Resilient User-Side Android Application Repackaging and Tampering Detection Using Cryptographically Obfuscated Logic Bombs

Qiang Zeng, *Member, IEEE,* Lannan Luo*, *Member, IEEE,* Zhiyun Qian, *Member, IEEE,* Xiaojiang Du, *Senior Member, IEEE,* Zhoujun Li, Chin-Tser Huang, *Senior Member, IEEE,* and Csilla Farkas, *Member, IEEE*

Abstract—Application repackaging is a severe threat to Android users and the market. Not only does it infringe on intellectual property, but it is also one of the most common ways of propagating mobile malware. Existing countermeasures mostly detect repackaging based on app similarity measurement, which tends to be imprecise when obfuscations are applied to repackaged apps. Moreover, they rely on a central party, typically the hosting app store, to perform the detection, but many app stores fail to commit proper effort to piracy detection. We consider building the application repackaging 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 the detection code from being manipulated by attacks*. We propose a creative use of *logic bombs*, which are otherwise regularly used in malware. The *trigger conditions* of bombs 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 the user side. The detection code, which is part of the bomb *payload*, is executed only if the bomb is activated. We introduce *cryptographically obfuscated logic bomb* to enhance the bomb: (1) the detection code is *woven* into the neighboring original app code, (2) the mixed code gets encrypted using a key, and (3) ***the key is deleted from the app and can only be derived when the bomb is activated***. Thus, attacks that try to modify or delete the detection code will corrupt the app itself, and searching the key in the application will be in vain. Moreover, we propose a *bomb spraying* technique that allows many bombs to be injected into an app, multiplying the needed adversary effort for bypassing the detection. In addition to repackaging detection, we present application tampering detection to fight attacks that insert malicious code into repackaged apps. We have implemented a prototype, named BOMBDROID, that builds repackaging and tampering detection into apps through bytecode instrumentation. The evaluation and the security analysis show that the technique is effective, efficient, and resilient to various bomb analysis techniques including fuzzing, symbolic execution, multi-path exploration, and program slicing. Ethical issues due to the use of logic bombs are also discussed.

Index Terms—Android app repackaging, tamper-proofing, logic bombs.

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. Over $14 billion of revenue loss is caused by app piracy each year [2]. The app repackaging 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 users' security and privacy [3], [4], [5], [6], [7], [8], [9]. Previous research showed that 86% of 1260 malware samples were repackaged from legitimate apps [10]. For example, the malicious adware family, Kemoge, which infected victims from more than 20 countries, disguised itself as popular apps via repackaging [11].

Because of the importance of the problem, many repackaging detection techniques have been proposed. Most of them are based on app similarity comparison [3], [12], [13], [14], [15], [16], and, thus, tend to be imprecise when obfuscations are applied to repackaged apps. Besides, they usually rely on a central party, typically the app store, to conduct detection; there are a plethora of alternative app markets, but their quality and commitment in repackaging detection are questionable [17]. Finally, users may download apps from places other than any markets (such as FTP and BitTorrent) and install them, bypassing the centralized detection. Due to these limitations, numerous repackaged apps escape detection and get installed on user devices [3].

We consider a decentralized detection scheme that adds repackaging detection capability into the app being protected.

* *Correponding author.*

- *Q. Zeng is with the Department of Computer Science and Engineering, University of South Carolina, Columbia, SC 29208.*
- *L. Luo is with the Department of Computer Science and Engineering, University of South Carolina, Columbia, SC 29208.*
- *Z. Qian is with the Department of Computer Science and Engineering, University of California Riverside, CA 92521.*
- *X. Du is with the Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122.*
- *Z. Li is with the Department of Computer Science, Beihang University, Beijing, China 100191.*
- *C. Huang is with the Department of Computer Science and Engineering, University of South Carolina, Columbia, SC 29208.*
- *C. Farkas is with the Department of Computer Science and Engineering, University of South Carolina, Columbia, SC 29208.*

This manuscript is an extension of the conference version published in the *Proceedings of International Symposium on Code Generation and Optimization (CGO 2018)* [1]. This manuscript adds the presentation of the new component for application tampering detection, and extends the system design, security analysis, and evaluation.

The repackaging detection becomes an inherent capacity of the app and, hence, does not rely on a third party. *The main challenge obviously is how to protect the repackaging detection capacity from attacks.* While the goal of decentralized detection is highly desired and some attempts have been made towards it, that challenge is not solved. For example, a state-of-the-art defense, SSN [18], proposes to conduct repackaging detection at a very low probability to hide the detection nodes. However, such probabilistic computation (based on the return value of `rand()`) can be turned deterministic through code instrumentation. Plus, SSN tries to conceal certain API calls (mainly `getPublicKey()`) through *reflection*. But by inserting code that checks the reflection call destination, all those calls can be revealed and manipulated. Actually, SSN can be bypassed in multiple other ways (detailed in Section 2.2). The failed attempts also show how difficult it is to conquer the challenge.

Our protection techniques are based on our observations that the attacker side is very different from the user side, which is mainly reflected in the following two observations:

O1 **Inputs and environments.** The inputs, hardware/software environments and sensor values are very diverse on the user side, while the attacker can only *afford* the time and money to run the app under a limited number of environments [19]. With regard to inputs, although the attacker can feed the program execution with numerous inputs, they do not necessarily trigger the execution of a large number of unique execution paths [20]. This leads to our another observation.

