Resilient Decentralized Android Application Repackaging Detection Using *Logic Bombs*

Qiang Zeng
Temple University, USA
qzeng@temple.edu

Lannan Luo
University of South Carolina, USA
lluo@cse.sc.edu

Zhiyun Qian
UC Riverside, USA
zhiyunq@cs.ucr.edu

Xiaojiang Du
Temple University, USA
dux@temple.edu

Zhoujun Li
Beihang University, China
lizj@buaa.edu.cn

Abstract

Application repackaging is a severe threat to Android users and the market. Existing countermeasures mostly detect repackaging based on app similarity measurement and rely on a central party to perform detection, which is unscalable and imprecise. We instead consider building the detection capability into apps, such that user devices are made use of to detect repackaging in a decentralized fashion. *The main challenge is how to protect repackaging detection code from attacks*. We propose a creative use of *logic bombs*, which are regularly used in malware, to conquer the challenge. A novel bomb structure is invented and used: the *trigger conditions* are constructed to exploit the differences between the attacker and users, such that a bomb that lies dormant on the attacker side will be activated on one of the user devices, while the repackaging detection code, which is packed as the bomb *payload*, is kept inactive until the trigger conditions are satisfied. Moreover, the repackaging detection code is *woven* into the original app code and gets *encrypted*; thus, attacks by modifying or deleting suspicious code will corrupt the app itself. We have implemented a prototype, named BombDroid, that builds the repackaging detection into apps through bytecode instrumentation, and the evaluation shows that the technique is effective, efficient, and resilient to various adversary analysis including symbolic execution, multi-path exploration, and program slicing.

CCS Concepts • **Security and privacy → Software and application security**; *Intrusion/anomaly detection and malware mitigation*;

Keywords Android app repackaging, code obfuscation

CGO'18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5617-6/18/02…$15.00
https://doi.org/10.1145/3168820

ACM Reference Format:
Qiang Zeng, Lannan Luo, Zhiyun Qian, Xiaojiang Du, and Zhoujun Li. 2018. Resilient Decentralized Android Application Repackaging Detection Using *Logic Bombs*. In *Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3168820

1 Introduction

Application repackaging poses a severe threat to the Android ecosystem. Dishonest developers unpack apps, replace the icons and author information with theirs, repackage them and then resell them to make profits; the whole procedure can be automated and done instantly. Moreover, attackers frequently insert malicious code into repackaged apps to steal user information, send premium text messages stealthily, or purchase apps without users' awareness, threatening user security and privacy [12, 18, 22, 25, 49–51]. Previous research showed that 86% of 1260 malware samples were repackaged from legitimate apps [61]. E.g., the malicious adware family, Kemoge, which infected victims from more than 20 countries, disguised itself as popular apps via repackaging [56].

To deal with the critical problem, many repackaging detection techniques have been proposed. Most of them are based on app similarity comparison [8, 12, 20, 36, 55, 60], and thus tend to be imprecise when handling obfuscated apps. Besides, they usually rely on a centralized trusted party to conduct detection, which is not scalable considering the sheer number of apps. Moreover, there are a plethora of alternative app markets, but their quality and commitment in repackaging detection are questionable [27]. Finally, users may download apps from places other than any markets and install them.

Thus, a decentralized repackaging detection scheme is desired: it adds repackaging detection into an app being protected, such that it becomes an inherent capacity of the app and thus does not rely on a third party. *The main challenge obviously is how to protect the repackaging detection capacity from the attacker.* While the goal of decentralized detection is highly desired and some attempts have been made towards it, the main challenge is not solved. For example, a state-of-the-art defense, SSN [28], proposes to conduct repackaging detection at a very low probability to hide the detection nodes; however, such probabilistic computation can be turned deterministic through code instrumentation. Plus, SSN tries to conceal specific API calls (mainly `getPublicKey`) through

reflection; but by inserting code that checks the reflection call destination, an attacker can easily reveal and manipulate all those calls. In short, SSN can be bypassed in multiple ways (detailed in Section 2.1). The failed attempts show how difficult it is to conquer the challenge.

Our observation is that the attacker side is very different from the user side; specifically, **(D1)** the hardware/software environments and sensor values are very diverse on the user side, while the attacker can only *afford* the time and money to test the repackaged app in a limited number of environments, and **(D2)** the attacker typically can only *afford* to test a small portion of an app, while users altogether play almost every part of the app. *Such observation is actually consistent with some well-known challenges in software testing*; that is, even with tremendous time and effort invested, a commercial program typically can only be tested under a very limited number of environments compared to the user side; and it is difficult to achieve a high code coverage. Thus, if we can exploit the differences *and* make sure the repackaging detection code is concealed and protected from being damaged on the attacker side and meanwhile ensure it to be executed on the user side, then the main challenge is solved.

To exploit the differences between the attacker side and the user side, we propose a novel use of *logic bombs*, which are normally used in malware. The trigger condition of a bomb is crafted, such that a bomb which keeps dormant on the attacker side will be activated on one of the user devices. For instance, a trigger condition can test whether the host app runs with some specific input or at a specific GPS location. While it is very costly for an attacker to trigger such a bomb by executing the app intensively, it is actually free to rely on the user devices to activate it, which exploits **D1**. Consequently, the repackaging detection code, as part of the bomb payload, is not executed unless the bomb is triggered. To exploit **D2**, bombs are inserted into various parts of an app (and our optimization phase will remove bombs that incur large overheads), such that many bombs can survive the adversary analysis of attackers.

In order to protect logic bombs, we additionally apply the following enhancements. First, the code of a logic bomb is encrypted and, more importantly, the key used to decrypt the code is not embedded in the app (which is unlike code packing used in virus); instead, the key can only be derived when the trigger condition is satisfied during program execution. Since the code is encrypted, attacks that try to search for specific API calls or bypass trigger condition will fail. Second, the bomb code is woven into the app code before being encrypted, such that attacks that simply delete suspicious code will corrupt the app execution.

As repackaging detection is performed online during app program execution, it provides good opportunities for rich actions upon detection of repackaging, such as slowing down, freezing, or crashing the app, which cause poor user experience and motivate users to uninstall the app. Moreover, the repackaging detection effort can be aggregated from user devices in many ways. First, the bad rating of a repackaged app due to the poor user experience will discourage other users from downloading the app. Second, the information about the repackaged app can be sent to the original developer, who can take further actions, e.g., requesting the host app market to take down the repackaged app. For apps downloaded from the Google Play app store, the Remote Application Removal Feature can wipe a malicious app from Android devices without any action on the user side [38], propagating the effect of detection from one device to others.

We have implemented the decentralized repackaging detection technique in a system named BombDroid, which adds the repackaging detection capacity to the bytecode of an app through *bytecode instrumentation*. Thus, it does not require the source code of apps to apply the technique, which means a third-party company may sell this service to developers who want to enhance their apps. We evaluated BombDroid on 963 Android apps. The evaluation results show that the protection provided by BombDroid is effective in repackaging detection, resilient to various adversary analysis, and incurs a very small speed overhead (~2.6%).

We made the following contributions.

