Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Securing android applications via edge assistant third-party library detection

Zhushou Tang[a,b], Minhui Xue[c], Guozhu Meng[d], Chengguo Ying[b], Yugeng Liu[a], Jianan He[e], Haojin Zhu[a,1,*], Yang Liu[d]

[a] *Shanghai Jiao Tong University, Shanghai, China*
[b] *PWNZEN InfoTech Co., LTD, Shanghai, China*
[c] *Optus Macquarie University Cyber Security Hub, Sydney, Australia*
[d] *Nanyang Technological University, Singapore*
[e] *China Electronics Technology Group Corporation No.32, Shanghai, China*

ARTICLE INFO

Article history:
Received 17 February 2018
Revised 19 June 2018
Accepted 7 July 2018
Available online 15 October 2018

Keywords:
Third-party library detection
Edge computing
Mobile security
Malware detection
Vulnerability detection

ABSTRACT

Third-party library (TPL) detection in Android has been a hot topic to security researchers for a long time. A precise yet scalable detection of TPLs in applications can greatly facilitate other security activities such as TPL integrity checking, malware detection, and privacy leakage detection. Since TPLs of specific versions may exhibit their own security issues, the identification of TPL as well as its concrete version, can help assess the security of Android APPs. However in reality, existing approaches of TPL detection suffer from low efficiency for their detection algorithm to impracticable and low accuracy due to insufficient analysis data, inappropriate features, or the disturbance from code obfuscation, shrinkage, and optimization.

In this paper, we present an automated approach, named PanGuard, to detect TPLs from an enormous number of Android APPs. We propose a novel combination of features including both structural and content information for packages in APPs to characterize TPLs. In order to address the difficulties caused by code obfuscation, shrinkage, and optimization, we identify the invariants that are unchanged during mutation, separate TPLs from the primary code in APPs, and use these invariants to determine the contained TPLs as well as their versions. The extensive experiments show that PanGuard achieves a high accuracy and scalability simultaneously in TPL detection. In order to accommodate to optimized TPL detection, which has not been mentioned by previous work, we adopt set analysis, which speed up the detection as a side effect.

PanGuard is implemented and applied on an industrial edge computing platform, and powers the identification of TPL. Beside fast detection algorithm, the edge computing deployment architecture make the detection scalable to real-time detection on a large volume of emerging APPs. Based on the detection results from millions of Android APPs, we suc-

* Corresponding author.
E-mail addresses: ellison.tang@gmail.com (Z. Tang), minhui.xue@mq.edu.au (M. Xue), gzmeng@ntu.edu.sg (G. Meng), anjiannian@gmail.com (C. Ying), liuyugeng@sjtu.edu.cn (Y. Liu), 13428466234@139.com (J. He), zhu-hj@cs.sjtu.edu.cn, zhuhaojin@gmail.com (H. Zhu), yangliu@ntu.edu.sg (Y. Liu).
[1] This work was partially supported by NSFC (Grant no. U1401253 and 61672350) and the Joint Funds of CETC (Grant No. 20166141B08020101).

https://doi.org/10.1016/j.cose.2018.07.024

cessfully identify over 800 TPLs with 12 versions on average. By investigating the differences amongst these versions, we identify over 10 security issues in TPLs, and shed light on the significance of TPL detection with the caused harmful impacts on the Android ecosystem.

1. Introduction

Mobile application development is undergoing a significant revolution. Functionalities of one APP are decoupled and modularized for code reuse and collaborative development, which leads to a burgeoning increase of third-party libraries (TPLs). According to the statistics by AppBrain[2], there are 400 TPLs providing ample functionalities, such as advertisement, game engines, and social network integration. These TPLs are widely deployed and reused in Android APPs. WuKong (Wang et al., 2015) states that TPLs have taken more than 60% of code in Android APPs, and they contribute 41% of APP code as reported in Li et al. (2016). Therefore, TPLs have prominently facilitated the development of Android APPs.

The use of TPL is a double-edged sword. In particular, the security of TPL has raised an increasing attention recently. In particular, during the high popularity of TPL, a vulnerable TPL may make thousands of APPs in the risk. According to a report released by PanguTeam, a severe vulnerability in TPL was found to widely exist in Tencent family APPs including Tencent Browser, QQ Hotspot, which can lead to unauthorized sensitive data access or application setup (Pangu, 2017b). Therefore, there is a large body of recent studies on TPL detection (Backes et al., 2016; Li et al., 2017; Ma et al., 2016; Narayanan et al., 2014) and security analysis of TPLs (Chen et al., 2016a; Liu et al., 2014; Meng et al., 2016; Rastogi et al., 2016).

However, it still faces the following challenges to identify TPLs in Android Applications. Firstly, the hand-crafted ambiguous code by attackers is usually mingled with the TPL code, which makes it easily escape the security inspection and distort the integrity of TPL. Additionally, the vulnerable TPLs or a specific vulnerable TPL version carried out by an APP make the APP in a fragile stat. Since most original TPLs or their historical version is unavailable for downloading, we use historical collected APP to build TPL corpus. With a large volume APPs in hand, how to distinguish the TPL code from the APPs remains the first challenge. Secondly, obfuscation, shrinkage, and optimization are widely performed on TPLs that impede the TPL detection. For instance, ProGuard, which is able to obfuscate, shrink, and optimize code, is observed to being applied on 23.22% APPs. Although a handful of studies propose approaches (Backes et al., 2016; Ma et al., 2016) that are resilient to obfuscation, they fail to consider the impact by shrinkage and optimization to code; Lastly but not least, there are around 1,300 new APPs and their integrated TPLs emerging per day (Dogtiev, 2017). This introduces a serious scalability issue especially when millions of detection instances should be supported. Therefore, an automated tool is demanding to identify TPLs in an accurate and scalable manner.

To overcome the above challenges, we propose the approach PanGuard which has been integrated into the online APP analysis platform Janus [3]. To make the proposed architecture more scalable, we adopt edge computing architecture to deploy our TPL detecting tool. Each TPL is depict as bit vectors, which reduce the bandwidth requirement for deploying feature from cloud to edge and vice versa. On the edge side, policy for skip analyzing and the checking algorithm, namely set analysis, bit vector operation in practice, ensuring the real time TPL detection. In addition to the novel edge-assistant architecture, the proposed PanGuard has addressed the technical challenges from the following aspects.

Firstly, to build TPL corpus, we propose a novel signature to depict the characteristics of TPLs, including both structural and content information. By recursively traversing the structure of packages in TPLs, and then computing their hash values, we obtain a signature for each node of the tree-like package. The signature demonstrates the hierarchy and content from this node, and is robust against literal changes (e.g., optimization) to TPLs. Therefore, the hash values serve to depict the concrete versions of TPLs. We apply our approach on a collected large dataset with 9,049,323 APPs to build the TPL corpus. The enormousness of our dataset is able to minimize the disturbance (e.g., a rare TPL variance) within TPLs, and makes the detection more accurate.

Secondly, to handle the obfuscation issue, we conduct a comprehensive investigation on “mutated” (i.e., obfuscated, shrunk, or optimized) TPLs mainly by ProGuard. It is found that ProGuard removes unused classes, fields, methods in its shrinkage stage or changes the modifier of field or method, optimizes the code sequence, evaluates constant values in advance, propagates constant values to callee, etc. in its optimization stage, and attributes to the original code. These changes blur the boundaries between the primary code and TPL, which makes the extraction of TPLs difficult. Therefore, we employ a decoupling algorithm to separate TPLs from the primary code in advance. Then, we identify invariants that are stable during the mutation, for example, stable Android SDK API invoked by TPLs. Stable Android SDK API are those that build the basis of functionalities of TPLs. These APIs are deprived and serve as a distinguishing feature to depict the mutated TPLs.

Lastly, to make the proposed architecture more scalable, we develop an off-the-shelf tool named PanGuard, which integrates all algorithms of TPL detection in Section 2. PanGuard generates features from APPs automatically, and performs a feature set matching algorithm to identify TPLs. PanGuard is also robust against multiple mutations (e.g., obfuscation, shrinkage, and optimization) to original TPLs. So far, PanGuard have analyzed over one million Android APPs, and

[2] https://www.appbrain.com/stats/libraries

[3] Janus (http://www.appscan.io), a large-scale mobile APP analysis platform released by Pangu Team in April 2017.

identified 800 TPLs with 9623 versions. Compared to other state-of-the-art tools of TPL detection. Moreover, by applying PanGuard to a real case, we find a buggy version of a TPL which is previously unknown to public and that the buggy version is integrated by over 13,000 APPs; even the APP enclosing the buggy TPL is shrunk and optimized by ProGuard, our tool can still detect them successfully.

We establishes an online APP analysis platform Janus, which has collected millions of Android APPs. In advance of analyzing Android APPs, Janus needs to identify and thereby eliminates integrated TPLs with the purpose of raising the accuracy of detection. PanGuard is a part of Janus which is responsible for identifying TPLs. TPL identification can vastly reduce storage on Janus since we only need to reserve a copy of the meta-data of a TPL for now. Since Janus has integrated an online interactive analysis environment *Akana* (Pangu, 2017a). This program analysis tool is computationally intensive and storage consuming. Fortunately, PanGuard has drastically reduced the pressure of storage and computation. In addition, by filtering out TPLs identified by PanGuard, our product Janus has worked more efficiently and effectively in malware detection and vulnerability analysis.

In summary, our contributions are listed as follows:

- We present an edge-assistant TPL detection architecture. Edge nodes in this architecture are responsible for TPL detection and cloud in this architecture is mainly responsible for TPL signature generation. On top of the edge computing architecture, our work is scalable to real time TPL detection for the unprecedentedly Android APP.
- We conduct a systematical and thorough study of factors that disturb the TPL detection, including obfuscation, shrinkage, and optimization. Especially for the optimization factor, which has never been mentioned by previous work. Our study discloses that 23.22% of TPLs have leveraged these factors. Among these TPLs, about one third of them cannot be detected by previous work for they are processed by optimization. These results can help to improve TPL detection algorithms.
- We propose an accurate and scalable approach to detect TPLs which is adapted to mutation introduced by ProGuard. (e.g., obfuscation, shrinkage, and optimization). In addition, we develop an automated tool, PanGuard, to analyze the TPLs in APPs in our data set. The experimental results show that PanGuard can find over 10 vulnerable TPLs.