O2 **Code coverage.** The attacker typically can only *afford* to analyze a small portion of the app, while users, with time, play almost every part of the app.

Such observations are consistent with some well-known challenges in software testing [21]. That is, even with significant time and effort invested, a commercial program typically can only be tested under a small number of environments and inputs compared to the user side. In practice, it is very difficult and expensive to achieve a high code coverage [22]. We propose to exploit the differences, which otherwise are deemed root causes of "challenges" in software testing, to protect the inserted repackaging detection capacity from attacks. To that end, we propose a creative use of *logic bombs*, which are normally used in malware, and the following two design strategies.

- **Exploitation of O1.** 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 app runs with a specific input or at a specific GPS location (we assume attackers can forge fake GPS values, and our system is resilient to such attacks). While it may be costly for an attacker to trigger a given bomb, it is actually free to rely on users who play the app to activate it.
- **Exploitation of O2.** The 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.

To protect the logic bombs, we additionally apply the following enhancements. First, each logic bomb is encrypted and, more importantly, the decryption key 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* (detailed in Section 4.2). As 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 original app code before being encrypted, such that attacks that simply delete suspicious code will corrupt the app execution. In short, through these design strategies, we can achieve the following three goals:

G1 **Resilience.** The user-side detection avoids single points of failure and is resilient to attacks.

G2 **Security through keys.** The strength of the scheme comes from the need of the key for decrypting the bomb payload rather than keeping the algorithm secret.

G2 **Non-stealthy defense.** Unlike conventional software tampering detection techniques that try to conceal the code for integrity checking [18], [23], [24], [25], which is difficult to achieve, we do not hide our bombs but deter attackers from deleting or modifying them.

In addition, we propose techniques for detecting code tampering, which is frequently caused by attackers that insert malicious code and hence implies extraordinary dangers. Plus, the proposed techniques can be generalized to detect tampering of other app files. We have implemented the decentralized detection technique in a system named BOMBDROID, which adds the detection capacity to 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 1,463 Android apps. The evaluation results and security analysis show that the protection provided by BOMBDROID is effective in repackaging detection, resilient to various adversary analysis, and incurs a very small speed overhead (~2.8%). We made the following contributions.

- We present the first *resilient* user-side Android app repackaging and tampering detection technique which builds the detection capacity directly into apps. Logic bombs are creatively used for a benign purpose, that is, protecting the detection capacity from attackers.
- We propose effective measures to enhance the bombs (called *cryptographically obfuscated logic bombs*) based on encryption and code weaving, such that our technique is resilient to various adversary analysis and code modification and deletion attacks. A novel *double-trigger* logic bomb structure is proposed to allow fine control of the trigger condition.
- We have implemented a prototype system, and analyzed and evaluated its effectiveness, resilience, and efficiency.

The rest of the paper is organized as follows. Section 2 describes the background on app signing and logic bombs, and introduces the threat model. Section 3 presents our goal, assumptions, and the system architecture. Section 4 introduces cryptographically obfuscated bombs for protecting the repackaging detection capability from attacks. Section 5

```
1 if(rand() < 0.01) {
2   funName = recoverFunName(obfuscatedStr);
3   // The reflection call invokes getPublicKey
4   currKey = reflectionCall(funName);
5   if(currKey != PUBKEY)
6     // repackaging detected!
7     // response is delayed
8 }
```

Listing 1: A vulnerable design.

presents how we detect app repackaging and code tampering, and Section 5.3 discusses the detection response. Section 6 covers the security analysis on the resilience of BOMBDROID to various evasion attacks. Section 7 introduces double-trigger bombs to further improve the resilience. Section 8 describes the implementation details of BOMBDROID. Section 9 presents the evaluation results. The related work and the discussion are presented in Section 10 and Section 11, respectively. The paper is concluded in Section 12.

2 BACKGROUND AND ADVERSARY MODEL

2.1 Background

Signing Applications. Each developer owns a unique public-private key pair. Before an app is released, it has to be signed using the developer's private key. The process of signing an app calculates the digests of the app files and then generates a signature based on these digests. When an app is installed, the Android system checks the digests and verifies the signature using the public key. Note that this itself does not detect code modification or repackaging, since attackers always re-sign the app upon repackaging, which replaces the digests and signature carried in the app. However, the public key contained in the APK file of the repackaged app is certainly different from the original one. Therefore, it is viable to detect repackaging by comparing the original public key against the one in the app's certificate. Similarly, by comparing with the original digests, we can detect code tampering.

Logic Bombs. A *logic bomb* is a piece of code consisting of a *trigger condition* and a *payload*; when the trigger condition is met, the bomb is *triggered* (or, *activated*) and the payload code gets executed. The input or event that makes the trigger condition satisfied is called a *trigger*. For example, given a time bomb that executes its payload at some specific time, its trigger is the predefined time. A payload usually corresponds to some malicious function, such as formatting a hard drive or sending out private information. While logic bombs have been widely used in writing malware, such as Trojans and worms, we employ them for a defense purpose, that is, constructing countermeasures against app repackaging.