- We present the first *resilient* decentralized Android app repackaging detection technique which builds repackaging detection code directly into apps.
- A creative use of bombs is proposed to exploit the differences between attackers and users, such that bombs keeping inactive on the attacker side will be activated on one of the user devices.
- We employ effective measures to enhance bombs, such that our technique defeats code deletion attacks and is resilient to various adversary analysis.
- We have implemented the techniques in a prototype system, and evaluated its effectiveness, efficiency, and resilience to various evasion attacks.

2 Threat Model, Goals and Architecture

2.1 Threat Model

Background. Before an app is released, it is signed using the developer's private key. When an app is installed, the signature is verified by the Android system using the public key stored in the certificate file carried in the APK file of the app. Once an app is installed, *its certificate is managed by the Android system and cannot be modified by app processes*. Each developer owns a unique public-private key pair; thus, when an app is repackaged by an attacker, the public key contained in the APK file of the repackaged app is certainly different from the original one. Therefore, it is possible to detect repackaging by comparing the original public key against the public key in the app's current certificate.

SSN's design. In order to discuss the threat model in a more concrete way, we first describe a *vulnerable* design, as shown in Listing 1, for building repackaging detection into the app code, which was adopted in SSN [28]. It detects repackaging by comparing the app's current public key to the original public key, `PUBKEY`, which is inserted into the code by SSN; the current public key is retrieved through a call to Android system service API `getPublicKey`. In order to hide the call

Listing 1. A vulnerable design.

```
1 if(rand() < 0.01) {
2    funName = recoverFunName(obfuscatedStr);
3    currKey = reflectionCall(funName);
4    if(currKey != PUBKEY)
5       // repackaging detected!
6       // response is delayed
7 }
```

from attackers, SSN proposed the following measures: (1) repackaging is only invoked probabilistically as shown in Line 1; (2) the function name "*getPublicKey*" is obfuscated, so attackers cannot find the word in the code; the call is issued through *reflection* (Line 3), which requires the function name to be recovered (Line 2), though. (3) after repackaging is detected, instead of responding to it, the response is delayed to confuse the attacker analyzing the protected app.

Next, we present attacks against repackaging detection. As our technique leverages logic bombs, we not only consider various common attacks, but also the state-of-the-art adversary analysis against bombs. We will also show how SSN is vulnerable to multiple types of attacks [34].

Text search. An attacker may search for specific text patterns, such as "*getPublicKey*", to locate repackaging detection code. In the case of SSN, it hides calls to `getPublicKey` through reflection calls.

Debugging. An attacker may install the repackaged app and run it on an emulator or a real device. Whenever suspicious symptoms arise, the attacker may use a debugger to trace back to the repackaging detection and response code. Such dynamic analysis works only when repackaging detection is executed. An attacker may try to *hook* critical calls the repackaging detection code relies on. For instance, an attacker may hook calls to `getPublicKey` in order to locate the repackaging detection code.

Blackbox fuzzing. An attacker may use blackbox fuzzing to run the repackaged app by providing a large number of random inputs to trigger as many logic bombs as possible [32, 52]. For every activated bomb, the attacker can trace back and disable it.

Path exploration. Various techniques have been proposed to explore execution paths in a program. A dynamic analysis based approach is to *explore multiple paths* during execution [33]. *Symbolic execution* has been widely applied to discovering inputs that execute program along specific paths [5, 6, 11, 19, 31]; it uses symbolic inputs to explore as many execution paths as possible, and resolves the corresponding path conditions to find the concrete inputs. Recent research has shown that symbolic execution is an effective approach to discovering conditional code and identifying trigger conditions [19]. *When symbolic execution is applied to SSN, Line 1 cannot stop symbolic executor from exploring (and hence exposing) the path containing repackaging detection.*

Circumventing trigger conditions. An attacker may simply circumvent trigger conditions and execute payloads directly. E.g., given a line of suspicious code, an attacker may perform *backward program slicing* starting from that line of code, and then execute the extracted slices to uncover the payload behavior [37]. Or, the attacker may apply *forced execution* to directly execute the code that is suspected to be payloads [47]. Take SSN as an example: an attacker can circumvent Line 1 to execute the following code; thus, SSN is vulnerable to such attacks.

Code instrumentation. An attacker may modify code to assist attack. In SSN, e.g., the attacker can insert code right before a suspicious reflection call to check the destination of the call, i.e., `if(funName==''getPublicKey'')`, and modify `funName` at will; or force `rand()` to return 0, such that *probabilistic invocation becomes deterministic.*

Code deletion. A trivial attack is to delete any suspicious code. Code deletion is not difficult to defeat. For example, we can transform some of the app code into a suspicious form, such that deletion of such code may lead to corruption of the app, or weave the logic bombs with the original app code.

2.2 Our Goals

The example of SSN, which is vulnerable to a variety of attacks, shows there exist plenty of pitfalls when designing a user-side repackaging detection technique and it also illustrates how challenging it is to propose a resilient design.

Our goals are as follows: **(G1)** it should be resilient to *path exploration*; **(G2)** the defense should be resilient to attacks via *text search*, *code instrumentation*, and *circumventing conditions*; **(G3)** it should be resilient to *debugging* and *fuzzing*; and **(G4)** it should defeat attacks based on *code deletion*. Different techniques and measures are employed in our system and they work together to achieve all the goals.

Assumptions: We do not handle users who are in collusion with pirates. We assume users do not *jailbreak* their devices; otherwise, the Android framework on the user device could be hacked to mislead our detection. However, attackers are allowed to hack and modify their own Android systems arbitrarily to assist adversary analysis.

2.3 Architecture

Figure 1 shows the procedure of building the repackaging detection capacity into an app. The input is the APK file of the app to be protected. BombDroid works on the binary code level. This is different from the prior state-of-the-art SSN [28], which can only work on source code. (1) The APK file is first unpacked to extract `classes.dex` (which is then converted to a collection of java classes) and a folder of resources containing the `CERT.RSA` file. (2) Then, BombDroid extracts the *public key* from `CERT.RSA`, and selects candidate locations for inserting logic bombs through static analysis of the binary code. (3) For each candidate location, a logic bomb is constructed and inserted by instrumenting the binary code. (4) The bomb code is then encrypted and *the encryption key is deleted from the app code.* The output is a protected app and will be sent to the legitimate developer to sign the app; note that the private key is kept by the legitimate developer and is not disclosed to BombDroid.

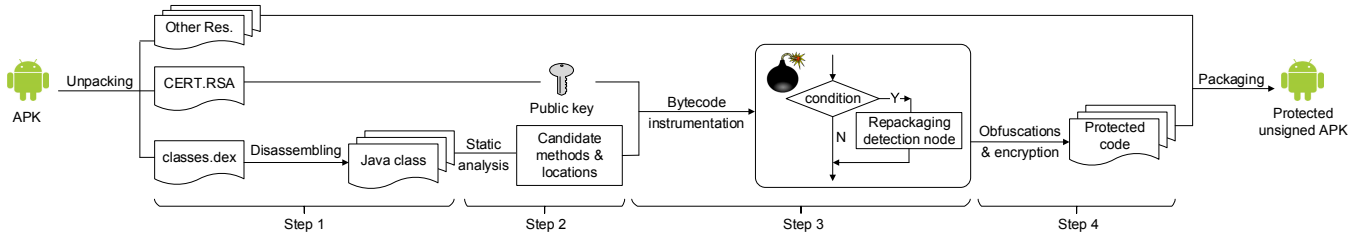


Figure 1. Architecture of BombDroid.