2. Background of third-party library detection

Third-party library (TPL) detection is to detect third party libraries employed by Android APPs. The current detection methodology can mainly be classified into four categories: string-based, token-based, tree-based, and semantics-based (Jiang et al., 2007; Roy and Cordy, 2007; Roy et al., 2009). However, they are computationally insufficient when an APP is processed by ProGuard. To ease the presentation, we introduce the following definitions.

Definition 2.1 (Original TPL). An off-the-shelf TPL used by an APP, which has not been processed by ProGuard with any modification. In this case, the embedded TPL remains the same as the off-the-shelf TPL.

Definition 2.2 (Obfuscated TPL). An off-the-shelf TPL used by an APP, which is processed by ProGuard using only obfuscation. In this case, the TPL in the APP is also obfuscated by ProGuard.

Definition 2.3 (Shrunk TPL). An off-the-shelf TPL used by an APP, which is processed by ProGuard using only shrinkage. In this case, the TPL in the APP is also shrunk by ProGuard.

Definition 2.4 (Optimized TPL). An off-the-shelf TPL used by an APP, which is processed by ProGuard using only optimization. In this case, the TPL in the APP is also optimized by ProGuard.

Definition 2.5 (Package Stem). "Package Stem" is part of a fully qualified name (Wikipedia, 2017), which is presented as a *namespace*. Class files in the "Package Stem" covers the most part of a TPL, which means that the "Package Stem" is short enough to encapsulate most of the class files in a TPL. Meanwhile, the namespace can be extended to detect another version of a TPL and the features associated with "Package Stem" are useful in detection, which means that the "Package Stem" should be long enough to avoid conflicts.

Definition 2.6 (Package Dependency Graph (PkgDG)). Package Dependency Graph, abbreviated to PkgDG, shows dependency between packages, fields, and methods that contribute to the graph.

2.1. *A Study of ProGuard*

ProGuard is an officially recommended tool for developers to shrink their APPs in practice (Google, 2017). Apart from shrinkage, ProGuard can also perform other functions–optimization, obfuscation, and verification to Android projects. The prevalent usage of ProGuard introduces challenges to TPL detection.

2.1.1. *The Obfuscation of ProGuard*

The obfuscation of ProGuard renames the classes, fields, and methods by using short and meaningless names, and it is prevailing in the Android APP development. Previously, Duet (Hu et al., 2014) claims that over 80% of TPLs in 100,000 Google Play APPs are indeed used without any modification. Our evaluation verified this data in Section 4.2.4.

2.1.2. *The shrinkage of ProGuard*

Since a TPL provides functional interface, of which a subset is available to developers, the shrinkage of ProGuard will tremendously reduce the storage space of an APP (see breakdowns of shrunk TPL by ProGuard in Table 1). The shrinkage of ProGuard mainly relies on detecting and removing unused classes, fields, methods, and attributes. In this scenario, the corpus built for original or obfuscated TPLs cannot be used to detect shrunk TPL directly. Much research uses subgraph isomorphism (Fan et al., 2017) and intersection detection (Backes et al., 2016) to identify shrunk TPLs.

```java
private String hostName = "smtp.xxx.com";
private String userName = "user@xxx.com";
private String password = "password";
private String fromAddress = "from@xx.com";
private String fromName = "from";
private String toAddress = "to@xx.com";
private String toName = "to";

public void sendSimpleEmail() throws Exception {
    SimpleEmail email = new SimpleEmail();
    email.setHostName(hostName);
    email.setAuthentication(userName, password);
    email.setCharset("utf-8");
   email.setFrom(fromAddress, fromName, "utf-8");
   email.addTo(toAddress, toName, "utf-8");
   email.setSubject("Subject");
   email.setMsg("Simple mail");
   email.send();
}
```

Listing 1 – Original code for sending a simple mail through "Apache Commons Email".

2.1.3. *The Optimization of ProGuard*

To date, the released version of ProGuard ranges from 1.0 to 5.3.3, with 47 versions in total. Since version 3.0.7, ProGuard has provided a function to optimize an APP at the bytecode level within and across methods. Specifically, ProGuard uses techniques, such as control flow analysis, data flow analysis, partial evaluation, static single assignment, global value numbering, and liveness analysis to optimize APPs. To our knowledge, no previous work has examined the optimization problem in TPL detection, which we detail as follows.

In the optimization step, ProGuard analyzes and optimizes the bytecode of methods. The latest version, ProGuard 5.3.3, provides 17 types of optimizations, including modifier changing, constant propagation, inlining, peephole optimizations, etc. The most prominent peephole optimization contains over 200 peephole optimizations, say pre-concatenating two concrete strings.

The optimization introduces complexity for TPL identification. For example, user space may intrude the TPL code through constant propagation (caller → callee); TPL code may intrude the user space by inlining (callee → caller). To take a controlled study of the optimization problem, we turn off the obfuscation option of ProGuard. Listing 1 and 2 are two examples that show code in TPL is inlined to user space by optimization of ProGuard. We conclude that optimization generally obscures the boundary between user space and TPL.

The optimization introduces more complexity if different versions of ProGuard are used for APPs. To continue with the aforementioned inlining, in ProGuard 3.0.7, the first appearance of inlining declares simple methods of getters and setters. However, in ProGuard 5.3.3, the inlining supports inlined constant fields, method parameters, return values, or inline methods that are short or only called once. TPL optimization by various versions of ProGuard turns out different outcomes. Furthermore, different optimization adopts different search algorithms (caller → callee / callee → caller), for which ProGuard optimizes APPs randomly. In this case, different TPLs can be issued by the same version of ProGuard.

In order to clarify the optimization problem, we take a systematic study of the effect of optimization of the current prevalent TPLs, by released versions of ProGuard updated from 3.0.7 to version 5.3.3. Table 3 shows how optimization distorts commonly-used features and defeats most practices of state-of-the-art TPL detection. Specifically, we find that the "final" method is the most prevalent and prominent signature for ProGuard optimization after an insight study of the his-

```java

ApacheMailTest v0 = new ApacheMailTest();
try {
    SimpleEmail v1_1 = new SimpleEmail();
    v1_1.setHostName(v0.hostName);
    ((Email)v1_1).setAuthenticator(new DefaultAuthenticator(v0.userName, v0.password)); //
        Interface defined in ‘‘org.apache.commons.mail’’ are inlined to user code.
    v1_1.setCharset("utf-8");
    v1_1.setFrom(v0.fromAddress, v0.fromName, "utf-8");
    v1_1.addTo(v0.toAddress, v0.toName, "utf-8");
    v1_1.setSubject("Subject");
    String v2 = "Simple mail"; // Inlined ‘‘setMsg’’ method.
    if(EmailUtils.isEmpty(v2)) {
        throw new EmailException("Invalid message supplied");
    }

    v1_1.setContent(v2, "text/plain");
    ((Email)v1_1).buildMimeMessage(); // Inlined ‘‘send’’ method.
    ((Email)v1_1).sendMimeMessage();
}
catch(Exception v1) {
    v1.printStackTrace();
}
```

Listing 2 – ProGuard Inlining code from "Apache Commons Email" to user space.

Table 1 – TPL shrunk by PROGUARD.

TPL	TPL Version	# classes	# classes left	# fields	# fields left	# methods	# methods left
Amazon s3	2.4.3	6945	538 (7.75%)	15525	1426 (9.19%)	50585	2895 (5.72%)
Amazon s3*	2.4.3	1244	499 (40.11%)	3384	1270 (37.53%)	8703	2572 (29.55%)
Apache Email	1.5	349	232 (66.48%)	1253	711 (56.74%)	2817	1801 (63.93%)
Apache Ftp	3.6	195	57 (29.23%)	1270	207 (16.30%)	1815	313 (17.25%)
Gson	2.8.1	186	160 (86.02%)	421	290 (68.88%)	1027	771 (75.07%)
Flurry Analytics	7.2.3	338	328 (97.04%)	940	927 (98.62%)	1445	1435 (99.31%)
OkHttp	2.7.5	234	182 (77.78%)	992	724 (72.98%)	2187	1416 (64.75%)
Nine Old Androids	2.4.0	145	133 (91.72%)	474	339 (71.52%)	930	688 (73.98%)
Google Cloud Messaging	5	4132	120 (2.90%)	10398	424 (4.08%)	26118	673 (2.58%)
Picasso	2.5.2	92	72 (78.26%)	352	247 (70.17%)	522	362 (69.35%)
Http Client	4.5.3	732	382 (52.19%)	1909	698 (36.56%)	4707	2501 (43.57%)
InApplication Billing	5	320	28 (8.75%)	853	78 (9.14%)	2449	118 (4.82%)
Means			53.19%		45.98%		45.82%
Standard Deviation			0.33		0.30		0.32

1 Data issued from a modification of PROGUARD 5.3.3. To perform this task, we prefer using the TPL's officially-recommended configuration for PROGUARD. 2 Each sample is fully functional. For example, "Apache Ftp" uses upload and download function. 3 When using PROGUARD, we only use code shrinkage, which means the result derives from only one iteration. 4 By observation, intersection detection analysis does not work for most of the TPL shrunk by PROGUARD.

Table 2 – Previous work on TPL detection.

Previous work	Methodology	Can detect original TPL	Can detect obfuscated TPL	Can detect shrunk TPL	Can detect optimized TPL
AdDetect Narayanan et al. (2014)	Machine learning	√	√	×	×
LibRadar Ma et al. (2016)	Use system API which is resilient to obfuscation	√	√	×	×
LibScout Backes et al. (2016)	Similarity analysis based on features which is resilient to obfuscation	√	√	×	×
LibD Li et al. (2017)	Use opcode which is resilient to obfuscation	√	√	×	×

1 LibScout (Backes et al., 2016) use profile matching to get the candidate similar TPLs, which is likely to accommodate to shrunk TPL detection. But statistical data in Table 1 shows that about 53.19% classes in TPLs are left by shrinking of ProGuard. Moreover, for each left class, lots of methods in the class are removed which will distort the profile of the class. All these make it's algorithm does not work on shrunk TPL detection.

torical of ProGuard. To avoid the fact that the method modifier is already defined as "final", which will lead to false positives, we collect all methods which are not finalized in the historical "Android Support Repository". Then, we check if an APP contains these methods and verify their modifiers. If the modifier is converted to "final", then the APP is optimized.

