APKLancet: Tumor Payload Diagnosis and Purification for Android Applications

Wenbo Yang*
Shanghai Jiao Tong University
800 Dongchuan Road
Shanghai, China
talentyang@hotmail.com

Juanru Li*
Shanghai Jiao Tong University
800 Dongchuan Road
Shanghai, China
jarod@sjtu.edu.cn

Yuanyuan Zhang*
Shanghai Jiao Tong University
800 Dongchuan Road
Shanghai, China
yyjess@sjtu.edu.cn

Yong Li
Shanghai Jiao Tong University
800 Dongchuan Road
Shanghai, China
1120339057@sjtu.edu.cn

Junliang Shu
Shanghai Jiao Tong University
800 Dongchuan Road
Shanghai, China
s.junliang@gmail.com

Dawu Gu†
Shanghai Jiao Tong University
800 Dongchuan Road
Shanghai, China
dwgu@sjtu.edu.cn

ABSTRACT

A huge number of Android applications are bundled with relatively independent modules either during the development or by intentionally repackaging. Undesirable behaviors such as stealthily acquiring and distributing user's private information are frequently discovered in some bundled third-party modules, i.e., advertising libraries or malicious code (we call the module *tumor payload* in this work), which sabotage the integrity of the original app and lie as a threat to both the security of mobile system and the user's privacy.

In this paper, we discuss how to purify an Android APK by resecting the tumor payload. Our work is based on two observations: 1) the tumor payload has its own characteristics, so it could be spotted through program analysis, and 2) the tumor payload is a relatively independent module so it can be resected without affecting the original app's function. We propose APKLancet, an automatic Android application diagnosis and purification system, to detect and resect the tumor payload. Relying on features extracting from ad libraries, analytics plugins and an approximately 8,000 malware samples, APKLancet is capable of diagnosing an APK and discovering unwelcome code fragment. Then it makes use of the code fragment as index to employ fine-grained program analysis and detaches the entire tumor payload. More precisely, it conducts an automatic app patching process to preserve the original normal functions while resecting tumor payload. We test APKLancet by the Android apps bundled with representative tumor payloads from online sandbox system. The result shows that the purification process is feasible to resect tumor payload and repair the apps. Moreover, all of the above do not require any Android system modification, and the purified app does not introduce any performance latency.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: (e.g., viruses, worms, Trojan horses);

General Terms

Security

Keywords

Program analysis; Third-party libraries; Malicious code; Android Security

1. INTRODUCTION

Smartphones such as Android phones are beneficial supplements to stationary computing: online payment, web surfing, email processing, etc. Being a functional cellular phone, smartphone maintains the contacts lists, geographic location, SMS and more, which are directly related to the user herself. As a result, with the booming of the market share of these mobile devices, Android attracts great attention of malicious activities that are rapidly increasing. Android applications are discovered gathering data such as GPS location, device identifiers, and even user's identity without proper notice or authorization from the end user. The unwanted behaviors, however, are usually introduced by the bundled third-party libraries or injected malicious code. These kinds of potentially undesirable third-party code, as we call *tumor payload* in this paper, introduce new security threat to benign apps and expose the users to privacy leakage or system compromising.

Typical tumor payload includes not only malicious code but also some advertising/analytics libraries. Most of the tumor payload choose to bundle with the popular apps which would bring in more profit for the publisher. The *Brightest*

*All three authors contributed equally to this work.

†Corresponding author.

ASIA CCS'14, June 4–6, 2014, Kyoto, Japan.

http://dx.doi.org/10.1145/2590296.2590314.

Flashlight Free app for Android, which has been downloaded between 50 to 100 million times, has been secretly sending location and device data to advertisers[5]. For third-party ad libraries or analytics plugins that seem to be benign, they may also introduce potential security breaches. For instance, ad libraries have been observed the behavior of downloading code over HTTP and dynamically loading and executing at runtime without checking the integrity[3]. Attacker may hijack the network and replace the executable with malicious code.

Due to the fact that Android apps are vulnerable to repackaging attack, tumor payload bundling becomes even more popular. Attackers could simply modify the apps using sophisticated decompilation tools to inject extra code or change the user interface through tweaking resource files. According to Zhou and Jiang's study[24], about 86.0% of the malwares they analyzed are repackaged versions of legitimate apps with malicious payloads. BitDefender's survey on Google Play[1] shows that more than 4,000 apps out of about 420,000 apps are plagiarized or simply re-engineered from other app developers, adding new advertising SDK to original apps for profit. In third-party application market, the situation is even worse. The experiment in[23] displays the fact that 5% to 13% of apps hosted on these studied marketplaces are repackaged. Since bundling tumor payload with an app is becoming a common phenomenon in Android ecosystem, tackling the tumor payload is not only the issue of protecting the secure execution environment of Android app, or of constraining the undesirable behavior, but also of protecting the ecosystem of the Android OS.

From the end users point of view, not only do they want to detect but also to forbid the imposed function (mostly from the tumor payload) while remaining the expected app function. Several studies on detecting the tumor payload have been proposed at module level pairwise comparison and large-scale analysis[8, 9, 14, 22]. The issue here is how to repress the undesirable behaviors and restore a purified app to its normal status.

A coarse-grained approach is to forbid the unnecessary permissions that tumor payload has declared. Android OS grants and controls the permissions to an entire APK, so the injected tumor payload shares the same privilege that the original APK has declared. Avoiding declaring unnecessary permission could somehow protect the apps from permission abuse by malicious activities, but notice that many apps provide their own function based on privileged permissions, and the injected code itself could share those permissions to conduct malicious activities, so this method is not recommended.

There also exist several fine-grained access control approaches[15, 10, 20] by rewriting the APK and inserting instrumentation routines to monitor the API invocation and dynamically applying security countermeasures when detecting threats. We should notice that these countermeasures do not account for the fact that APK is composed of modules of equal status but treated as an entire package. So it still goes to the end user's judgment call on granting permission to the "benign" or "malicious" functions in the APK. This is a difficult task even for the savvy user with the help of advanced information flow analysis, not to mention the non-savvy users who have little knowledge on the mechanism behind the veil. An alternative approach suggests to isolate the unwanted code (typically, the advertising libraries) and execute it in another process with limited privilege[21, 18, 17]. The advantage is to ensure the function of legally inserted third-party advertising libraries to bring profit to developers. The problem is that it commonly requires Android system modification and thus is difficult to implement.