3 Logic Bombs for Repackaging Detection

3.1 Straightforward Use of Logic Bombs

Listing 2. A naive use of logic bombs.

```
1 if (X == c) {
2   // repackaging detection
3 }
```

While logic bombs have been widely used in building malware and are very effective in practice for keeping malicious code dormant until "correct" conditions are met, a straightforward use of logic bombs will be vulnerable to attacks. For example, as shown in Listing 2, a logic bomb is used for repackaging detection, which should not be activated unless the trigger condition `X == c` is true, where `X` is a variable or an expression, and `c` is a constant value. Consistent with the analysis in Section 2.1, the piece of code is vulnerable to various automated attacks. For example, the attacker may modify the bytecode such that the trigger condition is always true; worse, the attacker can circumvent the trigger condition evaluation to analyze and reveal the enclosed code directly [37]. Actually, it shares all the weaknesses of SSN. Thus, a naive use of bombs will not work for our purpose.

3.2 Cryptographically Obfuscated Logic Bombs

Listing 3. A cryptographically obfuscated logic bomb.

```
1 if(Hash(X) == Hc) // this line is equivalent to "X==c"
2   // "code" is encrypted and can only be decrypted when X=c
3   p = decrypt(code, X);
4   execute(p);
5 }
```

To defeat such attacks, we present a type of cryptographically obfuscated logic bombs. Let us take the code in Listing 2 as an example to show how to transform a vulnerable bomb to a cryptographically obfuscated bomb. First, the trigger condition "`X==c`" in Listing 2 is transformed into `Hash(X) ==` H_c, where H_c = `Hash(c)`. Second, the repackaging detection code is encrypted (before the app is released) and can only be decrypted correctly when `X=c`; any attempts that try to decrypt the code with an incorrect key will fail. Finally, *the constant value* `c`*, which works as the key, is removed from the code*, which means that an attacker cannot expect to search the code to find the correct key to recover the encrypted code. Through such transformation, the code in Listing 2 is transformed into the code shown in Listing 3.

The transformation applies both cryptographic hashes and encryption. A cryptographic hash function has two properties that are critical for transforming a condition `X == c` to the obfuscated condition `Hash(X) ==` H_c, where H_c = `Hash(c)`. First, the *one-way function (pre-image resistance)* property means it is difficult to recover the constant value `c` based on H_c, which ensures that it is computationally infeasible to reverse the obfuscation and hence defeats constraint solvers relied on by symbolic execution. Second, the *second pre-image resistance* property makes it difficult to find another constant value `c'` whose hash value is also H_c; thus, the obfuscated condition is semantically equivalent to the original, ensuring the correctness of the transformation.

Below is a simple example. The left part is a code snippet extracted from a real app. The right part is the corresponding obfuscated version: only when `mMode` is assigned with 0xfff000, can the payload code be successfully decrypted.

```
if (mMode == 0xfff000) {
   payload;
}
```

→

```
if (Hash(mMode) ==
da4b9237bacccdf19c0760cab7aec4a8359010b0) {
   p = decrypt (encrypted_payload, mMode);
   execute (p);
}
```

When designing the cryptographically obfuscated logic bombs, we were inspired by user authentication invented by Roger Needham [48], which stores user passwords as hash values [15], such that user passwords are not exposed but the authentication can still be performed. We found such transformations were widely discussed by researchers working on *virtual black-box obfuscation* [4] and concealing malware [40]. Their work confirms the security of such transformations.

3.3 Trigger Conditions

A condition that can be used as a trigger condition for the transformation must check equality of two operands with one of them having a constant value; the equality checking includes `==` and comparison methods such as string's `equals`, `startsWith`, and `endsWith`. We call such a condition a *qualified condition* (QC). Without loss of generality, a QC is denoted as "ϕ `== c`" in the following presentation, where ϕ is an expression or variable and `c` has a *statically determinable* constant value.

Existing qualified condition. A logic bomb can use a QC in the original code to build its trigger condition, which is

called an *existing qualified condition*. While medium and large sized programs usually have many existing QCs, smaller programs may not, which limits the number of logic bombs that can be inserted.

Artificial qualified condition. The drawback can be resolved by inserting *artificial qualified conditions*: given a program location L, a variable ϕ, and a constant value c, assume $L \in scope(\phi)$ and $c \in dom(\phi)$, where $scope(\phi)$ denotes the program locations where ϕ can be accessed and $dom(\phi)$ is the set of all possible values of ϕ. Then, ϕ == c is an artificial QC that can be inserted at L and work as a trigger condition. In an app, program variables with many possible values (i.e., a high entropy) are suitable for this purpose. *Without knowing the program logic*, it is difficult to determine whether a condition is an existing or artificial one.

3.4 Countermeasures against Code Deletion

An easy-to-conceive attack is to *delete* all suspicious code that involves cryptographic hash computation and code decryption. We apply the following countermeasures.

Code weaving. The first countermeasure is to *weave* the payload (i.e., the repackaging detection and response code) into the original app code. It is particularly suitable when a logic bomb is built based on an existing qualified condition (QC). When instrumenting the bytecode and injecting code, the repackaging detection and response code is woven into the body of the `if` statement for the existing QC. After code weaving, if attackers delete conditional code that look suspicious, it will corrupt the app itself. The consequences of corrupting an app can be various, such as instability, visualization errors, incorrect computation, or crashes.

Bogus bombs. The second countermeasure is to *transform* some conditional code of the app and make it look like logic bombs, and we call them *bogus* bombs. Deletion of bogus bombs will corrupt the app as well. Through such countermeasure, it is difficult for attackers to determine whether a piece of suspicious code is a real or bogus logic bomb.

Therefore, instead of hiding the repackaging detection code, which is difficult to achieve, we deter attackers from deleting the code.

4 Repackaging Detection and Response

4.1 Repackaging Detection

Repackaging detection and response are inserted as the payload of logic bombs, and we consider the following methods to detect repackaging.

Public Key Comparison. As used in previous attempts that try to build repackaging detection into the app code [28, 39], we also make use of public key comparison to detect repackaging. Each developer (or software company) has her own public-private key pair; thus, once an app is repackaged and resigned, the public key of the app must be different from the original one. We can retrieve the public key K_r at runtime and compare it against the original public key K_o to detect whether the app has been repackaged. K_o is extracted from `CRET.RSA` in the input APK file and then hard coded into the detection code, while K_r is retrieved by invoking an Android Framework API `Certificate.getPublicKey`.

Once an app is installed, its certificate is managed by the Android system and cannot be modified by app processes. While there are techniques that can intercept Android API calls and return fake values, they usually depend on jailbreak [10] or require code modification [24]. Note that we assume user devices are not jailbroken; moreover, modification of encrypted code will not work. Still, it should be possible to intercept system service calls in other ways, which is further discussed below.