Because optimization distorts commonly used features for TPL detection and optimized APPs take up a large portion of APPs, a more fine-grained TPL detection approach is desired. To make the requirement clear, previous work and their abilities are summarized in Table 2.

2.2. *The Use of PROGUARD*

In this subsection, we attempt to show that customizing the use of PROGUARD will complicate the TPL detection.

The actual workflow of PROGUARD is shown in Fig. 1. For PROGUARD, shrinkage, optimization, and obfuscation are default options for processing class files (GuardSquare, 2017). Along the workflow pipeline, the number of iteration of shrinkage and optimization is controlled by "-

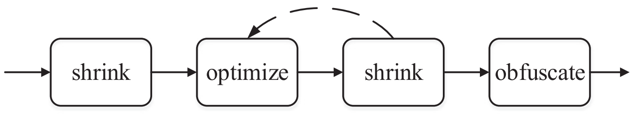


Fig. 1 – Workflow of PROGUARD

optimizationpasses" option. If this argument is specified, PROGUARD keeps optimizing or shrinking an APP until the loop-bound is reached or no optimization can be applied to the class file.

From the perspective of APP developing environment, the most-recent configuration file (such as "project.properties" file in eclipse) is generated by "Android Tool", which supplies a standard configuration for developers when using PROGUARD. For optimization, it turns off the option by default (with option "*-dontoptimize*").

From the perspective of a developer, the customized use of PROGUARD can trigger different consequences even if the TPL is processed by the same PROGUARD. Again, in the example of

Table 3 – TPL optimized by ProGuard.

TPL	Amazon s3	Amazon s3*	Apache Email	Apache Ftp	Gson	Flurry Analytics	OkHttp	Nine Old Androids	Google Cloud Messaging	Picasso	Http Client	In-Application Billing
TPL Version	2.4.3	2.4.3	1.5	3.6	2.8.1	7.2.3	2.7.5	2.4.0	5	2.5.2	4.5.3	5
# optimize iterations	7	7	3	5	5	2	6	4	4	3	4	4
# finalized classes	27	27	13	32	32	1	48	100	60	38	198	19
# unboxed enum classes	0	0	0	0	0	0	1	0	0	1	2	0
# privatized methods	313	206	17	26	109	0	55	52	81	24	235	14
# staticized methods	111	106	7	9	71	0	23	42	35	4	70	1
# finalized methods	1410	1234	137	160	399	1	646	359	360	167	987	64
# removed method parameters	138	132	0	2	2	0	30	0	51	10	32	1
# inlined constant parameters	26	26	4	5	7	0	22	24	7	0	32	5
# inlined constant return values	10	10	0	1	0	0	6	0	6	0	8	1
# inlined short method calls	2697	1364	26	168	142	0	553	253	48	96	402	17
# inlined unique method calls	639	632	32	83	122	0	342	113	170	91	280	20
# inlined tail recursion calls	0	0	0	0	5	0	2	1	2	0	0	0
# merged code blocks	6	6	4	1	2	0	14	0	5	3	5	0
# variable peephole optimizations	4165	2681	678	376	473	2	1293	651	412	287	1378	67
# field peephole optimizations	6	5	3	2	1	0	1	1	3	1	1	0
# branch peephole optimizations	657	644	223	64	180	0	294	66	128	84	263	13
# string peephole optimizations	480	318	5	28	82	0	179	2	139	45	205	0
# simplified instructions	202	136	30	26	21	2	210	73	40	12	138	26
# removed instructions	868	649	220	96	132	4	1016	80	481	83	472	65
# removed local variables	158	118	13	23	40	0	53	20	33	13	58	1
# removed exception blocks	31	31	9	6	1	0	12	0	4	3	23	0
# optimized local variable frames	645	549	288	87	197	1	331	192	177	86	416	37

1 Data issued from a modification of ProGuard 5.3.3. 2 The default configuration for using ProGuard (proguard-android-optimize.txt) turns off "vertically merged classes", "horizontally merged classes", "removed write-only fields", "privatized fields", "inlined constant fields", "arithmetic peephole optimizations", and "cast peephole optimizations" options, result for these options are not listed in the table. 3 APPs will not turn to be stable until "optimize iterations". Other data comes from the first iteration of optimization. 4 Optimization is performed based on the results shown in Table 1. The overlapping of shrinkage and optimization tremendously change the APP.

inlining, the default strategy for inlining a short method is to use the code block of the size no greater than 8. However, a developer can use his own strategy by specifying the argument of "*maximum.inlined.code.length*" to expand or shrink the code block for inlining.

In summary, different versions of ProGuard and various usage of ProGuard lead TPLs inconsistent, which is extremely hard to precisely predict the final result of a TPL processed by ProGuard, especially when the TPL is optimized. Because ProGuard is pervasively used for APP developing, it is highly desirable to establish new TPL corpus from the scratch and introduce stable features for TPL identification, ultimately rendering mutated TPLs to be readable.

3. Our approach and implementation

3.1. *System overview*

Aside from TPL detection methodology, an architecture is required to ensure the feasibly of TPL detection. In order to deploy TPL checking tool to protect end-user in real time, and promise performance benefits such as low latency and quick response time for customer, we make a profound survey on the bottleneck of real-world TPL detection in advance.

The data for TPL checking is coming from geographically distributed organizations, acquiring from the traffic flow. A high volume data is generated each day by observation. For example, about eighty millions URLs point to APP are discovered each day in CERNET network. The explosive proliferation of APPs for detection requires high bandwidth to transmit and becoming a big computing burden for our cloud. Since the pure cloud computing can't be competent for TPL detection for the unpredictable latency, bandwidth bottlenecks, an elaborately designed architecture is required for leverage resources for computation, networking, and storage.

Considering the geographically distributed organizations where data is generated have the ability to equip with sufficient computing resource, we propose an edge computing architecture which provide elastic resources at the edge of the network. This computing paradigm collaboratively provide elastic computation, storage and communication for TPL detection. Data generated and mainly analyzed at edge, which reduce the delay of data analytics and decrease the cost of data transmission and storage.

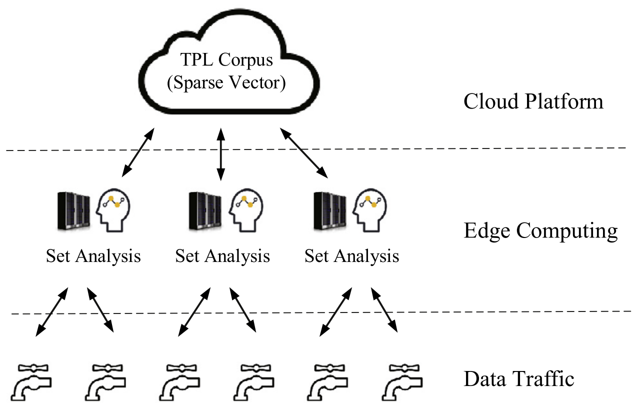


Fig. 2 – Edge computing architecture for real time TPL detection.

Table 4 – JANUS configuration.

Cluster	Node	Configuration
Hadoop	2 masters	4 cores CPU, 32 gb memory, 80 gb disk
	26 slaves	4 cores CPU, 12 gb memory, 8 tb disk
Elastic Search	3 masters	4 cores CPU, 16 gb memory, 500 gb disk
	172 slaves	4 cores CPU, 8 gb memory, 500 gb disk

Our TPL checking tool is now providing service for CETC, China Unicom, etc. The customer is geographically near to the spot where data is generated, and as Fig. 2 depicted, edge nodes are maintained by these organizations (customers). Meanwhile, some tasks, such as TPL signature exchanging, might benefit from the cloud running in the back-end, and the cloud is owned and maintained by us.

Cloud of this architecture is the Hadoop cluster of our JANUS platform, which is located in the cloud service provider UCloud (Ucloud Co., 2017). Cloud build TPL corpus and exchange signature between edge. The JANUS configuration is shown in Table 4.

Edge nodes are geographically distributed, and logically decentralized in that they are maintained by different organizations. The organizations have the advantage to collect APPs and the localized edge overcome the high bandwidth required for transmitting APPs from different channels to our cloud.

To coordinate different entities in the edge computing architecture and providing a high performance and effective system, we define policies for the system. They are synchronization policy, locking policy, and migration policy. (1) The synchronization policy can reduce resources for computation. For example, a pre-checked APP should not be checked again. In order to achieve this goal, hash or signature for the pre-checked APP should be shared between the edge nodes. The policy enforces data items synchronization at a threshold of 100. That is, when the accumulated new APPs reach 100, the enforced data synchronization is started. (2) The locking policy guarantees the integrity of data. If the edge generated data and synchronized data from other edge arrived at the same time, the unlocked write operation will distort the data collection in the edge node. (3) The migration policy can improve the overall system performance. In practice, edge nodes in our architecture are commonly equipped with low performance servers. To ensure the quick response time even when computation resource on the edge node has been exhausted, we offload the computation task to the back-end cloud.

3.2. *Establishing the TPL Corpus*

In this subsection, we try to establish the TPL corpus, which is indexed as a hash token, in order to detect both original or mutated (i.e., obfuscated, shrunk and optimized) TPLs. The built-in corpus is resilient against the obfuscation of PROGUARD.

We leverage "Big Code" to establish the corpus, based on the assumption that a TPL is used by many APPs. Apart from traditional work that uses qualified names as a feature of a TPL, instead, we use the combination of Android SDK API and string hashing as our features. After setting up the tree-based feature, we intentionally probe the most outstanding qualified name, known as "Package Stem", to identify other versions of the TPL and related hashing. The built-in corpus serves as a benchmark dataset for detecting original or obfuscated TPLs directly. The pipeline of establishing TPL corpus is illustrated in Fig. 3.

3.2.1. *Building the hash-tree package*

We use Android SDK APIs (Ma et al., 2016) and strings to derive the package-level feature, because Android SDK APIs and strings remain stable even if an APP is obfuscated by PROGUARD. The Android package is structured as a hierarchical tree. In the tree, a node is a folder and a leaf is a class file. We apply recursive depth-first search (DFS) algorithm to generate the hash tree, and then generate hash values based on the child hash values, in the way that hash values preserve the structural information of a TPL. Specifically, we do the following:

(i) For each leaf i, we generate a vector $\mathbf{v_i}$ of the functions, where each component encodes an Android SDK API. The length of the vector $\mathbf{v_i}$ is denoted $l_i = \|\mathbf{v_i}\|$. We then hash $\mathbf{v_i}$ and output a hash value $s_i^1 = SHA - 256(\mathbf{v_i})$; we also hash ordered strings str_i and output another hash value $s_i^2 = SHA - 256(str_i)$. We then hash the ordered s_i^1 and s_i^2, and generate the final hash value $s_i = SHA - 256(s_i^1, s_i^2)$.