2.2 Adversary Model

SSN's design. To discuss the adversary model in a more concrete way, we first illustrate a *vulnerable* defense designed in SSN [18], shown in Listing 1. The defense can be easily bypassed by attackers. It can be used by legitimate developers during *compile time* to build repackage detection capability into their apps.

Listing 1 illustrates a repackaging detection node inserted by SSN. It detects repackaging by comparing the app's current public key `currKey`, to the original one `PUBKEY`, which is embedded into the code (Line 5); a *hash* is applied to both `PUBKEY` and `currKey`, such that an attacker cannot search the literal value of `PUBKEY` to locate the detection nodes. The `currKey` is retrieved through a call to the Android system service API `getPublicKey`. In order to hide the call from attackers, SSN proposes the following measures. (1) Repackaging is only invoked probabilistically to hide the detection nodes from fuzzing analysis (Line 1). (2) The function name "*getPublicKey*" is obfuscated and the call is issued through *reflection* (Line 4), such that attackers cannot locate the call through text search. (3) Some calls in the app are converted to be issued through reflection calls as well; hence, attacks that delete reflection calls will not work. (4) When repackaging is detected, instead of taking a response immediately, the response is delayed to confuse the adversary analysis.

Next, we present the capabilities of attackers and attacks against our defense. As our technique leverages logic bombs, we consider not only various common attacks but also the state-of-the-art adversary analysis against bombs. We also show how SSN is vulnerable to multiple types of attacks.

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 and transform some normal calls into reflection calls as well, so it is resilient to such attacks. Note that text search for `PUBKEY` (Line 5) will fail, since instead of using the original public key, SSN derives a value from `PUBKEY` using a custom hash to eliminate the literal key value from the code.

API-hooking assisted analysis. 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 code. In particular, an attacker may try to *intercept* 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. The attacker is also free to forge the return values for API calls (such as fake GPS locations) to fool our system to assist his investigation. However, running a protected app in order to trigger all or most of the repackaging nodes is too costly, so we regard SSN rather resilient to such attacks.

Blackbox/greybox fuzzing. An attacker may use fuzzing to run the repackaged app by providing a large number of inputs to trigger as many detection nodes as possible [26], [27], [28]. For every activated node, the attacker can trace back and disable it. To handle such attacks, SSN proposes a "stochastic" mechanism by invoking `rand()` (Line 1). However, by manipulating the return value of `rand()`, attackers can turn the probabilistic activation of detection nodes into deterministic, such that whenever a path containing a detection node is executed, the node can be surely revealed to attackers. That is, the design goal of "stochastic" activation of detection nodes in SSN fails.

Whitebox fuzzing. Various techniques have been proposed to explore execution paths in a program. A dynamic analysis based approach is to *explore multiple paths* during execu-

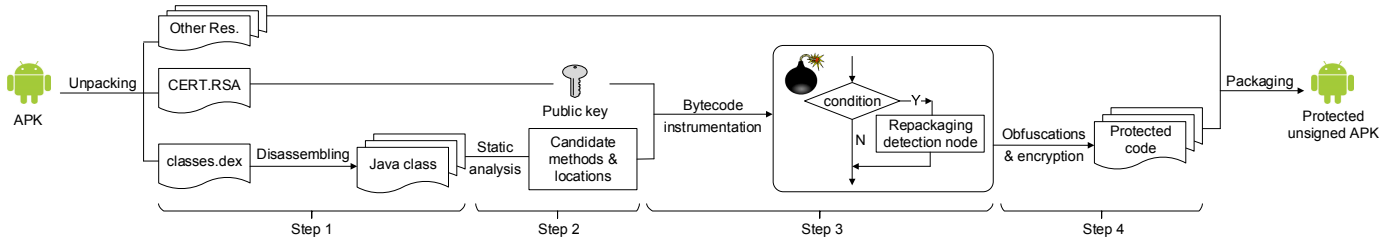


Fig. 1: Architecture of BOMBDROID.

tion [29]. *Symbolic execution* has been widely applied to discovering inputs that execute a program along specific paths [30], [31], [32], [33], [34]. It uses symbolic inputs to explore as many paths as possible, and resolves the corresponding path conditions to find the concrete inputs. *When symbolic execution is applied to SSN, Line 1 cannot stop symbolic executor from exploring (and hence exposing) the path containing repackaging detection.* Note that we also assume attackers can augment conventional fuzzing with selective symbolic execution (such as Driller [35]) to enhance the adversary analysis. Plus, recent research shows that symbolic execution is an effective approach to discovering conditional code and identifying trigger conditions [33].

Backward program slicing. An attacker may simply circumvent trigger conditions and execute payloads directly. Specifically, 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 [36]. Or, the attacker may apply *forced execution* to directly execute the code that is suspected to be payloads [37]. 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 bypass the detection. In SSN, e.g., the attacker can insert code right before a suspicious reflection call to check the destination of that call, i.e., `if(funName=="getPublicKey")`, and modify `funName` at will.

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

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.

3 GOALS, ASSUMPTIONS AND ARCHITECTURE

3.1 Goals and Assumptions

Goals: Our goals are as follows: **(G1)** it should be resilient to *whitebox fuzzing*; **(G2)** it should be resilient to attacks via *text search*, *code instrumentation*, and *backward program slicing*; **(G3)** it should be resilient to *API hooking assisted analysis* and *blackbox/greybox fuzzing*; and **(G4)** it should defeat attacks based on *code deletion*. We divide these attacks into different groups, and will show how different groups of attacks can be defeated by different techniques.