Code Digest Comparison. The second way is to compare the original digest D_o of a resource or code file against the one D_r retrieved from `MANIFEST.MF` at runtime. However, the challenge is that it is unlikely to predict the digest of a code file and then hard code it into the code file. To solve it, we leverage *steganography* to hide D_o in `strings.xml` (a file storing string literals for the app), and then extract it at runtime and compare it against D_r. Note that it is unnecessary to compare the complete digest value; thus, D_o can encode part of a digest value into a string. Similarly, we can detect repackaging by checking whether the app icon and author information have been changed; it is very similar to checking digest values and thus omitted here.

As `MANIFEST.MF` is managed by the Android system, app processes cannot manipulate it. In addition, an attacker does not know how to manipulate strings in `strings.xml` even when they look suspicious, as the logic for recovering the digest value information from D_o is encrypted as part of the repackaging detection code.

Code Snippet Scanning. The third way is to scan code snippets and compute their hash values, which are then compared against the pre-computed values hidden in the data section or `strings.xml`; this is a mature code-integrity detection technique [7]. Unlike the previous two methods, code scanning does not rely on Android system service calls. We can combine public-key/digest checking with code snippet scanning: since any code modification can be detected by public key/digest checking, instead of scanning the whole app, the scanning can be focused on checking the integrity of bombs that compare the public keys and digests. In addition, it is indeed possible to intercept calls to `getPublicKey` through vtable hijacking [54]; scanning can be used to check the integrity of the vtable or the function body. That is, the scanning can be extended to scan other program elements.

4.2 Response

The responses upon repackaging detection should cause negative user experiences. For example, the response may set a reference variable to be NULL, cause memory leak (e.g., by allocating a large data structure and pointing to it using a static reference field), set up a timer that will terminate the process, or launch a thread executing an endless loop.

In addition, the response can warn users about the repackaging. Many ways can alert users through, e.g., `TextViews`, `PopupWindows`, and `Dialogs`. The response can also send

a brief description of the repackaged app to the developers, who can take further actions, such as requesting the store to take down the repackaged app before being widely distributed. *How to collect software piracy information in an inexpensive way has been a challenge for many app companies, and our technique can serve as a solution for them.*

5 Security Analysis

We then examine how the goals described in Section 2.2 are achieved by our design. First, all path exploration techniques [5, 6, 11, 33], including symbolic execution and multi-path execution, rely on resolving constraints correctly. In our case, the constraint `Hash(X) ==` H_c, has to be resolved in order obtain the key to decrypt and analyze the payload code. However, as cryptographic hash functions cannot be reversed, no constraint solvers can solve it. Therefore, we have achieved **G1** successfully.

Second, as the repackaging detection and response code is encrypted, attacks that rely on *text search*, *code instrumentation*, and *circumventing conditions for forced execution* will all fail. Thus, **G2** is achieved as well.

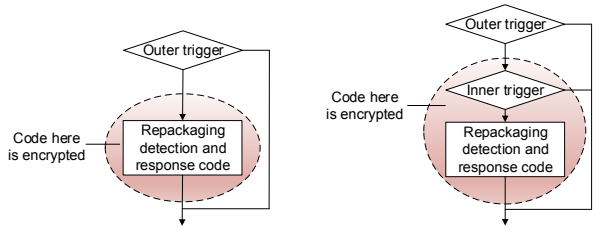
Third, we insert many logic bombs into different parts of an app. It is not surprising that through *blackbox fuzzing*, an attacker may be able to trigger some of them, but it is very difficult to reach a high ratio. This is also consistent with one of the well-known facts about software testing: it is difficult to reach a high code coverage. *Without the assistance of symbolic execution, even a line as simple as this,* `if (x==0x56789abc)`*, may take the fuzzer billions of tries to satisfy the condition.* Our evaluation also shows that only a small portion of bombs can be triggered through black-box fuzzing. Therefore, **G3** is also achieved. On the other hand, *given a large number of diverse users that play an app in many different contexts, it is natural that most of the logic bombs will be triggered on the user side for checking repackaging.*

Fourth, as described in Section 3.4, *code deletion* is defeated by code weaving and bogus bombs; thus, **G4** is achieved.

5.1 Attacks against Keys

As the key of of a logic bomb is important, we consider attacks specifically against keys. Attackers may try to figure out the key used in each logic bomb. One approach is *brute force attacks*. Given an obfuscated condition `Hash(X) ==` H_c, attackers may compute `Hash(X)` for all possible values of `X` to identify a value that satisfies `Hash(X) ==` H_c. Thus, the strength of the hash operation is determined by the set of possible values that `X` may take, denoted as $dom(\texttt{X})$. Let t be the time needed to verify one value of `X`, then the brute force attack for cracking a key will take $|dom(\texttt{X})| * t$ time.

One way of determining the upper bound of $|dom(\texttt{X})|$ is based on the number of bits of `X`. For example, if `X` is an 32-bit integer, the brute force attack may take up to $2^{32}t$ time. Generally, if `X` has n bits, the attack needs $2^n t$ time. Thus, an obfuscated condition that depends on a string variable tends to be more resistant than a condition that involves a boolean variable. To reduce the search time, attackers may attempt to apply *rainbow attacks*, which use a precomputed table for reversing hash functions. However, it is well known that such attacks can be defeated by mixing a unique plaintext *salt* (for each bomb) into the hash computation.



(a) Single-trigger bomb **(b)** Double-trigger bomb

Figure 2. Two types of logic bombs. We used double-trigger bombs in our implementation.

6 Enhancement: Double-trigger Bombs

As symbolic execution, which is commonly used in whitebox fuzzing for a high code coverage, cannot be used to "crack" our bombs, we estimate that attackers may invest more on blackbox fuzzing, e.g., by renting cloud services to run multiple fuzzers concurrently for a prolonged period of time. Knowing that blackbox fuzzing is inefficient in analyzing bombs, to make our defense even more resilient we propose *double-trigger bombs*. But note that the security of our system takes it as an enhancement (rather than a must).

Fig. 2a shows the structure of a single-trigger bomb presented above (in Section 3.2), while Fig. 2b shows a double-trigger bomb structure. In a double-trigger bomb, an extra *environment-sensitive* inner trigger condition is inserted and the logic bomb is activated only if *both* trigger conditions are met. In a double-trigger bomb, both the inner trigger condition and payload (i.e., the repackaging detection and response code) are encrypted.

With double-trigger bombs, we can finely exploit the sharp differences between the attacker side (who runs apps in a limited number of different environments) and the very diverse user side. While the outer trigger condition is satisfied only if the control flow reaches the trigger condition with `X` equals to `c` (Line 1 in Listing 3), the inner trigger condition is met only if the app runs on a device with some specific environment in terms of the system build number, IP address, GPS location, etc. Given the huge number of possible environment variable values, attackers have to invest enormously to *blindly* keep trying different combinations of environments for triggering logic bombs. As another example, a bomb can be constructed such that it sets off only if the app is played at some specific time. Thus, running an app for a longer time does not necessarily trigger it.