(ii) For each non-leaf node i with child nodes i_c, we simply take in all ordered hash values of its child's nodes s_{i_c} and output $s_i = SHA - 256(s_{i_c})$. Likewise, the length of the vector $\mathbf{v_i}$ equals to the length of the vector of its child's nodes, denoted $l_i = \|\mathbf{v_{i_c}}\|$. We say each hash value s_i represents the signature of each folder and l_i represents the number of Android SDK APIs used of its class file. Again, we still preserve the tree-based structure, that is qualified name for each folder p_i, given TPL. Finally, we obtain the tuple $\langle s_i, l_i, p_i, v_i \rangle$, for any node i.

(iii) For each signature s_i of each folder, we try to add up all the APPs associated with s_i. We then extend the tuple to $\langle s_i, l_i, p_i, v_i, a_i \rangle$, where a_i is the number of APPs associated with s_i. The value of a_i is largely fluctuated by different versions of APPs containing commonly-used code, wherein each APP is characterized by the qualified name and the signature s_i.

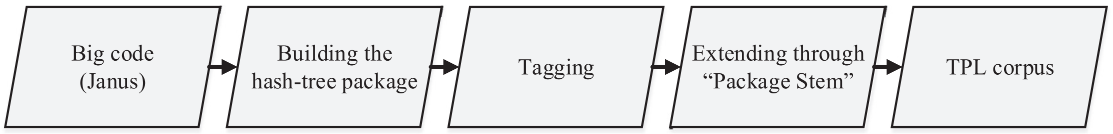

Big code (Janus)
Building the hash-tree package
Tagging
Extending through "Package Stem"
TPL corpus

Fig. 3 – Establishing TPL corpus.

3.2.2. *Tagging the TPL*

To accommodate more meta-information into the TPL corpus with minimum human efforts, we tag the TPL as follows:

(i) We apply breadth-first search (BFS) algorithm to probe the qualified name. The search depth is determined by the size of p_i for each node i. We prioritize the search by sorting a_i in descending order, which guarantees us to pass the most popular TPL first.

(ii) We then probe the "Package Stem" by moving forward until l_i is changed or the current node i has multiple child nodes. Note that lower-level features or signatures can be polluted by the mixture of TPLs. We de facto probe from the lower-level of the hierarchical tree to the "Package Stem".

(iii) We build a dependency graph for the "Package Stem" and use NetworkX (Hagberg et al., 2013) to identify cycles in the PkgDG, attempting to merge "Package Stem". That is, a TPL is a mixture of completely distinct "Package Stem".

(iv) We tag the TPL manually, with the probed "Package Stem". We store the current representation as $\langle s_i, p_i, v_i, d_i \rangle$ for a given TPL, where d_i is the semantic description and p_i is the ultimate "Package Stem" of the TPL.

(v) We further extend to find other tuples by only considering both p_i and d_i, regardless of s_i and v_i, then obtain a collection of tuples $\langle s_i, d_i, v_i \rangle$, where s_i is a signature associated with a specific version of a TPL. The derived representation of the TPL corpus serves as a benchmark dataset for detecting original or obfuscated TPLs directly. Note v_i is the feature for detecting shrunk or optimized TPLs.

To date, most polluted TPLs are sanitized by using cycle search. Shrunk and optimized TPLs are sanitized by feature set matching. We successfully create a one-to-one mapping between a hash value (or a feature set) and a specific TPL version.

3.3. *Identifying shrunk or optimized TPLs*

With the TPL corpus established in hand, we attempt to take a deep dive into how we use it for proposing our TPL detection, where no previous research considers the effect of shrunk or optimized TPLs in the wild. Recall that the shrinkage and optimization of ProGuard introduce the complexity of TPL detection. In order to detect shrunk or optimized TPLs, we first decouple an APP, and then use the stable generated feature to identify the calibrated TPL.

3.3.1. *Decoupling an APP*

Because an APK file processed by ProGuard can be mixed up with shrunk, optimized, obfuscated user space, the boundary between these modules become obscure. In order to identify a TPL within an APK file, we use module decoupling technique to divide the APK file into different modules. Inspired by PiggyApp (Zhou et al., 2013) and the follow-up work LibSift (Soh et al., 2016), we decouple APPs through the "Package Stem" probing.

Different from Section 3.2.2, we probe the "Package Stem" from the second level of the package tree. Fig. 4 shows an anecdotal example when we decouple an APP. In this case, a node represents a module and a directed edge represents the dependency between two modules. Direction of the edge helps us to find cycles between different modules (by the assumption that the TPL will not invoke each other). In the mean time, if there is a cycle in the siblings, we stop decoupling and take the current node as a "Package Stem" of this module. The weight assigned to each edge is the degree of dependencies between modules. The dependency between modules includes fields and method dependency. We present the module decoupling technique used in Algorithm 1 .

Algorithm 1: Decoupling an APP

```
Input: Android package structure and metadata
Input: Upper-threshold of PDG: $U$
Output: Modules and their dependencies: $M = \varnothing$
foreach node i visited by recursive DFS do
    $i_c \leftarrow$ child's nodes of node i;
    $w_{i_c} \leftarrow \sum PDG(i_c)$;
    if $w_{i_c} > U$ then
        $M \leftarrow M \cup i_c$;
    end
    else if $cycle(i_{c_i}, i_{c_j}) = True$ then
        $M \leftarrow M \cup i_c$;
    end
    else
        continue;
    end
end
foreach $(m_i, m_j), m_i \in M, m_j \in M$ do
    if $cycle(m_i, m_j) = True$ then
        $M \leftarrow M - m_i$;
        $M \leftarrow M - m_j$;
        $M \leftarrow M \cup m_{ij}$;
    end
end
```

3.3.2. *Feature generation*

Current research chooses to use parameter types, return types, constant strings, access modifiers, and instruction streams as features to detect TPL (Backes et al., 2016; Bichsel et al., 2016; Zhou et al., 2015). However, the optimization of ProGuard changes the layout of TPL drastically (see Fig. 3). This leads to the fact that the precise graph

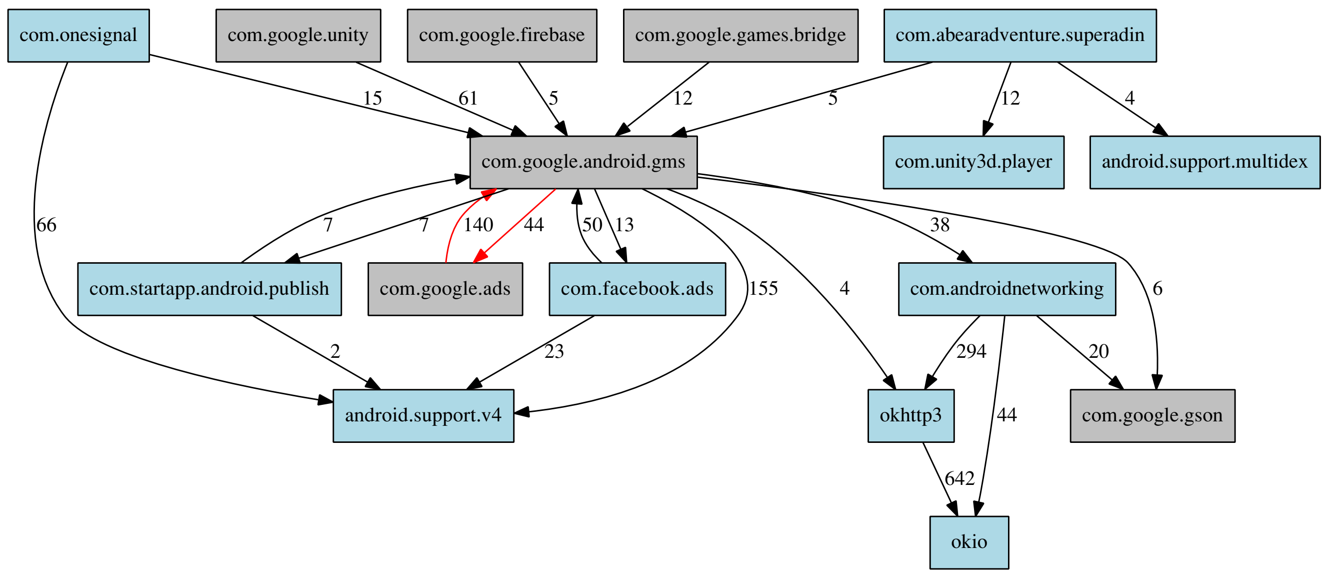
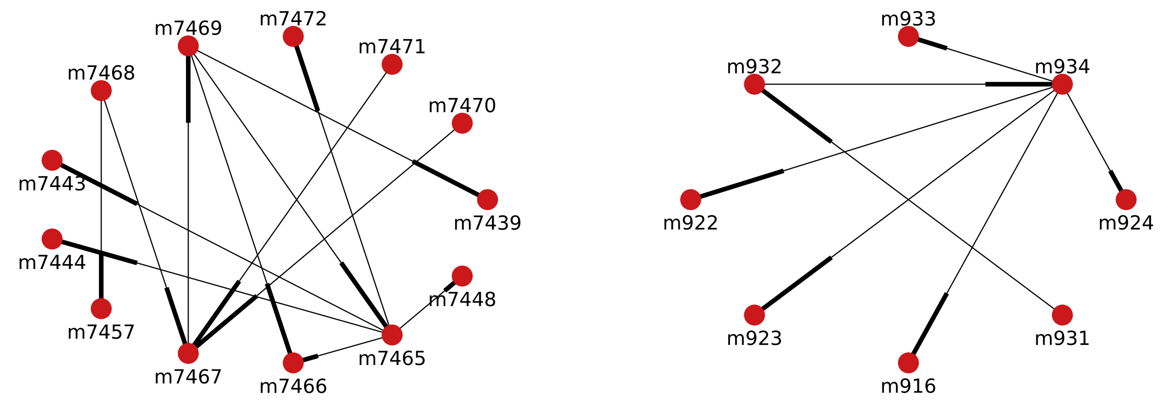


Fig. 4 – Building apackage dependency graph for an APP.