Assumptions: We assume user devices are not in collusion with pirates. Specifically, we assume users devices are not rooted; otherwise, the OS or Android framework on the user device could be modified to mislead our detection. Actually, only 7.6% of Android devices are *rooted* [38], and those device do not necessarily collude with developers of repackaged apps. However, *attackers are allowed to hack and modify their own Android systems arbitrarily to assist adversary analysis.*

3.2 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 [18], which can only work on source code.

The procedure involves the following four steps. (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) BOMBDROID extracts the *public key* from `CERT.RSA`, and selects candidate locations for inserting logic bombs through analysis of the app. (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.

4 LOGIC BOMBS FOR REPACKAGING DETECTION

4.1 Naive Use of Logic Bombs

Listing 2: A naive use of logic bombs.

```
1 if (X == c) {
2   // payload consists of repackaging detection and
      response code
3   payload;
4 }
```

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 naive use of logic bombs is 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 2.2, the piece of code is vulnerable to various attacks. For example, the attacker may modify the bytecode such that the trigger condition is never met; worse, the attacker can circumvent the trigger condition evaluation to analyze and reveal the enclosed code directly [36]. Actually, it shares all the weaknesses of SSN. Thus, a naive use of bombs will not work for our purpose.

4.2 Cryptographically Obfuscated Logic Bombs

Listing 3: A cryptographically obfuscated logic bomb.

```
1 if(Hash(X) == Hc) // Hc= Hash(c)
2   // payload is encrypted and can only be decrypted
       when X=c
3   p = decrypt(encrypted_payload, X);
4   execute(p);
5 }
```

To defeat such attacks, we present a cryptographically obfuscated logic bomb structure. 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 [39], which stores user passwords as hash values [40], 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* [41] and concealing malware [42]. Their work confirms the security of such transformations.

4.3 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 limitation can be resolved by inserting *artificial qualified conditions*: given a program location L, a program 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. Based on this technique, we can perform *bomb spraying*; i.e., inserting many bombs into apps and thus significantly increasing the needed efforts of the adversary trying to bypass the detection.

If "ϕ" can take very few possible values, e.g., it is a boolean expression, it is trivial for attackers to guess the key with a few tries. However, for artificial QCs, We can select "ϕ" with a large $\in dom(\phi)$. This is discussed in Section 6.2.

4.4 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 into the form of logic bombs, which we call *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.

It is clear that the repackaging detection code is part of the code that looks very suspicious. *Instead of hiding the protection code, our technique deters attackers from deleting the suspicious code.*

5 Detection and Response

Repackaging and tampering detection as well as the response are inserted as the payload of logic bombs. Section 5.1 and Section 5.2 present how to detect repackaging and code tampering, respectively. How to construct the response is described in Section 5.3.

5.1 Detection of Repackaging

As in SSN [18], 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`.

5.2 Detection of Code Tampering

5.2.1 Motivation

In the process of repackaging apps, attackers may insert malicious code into apps to launch various attacks. Previous research showed that 85% of malware propagated in the form of repackaged apps [10]. So we regard *repackaged apps with code modifications as extraordinary dangerous.* Android markets should *promptly* take down such apps, while users should be informed *clearly* if the devices contain such apps. Thus, it would be beneficial if we can not only detect repackaged apps, but also make more fine-grained detection to determine whether the repackaged apps' code has been modified.

5.2.2 Main Idea

Conventional code-tampering detection is mostly built on code-snippet scanning [43]. The advantage is that the code scanning can conveniently masquerade itself as normal memory reading, since the `text` section is mapped into the address space. However, this approach does not work well for Android apps: instead of being mapped to memory, their bytecode is *re-compiled* into native code upon installation, so *it is not portable to estimate the checksum of the resulting native code*. Plus, it is difficult to scan memory stealthily in Java programs due to its type-safety enforcement.

App signing (Section 2) generates a folder, `META-INF`, consisting of three files; one of them is `MANIFEST.MF` containing the digest of each app file. That is, each file in the app has a digest stored as part of each app. When an app is installed, all digests are verified and recorded by the Android system and cannot be modified by app processes afterward. If a file is modified during repackaging, its digest must change. Thus, instead of performing code scanning, we resort to using code file digests to detect code tampering. We leverage the Android system to retrieve the code file digests and compare them against the original digests we store in the app. If they are not equal, code tampering is detected.

While the *real* digest C_r of a file is retrieved at runtime by invoking Android system services, we store the information about the *original* digest C_o in the protected app. However, due to *circular dependency* it is impossible to put the digest of a file inside that file; we thus propose to put C_o out of the code file. We propose two different resilient ways, *salted hashing* and *steganography*, to achieve it.

Note that the *code files* above include both `classes.dex` and native library files. Actually, the code tampering detection approach can be easily generalized to check the integrity of other files, such as images and music files, since each file has its own digest stored in `MANIFEST.MF`.

5.2.3 Using Salted Hashing to Store Digests

The digest checking is coded as `if(Hash(`C_r`, st) == P(S))`, where `st` is a salt (i.e., a unique random integer) and C_r is the real digest retrieved at runtime, and the function call `P(S)` retrieves the stored digest D_o from the string named `S` in `strings.xml`, which contains all string literals needed by the app (note that other files in the app or a newly added file may also be used).