The inner trigger condition is a quantifier-free first-order-logic formula consisting of one or more constraints, which are concatenated by `&&` or `||`; each constraint is in the form of "`f(env) op r`", where `op`$\in \{<, >, ==, !=\}$, `r` is a constant

value, and `f()` is a function of `env`. Below are some example environment variables that can be used in an inner condition.

- Hardware environment and status. Different user devices have different manufacturers, boards, boot loader versions, brand names, CPU types, display metrics, MAC addresses, serial numbers, flash sizes, etc.
- Software environment, e.g., SDKs, API levels, OS versions, IP addresses, etc.
- Time and sensors. A trigger condition can be constructed based on time and sensor information, such as GPS, light, and temperature.

7 Implementation

We leverage `Javassist` [23] for instrumenting bytecodes. Additional code for realizing our approach comprises 6,354 lines of Java code measured with CLOC [9].

7.1 Candidate Methods

Given an app, in order to avoid a high overhead, we first use profiling to find *hot* methods, i.e., the most frequently invoked ones, and exclude them from instrumentation; the rest are *candidate methods* used to insert logic bombs. Specifically, we first use Dynodroid [32] to generate a random stream of 10,000 user events and feed them to the app. Meanwhile, we use Traceview [44] to log the execution trace, which includes the invocation count of each method. The top 10% most frequently invoked methods are considered as hot methods; the rest methods are candidate methods.

7.2 Outer Trigger Conditions

BombDroid first searches *existing* qualified conditions and then constructs *artificial* ones for building bombs.

Existing qualified conditions. We use Soot [42] to generate the CFG of each candidate method and locate all qualified conditions that include equality checking; specifically, we search for instructions containing `IFEQ`, `IFNE`, `IF_ICMPEQ`, `IF_ICMPNE`, and `TABLESWITCH`. As a heuristic optimization, we avoid inserting bombs into loops in a procedure.

Logic operations such as `&&` or `||` combine more than one simple condition. For logical `and` operators, e.g., `if (X == a && Y == b)`, as both simple conditions (`X == a`, and `Y == b`) must be satisfied to execute conditional code, either of the two can serve as the outer condition.

For logical `or` operators, e.g., `if (X == a || Y == b)`, as showed below, since either of the two simple conditions may execute conditional code, we duplicate the conditional code and use `if...else if` constructs; this way, the payload can be injected into either of the two branches.

```
if ( X == a || Y == b ) {          ⟶   if ( X == a ) {
   foo ();                                 foo ();
}                                      } else if ( Y == b ) {
                                           foo ();
                                       }
```

Artificial qualified conditions. $\alpha = 0.25$ (α is configurable) of the candidate methods are randomly picked for inserting artificial qualified conditions. In each selected method, a program location that is not in a loop of the method is randomly chosen for inserting an artificial QC. At each selected program location, we collect the possible values that each accessible field takes through profiling; fields that have the largest numbers of unique values are considered to have higher entropies and are used to construct artificial QCs. To construct an artificial QC, one of the field values is randomly selected as the constant value.

7.3 Inner Trigger Conditions

Inner trigger conditions depend on environment variables. We collect information about environment variables from online resources. E.g., the Android official website maintains a Dashboards page, which provides information about the ratio of devices that share some property in terms of the OS version, API level, or screen size [14]; AppBrain provides statistics about manufacturers [43]. We also allow developers to override the collected information; for example, an app may be used on tablets only or target users at specific countries.

Base on the information collected, we construct inner trigger conditions, each of which will be satisfied at some specific probability p. The probability range is customizable by developers; in our implementation, $p \in [0.1, 0.2]$. For example, when building an inner trigger condition that depends on the IP address $A.B.C.D$, the condition $101 < C < 132$ has $p = 30/256$. Note that it does not mean that, given a device, when an inner trigger condition is evaluated for $1/p$ times on the device the bomb will be activated once; instead, the bomb may never be activated on that device until the environment condition is met. But when the condition is evaluated for $1/p$ times under the *diverse* user environments, it is expected the bomb gets activated once.

7.4 Hash, Encryption, Obfuscations, and Repackaging Detection

We use SHA-128 as the hash function and AES-128 for encryption. We use `key = Hash(c|S)`, where `S` is a salt, to transform a constant value `c` with a various size into a uniform 128-bit key to be used for encryption. Our current prototype implemented the repackaging detection method based on public-key comparison; as this work is focused on presenting the new logic bomb structure and its effectiveness in concealing bombs on the attacker side, and we leave the inclusion of other detection methods for future work.

7.5 Bytecode Instrumentation

We first use `apktool` [2] to unpack the APK file to generate `classes.dex` and `CERT.RSA` (the latter is used to extract the original public key using `openssl`). We then use `dex2jar` [16] to convert `classes.dex` to a collection of java classes, which are used for instrumentation.

We leverage `Javassist` [23] for bytecode instrumentation. Javassist allows us to write the repackaging detection code in Java, and it then compiles the code on-the-fly during instrumentation. The generated code, after being mixed with part of the original code, is encrypted into a string, which is inserted into the app code (during execution time, when a

Table 1. Static characteristics.

Category	# of apps	Avg LOC	Avg # of candidate methods	Avg # of exist. qualified conditions	Avg # of env. var.
Game	105	3,043	95	56	16
Science&Edu.	98	4,046	86	44	8
Sport&Health	87	5,467	113	40	11
Writing	149	7,099	149	67	6
Navigation	121	9,374	185	52	9
Multimedia	108	10,032	203	72	17
Security	152	11,073	242	86	12
Development	143	14,376	373	93	11

bomb is triggered, the string will be decrypted and stored in a separated `.dex` file, which is then loaded and invoked; note that ART supports dynamic loading of `.dex` files). *There definitely exist other ways to implement the system. For example, the encrypted code can also be inserted into the* `.data` *section of a shared library of the app.* To make BombDroid work for Android apps, during instrumentation, `android.jar` is included into the build path, and all the packages that payloads relying on such as `PackageManager` are imported. Finally, after instrumentation, we use `dx` to convert the modified Java classes into a new `classes.dex`.

8 Evaluation

We have applied BombDroid to a set of Android apps, 963 totally, downloaded from F-Droid [17]. We first present the static characteristics of the apps, and then describe the evaluation results about effectiveness, resilience, and side effects.

8.1 App Program Characteristics

Table 1 shows the static characteristics of the apps. For each category, it shows the number of apps, the average number of lines of Java code, the average number of candidate methods, the average number of existing QCs, and the average number of environment variables used by apps.

All the apps use environment variables, and apps in the Multimedia category use the most of them on average. Larger sized programs tend to have a larger number of candidate methods and existing qualified conditions. Note that BombDroid allows to insert artificial QCs.

To construct artificial QCs, program variables are used. As an example, we visualize how program variables of `AndroFish` change their values with time in Figure 3. In the main interface, multiple fishes move around, and players need to click these fishes to gain scores. The six program variables in Figure 3 store different information of the currently visible fish, such as its moving direction, width, height, speed, and position. We use Dynodroid [32] to run the app for an hour, and record program variable values once per minute. It shows that while some variables have many unique values, others take few different values. We choose those with the largest numbers of unique values to construct more resilient artificial qualified conditions.