We found that in most cases tumor payload is simply inserted as an independent module (otherwise the decompilation and re-compilation process may fail and it will be hard to be deployed in large scale) in APK files. When executed, this module runs in a separated work flow and is loosely relevant to the original work flow. Moreover, the styles of integration of the tumor payload are limited and usually the integrating procedure is reversible. Thus, it is able to purify an app through properly analyzing the way of integration and precisely locating and resecting the tumor payload.

In this paper we propose *APKLancet*, a tumor payload diagnosis and purification system for Android app. APKLancet is an APK rewriting system focusing on resecting tumor payload (the resecting process is called *purification* in this work) and further repairing the purified app after the resecting. It diagnoses an APK to spot potential tumor code with tumor payload feature database. Then it partitions the entire tumor payload with program analysis. Rather than directly cutting the tumor code off and leaves the normal function unchanged, APKLancet will automatically detect the integration style of the tumor payload and correspondingly patches the APK file to prove that it can work properly after the purification. Finally, APKLancet conducts a verification process to assure that the purification does resect the unwanted behavior, and the operated app is able to work properly.

The contributions of this work include:

- We conduct a systematic study on the characteristics of different kinds of tumor payload as ad libraries, analytics plugins and malicious code. This serves as a first step towards resecting the tumor payload. Based on this study, we summarize typical integration styles of the tumor payload. We propose an effective purification and restoration process that adopts different strategies to trim the tumor payload and sew the APK back.

- We build a tumor payload feature database using the knowledge of ad libraries, analytics plugins and an approximately 8,000 malware samples to help finding suspicious code fragment in the app. The diagnosed suspicious code fragment is then used as index to partition the entire tumor payload from the app using code and data dependency analysis.

- The proposed APK purification approach directly operates on apps and does not need to modify the system. Compared with other access control schemes, our purification approach operated by APKLancet does not bring any extra runtime overhead to either the app or the system.

- We evaluated the effect of resecting a variety of representative tumor payloads such as malicious code *Geinimi* and ad library *Wooboo*. Our work shows that tumor payload is able to be split. Based on this fact, APKLancet provides a novel way of access control at the code level.

2. PRELIMINARIES

2.1 Tumor Payload

We define the term of tumor payload as the code that is loosely linked to the primary function of the host app and performs undesirable behaviors (e.g., collecting information without proper notice or authorization from the end user). Malicious code, ad libraries and analytics plugins that contain privacy-violated function (although they may bring profit to developer or help collecting information) can all be regarded as tumor payload.

2.1.1 Malicious Code

Among all kinds of potentially unwanted code, malicious code is perhaps the most dangerous one. Malicious code refers to the code that intentionally conducts undesirable behaviors without approval. On Android OS, malicious code has evolved from simple functions such as sending SMS without the authorization to sophisticated functions such as getting root privileges by exploits, receiving and executing commands from remote server. More works[12, 19] have described the details on such threats caused by malicious code. One example is the *Geinimi* malware, the first sophisticated malware for Android found in the wild in 2010. Many legitimate apps are infected by being repackaged and the malicious code that includes a backdoor-like functionality. Following Geinimi, the repackaging-based infections were soon arising by other variants such as the Trojan *ADRD*. Researchers found that 86.0% repackaged apps have included malicious payloads after analyzing more than 1200 Android malware samples in 49 different malware families[24].

2.1.2 Ad Library

Ad library is the most common third-party code among Android apps. According to [2], about 50% of the apps contain at least one ad library. Although users dislike ads popping in their apps, developers can gain profit from it (by user's clicking the ads) so it is not reasonable to classify the ad library as malicious code. However, if an app without advertising is modified by injecting ad library for profit, the intellectual property rights of the original author are violated, which harms the whole Android ecosystem. In this situation, the injected library is obviously considered to be unwanted code by both the end users and the developers.

2.1.3 Analytics Plugin

Being different from the ad library, analytics plugin is to help developer collect user engagement data from their applications and make decision according to huge amount of collected information. For instance, the Google Analytics SDK for Android can help Android developers to collect data such as the number of active users, location of the users who use the application, etc. Analytics plugin generally runs in background and does not intervene in any operation at the front-end. Same as ad lib, analytics plugin should not be defined as malicious code. The potential risk of analytic plugin is that it may collect data related to user's privacy.

2.2 Observations

Tumor payload has obvious characteristics. From the viewpoint of program analysis, the tumor payload is a module that is loosely linked to the primary function of the host app and can be effectively partitioned[22]. From the perspective of pattern matching, typical tumor payload often contains special meta-data information declared in the xml file of the APK. And from the viewpoint of behavior monitoring, tumor payload generally involves privileged permission requirement and sensitive API invoking.

Another observation is that although tumor payloads may be various, they are basically using the same repackaging techniques, and most repackaging processes are reversible. The typical repackaging process involves unpacking the original APK to extract bytecode part and resource part, decompiling bytecode, modifying existing code (injecting new code) and resource files, and repacking the modified contents into a new package. Most of the repackaging procedures do not make too much modification to the inside of the original code but only add extra functional modules due to the applicability and dissemination. The function of the original APK is often preserved, and the injected code is normally a third-party module that is well encapsulated. Therefore, it is feasible to split the tumor payload from the app and purify the host app because the injection is detachable.

3. APK PURIFICATION

APK purification is the process of resecting tumor code from an app. The goal is to resect the undesirable behaviors brought by the tumor code while preserving the original function. It relies on the observation that tumor code is not tightly interweaving with the major function of the APK. To this end, we propose **APKLancet**, a tumor code excision system to conduct the app purification process. APKLancet's work flow consists of four stages(Fig 1). At the first stage APKLancet mainly relies on existing tumor code features to diagnose the APK file and filter out possible code fragment related to tumor payload. At the second stage the filtered suspicious code fragment is used as index to partition the entire tumor payload from the benign code, the partitioning process is conducted using program analysis. The third stage is tumor code resection. The main purpose of this stage is to resect the tumor payload, and patch the benign code of the APK to prove the correctness of the normal control flow. Final stage, the purified APK is verified and the effect of tumor payload resection is measured.

3.1 The APKLancet System

APKLancet fulfils tumor payload excision task. The APKLancet system contains three main components: *APK analyzer*, *APK rewriter* and *APK verifier*.

3.1.1 APK Analyzer

The APK analyzer of APKLancet is used to analyze APK file comprehensively. The implementation is based on Androguard[11]. The APK analyzer parses manifest file and resource files to acquire declaration information of an app. Relying on a feature database built from existing knowledge of tumor payload, it then locates suspicious code fragment and partitions the entire tumor code payload according to the results from smali code analysis. Moreover, it analyzes the type of tumor payload and conducts a patching process to prove the purified APK work properly.