After detection nodes are inserted, we obtain the original digest C_o. The final step is to construct the value of `S` as the salted hash value, `Hash(`C_o`, st)`, using the Base64 binary-to-text encoding and insert `S` into `strings.xml`.

Obviously, the value of `S` looks suspicious from other normal string values; but attackers do not know how to manipulate it unless they know the salt value, which is encrypted as part of the detection code. Therefore, instead of trying to make our approach stealthy, this design is *non-stealthy*: even if attackers suspect where the information stored, they cannot manipulate it successfully.

5.2.4 Using Steganography to Store Digests

The second approach adopts Steganography. Steganography is the strategy of hiding a secret message inside a file to conceal the existence of the secret message without drawing suspicion to others. In our case, the secret message is the information about the digest `T(`C_r`)`, where `T()` is a transformation function such as a custom hash function. There are many different methods for steganography. Below we give two concrete examples for conducting it.

Note that such information hiding can eliminate or mitigate the suspicion by attackers, which can be regarded as an advantage of this approach compared to using salted hashing. However, the security does not rely on stealthiness, as even an attacker knows which part of the app is used to hide the digest information, he does not know how to manipulate it since the custom hashing code for converting the digest is encrypted.

Storing Digest Information in `strings.xml`. The digest checking is coded as `if(T(`C_r`) == P(S))`, where `P(S)` represents how to retrieve the characters based on a string named `S` in `strings.xml`. E.g., if the logic of retrieving the characters is to get the second and seventh letters from the string named "`info`" and assume, *after the code file is finalized*,

we calculate T(C_o) as 'en', then we can make up a string accordingly. The benefit is that, given the characters, we can hide them into a sentence that looks normal as shown below.

```
<string name="info">Help animals!</string>
```

Storing Digest Information in the Launcher Icon. We can also hide the original digest in the launcher icon of the app. The digest checking is coded as `if(T(`C_r`) == F(`ρ`))`, where ρ indicates the positions where the converted digest information is embedded at the launcher icon. We adopt a very simple Least Significant Bit (LSB) based image steganography technique [44]; it is known that human eyes cannot notice the difference of an image when the LSBs of pixels are modified [44]. Thus, we use the LSBs of pixels at ρ to store the value of `T(`C_r`)`.

Android uses 32-bit PNG icons, where each pixel is made up of 4 bytes representing *alpha*, *red*, *green*, and *blue* (with each using one byte), and is stored as: *(alpha << 24) | (red << 16) | (green <<8) | blue*, so there are four LSBs for each pixel (the right-most bit of alpha, red, green and blue each). E.g., to embed four bits of `T(`C_r`)`, we simply select one pixel, and then overwrite the four LSBs of the pixel with the four bits. The process is illustrated as follows. Given a selected pixel below,

00010110 01110111 01110100 01110001

assuming the 4-bit binary number to be embedded is: 0011, we hence overwrite the four LSBs of the pixel with the four bits, respectively (where bits in bold have been changed):

00010110 0111011**0** 0111010**1** 01110001

In order to manipulate the icon, the attacker needs to know the code of `T(`C_r`)` and the positions of the selected pixels ρ; both are part of the encrypted code. In short, the code tampering detection code leverages the power of logic bombs to keep resilient to attacks.

5.3 Response to Detected Repackaging

The responses should cause negative user experiences, such that users give a bad rating or even flag the app as malicious. 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. It is worth mentioning that *the resilience of our approach relies on the difficulty of triggering any given logic bomb on the attacker side*, rather than increasing the difficulty of analyzing and debugging the inserted errors. This not only simplifies the design, but also makes the security analysis more clear (see Section 6).

In addition, if code tampering is detected, the response should warn users of the high risk. 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. This way, the effect of repackaging detection is virtually propagated from one device to others. Moreover, *how to collect software piracy information in an inexpensive and scalable way has been a challenge for many app companies, and our technique can serve as a solution for them.*

6 Security Analysis

6.1 General Attacks

We then examine how the goals described in Section 3.1 are achieved in our design. First, **whitebox fuzzing** techniques [29], [30], [31], [32], including symbolic execution and multi-path execution, typically rely on resolving constraints correctly for the purpose of path exploration. In our case, even given a *powerful* symbolic executor that can explore any execution path (until it reaches a logic bomb) efficiently, when it encounters a logic bomb, the *essential* problem becomes this: the constraint `Hash(X) ==` H_c introduced by the bomb, where H_c is the hash value of a constant value `c` (we use the notations in Section 4), 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.*

It is widely known that conclic execution is powerful in dealing with specific non-linear constraints. Particularly, given a condition `Hash(X) == Y`, where `X` and `Y` are both symbolic inputs, if a previous execution has explored the *false* branch with `X = 2` and `Y = 10`, i.e., `Hash(2) != 10`, then the next execution can keep the value of `X`, but change the value of `Y` to make `Hash(X) == Y`; e.g., assume `Hash(2) == 5`, then the variables are assigned as `X = 2` and `Y = 5`. This way, the next execution will explore the *true* branch. It seems that conclic execution can handle hash functions. But note that there is a subtle and critical difference: the constraint `Hash(X) == Y` involves two symbolic inputs that allow flexible value assignment, while the right operand of the constraint `Hash(X) ==` H_c is a constant hash value and resolving it needs to reverse the cryptographic hash function, which is infeasible. Note that the correct value of X (i.e., the key used for encryption and decryption) needs to be computed in order to decrypt the encrypted code correctly; any attempts that try to decrypt the code with an incorrect key will fail.