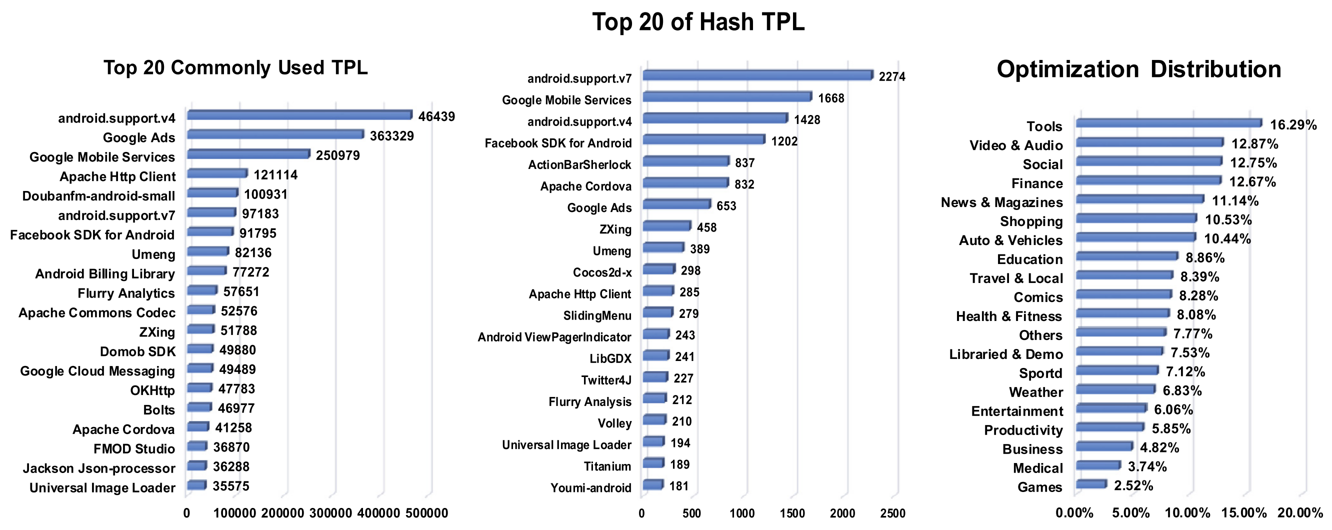
(a) The original call graph of "com/amazonaws/regions/ RegionMetadataParser" class

(b) The result of shrinkage and optimization with one iteration ($n = 1$ in Figure 1)

Fig. 5 – Inline optimization fails the call graph subgraph isomorphism to work on a class ("com/amazonaws/regions/RegionMetadataParser") of a TPL. Some nodes, such as "m7471", not invoked are removed by shrinkage. Since node "m7465" is invoked by only one time, ProGuard inlines this method to "m7469", which fails subgraph isomorphism.

detection does not scale well to optimized TPLs. As we see from the anecdotal example shown in Fig. 5, it indicates that inlining renders subgraph isomorphism to fail on class "com/amazonaws/regions/RegionMetadataParser". To continue with this APP, we show after one iteration of optimization by ProGuard, there are 465 classes left. By utilizing NetworkX, we find 34 classes are not belong to subgraph isomorphism, 12 classes are timeout (20 seconds) for subgraph isomorphism detection. Out of the 419 subgraph isomorphism classes, 150 classes contain less than 3 nodes that have less information, which makes subgraph isomorphism detection unreasonable. In addition, data dependency and the entire dependency graph for the APP are also affected by optimization.

Instead, we use the invocation of Android SDK APIs in TPLs. These Android SDK APIs are not directly processed by ProGuard and we find they are resistant to shrinkage and optimization. Here, we emphasize that, on one hand, the kind of invocation of an Android SDK API in the TPL may decay due to either method inlining from TPL code to user space or method removing through dead code elimination. On the other hand, the number of the same invocation may expand by method inlining from the callee to all callers within the same TPL. In optimization scenario, this principle does not scales to string, for a new string may appear by peephole optimization of ProGuard ("a" + "b" will be replaced with "ab"). Subgraph Isomorphism perhaps works for identifying shrunk TPLs but not for optimized TPLs. Because of these reasons, to facilitate scalability, we instead use various types of Android SDK APIs without considering the number of invocation for each Android SDK API, to identify shrunk or optimized TPLs.



(a) TPL popularity (b) The distribution of TPL versions (c) The distribution of optimization

Fig. 6 – The distributions of TPLs

4. Evaluation

In this section, we evaluate the performance of PanGuard and compare it with previous works. The APP 's metadata is distilled and stored in the HBase of our edge computing architecture.

4.1. *Bandwidth and computing requirement analysis*

In order to collect Android SDK API in an APP, we extract method with "public" modifier from "android.jar" file of version android-22. There are 32,203 Android SDK APIs are extracted. Bit vector representation for each TPL takes 4026 bytes. To go further on shrinking the data usage for TPL signature, we use sparse vector instead. For example, the Android SDK API used by different version of "Android Support V4" is ranging from 770 to 1805 with an average of 1107, 2214 bytes on average is needed for a specific "Android Support V4" TPL version. Compared to other tree-based method which maintain relation between node, our representation tremendous reduced the bandwidth for feature distribution from cloud to edge and the storage of edge.

In order to detect TPL, feature is converted to bit vector on edge node and the comparison is a bit vector operation. Compared to other tree-based method for TPL identification with complexity ($\exp((\log n)^{O(1)})$), our comparison is constant complexity, say O(1). This is acceptable in edge for TPL checking.

4.2. *The Distributions of TPLs*

4.2.1. *TPL popularity*

For feature v_i matching that is not practical on a large dataset, we do a measurement on the signature s_i. The distribution of TPLs in APPs is shown as Figure 6(a). We find that TPLs in our dataset are approximately 20% more than those in previous research (Backes et al., 2016; Li et al., 2016). Taking as example the TPL "Apache Http Client" as shown in Table 5, we find that

Table 5 – Qualified names of "Apache Http Client'.'

Path	# detection
Lorg/apache/http	87,544
Lorg/apache/commons/httpclient	25,344
La/a/a	3467
Lcom/flurry/org/apache/http	896
Lorg/a/b	689
Lorg/a/a	518
Lorg/apache	495
Others	2161

1 By using Reflector, class in "Lorg/apache/http" can be moved to "Lorg/apache/commons/httpclient". 2 "Lorg/a/b" are detected because they have the same s_i within the TPL corpus. APPs turn off shrinkage and optimization options to make the same s_i as the TPL corpus. 3 Hashing procedure ensures that the layout under each package is the same.

detecting TPLs by qualified name is not sufficient due to the use of Refactor in Java. Moreover, developers rarely process an APP by ProGuard using obfuscation alone. In fact, we show that ProGuard outperforms LibRadar (Ma et al., 2016) in the case— If two signatures s_i are the same for any node i, PanGuard not only ensures the same Android SDK API in these TPLs, but also preserves the structural information of its children, leading to a lower false positive rate.

4.2.2. *The distribution of raw TPL versions*

Before sanitization, each raw TPL version is shown in Fig. 6(b). Strikingly, we find 1,428 unique hashes appearing in the raw TPL "`Android.support.v4`", while there are only 61 versions of the officially-released one. We attribute this to the fact that shrinkage and optimization of ProGuard largely contribute to the difference of the version numbers.

4.2.3. *The distribution of sanitized TPL versions*

We try to merge raw TPL versions to sanitize ones produced by the approach shown in Section 3.3. Simply put, if a path of

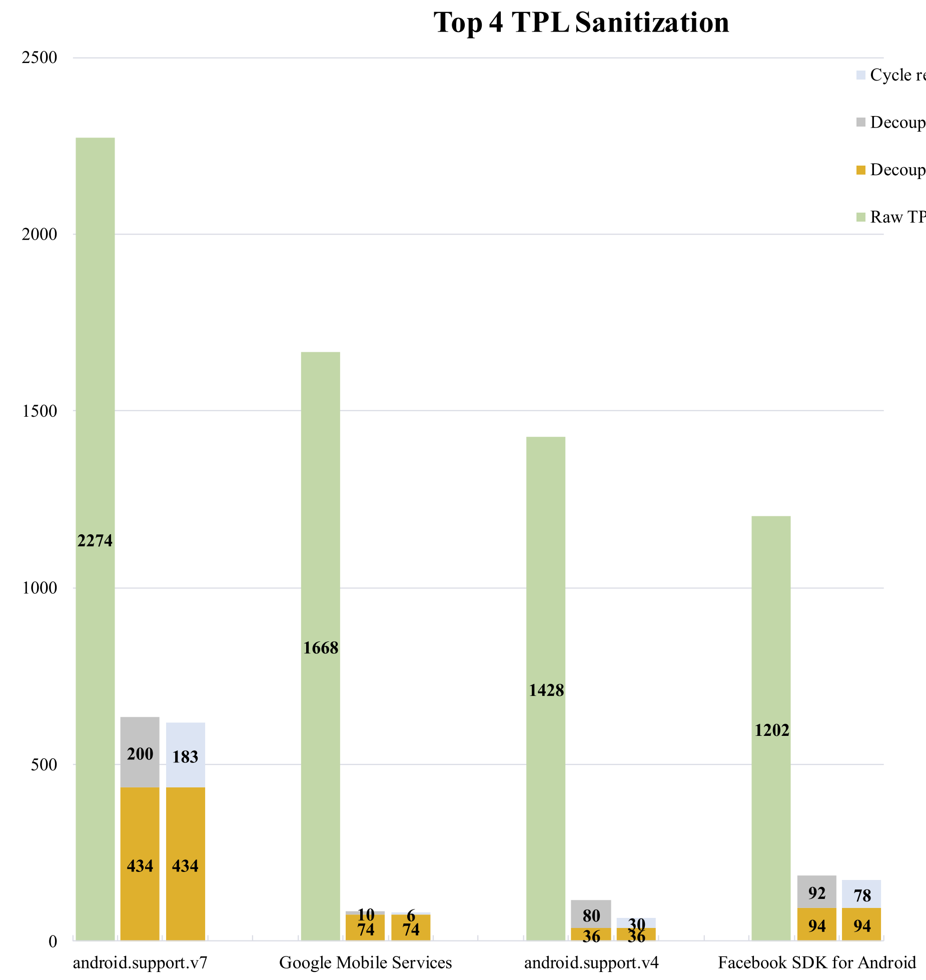


Fig. 7 – TPL sanitization process.