3.1.2 APK Rewriter

The APK rewriter is responsible for unpacking and decompiling APK file, and repacking the purified file into a new app. It is mainly based on APKtool[4]. The APK

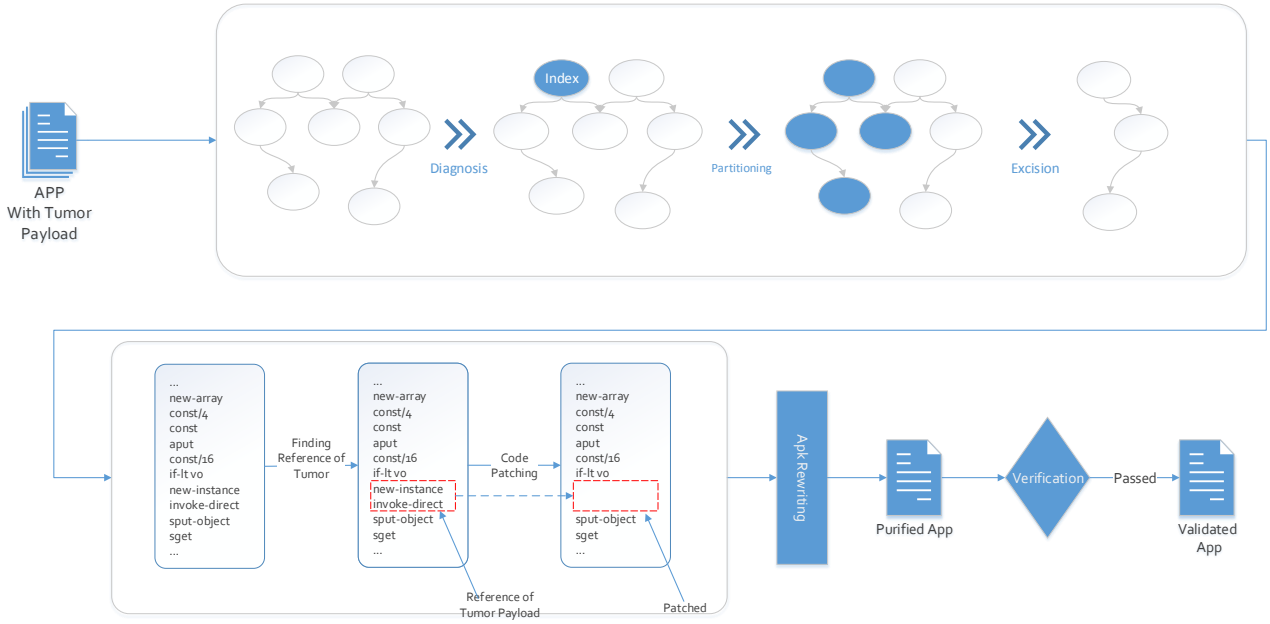

Figure 1: APK purification process

rewriter first unpacks the target APK file and decompiles the executable part into *smali* code, decoding manifest file and related resource files for the APK analyzer. Then, it receives the analyzing results from the APK analyzer, resects the tumor payload and repairs the "incision" in remained benign part. Finally, it reassembles the purified files into a new app.

3.1.3 APK Verifier

The APK verifier is mainly an automatic Android logging system to record and analyze app's execution status. It validates the purified app through dynamically recording its log output and corresponding events such as network access, and captures any possible exception that is introduced.

3.2 APK Diagnosis

To detect suspicious tumor payload, APKLancet conducts an approach that relies on existing knowledge database rather than analyzing the characteristics. The reason is that resecting tumor code from an app is aggressive and risky compared with solely detecting it. Any improper code resection will damage the benign function. What's worse, the effect of code resection may not be detected immediately but afterwards. Therefore, the strategy APKLancet adopts should be extremely conservative to prove that the diagnosis will not suffer from false positive.

To meet this requirement, APKLancet diagnoses an app using the existing knowledge of tumor payload. It first builds a tumor payload feature database summarized from known malicious code and popular third-party libraries. For malicious code, we have collected more than 8000 malware samples with 184 malware families from an automatic Android program analysis sandbox[6]. To build the database, we first randomly choose one app or two from each malware family and extract the feature. Then we use the knowledge to guide APKLancet to detect malicious payload in other malwares. For third-party libraries, we defined the tumor libraries after investigating popular ad libraries and analytics plugins since these libraries are more likely to be injected into benign apps for the profit. However, considering it may cause the result that legitimate apps bundled with tumor third-party libraries will also be the target of our system, we leave the issue discussed in Section 5.

The content of the database is the representative code fragment, as we called *index class*, of those tumor payloads. In the following, we discuss the details about extracting index class from malicious code and tumor libraries.

- *Malicious Code.* An index class of malware is responsible for certain events or functions connected to the malicious behaviors of the tumor payload. The advantage of building the feature database according to index classes instead of the whole payload is that it is common to malware that generating new variants. Even the tumor payload may be various, the index classes of one malware family are generally unchanged. Thus scanning these kinds of classes in tumor payload is essential for extracting feature.

  Since malicious code injecting aims to affect as much apps as possible, tumor payload of malware usually registers its own class as an entry point for convenience to assure it could be executed in most cases, which can avoid manual analysis to various app victims. Unlike application on traditional commodity computer platform, an Android application can have multiple entry points. APKLancet focuses on five types of classes (including inherited types) that can be registered as entry point. The first four are the basic components of Android: *Activity*, *Service*, *Broadcast Receiver* and *Content Provider*. The final class type, *android.app.Applic-*

ation, is the appointed class type of an app's main entry point and also considered in feature extracting. APKLancet only needs to scan these classes in a tumor payload and adds them into the tumor code feature database.

- *Third-party Library.* Unlike the malicious code, tumor third-party libraries such as popular ad libraries and analytic plugins are always well-documented, developer-friendly and less irregular. As a result, they usually offer a set of unified interfaces to developers for ease of use. It is not difficult for us to construct the tumor code feature database by selecting some of the most typical classes in the single library as the index classes(i.e., *AdView* in *AdMob*).

After constructing the feature database, APKLancet checks every class of an app that is one of the five entry point classes or representative classes of third-party libraries to find out the clue of tumor code. Different from signature-based detection of the malicious code that is frequently used in Antivirus, APKLancet compares the content of a class of the app with the feature database to identify the existence of any tumor code. In order to defeat the widely used code obfuscation technique, APKLancet leverages the fuzzy hashing content comparison technique proposed by DroidMOSS[23] to detect whether a class in an app is similar to one of the tumor classes in our database. If such a class is detected, APKLancet will label it as the index class to help conduct tumor payload partitioning in the next stage.