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

Third, it is not surprising that through **blackbox/grebox fuzzing**, an attacker is able to trigger some logic bombs. But it has been a known challenge in the software testing area how to achieve a high test coverage [22]; thus, we insert many logic bombs into different parts of an app. Regardless of conventional fuzzing (such as AFL [28]) or fuzzing enhanced by symbolic execution (such as Driller [35]), they are very inefficient for generating inputs that can satisfy the trigger condition `Hash(X) ==` H_c of a bomb. For example, given a trigger condition converted from `X == 0x12345678`, it may take a fuzzer billions of times of tries to satisfy it. **API-hooking assisted analysis** allow attackers to *forge* the return values for API calls, but it does not help analyze a given logic bomb, since the trigger condition `Hash(X) ==` H_c does not tell which value of `X` should be used when forging the return value. *Therefore, **G3** is also achieved.*

Fourth, as described in Section 4.4, **code deletion** is defeated by code weaving and bogus bombs (**G4**).

6.2 Attacks against Keys

As the key 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. Therefore, the obfuscation strength of a logic bomb can be measured based on the size of $dom(\texttt{X})$.

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 that contains all the mappings between the values of `X` and their hash values for the purpose of 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 (i.e., `Hash(X, salt)`), so the precomputed table will not work.

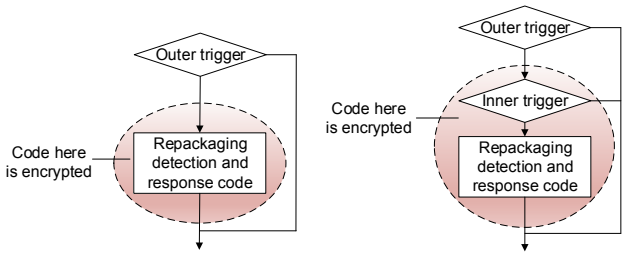
6.3 Attacks via User-Side API Interposition

Once an app is installed on the user side, its certificate is managed by the Android system and cannot be modified by app processes. However, since the detection code relies on APIs such as `getPublicKey`, we need to consider API interposition and manipulation on the user side that returns a fake return value to fool our system. While most of the Android API hooking methods have been proposed for security analysis and defense purposes, we consider the possibility that attackers employ them to evade our detection. Note that Android API hooking has been an active research problem, so we do not intend to make an exhaustive list, but discuss based on the following main categories.

API interposition based on roots. Many Android API interposition systems rely on the root privilege to work [45], [46], [47]. For example, ARTDroid [47] illustrates the interposition of API calls by manipulating vtable entries; it requires the root privilege. we assume user devices are not rooted (Section 3.1). Android rooting may brick the user devices, avoid warranty, and allow more possible access rights to malware; actually, only 7.6% of Android devices are rooted [38], and these device owners do not necessarily grant the root privilege to the repackaged apps.

API interposition based on code rewriting. Another approach to manipulating the API calls is via code rewriting [48], [49], but modification of encrypted code will corrupt the app and, hence, is not viable.

API interposition based on reloading system libraries. Reference Hijacking [50] proposes to load customized system classes from specified paths in the early stage of application launching, such that system API calls are hooked and handled by the customized system classes. The technique is proposed for security purposes (e.g., tainting and patching), but theoretically it may be leveraged to fool our system. However, attackers usually prefer the piracy to be stealthy, while the action of reloading system libraries is unusual and very suspicious. So we doubt it can become an attractive technique to attackers for the repackaging purpose.



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

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

API hooking based on sandboxed processes. Boxify [51] proposes to run the app to be monitored as a de-privileged *isolated process*, and meanwhile run a *broker process* that interposes all Android API calls from the isolated process. Like Reference Hijacking, Boxify is proposed for defense purpose for sandboxing untrusted apps. Isolated processes are rarely adopted by apps. NJAS [52] is another sandboxing technique that relies on `ptrace`. we argue that such unusual features are barely interesting to attackers, who want their repackaged apps keep stealthy from security analysis.

In short, evasion attacks by API interposition to return fake values either requires unusual features (such as the root privilege, isolated processes, and reloading system libraries) or rewriting the encrypted code. Thus, they should not constitute a likely threat to our system.

7 Enhancement: Double-trigger Bombs

As symbolic execution, which is commonly used in whitebox fuzzing for a higher code coverage, cannot be used to "crack" our bombs (Section 6), we estimate that attackers may invest more on blackbox/greybox fuzzing, e.g., by renting cloud services to run multiple fuzzers concurrently for a prolonged period of time. Even though we have shown that fuzzing is very inefficient in triggering our bombs, to make our defense even more resilient to fuzzing at scale, we propose *double-trigger bombs*.