Table 6 – TPL edit distance.

Distance between TPL signature	Max Distance	Min Distance	Mean	Standard Deviation
Android.support.v7 VS. Google Mobile Service	1672	325	1009.99	206.70
Android.support.v7 VS. Android.support.v4	1853	340	1252.77	299.13
Android.support.v7 VS. Facebook SDK for Android	2278	318	1411.81	276.74
Google Mobile Service VS. Android.support.v4	2031	141	1148.52	393.34
Google Mobile Service VS. Facebook SDK for Android	1930	33	1030.26	273.94
Android.support.v4 VS. Facebook SDK for Android	2507	140	1475.85	371.92

a decoupled module of an APP matches the "Package Stem" of a TPL, we then search cycles that contain this module. In most cases, we regard that a TPL is polluted if the model is discovered in the cycle, and then the related versions of this TPL will be removed. This sanitization process is shown in Fig. 7. For the example of "`Android.support.v4`", we show that about 64% of the APPs containing a raw version of a TPL is decoupled successfully. Out of the successfully decoupled 80 samples, we find that approximately 63% of them are in a cycle, which indicates that this TPL is largely polluted by code injection. Finally, we try to measure *edit distance* between two instances with respect to different TPLs. As we see from Fig. 6, the edit distance between two TPLs is large enough, which indicates they are remarkably distinguishable.

Table 7 – ProGuard signature.

".*/a;- > .*" and ".*/b;- > .*"

4.2.4. *The actual proportion of TPLs distorted by* ProGuard

We find that 5,948,438 out of 7,602,323 (78.25%) APPs hit the following ProGuard signature (see Table 7).

However, this data is a combination of ProGuard, processed by TPL providers and APP developers. Since only the developer processing can distort the TPL, we adopt Duet's term (Hu et al., 2014) to separate the TPL processing into developer's post-processing and provider's post-processing. In

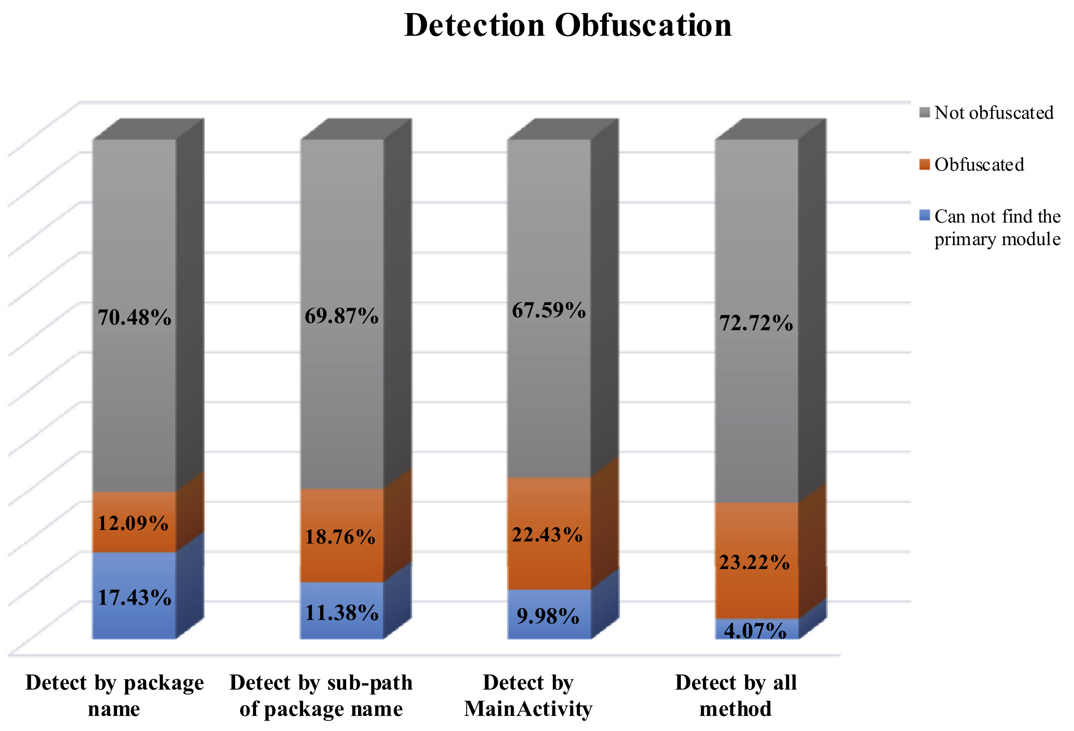


Fig. 8 – Provider's post-processing.

order to make sure the actual fraction of developer's post-processing, we use three methods to find developer's code (primary modules) of an APP. This measurement is performed on a collection of 4,795,627 APPs. We plot the results in Fig. 8. The first column of Fig. 8 shows the code we check under "package" defined in "AndroidManifest.xml", indicating that 15.36% APPs cannot be found any code under "package" approximately. The second column of Fig. 8 shows that we shorten the length of "package" to 2. The third column of Fig. 8 shows that path "MainActivity" resided are used to check the post-processing, indicating that using "MainActivity" as a clue to find primary modules is more reasonable. The rightmost column of Fig. 8 shows that approximately 23.22% APPs in our dataset are processed by ProGuard.

By observation, when using ProGuard to process an APP, developers always enable the obfuscation with shrinkage or optimization options. This will drastically break the integrity of TPLs. Therefore, efforts on shrinkage or optimization of TPLs problems cannot be emphasized more.

4.2.5. *The distribution of optimization*

By using the checking method defined in Section 2.1, we find that out of a collection of 7,723,699 APPs, 5,353,359 APPs contain the invocation defined by "Android Support Repository" and 442,605 APPs (8.26%) are optimized. Synthesized with data in the previous section, we find that approximately 36% APPs are processed by optimization enabled within ProGuard. The optimization of different categories is shown in Fig. 6(c). As we can see from Fig. 6(c), tool developers are more inclined to optimize their code.

Another measurement is performed on newly coming APPs for our Janus platform. 49,538 APPs out of 304,845 (16.25%) are optimized. It is very likely to be a trend for developers to optimize their code.

4.3. *Detecting vulnerable TPL*

This work is performed on our cloud service to help developer to exclude vulnerable TPL. In order to verify the capability of PanGuard, we apply our tool to a real-world TPL "AliPay SDK for Android". The "AliPay SDK for Android" released by *Alipay*, which is one of the biggest third-party mobile and online payment platforms in the world. Android developers integrate this TPL to provide online payment for customers. To evaluate its efficiency, our task is performed on a collection of 1,004,498 samples, most of which were collected before January 2016 by our *Janus* platform.

In the TPL corpus building stage, we collect 1031 signatures for this TPL. After the cycle search and feature set matching, we build a corpus with 199 versions of this TPL. In the cycle search process, two signatures of the collection are in a cycle of related TPLs. The instances integrated these polluted TPLs are *SHA* − 1: `9e1dab145cc524d0ad5e a934510b1247f81be1dc` and `b9ce54e8e4e3a21dbe85c 8dc8814000db419b84f`. After inspecting the code, we find that the APP injects code to redirect the payment to their own code, which leads to an integrity breakdown of this TPL. In the feature set matching, the version is enclosed in an instance *SHA* − 1: `01dd3693451b4c3447013297 bdda7005a2e6b32c` and say, its subset instance 102-hash *SHA* − 1: `0f64ca13e6851aaa776ffd351747c5a0 b32772b1`. We believe that all these versions are generated by ProGuard. Although we have no evidence on the exact official version of this TPL corpus, we find this version is a buggy one.

The sketch of the most-recent version of this TPL provides an interface for merchants to sign order information in their own servers, in such a way keeping private keys safely at the merchants' sides. However, after investigating this version of the TPL, we find that if a developer uses this version of "AliPay SDK for Android", he must embed his private key in the APP to ensure the integrity of the order message, which leads to private key leakage by reverse engineering. We performed a check on the randomly-selected samples of this version but with different signatures. As a result, we find that most of the APPs use this version of TPLs with their private keys embedded in their APPs.

With this version of the TPL corpus in hand, we go further to identify the shrunk or optimized TPLs. In this stage, we implement it by feature set matching. For example, the instances of *SHA* – 1: `4d5cede3d5ca53d1a6000fe5f5c4a5b396a78df2` and *SHA* – 1: `75591cd7ffc0c142b8880be73a2a5d389e9366ab` are drastically optimized by ProGuard, but we can still detect the buggy version in this APP.

Finally, we find that 13,578 APPs integrate this buggy version of the TPL and most of these APPs are likely to leak private keys in their code. Among the vulnerable APPs, over 20% of them are shrunk or optimized by ProGuard.

4.4. *Detecting virus who breaks the integrity of a TPL*

Since the TPL corpus has been built, it can be used to detect polluted TPLs in the detection stage. The TPL corpus is collected by cloud and distributed to the edge node, in the edge node, we compare the feature of a module with the candidate TPL corpus. If the feature of a given module is not one of all the versions in the TPL corpus, the module is considered to be polluted. Take the virus GhostClicker Micro (2017) (*SHA* – 1:`0a6583a741debc90498fb693eb56509f603fc404`) as example, after comparing with the closest TPL corpus, we find that the virus inserts 289 Android SDK APIs in "com.google.android.gms" and 461 Android SDK APIs in "com.facebook" package, which completely breaks the integrity of the TPL.

Additionally, as Fig. 7 shows, about 22% of the successfully decoupled TPLs are in a cycle. This is unreasonable for these 4 TPLs because the TPLs contained by this cycle is likely polluted. A quick checkout finds that frameworks, such as "Titanium" or "ActionBarSherlock", which inject their code into these TPLs, contribute about 50% of the cycles, and other code injection takes up the rest, which are suspicious. To continue with the example of GhostClicker, we show in Fig. 9, there are 3 cycles in the dependency graph. The module "com.google" and "`com.facebook.ads`" are mutually dependent. More specifically, the following method enclosed in "Android Mobile Service" contains a particular invoke instruction as follows, calling the method defined in the module "`com.facebook.ads`".

```
Lcom/google/android/gms/logs/fb;-><init>(Landroid/
content/Context;)V
```

```
Lcom/facebook/ads/InterstitialAd;-><init>(Landroid/content/
Context;Ljava/lang/String;)V
```

This is abnormal because the module "`com.google.gms`" actually does not depend on the module "`com.facebook.ads`". This illogical dependency constructs a cycle for the two modules, and the cycle can serve as a criterion to detect code injection to TPLs and purify the TPL corpus.