3.3 Tumor Payload Partitioning

After obtaining suspicious index classes, the next thing is to partition all of the tumor payload inside an app. In detail, the task is to identify the entire tumor payload including inserted executable code and its corresponding declaration in manifest file, extra injected resource files and additional libraries. To fulfil this task, APKLancet makes use of program analysis technique to traverse the entire payload from the detected index classes. According to the type of tumor code, APKLancet deals them with different strategies.

3.3.1 Third-party Library Partitioning

The method APKLancet partitions third-party library is mainly based on the knowledge of its insertion style. In most cases, third-party library is a relatively independent module which allows the partitioning with feasibility. The bundling of a third-party library generally involves adding three categories of information: a supporting Java jar file, special meta-data tag, possible elements declared in layout XML files and a small amount of modification to the original code mainly for inserting new advertising *View* class.

Because library provider more or less publishes document to illustrate how to integrate the lib into an app, APKLancet leverages this information for partitioning the inserted library with ease. The inserted libraries are usually Java jar files attached to the APK and can also be decompiled. If APKLancet identifies suspicious index class in these libraries, the whole Java jar file is labeled as the tumor payload (i.e., the whole package *com.google.ads.**). APKLancet further labels the inserted instructions in benign classes according to the reference relationship. In general, each class of benign code is searched to find any reference to any of the whole tumor libraries. For instance, an *AdView* class is created as follows:

```
adView = new AdView(this);
adView.setAdUnitId(MY_AD_UNIT_ID);
adView.setAdSize(AdSize.BANNER);
```

APKLancet could recognize the reference relationship of the class *AdView* with the payload library, and hence label these instructions(in smali code) as part of the tumor payload. Finally, the meta-data tag and the elements in resource files are also labeled as part of the tumor payload.

3.3.2 Malicious Code Partitioning

For malicious code in an infected app, the situation is much more sophisticated. Unlike the situation of third-party library, systematic knowledge is lacked in partitioning malicious code. There is no document information for malicious code and it usually uses transformation technique to avoid being detected. Furthermore, instead of being injected as an independent Java package (as the style of third-party libraries bundling), malicious payload is sometimes injected into an existing Java package of benign parts to make the identification more difficult, even the payload is still a relatively independent module from the viewpoint of functionality. Hence APKLancet makes use of program dependency analysis technique to help partition. APKLancet uses Algorithm 1 to search malicious code in an APK. The input of the algorithm include a set O of all classes in an APK and a set E containing the already identified malicious index classes. The output is a set M that contains the entire malicious class payload. The core part of the algorithm is the function $Find\_invoke\_dest()$. It is based on the program dependency graph of the app to extend the malicious class set. If the already identified malicious class invokes any method of a class that is not in system library, the invoked class should be added into the malicious class set.

Algorithm 1 Malicious Code Class Searching

Require:
The set of class in an APK, O;
The set of malicious class in O, E;
Ensure:
The set of malicious code class, M;

```
1: M = E
2: repeat
3:    D = ∅
4:    for all m in M do
5:       D ← Find_invoke_dest(m);
6:    end for
7:    for all d in D do
8:       if d in O AND d not in M then
9:          M ← d
10:      end if
11:   end for
12: until M is not modified
13: return M;
```

We choose a Java *class* as the basic unit in the algorithm for the reason that malicious payload is always composed of several complete classes. It is more likely for malware to construct a new class to perform malicious behavior instead of adding some malicious methods to specific benign

classes because adding independent module is more suitable for large-scale deployment and needs less complicated manual analysis to various original apps to assure the modified apps still work.

3.4 Tumor Payload Resection

Although during tumor code partitioning stage an entire payload has been identified, tumor payload resection is non-trivial compared with the permission removal of app[16]. A payload contains both relatively independent third-party libraries and bundled code that is tightly interweaved with the host class. Therefore, simply resecting the entire payload will lead to an improper execution or even crash the app. Thus after partitioning tumor code payload, there still remains fixing work to be done. The key point is to repair the inserted code in benign code and information in the manifest file. APKLancet adopts a three-step repairing strategy, which is introduced in the following subsections.

3.4.1 Entry Point Reverting

An app declares many kinds of information in its manifest file. An inserted tumor code payload also declares its components in this file and adds or modifies entry points. In detail, we consider the following frequently used entry point modification approaches:

1. *Directly added entry point.* Tumor payload will directly register new *Service*, *Broadcast Receiver* and *Activity* in manifest file. Under certain condition, the tumor payload class is invoked.

2. *Main entry point tampering.* Some tumor code simply changes the original app's main entry point class (declared in the manifest file's <*application*> tag or changed the ACTION_MAIN Intent) to its own class. Through tampering the execution flow, it makes sure that the inserted module is invoked.

3. *Entry point inheritance modification.* One trick used by malicious code is to hide malicious entry point class through class inheritance. In this case, the original main entry point class is kept and malicious classes can hide themselves without being declared in manifest file. However, the inheritance relationship of main entry point class is modified. For instance, in Fig 2 the entry point class *com.normal.Activity* is originally inherited from *android.app.Activity*, after the infection its superclass is changed to *com.mal.Activity* which is a malicious class. When the entry point class is executed, the malicious function in its super class is firstly invoked.

In order to make a fixed app work properly, APKLancet needs to handle manifest file to resect the entry point items related to tumor payload. Moreover, if the main entry point class is modified to be a tumor code class, APKLancet tries to recover the main entry point through searching a *launcher* class. Once a launcher class is found, APKLancet will use it as the new main entry point class. If no launcher class can be found, APKLancet will further search for benign *Activity* referred by current entry point class code, and uses the first found *Activity* as new entry point class. If again there is no candidate, APKLancet uses the first *Activity* declared in manifest file as the new entry point class. For the case of *Entry point inheritance modification*, we need to find the first super class of the original entry point class which is not in the tumor payload(i.e., *android.app.Activity* in Fig 2), and recover the inheritance relationship.