Fig. 2a shows the structure of a single-trigger bomb presented above (in Section 4.2), 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 == c` (Line 1 in Listing 3), the inner trigger condition is met only if the app runs on a device

under some specific environment in terms of the system build number, IP address, GPS location, etc. Given the huge number of possible environment variable values and their combinations, attackers have to invest enormously to keep trying different 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 the bomb. Even when the attacker happens to fake the time correctly, it requires correct input to satisfy the outer trigger condition of that bomb.

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.

Since the inner trigger condition is part of the encrypted code, given a bomb it is unlikely for the attacker to correctly estimate which of the many environment variables has been used and which value range is the trigger input. We also emphasize that the security of our system takes this as an enhancement (particularly against fuzzing at scale), rather than a must. As analyzed in Section 6, neither conventional fuzzing or fuzzing assisted by symbolic execution is effective in defusing the inserted bombs.

8 IMPLEMENTATION

This section describes the implementation details of BOMBDROID. Our current prototype implemented the two repackaging detection methods described in Section 5, including detecting repackaged apps and detecting repackaged apps with illegal code modification via salted hashing. The implementation comprises 9,668 lines of Java code.

8.1 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; All the other methods in the apps are *candidate methods* used to insert logic bombs. Specifically, we first use Dynodroid [27] to generate a random stream of $10,000$ user events and feed them to the app. Meanwhile, we use Traceview [53] 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 and are excluded. All other methods, no matter they are reached by the dynamic analysis or not, are candidate methods used to insert bombs.

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

→

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

8.2 Outer Trigger Conditions

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

Existing qualified conditions. We use Soot [54] 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. This way, the payload can be injected into either of the two branches.

Artificial qualified conditions. While medium and large sized programs usually have many existing qualified conditions, smaller programs may not. To solve the issue, we choose to inject artificial ones. Specifically, $\alpha = 0.25$ (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.

8.3 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 [55]; AppBrain provides statistics about manufacturers [56]. We also allow developers to override the collected information; e.g., 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, $p \in [0.1, 0.2]$. E.g., 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.

8.4 Hashing and Encryption

We use SHA-256 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 (only using the first 128 bits of the hash value) to be used for encryption.

8.5 Bytecode Instrumentation

We first use `apktool` [57] 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` [58] to convert `classes.dex` to a collection of Java classes, which are used for instrumentation.

We leverage `Javassist` [59] for bytecode instrumentation. Javassist allows us to write the repackaging detection code in Java (in the form of source code), and then compiles the source code into bytecode on-the-fly during instrumentation. Note that the host app code used for the instrumentation is a collection of Java classes; the advantage of Javassist is that it allows the inserted detection code to be written in Java and then it compiles the source code on-the-fly.

The generated code, after being mixed with part of the original app code, is encrypted into a string, which is inserted into the app code. Then during execution time, when a 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`, which is then packed with other app resources into a protected app. The protected app 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.

9 EVALUATION

We have applied BOMBDROID to a set of Android apps, 1,463 totally, downloaded from F-Droid [60]. We first present statistics about the characteristics of the app programs in Section 9.1, and then describe the measurement of the effectiveness of our system. In Section 9.3, we measure the resilience of logic bombs to adversary analysis. The overhead data is presented in Section 9.4.

9.1 App Program Characteristics

Table 1 shows the static characteristics of the 1,463 apps in the eight categories. For each category, it shows the number of apps, the average number of lines of Java code (of apps in the category), the average number of candidate methods, the average number of existing qualified conditions used as outer trigger conditions, and the average number of environment variables used by apps.

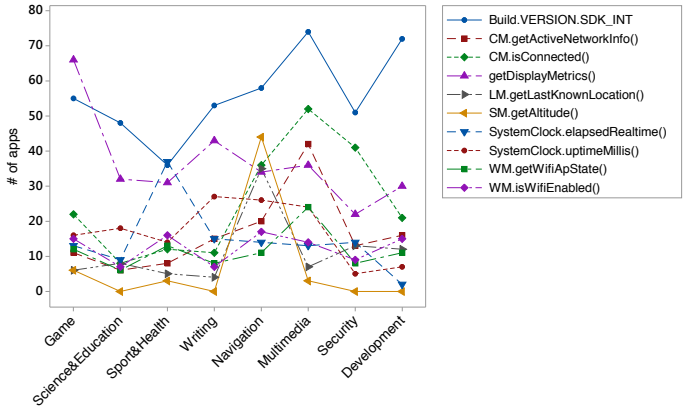


Fig. 3: The top ten most frequently referenced environment variables. (Legend: CM, LM, SM, and WM represent `ConnectivityManager`, `LocationManager`, `SensorManager`, and `WifiManager`, respectively.)

The first observation is that, unsurprisingly, larger-sized programs tend to have a larger number of candidate methods and existing qualified conditions. Note that BOMBDROID allows to insert artificial QCs which can be used as outer trigger conditions as well.

Figure 3 further shows the top ten most frequently used environment variables (denoted by Android Framework APIs) and, for each environment variable, the number of apps in each category using that variable. The most frequently referenced environment variable is `Build.VERSION.SDK_INT`; 43% of the apps in our data set use this variable, based on which apps can provide the best features and functionalities across different platform versions. 56% of the game apps read display metrics in order to adjust app user interfaces according to screen density and resolution. Almost half of the media apps request the state of network connectivity and network configurations, e.g., the connected network's link speed, IP address, negotiation state, etc., to control the usage of network resources and perform network operations.