5. Discussion

As a leading mobile security company in China, our team is responsible for daily processing TPLs to real-time update the platform for mobile security. We cooperate with Alipay in identifying the integration of old versions of Alipay SDK in a customer service APP, which is a customized service for Alipay. To check the version of this TPL, the cooperator provides the interface for version query in Alipay SDK. The detection usually fails when a customer service APP is shrunk or optimized. Moreover, the buggy version of this TPL mentioned above does not provide interface for version query.

When providing security audit for our customers, we find that many customers leak credentials when using "Alibaba Cloud Object Storage Service (OSS)". Then we pay a close attention to the "Alibaba Cloud OSS" credential leakage problem. After browsing developer documentation of "Alibaba Cloud OSS", we notice that "Alibaba Cloud OSS" TPL provides a test interface that misleads developers to leak their credentials. An anecdotal survey on this interface invoking shows that about 25% developers leak credentials that use "Alibaba Cloud OSS" TPL (CNVD-2017-06366, 09774, 10187, 11666, 11811, etc.). To date, "Alibaba Cloud OSS" has stealthily removed description for this interface, but that simple mitigation is not satisfying. To our knowledge, although the new version of "Alibaba Cloud OSS" TPL provides developers with recommendations when using ProGuard to ensure less mutation on this version of TPL, the old version, "MBAAS_OSS_Android_1.0.0_" for example, of "Alibaba Cloud OSS" TPL does not have any instructions for processing their TPL. This makes previous work fails to detect credential leakage (the TPL and interface) when a developer integrates the old version of "Alibaba Cloud OSS" TPL and then uses ProGuard to process. Another situation is when helping virus analyzer to extract domain in a virus, TPL extraction can help analyzer focus on domain resided in developer scope, rendering the analyzer to perform a fast response for this virus. For a security audit, our customers also care about the vulnerabilities introduced by developers or TPL providers. Additionally, by using TPL identification work, we also provide security un-related service to help our customers to compose a loosely coupled APP. In summary, TPL identification is an important work for both TPL providers and customers in mobile security analysis.

5.1. *Limitations*

The nature of TPL identification is complex, for which there is no all-in-one solution. In practice, our PanGuard is limited in the following aspects:

- The threshold *U* is very hard to set. In the experiment, a concrete value of *U* can successfully decouple one TPL, but over-decoupled another TPL within a same APP and vice versa. The successful decoupling rate depends on the TPL developers.

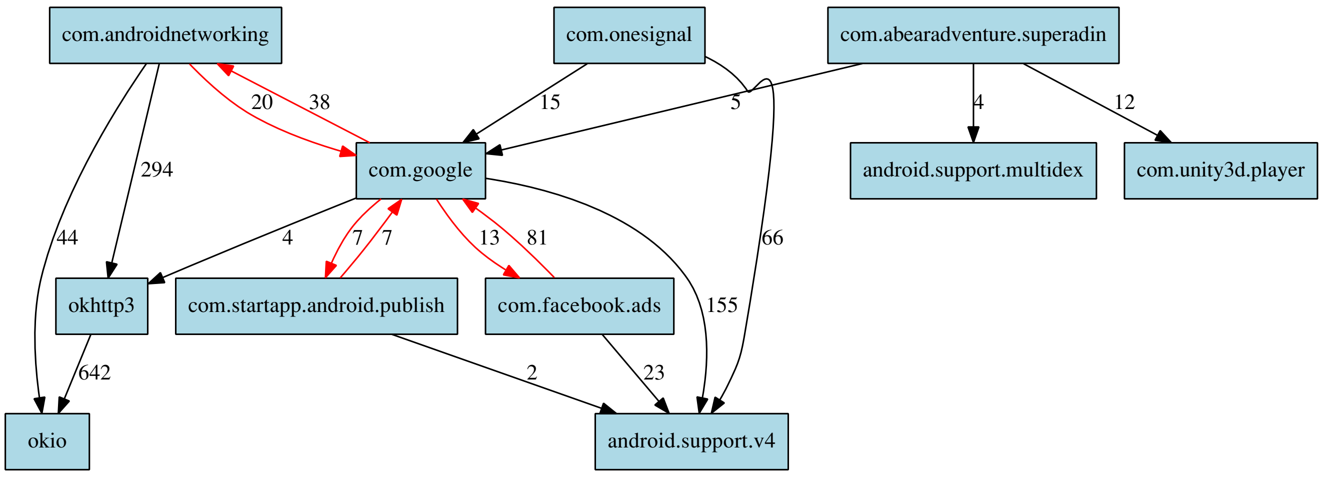


Fig. 9 – Cycles in GhostClicker's dependency graph.

- A TPL is distributed in completely distinct folders. For example, the most-recent version of "AliPay SDK for Android" needs two or more signatures for this TPL, which raises the complexity for representation and detection.
- A trivial modification on TPLs, usually taken security into consideration, does not affect the use of Android SDK APIs of TPLs, but does affect detection in establishing versions.
- In the practice of TPL detection, we find some APPs contain code in their root folders (e.g., Twitter, PayPal), which will distort our TPL corpus establishment and decoupling process.
- The biggest challenge is that it is hard to verify the results of the process of TPL corpus establishment. In some situations, even TPL providers cannot offer all versions of their TPLs.
- The decoupling process is a time-consuming process, and the cycle search is memory-consuming process, which does not scale well to a large dataset.

5.2. *Future Work*

- In the paper, we set the upper-threshold of PDG U related to the number of child's nodes, by the rule of thumb. In the future, we will try to search the optimal value of this variable.
- We will try to verify all cycles in the modules generated by decoupling and find more security threats to TPLs.
- TPL identification is the first step of our TPL security study. In practice, we notice that most of TPL providers have noticed security problems of their TPLs and made remedies for their TPLs. To ensure compatibility, TPL providers usually preserve unsafe interfaces for developers. But usually developers still keep using unsafe interfaces even after receiving the warnings given by TPL providers. In such a case, to identify which API is invoked is helpful for security study. This task becomes even harder if an APP is shrunk or optimized.

6. Related work

Although security study for android is growing by leaps and bounds (Chen et al., 2018), in this section, we only present the works which are related to our TPL detection.

Clone detection. Clone detection techniques are widely used in Android, which are also a doable way to detect TPLs. DROIDMOSS Zhou et al. (2012) calculates hash values for instructing sequences of a certain length, and figures out a fingerprint for the whole APP by combining these hashes. The fingerprint is compared pairwise to identify the clone in between. JUXTAPP (Hanna et al., 2012) uses k-grams opcode sequences and feature hashing as a signature for APPs, and identifies the clones existing in APP stores. DNADROID (Crussell et al., 2012) and ANDARWIN (Crussell et al., 2013) both create the *program dependency graph* as a signature for each method, and then DNADROID employs *subgraph isomorphism* to determine the clones, and ANDARWIN further extracts vectors from a graph to speed up the similarity computation.

Feature similarity computation. Li et al. (2016) employ a clustering algorithm with some empirical assumptions to obtain standalone packages, and then use these packages as signature to further detect TPLs. They propose a naming rule to judge whether the APP is obfuscated or not. LIBRADAR (Ma et al., 2016) only takes into account the Android APIs which are obfuscation-resilient for each package, and then performs multi-level clustering to identify TPLs. The selected Android APIs can effectively reduce the impact of obfuscation. Different with LIBRADAR, our approach also considers other prominent features such as *string* in case few, or even no, Android APIs are invoked in TPLs. LIBD (Li et al., 2017) extracts internal code dependencies as features and uses hashed features to identify TPLs. LIBSCOUT (Backes et al., 2016) leverages *class hierarchy analysis* to build *Merkle* trees with a fixed depth of three as a profile for each library. One matching algorithm is proposed to calculate the similarity with collected libraries. The pre-collected 800 distinct libraries of 9,623 versions form a tangible database that ensures an accurate detection result. Compared to their works, PANGUARD is not resilient to obfuscated APPs, but also to shrunk and optimized

APPs which takes up a large proportion 23.22% in Android APPs.

Machine learning-based detection. PEDAL (Liu et al., 2015) takes into account the code features in Android SDKs, and the features are used to train a classifier to identify TPLs. AdDetect (Narayanan et al., 2014) and LibSift (Soh et al., 2016) propose using hierarchical packages and package dependency information as features and building a classifier to classify TPLs. In addition, they can also separate the primary code (i.e., the main functionality of APP) from TPLs. Compared to their works, we only take code similarity computation to achieve effectiveness and accuracy simultaneously.

Edge computing. To address the concerns of response time requirement, battery life constraint, bandwidth cost saving, as well as data safety and privacy, industry and academia have recently proposed edge computing. Luo et al. (2017) enables cloud to automatically offload computations to the edge servers. Chen et al. (2016b) study a multi-user computation offloading problem in a multi-channel wireless interference environment. Zhang et al. (2016) develop an energy-efficient computation offloading mechanism for mobile edge computing in 5G heterogeneous networks. Sardellitti et al. (2015) propose to reduce the energy consumption by jointly considering the radio resources and computational resources. Mao et al. (2016) exploit renewable energy to help reduce the energy consumption of mobile devices. Luo et al. (2017) design an energy-efficient autonomic offloading scheme that can automatically offload computational tasks to edge servers.

7. Conclusion

In this paper, we propose PanGuard, a novel and automated approach to detect TPLs in Android APPs. PanGuard leverages both structural and content information as a feature, and performs a feature set matching algorithm to identify TPLs. Invariants are studied and determined to overcome the disturbance caused by obfuscation, shrinkage, and optimization to code. PanGuard successfully identifies over 10 security issues in TPLs. PanGuard has been already deployed on our platform Janus, which has promoted the efficiency of Janus for malware detection and vulnerability analysis.

REFERENCES

Backes M, Bugiel S, Derr E. Reliable third-party library detection in android and its security applications. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. ACM; 2016. p. 356–67.

Bichsel B, Raychev V, Tsankov P, Vechev M. Statistical deobfuscation of android applications. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. ACM; 2016. p. 343–55.