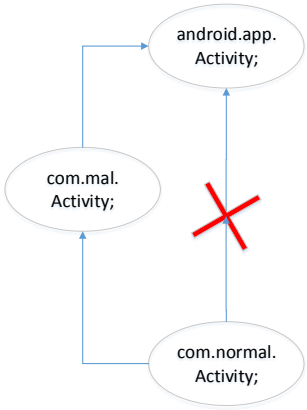

Figure 2: Entry Point Class Inheritance Modification

3.4.2 Benign Code Patching

APKLancet can identify every invoke instruction in decompiled smali code and check whether a benign code class invokes a method of tumor code class. Further, the invoking can be divided into two cases: 1) the invoked method is an inherited method that overwrites the base method, and it belongs to a class of entry point class type (inherits from *Activity*, *Service*, *Broadcast Receiver*, *Content Provider* or *Application* class), 2) the invoked method belongs to a common class of tumor code payload.

For the first case, the invoked methods in tumor code generally overwrite the methods of base class (such as *onCreate()*) and are special for Android application execution model. Resecting this kind of methods will crash the app. In this situation, APKLancet will replace the invoked methods in tumor code class with the methods of its direct base class (already fixed in inheritance recovery). For the second case, we consider the invoking instruction and its data-dependent instructions are inserted. Therefore, APKLancet analyzes the following instructions that depend on the return value of this invoking instruction, and patches the invoking instruction as well as following data-dependent instructions with *nop* instruction.

APKLancet will also scan the decompiled smali code to find any object reference relationship between benign code class and tumor code class. Again, if an object reference is found, APKLancet uses data dependency analysis to find other instructions and objects that are data-dependent to this object and then get it resected.

3.4.3 Payload Resection

After the fixing of benign code classes, APKLancet resects tumor code directly. In addition, APKLancet checks resource files related to tumor code. Native libraries(.so) and Java libraries(.jar) that are used by tumor code **only**, file under the *lib* and *assets* directory that are only referred by tumor code are all considered as part of the tumor payload, and should be resected.

Tumor payload	Classes resected	Items resected in manifest file	Resources resected	Reference patching	Entry point patching
ADRD	25	5	0	No	No
BaseBridge Variant.A	17	0	1	Yes	Yes
BaseBridge Variant.B	69	10	3	Yes	Yes
BaseBridge Variant.C	20	3	4	Yes	No
DroidDream	10	3	5	No	Yes
DroidKungFu	13	3	5	Yes	No
DroidKungFu2	13	3	5	Yes	No
Geinimi Variant.A	105	4	0	Yes	Yes
Geinimi Variant.B	85	3	0	No	Yes
PJAPPS	22	3	0	No	No

Table 1: Characteristic of Different Tumor Payload

3.5 Verification

APKLancet introduces a verification process to assure that the purified app is able to work properly and the purification process does remove the unwanted behavior. It evaluates the purification from the aspects of both feasibility and effectiveness.

3.5.1 Feasibility

APKLancet adopts a very conservative strategy, which does not allow new exception to crash the host app after the purification. The verification work for purified app contains two steps. First, the tested app is launched before purification and operated manually, and the Android's *logcat* output is collected. Second, after the purification the tested app is launched again with the same operation. APKLancet will check the *logcat* output and compare it with the previous record to determine whether the application is being executed normally or throws any exception or crashes due to the purification.

APKLancet does not adopt random test methodology such as testing using Android *Monkey*. Instead, to assure the purified APK is still able to work properly, APKLancet needs the involvement with manual test to deal with complex GUI interactions such as *password login*.

3.5.2 Effectiveness

APKLancet evaluates the effectiveness of the tumor code excision both statically and dynamically. The static evaluation approach adopted by APKLancet takes advantage of online malicious code analysis engine. The purified APK, if containing tumor code beforehand, is submitted to VirusTotal[7] to check whether the malicious feature still exists. This approach evaluates whether the purification does identify the malicious feature. The purified app is also executed on real device for APKLancet to collect *logcat* output and verify whether the features such as advertising is resected.

4. EVALUATION

4.1 General Testing

We tested APKLancet with repackaged apps from online sandbox system Sanddroid[6]. Table 1 shows the result that how APKLancet splits the typical tumor payloads. The result shows that for each payload at least 10 classes are resected during the purification process. To be more clear, we define different purification results of one malware family as different variants and label it in the form of *Variant.A*, etc. From the results, we can see that nearly all the malwares added or modified entry point of the original apps and it is quite common for malwares to conduct simple modification to benign part of the original app or add extra resource files such as native libraries or jar files to inject malicious behaviors.

In order to further effectively evaluate APKLancet, we randomly choose 16 apps from each malware family in our malware collection (we elaborately avoid those samples we used to build feature databases though the malware family they belong to should be included in our feature databases). The characteristic of the chosen apps is that not only do they contain either malicious code or third-party libraries (some of them contain both), but all of these apps are also able to execute well on latest version of the Android OS on mainstream devices. We evaluate the sophisticated result of malicious code purification. First, all of the tested apps worked properly and did not terminate by exception according to our manual test and Android's *logcat* output. Also, as Table 2 shows (malware family information given by AegisLab Antivirus), all of the samples experience a dramatic decrease in VirusTotal detection result after the purification, which means the tumor payload purification is effective and malicious behaviors are resected from these malwares to some extent. Although more than half of the samples are not detected by any anti-virus engines in VirusTotal, we also notice that a few anti-virus engines in VirusTotal still regard the purified apps as malicious. After manually checking those purified samples, we find that the same anti-virus engine classifies the sample as a different malware family in most cases. The root cause for the alert information is that some third-party libraries such as analytics plugins and ad libraries are still remained in the app, but are not defined as tumor payload either by APKLancet or other majority of Antivirus. Therefore, we treat this as false positive alert.

We also evaluate the effectiveness of resecting typical ad libraries. After purification of the tested app with *AdMob* library, advertisement in app's UI is gone.(See Fig 3). What's more, such outputs (i.e., URL of *AdMob* or Javascript code) that indicate the existence of *AdMob*, are not found in *logcat* output after purification. At the meantime, other log information created by benign code generally appears repeatedly before and after the purification. Another case is app with

Package Name	Malware Family	Before Purification	After Purification
com.appspot.swisscodemonkeys.steam	ADRD	26/47	3/47
com.bytedroid.liveprints	ADRD	35/47	0/44
com.caiping	BaseBridge	30/45	1/48
com.computertimeco.minishot.android	BaseBridge	32/48	0/47
com.game	BaseBridge	34/48	0/47
com.hyxen.taximeter.app	BaseBridge	34/48	0/45
com.power.SuperSolo	DroidDream	38/47	0/46
super.mobi.eraser	DroidDream	37/47	3/47
com.andtutu.stetris	DroidKungFu	33/47	0/46
com.eguan.update	DroidKungFu	34/47	0/47
com.bottleworks.dailymoney	DroidKungFu2	35/47	3/46
com.allen.txthej	DroidKungFu2	36/47	4/46
chaire1.mm	Geinimi	33/47	0/48
com.aac.cachemate	Geinimi	33/47	1/47
com.electricsheep.dj	Geinimi	33/48	1/47
cn.jingling.motu.photowonder	PJAPPS	32/47	0/46