Table 2 shows the number of injected bombs in eight randomly selected apps from each of the eight categories. For the sake of consistency, we use the eight apps to demonstrate

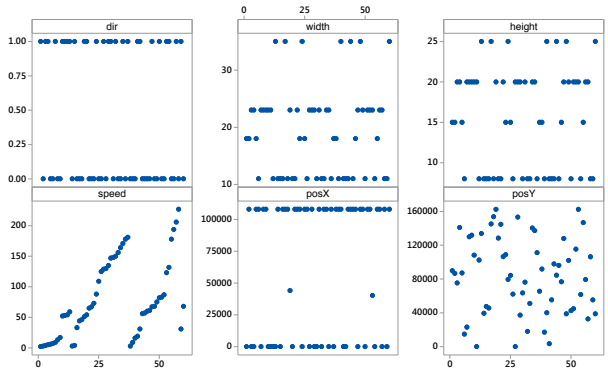


Figure 3. Visualization of how the values of six program variables of AndroFish vary with time. The x-axis and y-axis represent the time (mins) and variable values, respectively.

Table 2. Injected logic bombs.

App	# of logic bombs injected	# of existing qualified conditions	# of artificial qualified conditions
AndroFish	67	36	31
Angulo	43	25	18
SWJournal	58	28	30
Calendar	104	63	41
BRouter	263	144	119
Binaural Beat	82	52	30
Hash Droid	65	37	28
CatLog	73	35	38

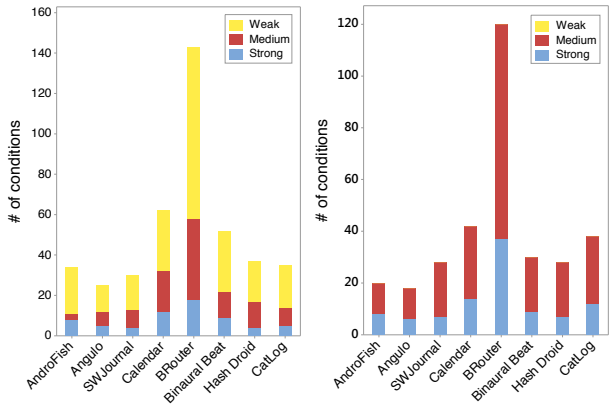
Table 3. Triggering the first logic bombs.

App	Min time (sec)	Max time (sec)	Avg time (sec)	Success times
AndroFish	12	213	89	50/50
Angulo	17	778	125	50/50
SWJournal	8	369	93	50/50
Calendar	11	452	136	50/50
BRouter	23	590	142	50/50
Binaural Beat	9	241	75	50/50
Hash Droid	17	436	158	50/50
CatLog	26	522	164	50/50

all the evaluation results in the rest of the section. Take `AndroFish` as an example; 67 bombs are injected into the app totally, consisting of 36 bombs based on existing qualified conditions and 31 on artificial ones.

8.2 Effectiveness

We then measure how soon repackaging detection is performed when users run an app; i.e., how long it takes to trigger the first logic bomb. We use BombDroid to embed logic bombs into the eight apps and repackage them. We let four human testers play the repackaged apps; each tester players two apps on emulators. Each app is played until the first logic bomb is triggered, and the time taken to trigger the first bomb is recorded; each app is measured for 50 times, and the testers are asked to vary the emulator configurations (device types, SDK versions, and CPU/ABI, etc.) between the runs. The minimum/maximum time to trigger the first logic



(a) Existing qualified conditions **(b)** Artificial qualified conditions

Figure 4. Strength of outer trigger conditions.

bomb and the number of successful detection times (if no bomb is triggered within 60 minutes it is considered as a failure) are listed in Table 3.

The results are encouraging showing that victim users can be quickly alerted when using a repackaged app. The response time is as short as 8 seconds, while the maximum times to trigger the first bombs are all within 13 minutes.

8.3 Resilience

Security analysis (see Section 5) of our approach has shown that two types of attacks are more effective than others: brute force attacks and fuzzing. We thus evaluated the resilience to these attacks.

8.3.1 Resilience to Brute force Attacks

Given an outer trigger condition `Hash(X) ==` H_c, attackers may try to search the value of `X` from *dom*(`X`) that satisfies the condition. To evaluate the resistance, we define three levels of strength based on the type of the data used in the condition. An obfuscation is considered *strong*, *medium*, or *weak* if the QC depends on string, integer, or boolean constant values, respectively. Figure 4 shows the analysis results on the eight apps. Figure 4a shows that a high percentage of the existing QCs have a weak obfuscation. Figure 4b shows that the artificial QCs all have medium to strong obfuscations. Note that the number of artificial qualified conditions are adjustable, so developers can insert more artificial ones if they can afford a larger overhead.

8.3.2 Resilience to Fuzzing and Human Analysis

We first measure the number of *outer trigger conditions* satisfied during one hour analysis by state-of-the-art Android fuzzing tools, including Monkey [46], PUMA [21], AndroidHooker [1], and Dynodroid. Table 4 shows the results. It can be observed that Dynodroid performed slightly better.

We then look into the number of *logic bombs* triggered (when *both* trigger conditions were met) by Dynodroid. Figure 5 shows the results; each line corresponds to the percentage of triggered bombs for each app. In the first 5 minutes

Table 4. Percentages of satisfied outer trigger conditions (AH represents AndroidHooker).

App	Monkey	PUMA	AH	Dynodroid
AndroFish	28.4	31.3	32.8	35.8
Angulo	30.2	34.8	30.2	37.2
SWJournal	27.7	31.0	29.3	34.5
Calendar	31.7	35.6	33.7	38.5
BRouter	19.4	22.1	20.9	26.6
Binaural Beat	24.4	26.8	26.8	34.1
Hash Droid	29.2	33.8	32.3	38.5
CatLog	26.0	27.4	30.1	38.4

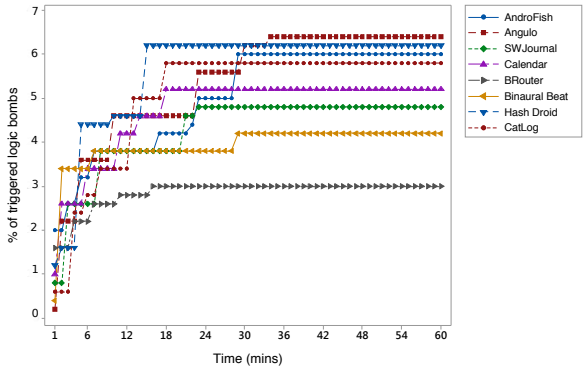


Figure 5. Number of bombs triggered by Dynodroid in one hour.

Table 5. Execution time overhead.

App	T_a (sec)	T_b (sec)	Overhead (%)
AndroFish	124	126	1.7
Angulo	125	128	1.6
SWJournal	115	118	2.6
Calendar	148	151	2.2
BRouter	132	135	1.8
Binaural Beat	163	166	1.9
Hash Droid	155	157	1.4
CatLog	131	134	2.3

a small number of bombs are triggered, and the growth of the numbers slows down quickly. After 35 minutes, no new bombs are triggered. At most 6.4% bombs are triggered, meaning the majority of bombs keep dormant and the apps are resilient to such attacks.