Chen K, Wang X, Chen Y, Wang P, Lee Y, Wang X, Ma B, Wang A, Zhang Y, Zou W. Following devil's footprints: cross-platform analysis of potentially harmful libraries on android and IOS. In: Proceedings of the IEEE symposium on security and privacy (SP), 2016. IEEE; 2016a. p. 357–76.

Chen X, Jiao L, Li W, Fu X. Efficient multi-user computation offloading for mobile-edge cloud computing. IEEE/ACM Trans Netw 2016b;24(5):2795–808.

Chen Y, Li T, Zhang R, Zhang Y, Hedgpeth T. Eyetell: video-assisted touchscreen keystroke inference from eye movements. In: Proceedings of the EyeTell: video-assisted touchscreen keystroke inference from eye movements. IEEE; 2018. p. 0.

Crussell J, Gibler C, Chen H. Attack of the clones: Detecting cloned applications on android markets. In: Proceedings of the European symposium on research in computer security. Springer; 2012. p. 37–54.

Crussell J, Gibler C, Chen H. Andarwin: Scalable detection of semantically similar android applications. In: Proceedings of the European symposium on research in computer security. Springer; 2013. p. 182–99.

Dogtiev A.. App Download and Usage Statistics 2017. http://www.businessofapps.com/data/app-statistics/; 2017. [Accessed 18 Octorber, 2017].

Fan M, Liu J, Wang W, Li H, Tian Z, Liu T. Dapasa: detecting android piggybacked apps through sensitive subgraph analysis. IEEE Trans Inf Forensics Secur 2017;12(8):1772–85.

Google. Shrink your code and resources. https://www.developer.android.com/studio/build/shrink-code.html; 2017. [Accessed 3 August, 2017].

GuardSquare. Proguard manual. https://www.guardsquare.com/en/proguard/manual/usage; 2017. [Accessed 27 August, 2017].

Hagberg A., Schult D., Swart P., Conway D., Séguin-Charbonneau L., Ellison C., Edwards B., Torrents J.. Networkx. high productivity software for complex networks. Webová strá nka https://networkx lanl gov/wiki 2013.

Hanna S, Huang L, Wu E, Li S, Chen C, Song D. Juxtapp: a scalable system for detecting code reuse among android applications. In: Proceedings of the international conference on detection of intrusions and malware, and vulnerability assessment. Springer; 2012. p. 62–81.

Hu W, Octeau D, McDaniel PD, Liu P. Duet: library integrity verification for android applications. In: Proceedings of the 2014 ACM conference on security and privacy in wireless & mobile networks. ACM; 2014. p. 141–52.

Jiang L, Misherghi G, Su Z, Glondu S. Deckard: Scalable and accurate tree-based detection of code clones. In: Proceedings of the 29th international conference on Software Engineering. IEEE Computer Society; 2007. p. 96–105.

Li L, Bissyandé TF, Klein J, Le Traon Y. An investigation into the use of common libraries in android apps, 1. IEEE; 2016. p. 403–14.

Li M, Wang W, Wang P, Wang S, Wu D, Liu J, Xue R, Huo W. Libd: Scalable and precise third-party library detection in android markets. In: Proceedings of the 39th international conference on software engineering. IEEE Press; 2017. p. 335–46.

Liu B, Liu B, Jin H, Govindan R. Efficient privilege de-escalation for ad libraries in mobile apps. In: Proceedings of the 13th annual international conference on mobile systems, applications, and services. ACM; 2015. p. 89–103.

Liu B, Nath S, Govindan R, Liu J. DECAF: detecting and Characterizing Ad Fraud in Mobile Apps. In: Proceedings of the 11th USENIX conference on networked systems design and implementation; 2014. p. 57–70.

Luo C, Salinas S, Li M, Li P. Energy-efficient autonomic offloading in mobile edge computing. In: Proceedings of the 2017 IEEE 15th International Conference on Dependable, Autonomic and Secure Computing, 15th International Conference on Pervasive Intelligence and Computing, 3rd international Conference on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech); 2017. p. 581–8. doi:10.1109/DASC-PICom-DataCom-CyberSciTec.2017.104.

Ma Z, Wang H, Guo Y, Chen X. Libradar: fast and accurate detection of third-party libraries in android apps. In: Proceedings of the 38th international conference on software engineering companion. ACM; 2016. p. 653–6.

Mao Y, Zhang J, Letaief KB. Dynamic computation offloading for mobile-edge computing with energy harvesting devices. IEEE J Sel Areas Commun 2016;34(12):3590–605.

Meng W, Ding R, Chung SP, Han S, Lee W. The price of free: privacy leakage in personalized mobile in-apps ads. Proceedings of the NDSS, 2016.

Micro T.. Ghostclicker adware is a phantomlike android click fraud. http://www.blog.trendmicro.com/trendlabs-security-intelligence/ghostclicker-adware-is-a-phantomlike-android-click-fraud/; 2017. [Accessed 22 August, 2017].

Narayanan A, Chen L, Chan CK. Addetect: Automated detection of android ad libraries using semantic analysis. In: Proceedings of the ieee ninth international conference on intelligent sensors, sensor networks and information processing (ISSNIP), 2014. IEEE; 2014. p. 1–6.

Pangu T.. Akana: Interactive analysis environment. http://www.auditdroid.mobiseclab.org:8080/android/; 2017a. [Accessed 3 August, 2017].

Pangu T.. Wormable browser. http://www.blog.pangu.io/wormable-browser/; 2017b. [Accessed 3 August, 2017].

Rastogi V, Shao R, Chen Y, Pan X, Zou S, Riley R. Are these ads safe: detecting hidden attacks through the mobile app-web interfaces. Proceedings of the NDSS, 2016.

Roy CK, Cordy JR. A survey on software clone detection research. Queenïs Sch Comput TR 2007;541(115):64–8.

Roy CK, Cordy JR, Koschke R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Sci Comput Program 2009;74(7):470–95.

Sardellitti S, Scutari G, Barbarossa S. Joint optimization of radio and computational resources for multicell mobile-edge computing. IEEE Trans Signal Inf Processr Netw 2015;1(2):89–103.

Soh C, Tan HBK, Arnatovich YL, Narayanan A, Wang L. Libsift: automated detection of third-party libraries in android applications. In: Proceedings of the 23rd Asia-Pacific software engineering conference (APSEC), 2016. IEEE; 2016. p. 41–8.

Ucloud Co. L.. Ucloud. https://www.ucloud.cn/; 2017. [Accessed 3 August, 2017].

Wang H, Guo Y, Ma Z, Chen X. Wukong: a scalable and accurate two-phase approach to android app clone detection. In: Proceedings of the 2015 international symposium on software testing and analysis. ACM; 2015. p. 71–82.

Wikipedia. Fully qualified name. https://www.en.wikipedia.org/wiki/Fully_qualified_name; 2017. [Accessed 3 August, 2017].

Zhang K, Mao Y, Leng S, Zhao Q, Li L, Peng X, Pan L, Maharjan S, Zhang Y. Energy-efficient offloading for mobile edge computing in 5g heterogeneous networks. IEEE Access 2016;4:5896–907.

Zhou W, Zhou Y, Grace M, Jiang X, Zou S. Fast, scalable detection of piggybacked mobile applications. In: Proceedings of the third ACM conference on Data and application security and privacy. ACM; 2013. p. 185–96.

Zhou W, Zhou Y, Jiang X, Ning P. Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of the second ACM conference on data and application security and privacy. ACM; 2012. p. 317–26.

Zhou Y, Wu L, Wang Z, Jiang X. Harvesting developer credentials in android apps. In: Proceedings of the 8th ACM conference on security & privacy in wireless and mobile networks. ACM; 2015. p. 23.

Zhushou Tang is pursuing his Ph.D. degree at the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China, advised by Professor Haojin Zhu. He is also co-founder of Pwnzen Infotech Inc. He is focusing primarily on areas of smartphone security, Android / iOS malware, vulnerability and program analysis.

Minhui Xue is currently a Research Fellow with Optus Macquarie University Cyber Security Hub at Sydney, Australia. He received his PhD degree at the School of Computer Science and Software Engineering of East China Normal University in June 2018. His current research interests are security and privacy, deep learning security and testing, and Internet measurement. He is the recipient of the ACM SIGSOFT distinguished paper award and IEEE best paper award, and his work has been featured in the mainstream press, including The New York Times, Science Daily, PR Newswire, and Yahoo.

Guozhu Meng received the bachelor's and master's degrees from the School of Computer Science and Technology from Tianjin University, China, and the Ph.D. degree from Nanyang Technological University (NTU), Singapore in 2017. Before his study of Ph.D., he was with Temasek laboratory, National University of Singapore (NUS), for one year as an Associate Scientist in 2012. Since 2017, he continued with NTU as a research fellow. His research interests include mobile security, software engineering, program analysis, and vulnerability detection.

Chengguo Ying is a security expert of Pwnzen Infotech Inc. He is focusing primarily on areas of cloud computing, smartphone security, Android malware, vulnerability and program analysis. He received bachelor degree from Shandong University, China, in July 2012.

Yugeng Liu is pursuing his bachelor degree at the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. He is focusing on IoT Security analysis, and Android application analysis. He is interested in IoT Security and Android Security, aiming at combining them together.

Jianan He received the master's degrees from the School of Wuhan University of Technology, China. He is now serving at China Electronics Technology Group Corporation No.32. He is major in mobile application testing, source code auditing.

Haojin Zhu is currently a Professor with Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. He received his B.Sc. degree (2002) from Wuhan University (China), his M.Sc.(2005) degree from Shanghai Jiao Tong University (China), both in computer science and the Ph.D. in Electrical and Computer Engineering from the University of Waterloo (Canada), in 2009. His current research interests include network security and data privacy. He serves as the Associate/Guest Editor of IEEE Internet of Things Journal, IEEE Wireless Communications, IEEE Network, and Peer-to-Peer Networking and Applications.

Yang Liu received the bachelorâ;;s degree in computing and the Ph.D. degree from the National University of Singapore (NUS), in 2005 and 2010, respectively. He continued with his postdoctoral work with NUS. Since 2012, he has been with Nanyang Technological University, and now work as an Associate Professor. His research focuses on software engineering, formal methods, and security. In particular, he specializes in software verification using model checking techniques. This work led to the development of a state-of-the art model checker, Process Analysis Toolkit.