Table 2: VirusTotal Detection Result

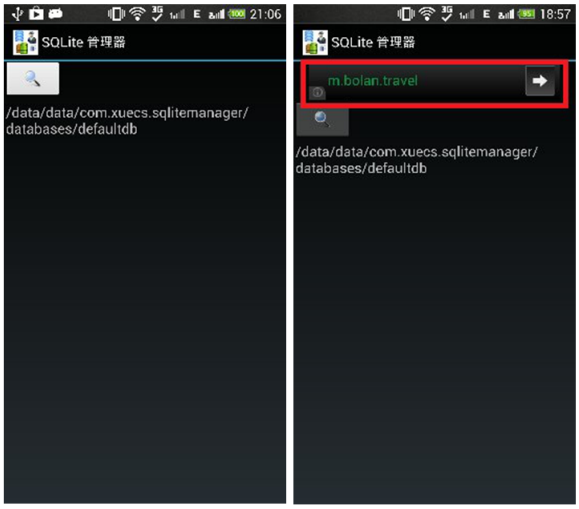

Figure 3: Advertisement Resection

Flurry library. If the *Flurry* library is loaded, *logcat* outputs the following information:

```
D/FlurryAgent: Starting new session
D/FlurryAgent: Sending report to: http://data.flurry.com/aar.do
D/FlurryAgent: Report successful
```

Such information would not be found in the purified app even for the same user operation is performed.

4.2 Case Study I: BaseBridge and Wooboo

The first case we studied is a repackaged app *com.caiping*. APKLancet extracts the fuzzy hashing feature of every entry point class of the apps. Compared with the feature database built before, in this app 10 classes (*com.android.view.custom.**) are detected to be tumor code index classes. Then APKLancet identifies the payload using Algorithm 1 taking these index classes as input, and finds out class *jackpal.androidterm.Exec* and all classes of *com.sec.android.providers.drm.** are also tumor code.

After locating tumor code payload, APKLancet discovers that the direct base class of main entry point *caiping* class, the *BaseAActivity* class, which is not declared in the manifest file, is suspicious and connected to the feature of *BaseBridge* malware family.

```
.class public Lcom/caiping/caiping;
.super Lcom/android/view/custom/BaseAActivity;
.source "caiping.java"
```

That means the malicious code modifies the class inheritance to make the *BaseAActivity* class execute before the original entry point class. If APKLancet simply resects the *BaseAActivity* class, the inheritance of *caiping* class is broken. APKLancet thus redirects the base class of *caiping* class to the base class of *BaseAActivity*, *android.app.Activity*.

After fixing the original inheritance of the main entry point class, APKLancet checks references in benign code and finds 4 method-invoking references (*onCreate*, *<init>*, *onCreateOptionsMenu* and *onOptionsItemSelected*) of the *BaseAActivity* class in benign code. All of these references locate in the main entry points class *com.caiping.caiping*. As we mentioned above, class *caiping* is a derived class of the malicious class *BaseAActivity*, and the base class of *BaseAActivity* is *android.app.Activity*. These 4 method-invoking references should point to the methods in class *android.app.Activity*, APKLancet hence patches the invoking instruction through replacing the invoked method's class from *BaseAActivity* to *android.app.Activity*. For the tumor object reference in this app, no data-dependent relationship is found and APKLancet just resects any direct reference of *Lcom/android/view/custom/BaseAActivity* in the benign code. APKLancet also cleans the declaration of the tumor code class in manifest file.

Finally, according to the string searched in smali code,

```
const-string v0, "androidterm"
invoke-static {v0},
    Ljava/lang/System;->
        loadLibrary(Ljava/lang/String;)V
```

The related *androidterm* library (libandroidterm.so) should be resected.

After the purification and repackaging, the new app is able to work. However, the test result of VirusTotal demonstrates that 15 out of 47 antivirus engines still report the app

as malware and most of them classified it as *Wooboo* malware. We manually check the app and find that it contains an unauthorized advertising library *Wooboo*. So we add the *Wooboo* library's feature into APKLancet's database and re-execute the purification process. Relying on the knowledge of *Wooboo* directly, we regard the *WoobooAdView* as its index class and label the whole package where the index class locates as the payload. This time APKLancet partitions all classes of *com.wooboo.adlib_android.** as tumor payload.

In benign code APKLancet find four references of the tumor payload. All these references directly refer to the class *com.wooboo.adlib_android.WoobooAdView*, hence APKLancet recursively resects dependent instructions and definitions. After the purification, the purified app still works properly under manual testing. And all 47 engines of VirusTotal report no threat for this purified app.

4.3 Case Study II: Geinimi

The second case is an app *com.electricsheep.dj* containing malicious code of *Geinimi* malware family. According to APKLancet's feature database, the following classes are identified as the index classes,

```
com.geinimi.AdServiceReceiver;
com.geinimi.AdService;
com.geinimi.custom.Ad1020_102001;
```

Then, using the index classes as input, APKLancet executes Algorithm 1 and partitions classes *com.geinimi.** as the tumor code payload. After the partitioning of tumor code payload, APKLancet checks references in benign code and finds no reference to the tumor code payload. However, APKLancet finds that in manifest file the main entry point will be missing if the tumor code is resected. According to the fixing policy of APKLancet, the replaced main entry point should be first found in all launcher classes declared in its manifest file(See Fig 4). In this case, APKLancet finds *DroidJumpActivity* is the proper candidate and defines it as the new main entry point class. Further analysis shows that the inserted malicious entry point class will first perform its own function, and then starts the *DroidJumpActivity* class. The code fragments selected from several smali files to illustrate the behavior of launching the original entry point is showed as followed(code fragment of exception handler part is omitted):

```
AdService.smali:
...
  const-string v0,
    "com.electricsheep.dj.DroidJumpActivity"
  sput-object v0,
    Lcom/geinimi/AdService;->a:Ljava/lang/String;
...

AdActivity.smali:
...
  sget-object v1,
    Lcom/geinimi/AdService;->a:Ljava/lang/String;
  invoke-static {v1}, Ljava/lang/Class;->
    forName(Ljava/lang/String;)Ljava/lang/Class;
  move-result-object v0
  new-instance v1, Landroid/content/Intent;
  invoke-direct {v1, p0, v0},
    Landroid/content/Intent;->
      <init>(Landroid/content/Context;Ljava/lang/Class;)V
  invoke-virtual {p0, v1},
    Lcom/geinimi/AdActivity;->
      startActivity(Landroid/content/Intent;)V
...
```

It proves that APKLancet's fixing policy effectively recovers the original entry point.