We next let four human analysts to manually run the apps in order to trigger the survived bombs. They are skilled in debugging and test input generators. Each analyst took care of two apps and spent 20 hours on each one. They are informed of the detailed implementation of BombDroid, and allowed to apply any tools to assist investigation and mutate environment variables' values. The results show that at most 9.3% bombs are triggered. Mutating environment variables values is slightly helpful to trigger more bombs. However, *considering that there are so many possible environment variables and each may have a large domain, attacker cannot configure the environments in a guided way.*

8.4 Side Effects

The side effects of BombDroid on apps are measured in the following three aspects: false positives, code size change, and execution time overhead.

As the response code injects errors into program execution, it is critical to ensure that such response code is never executed on apps that have not been repackaged, i.e., ensuring zero false positives. We thus run Dynodroid on each app protected by BombDroid for ten hours, and log whether the response code is executed. No false positives are triggered.

The code size increase ranges from 8% to 13% and averages 9.7% among all the apps. To evaluate the execution time overhead, we employ Dynodroid to generate a sequence of 20,000 user events, and feed the same user events to both the original and protected apps fifty times to measure the average execution time. The average execution time of the original app and protected app are denoted as T_a and T_b respectively; the execution time overhead is calculated as $O = (T_b - T_a)/T_a$. Table 5 shows the results (2.6% overhead at most). It can be seen that the overhead is very small. We attribute the small overhead to three reasons: (1) the logic bombs are not injected into hot methods, (2) the payloads are not executed until the conditions are met, and (3) the code decryption is one-time effort by caching it in memory.

9 Related Work

9.1 Logic Bombs

Various approaches have been proposed for discovering trigger-based behaviors [3, 5, 6, 11, 19, 26, 35, 37, 41, 47, 57]. `Bitscope` uses static analysis and symbolic execution to understand the behavior of malware binaries [5]. `MineSweeper` utilizes binary instrumentation and mixed concrete and symbolic execution for detecting trigger-based behavior [6]. By combining symbolic execution, path predicate reconstruction, and control-dependency analysis, `TriggerScope` can identify time-, location-, and SMS-related triggers in apps [19]. Our obfuscation on trigger conditions makes the path constraints unresolvable. `HSOMINER` [35] combines machine learning and program analysis to discover hidden sensitive operations in apps; however, it cannot handle trigger conditions that involve program variables as in our bombs.

Some techniques try to *circumvent trigger conditions and directly execute payloads*. Rasthofer et al. propose `HARVESTER`, which performs backward program slicing starting from the line of suspected code, and then executes the extracted slices to uncover the payload behavior [37]. Wilhelm and Chiueh propose a forced sampled execution approach that forces execution along different paths [47]. As BombDroid applies encryption on payloads, it is infeasible to directly execute payload without discovering the key used for decryption.

9.2 Repackaging Detection

Different app repackaging detection techniques use different features and methods for comparing app code [12, 13, 20, 29, 30, 36, 53, 59, 60]. For example, `DroidMOSS` uses hashing of app instruction sequences to detect repackaging [60]. Potharaju et al. use program syntactic fingerprints to detect plagiarized apps [36]. Chen et al. use the program dependency graphs to detect repackaging [8]. `AppInk` [58] and `DroidMarking` [39] inject watermarking into apps so that a trusted party with the knowledge of watermarking can help detect repackaging. `AnDarwin` detects similar apps using the semantic information [13]. Most of them rely on a centralized effort, and the detection techniques can be easily evaded by obfuscations. `SSN` [28] attempts to build repackaging detection into the app code, but as detailed in Section 2.1, it is vulnerable to a variety of attacks. BombDroid implements the first *resilient* decentralized repackaging detection.

10 Discussion

Ethical issues. First, when users use a repackaged app protected by our system, they experience crashes, slowdown, and other negative user experiences. We consider this acceptable, as causing negative user experiences is a common practice for handling pirated software in industry. Second, user devices are made use of to detect repackaging. This should not be a problem if we regard repackaging detection as helping users check potentially harmful apps.

Limitations. It is widely recognized that any software-based protection can be bypassed as long as attackers are determined enough, which is also true with BombDroid. We assume attackers are interested in repackaging apps only if it is cost-effective, e.g., when the cost of repackaging is less than that of developing apps from scratch. For example, although BombDroid has good resiliency to many attacks, we do not guarantee that determined attackers cannot crack the keys of bombs through, say, brute force attacks. As we inject artificial qualified conditions, which have medium to strong obfuscation strength, it is difficult to enumerate all possible values. But understanding the semantics of the branch conditions can help reduce the number of possible values and assist attackers to guess the keys.

Future work. We plan to improve the prototype by adding other detection methods, such as code snippet scanning, which does not rely on any Android APIs to detect code modifications. An interesting direction is to explore how to mute other bombs strategically once a bomb is triggered, so that even more bombs can survive. Finally, we plan to apply *custom packers* [45] widely used in virus to our logic bombs.

11 Conclusion

Building repackaging detection into apps brings many advantages over the centralized scheme, but how to make the detection code resilient to various adversary analysis is not resolved in prior work. We creatively propose to use bombs to conceal repackaging detection code. Cryptographically obfuscated bombs are used to construct resilient bombs, while double-trigger bombs achieve finer control of the triggering of bombs. We have built a prototype and evaluated it. The evaluation results show that the technique is efficient, effective and resilient. We expect it to benefit numerous honest Android app developers.

Acknowledgments

We would like to thank anonymous reviewers for their invaluable and constructive comments.

References

[1] AndroidHooker. 2016. https://github.com/AndroidHooker.

[2] Apktool. 2017. https://ibotpeaches.github.io/Apktool/.

[3] Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2010. Efficient Detection of Split Personalities in Malware. In *NDSS*.

[4] Zvika Brakerski and Guy N Rothblum. 2014. Virtual Black-Box Obfuscation for All Circuits via Generic Graded Encoding.. In *TCC*, Vol. 8349. 1–25.

[5] David Brumley, Cody Hartwig, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, Dawn Song, and Heng Yin. 2007. BitScope: Automatically dissecting malicious binaries. In *Tech. Rep. CMU-CS-07-133*.

[6] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. 2008. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*.

[7] Hoi Chang and Mikhail J. Atallah. 2002. Protecting Software Code by Guards. In *Security and Privacy in Digital Rights Management*.

[8] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *ICSE*.

[9] CLOC—Count Lines of Code. 2013. http://cloc.sourceforge.net/.

[10] Valerio Costamagna and Cong Zheng. 2016. ARTDroid: A Virtual-Method Hooking Framework on Android ART Runtime.. In *IMPS@ ESSoS*.

[11] Jedidiah R. Crandall, Gary Wassermann, Daniela AS de Oliveira, Zhendong Su, S. Felix Wu, and Frederic T. Chong. 2006. Temporal search: Detecting hidden malware timebombs with virtual machines. In *ACM Sigplan Notices*.

