professor at School of Information, Renmin University of China. His research interests focus on program analysis, vulnerability detection, mobile security and AI security.

Jingzheng Wu received his Ph.D. degree in computer software and theory from the Institute of Software, Chinese Academy of Sciences, Beijing, in 2012. He is a research professor at the Institute of Software, Chinese Academy of Sciences, Beijing. His primary research interests include system security, vulnerability detection and covert channels.

Yanjun Wu received his Ph.D. degree in computer software and theory from the Institute of Software, Chinese Academy of Sciences, Beijing, in 2006. He is a research professor at the Institute of Software, Chinese Academy of Sciences, Beijing. His primary research interests include operating system and system security.