4.4 Case Study III: AdMob and Flurry

In this subsection, two popular third-party libraries, *AdMob* and *Flurry*, are studied to illustrate the effectiveness of APKLancet's purification. Actually *AdMob* and *Flurry* are benign third-party ad libraries, but they may be injected into legitimate apps by attackers for profit. What's more, they are typical full functional third-party ad libraries and thus are good cases for demonstrating the excision process and evaluating the purification.

The case for illustrating *AdMob* purification is an app whose package name is *com.xuecs.sqlitemanager*. APKLancet lists the class *AdView* as index classes in the feature database and identifies the whole package of *com.google.ads.**, which contains the index class, as tumor payload. Then APKLancet resects all the method invoking and object references from the benign code to the tumor code, including 3 *invoke* instructions with the same pattern, 1 object reference and several further dependent instructions in benign code. For the case of *Flurry*, the sample is an app *com.electricsheep.dj*. Also, APKLancet constructs the feature database by listing *FlurryAgent* as index class and then resects the tumor payload *com.flurry.android.**. After that, APKLancet searches the benign code and finds several method invoking instructions related to tumor payload, including *onStartSession*, *onEvent*, *onEndSession*, etc.

One problem of the ad libraries purification is that Java class is often referred in resource files (i.e., layout XML or manifest). And this kind of information is not fixed and cannot be detected with the knowledge of feature database. For instacne, in *main.xml* file an *AdMob*'s *AdView* element is referred as:

```
<com.admob.android.ads.AdView android:id="@id/ad"
    android:layout_width="fill_parent"
    android:layout_height="48.0dip"
    admobsdk:backgroundColor="#ff00b8f5"
    admobsdk:textColor="#ffffffff"
    admobsdk:keywords="Android game droid jump" />
```

To solve this issue, the analysis of APKLancet for tumor payload not only deals with reference in code part but also in resource part. The patching of resource files is relatively easy because APKLancet could resect this kind of declaration directly.

5. LIMITATIONS

In this paper we do not focus on identifying the repackaged app which have been extensively studied[25]. One essential requirement for the repackaging detection is that the detection needs the knowledge of the original app. Actually, there is no one-size-fits-all approach to find the knowledge of the original app. It seems that the certificate of an APK could help identify its provenance. The problem is that Android allows either a certificate from the certificate authority or a self-signed certificate. Thus it is not able to trust an APK only through the certificate. Best practice suggests that Google Play is the trust source for validating APK. But even some APKs on Google Play are also repackaged version of other applications. And not all of the applications can be found on Google Play. Actually in China some famous applications(e.g., WeChat with more than 500 million users) cannot even be downloaded from Google Play. Thus, Our

```
<application android:label="@string/app_name" android:icon="@drawable/icon">
    <activity android:theme="@*android:style/Theme.NoTitleBar.Fullscreen" android:label="@string/app_name"
    android:name=".DroidJumpActivity" android:screenOrientation="portrait">
        <intent-filter>
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <activity android:label="@string/app_name" android:name=".SettingsActivity" android:screenOrientation="
    portrait" />
    <meta-data android:name="ADMOB_PUBLISHER_ID" android:value="a14b5c886f95c4c" />
    <activity android:theme="@*android:style/Theme.NoTitleBar.Fullscreen" android:name="GameActivity"
    android:screenOrientation="portrait" />
    <receiver android:name="com.geinimi.AdServiceReceiver">
        <intent-filter>
            <action android:name="android.intent.action.BOOT_COMPLETED" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </receiver>
    <service android:enabled="true" android:name="com.geinimi.AdService" android:permission="android.
    permission.INTERNET" />
    <activity android:label="@string/app_name" android:name="com.geinimi.custom.Ad1020_102001">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
</application>
```

Figure 4: Main Entry Point Fixing

APKLancet system focuses on the tumor payload ignoring whether it is a repackaged app or not. APKLancet adopts the policy of splitting any unrelated modules and resecting them to assure the security.

Since our work does not include repackaging detection, we do not distinguish whether the third-party libraries such as ad libraries are imported by the original developers or injected by malicious authors. We argue that the choice of purification of third-party libraries should depend on the end user. APKLancet hence just prompts the existence of any potentially unwanted ad library based on the permissions it requires, and the end user may decide whether to resect it or not.

Our work assumes that the tumor code is embedded as a module not closely interweaving with the host app. Although this assumption stands in most cases, some undesirable behaviors may still remains inside the application if the tumor code author binds the tumor code with the benign function cautiously. However this may not happen frequently for the cost is generally very high(a profitless work seldom lasts). What's more, we purify the apps according to the index classes in feature database, so benign apps that generally do not include the index classes will not be affect by APKLancet.

Finally, this work does not focus on the detection of malware. Our APKLancet relies on the tumor payload feature database built on existing knowledge of the undesirable code. As a result, APKLancet could only deal with the analyzed malicious tumor payload, and because APKLancet is mainly a static analysis based system, it is not able to control the dynamically loaded code. However, the goal of APKLancet is only to purify the application with undesirable code so that it is good enough to be used again. It adopts conservative strategy to only deal with apps that it can be purified. It is a simple but effective approach to prevent the known malicious tumor payload. APKLancet may not able to purify any tumor payload perfectly, but it helps deal with the known ones and can co-operate with other access control system to counter other unknown undesirable code.

6. RELATED WORK

6.1 Malware Analysis

Previous work on tumor payload analysis has mainly focused on malicious code analysis. Automated analysis tools are also developed to help detect malicious code and different evaluation schemes are adopted. **DroidRanger**[25] detects malicious applications by using both permission-based behavioral footprint and heuristics-based filtering scheme. **RiskRanker**[13] performs large-scale security risk analysis for zero-day malware detection. Researchers have also systematically characterize Android malwares from various aspects including installation methods, activation mechanisms as well as the nature of carried malicious payload[24].

Our study does not lead to malware detection. The purpose is to resect the tumor payload with the help of existing summarized knowledge. On one hand, malware detection and analysis contribute a lot to tumor payload identification. On the other hand, our APKLancet system pays attention to tumor payload, which contains not only malicious code but also potentially risky third-party libraries. It needs to address challenges brought by tumor payload purifying and host apps fixing.