[12] Jonathan Crussell, Clint Gibler, and Hao Chen. 2012. Attack of the Clones: Detecting Cloned Applications on Android Markets. In *ESORICS*.

[13] Jonathan Crussell, Clint Gibler, and Hao Chen. 2013. AnDarwin: Scalable Detection of Semantically Similar Android Applications. In *ESORICS*.

[14] Dashboards. 2017. http://developer.android.com/about/dashboards/index.html.

[15] Key derivation function. 2017. https://en.wikipedia.org/wiki/Key_derivation_function.

[16] dex2jar. 2017. https://github.com/pxb1988/dex2jar.

[17] F-Droid. 2017. Free and Open Source Software Apps for Android. https://f-droid.org/.

[18] Parvez Faruki, Vijay Ganmoor, Vijay Laxmi, M. S. Gaur, and Ammar Bharmal. 2013. AndroSimilar: Robust statistical feature signature for Android malware detection. In *SIN*.

[19] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2016. TriggerScope: Towards Detecting Logic Bombs in Android Applications. In *S&P*.

[20] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. 2013. Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications. In *DIMVA*.

[21] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. 2014. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*.

[22] Heqing Huang, Sencun Zhu, Peng Liu, and Dinghao Wu. 2013. A framework for evaluating mobile app repackaging detection algorithms. In *Trust and Trustworthy Computing*.

[23] Javassist. 2017. http://jboss-javassist.github.io/javassist/.

[24] Taeyeon Ki, Alexander Simeonov, Bhavika Pravin Jain, Chang Min Park, Keshav Sharma, Karthik Dantu, Steven Y Ko, and Lukasz Ziarek. 2017. Reptor: Enabling API Virtualization on Android for Platform Openness. In *MobiSys*.

[25] Shuang Liang and Xiaojiang Du. 2014. Permission-combination-based scheme for android mobile malware detection. In *Communications (ICC), 2014 IEEE International Conference on*. IEEE.

[26] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. 2011. Detecting environment-sensitive malware. In *International Workshop on Recent Advances in Intrusion Detection*.

[27] Martina Lindorfer, Stamatis Volanis, Alessandro Sisto, Matthias Neugschwandtner, Elias Athanasopoulos, Federico Maggi, Christian Platzer, Stefano Zanero, and Sotiris Ioannidis. 2014. AndRadar: fast discovery of android applications in alternative markets. In *DIMVA*.

[28] Lannan Luo, Yu Fu, Dinghao Wu, Sencun Zhu, and Peng Liu. 2016. Repackage-proofing Android Apps. In *DSN*.

[29] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM.

[30] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering* (2017).

[31] Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao, Min Yang, Xinyu Xing, and Peng Liu. 2017. System Service Call-oriented Symbolic Execution of Android Framework with Applications to Vulnerability Discovery and Exploit Generation. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 225–238.

[32] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *FSE*.

[33] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Exploring multiple execution paths for malware analysis. In *S&P*.

[34] Ravshanbek Norboev, Zakia Hossain, Lannan Luo, and Qiang Zeng. 2017. *On the Robustness of Stochastic Stealthy Network against Android App Repackaging*. Technical Report. Temple University.

[35] Xiaorui Pan, Xueqiang Wang, Yue Duan, XiaoFeng Wang, and Heng Yin. 2017. Dark Hazard: Learning-based, Large-scale Discovery of Hidden Sensitive Operations in Android Apps. In *NDSS*.

[36] Rahul Potharaju, Andrew Newell, Cristina Nita-Rotaru, and Xiangyu Zhang. 2012. Plagiarizing smartphone applications: attack strategies and defense techniques. In *In Engineering Secure Software and Systems*.

[37] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting runtime values in android applications that feature anti-analysis techniques. In *NDSS*.

[38] Remote Application Removal Feature. 2010. http://android-developers.blogspot.com/2010/06/exercising-our-remote-application.html.

[39] Chuanggang Ren, Kai Chen, and Peng Liu. 2014. Droidmarking: Resilient Software Watermarking for Impeding Android Application Repackaging. In *ASE*.

[40] Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. 2008. Impeding Malware Analysis Using Conditional Code Obfuscation. In *NDSS*.

[41] Chengyu Song, Paul Royal, and Wenke Lee. 2012. Impeding Automated Malware Analysis with Environment-sensitive Malware.. In *HotSec*.

[42] Soot. 2017. http://sable.github.io/soot/.

[43] Top Manufacturers. 2017. http://www.appbrain.com/stats/top-manufacturers.

[44] Traceview. 2017. http://developer.android.com/tools/help/traceview.html.

[45] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. 2015. SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 659–673.

[46] UI/Application Exerciser Monkey. 2017. http://developer.android.com/tools/help/monkey.html.

[47] Jeffrey Wilhelm and Tzi cker Chiueh. 2007. A forced sampled execution approach to kernel rootkit identification. In *International Workshop on Recent Advances in Intrusion Detection*.

[48] Maurice Vincent Wilkes. 1972. Time-sharing computer systems. (1972).

[49] Longfei Wu, Xiaojiang Du, and Xinwen Fu. 2014. Security threats to mobile multimedia applications: Camera-based attacks on mobile

phones. *IEEE Communications Magazine* 52, 3 (2014).

[50] Longfei Wu, Xiaojiang Du, and Jie Wu. 2014. MobiFish: A lightweight anti-phishing scheme for mobile phones. In *Computer Communication and Networks (ICCCN), 2014 23rd International Conference on*. IEEE.

[51] Rubin Xu, Hassen Saïdi, and Ross Anderson. 2012. Aurasium: Practical Policy Enforcement for Android Applications. In *USENIX Security*.

[52] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*.

[53] Shengtao Yue, Weizan Feng, Jun Ma, Yanyan Jiang, Xianping Tao, Chang Xu, and Jian Lu. 2017. RepDroid: an automated tool for Android application repackaging detection. In *Proceedings of the 25th International Conference on Program Comprehension*. IEEE Press.

[54] Chao Zhang, Dawn Song, Scott A Carr, Mathias Payer, Tongxin Li, Yu Ding, and Chengyu Song. 2016. VTrust: Regaining Trust on Virtual Calls.. In *NDSS*.

[55] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. 2014. ViewDroid: Towards Obfuscation-Resilient Mobile Application Repackaging Detection. In *WiSec*.

[56] Yulong Zhang. 2015. Kemoge: Another Mobile Malicious Adware Infecting Over 20 Countries. https://www.fireeye.com/blog/threat-research/2015/10/kemoge_another_mobi.html.

[57] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. 2012. SmartDroid: an Automatic System for Revealing UI-based Trigger Conditions in Android Applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*.

[58] Wu Zhou, Xinwen Zhang, and Xuxian Jiang. 2013. AppInk: watermarking android apps for repackaging deterrence. In *ASIA CCS*.

[59] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. 2013. Fast, Scalable Detection of "Piggybacked" Mobile Applications. In *CODASPY*.

[60] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. 2012. Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces. In *CODASPY*.

[61] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android Malware: Characterization and Evolution. In *S&P*.