Our work is the first to measure the effectiveness of tumor payload purification. Although our work relies on existing knowledge of tumor code, we believe that most tumor payload in repackaged applications are not novel, especially for its injection style.

6.2 Repackaging Detection

Tumor payload detection and analysis is closely connected to the problem of detecting repackaged legitimate app, which is often injected with malicious payload. Studies vary from pair-wise comparison to scalable detection. **DNADroid**[8] computes the similarity between two Android applications by comparing program dependency graph to detect applica-

tion copying and cloning. And **DroidMOSS**[23] systematically detects repackaged apps on third-party Android marketplaces by applying fuzzy hashing technique to measure an app's similarity and then compares apps in pair. **AnDarwin**[9] uses a scalable approach rather than comparing apps pairwise to detect similar Android applications based on semantic information. **PiggyApp**[22] scalably detects *piggybacked* Android applications by organizing various feature vectors from apps into a metric space and applying line-arithmic search algorithm. The prior works mainly focus on the similarity of the repackaged app and the original one, while our work assumes that the target of our system contains not only the repackaged apps but also some apps with tumor payload injected by their authors. In this case, APKLancet can deal with the tumor payload without the support of the reference app.

6.3 Advertisement Splitting

Advertisement is controversial in Android app. An ad library embedded in an app may send information about the device and user to the ad server, thus raising concerns about user privacy. Studies suggest isolating the advertising or even separating it from the application process.

AdSplit[18] extends Android to separate advertising from applications and leverage QUIRE's mechanisms to let the remote server validate the authenticity of client-side behavior. **AFrame**[21] provides a developer a friendly method to isolate untrusted third-party code from the host applications including process, permission, display and input isolation. **AdDroid**[17] introduces a new advertising API and corresponding advertising permissions to separate privileged advertising functionality from host applications. All of these schemes require to extend the Android system trying to separate advertisement from host apps. In comparison, APKLancet does not need to modify the Android system and introduces no performance latency.

6.4 App Rewriting

Many fine-grained permission control systems use application rewriting technique to implement sensitive API invoking management. **Aurasium**[20] repackages applications to attach user-level sandboxing and policy enforcement code for the aim of monitoring applications' behavior about security and privacy violation. **In-App Reference Monitors**(I-ARM)[10] rewrites the Dalvik bytecode of applications to enforce the security policies the framework users specified towards a set of security-sensitive API methods. **Dr. Android**[15] introduces a novel framework to address the issue that many applications are allowed broader access than required by adding finer-grained permissions. APKLancet also rewrites the app. But instead of adding instrumentation code or extra libraries to monitor the behavior of the app, it directly resects the undesirable part to enforce access control policy, which is more concise.

7. CONCLUSION

Android apps are vulnerable to repackaging attack and the undesirable code (tumor code) bundling is becoming a popular way of spreading malicious behavior. Based on the fact that the tumor code in Android APK is usually characteristic and relatively independent in the app, it is possible to resect the tumor from the original APK. In this paper, we propose an effective tumor code diagnosis and purification system called APKLancet. It consists of three components, the APK analyzer, APK rewriter and APK verifier, and carries out the workflow by four stages: 1) diagnosing the tumor code relying on an existing knowledge database, 2) partitioning the tumor payload from the host app, 3) excision of the tumor code and restoring the benign function, and 4) verifying the app's benign function. APKLancet has been applied to apps with typical tumor payloads and our analysis indicates that it is feasible to defend the undesirable behaviors through app's purification.

8. ACKNOWLEDGEMENT

This work is supported by National Natural Science Foundation of China (No.61103040), National Science and Technology Major Projects (Grant No.2012ZX03002011), and Technology Innovation Project of Shanghai Science and Technology Commission (No.13511504000). We also appreciate Wenjun Hu, the author of SandDroid[6], for providing the valuable malware samples.

9. REFERENCES

[1] 1.2 percent of google play store is thief-ware, study shows. `http://tinyurl.com/kvf7xvc`. Online; accessed Nov-2013.

[2] Ad networks - android library statistics. `http://www.appbrain.com/stats/libraries/ad`.

[3] Ad vulna: A vulnaggressive (vulnerable & aggressive) adware threatening millions. `http://tinyurl.com/pv4wts3`. Online; accessed Nov-2013.

[4] android-apktool, a tool for reverse engineering android apk files. `http://code.google.com/p/android-apktool/`. Online; accessed Nov-2013.

[5] Android torch app with over 50m downloads silently sent user location and device data to advertisers. `http://tinyurl.com/mhfyv3r`. Online; accessed Nov-2013.

[6] Sanddroid - an automatic android program analysis sandbox. `http://sanddroid.xjtu.edu.cn/`. Online; accessed Nov-2013.

[7] Virustotal - free online virus, malware and url scanner. `https://www.virustotal.com/` note = Online; accessed Nov-2013,.

[8] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. In *Computer Security–ESORICS 2012*, pages 37–54. Springer, 2012.

[9] J. Crussell, C. Gibler, and H. Chen. Andarwin: Scalable detection of semantically similar android applications. In *Computer Security–ESORICS 2013*, pages 182–199. Springer, 2013.

[10] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen. I-arm-droid: A rewriting framework for in-app reference monitors for android applications. *Mobile Security Technologies*, 2012, 2012.

[11] A. Desnos. Androguard: Reverse engineering, malware and goodware analysis of android applications... and more (ninja!).

[12] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild.

In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2011.

[13] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 281–294. ACM, 2012.

[14] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. Juxtapp: a scalable system for detecting code reuse among android applications. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 62–81. Springer, 2013.

[15] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2012.

[16] K. Kennedy, E. Gustafson, and H. Chen. Quantifying the effects of removing permissions from android applications.

[17] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 71–72. ACM, 2012.

[18] S. Shekhar, M. Dietz, and D. S. Wallach. Adsplit: Separating smartphone advertising from applications. *CoRR, abs/1202.4030*, 2012.

[19] G. Suarez-Tangil, J. Tapiador, P. Peris-Lopez, and A. Ribagorda. Evolution, detection and analysis of malware for smart devices. 2013.

[20] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[21] X. Zhang, A. Ahlawat, and W. Du. Aframe: Isolating advertisements from mobile applications in android. 2013.

[22] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, scalable detection of piggybacked mobile applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 185–196. ACM, 2013.

[23] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 317–326. ACM, 2012.

[24] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.

[25] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, 2012.