Overall, all the apps use environment variables (retrieved through Android Framework APIs) and apps in the Multimedia category use the largest number of them on average. It shows that the logic of Android apps is indeed environment sensitive. In our constructed logic bombs, we use either environment variables that do not need permissions or those already used by the app itself, such that the protected app does not need to request extra permissions.

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 4. In the main interface, multiple fishes move around, and players need to click these fishes to gain scores. The six program variables in Figure 4 store different information of the currently visible fish, such as its moving direction, width, height, speed, and position. We use Dynodroid [27] 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. Plus,

TABLE 1: Static characteristics.

Category	# of apps	Avg LOC	Avg # of candidate methods	Avg # of existing qualified conditions	Avg # of environment variables
Game	175	3,038	98	62	18
Science&Education	158	4,156	92	45	10
Sport&Health	147	5,405	108	38	10
Writing	219	7,159	159	65	7
Navigation	181	9,246	177	55	11
Multimedia	168	10,152	211	76	19
Security	212	11,044	232	83	13
Development	203	14,457	391	96	11

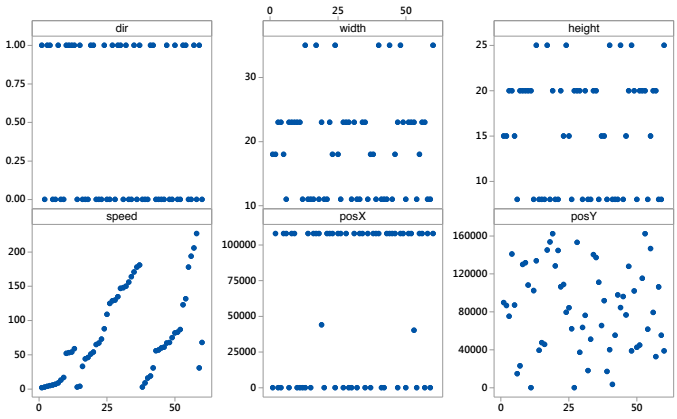


Fig. 4: Visualization of how the values of six program variables of AndroFish vary with time. The x-axis and y-axis represent the time (mins) and variable values, respectively.

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

some variables change values randomly (e.g., *posY*), whereas others have different values at different stages of the app running (e.g., *speed*), and it takes time for such variables to reach particular values, which implies that, for trigger conditions that depend on such variables, a user (or attacker) may take some particular time and input to activate them. It illustrate the necessity of profiling program variable 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 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.

9.2 Effectiveness

We next 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 plays 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 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.

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

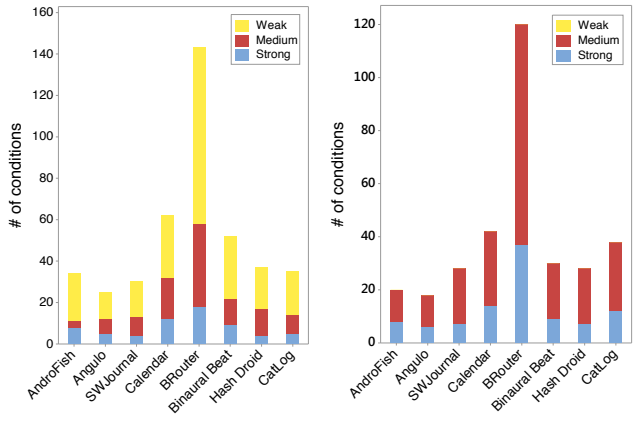
The results are encouraging showing that users can be quickly alerted when using a repackaged app. The response time is as short as 8 seconds, while the maximum times to trigger the first bombs are all within 13 minutes.

9.3 Resilience

Security analysis (see Section 6) 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.

9.3.1 Resilience to Brute force Attacks

Given an outer trigger condition `Hash(X) ==` H_c, attackers may try to search the value of `X` from $dom(\texttt{X})$ 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 qualified condition depends on string, integer, or boolean constant values, respectively. Figure 5 shows the analysis results on the eight apps. Figure 5a shows that a high percentage of the existing QCs have a weak obfuscation. Figure 5b 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.



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

Fig. 5: Strength of outer trigger conditions.

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

9.3.2 Resilience to Fuzzing and Human Analysis

We first measure the number of *outer trigger conditions* satisfied during one hour analysis using state-of-the-art Android fuzzing tools, including Monkey [61], PUMA [62], AndroidHooker [63], and Dynodroid. Table 4 shows the results. It can be observed that Dynodroid performed slightly better than other tools.

We then look into the number of *logic bombs* triggered (when *both* the outer trigger and inner trigger conditions were met) by Dynodroid. Figure 6 visualizes the results with each line corresponding to the percentage of triggered bombs for one of the eight apps. It shows that in the first 5 minutes a small number of bombs were triggered, and the growth of the numbers slowed down quickly. After 35 minutes, all apps did not have new bombs triggered. At most 6.4% bombs were triggered during the analysis, which means that the majority of bombs kept dormant, showing that the apps were 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 hundreds of different environment variables and each variable may have a large domain (possible values), it is unknown how to mutate the environment variable values effectively.* For example, the number of possible combinations of the MAC address and IP address is up to $2^{48} * 2^{32}$. It is essentially a type of brute force attacks, which is too expensive and inefficient to be a feasible option.

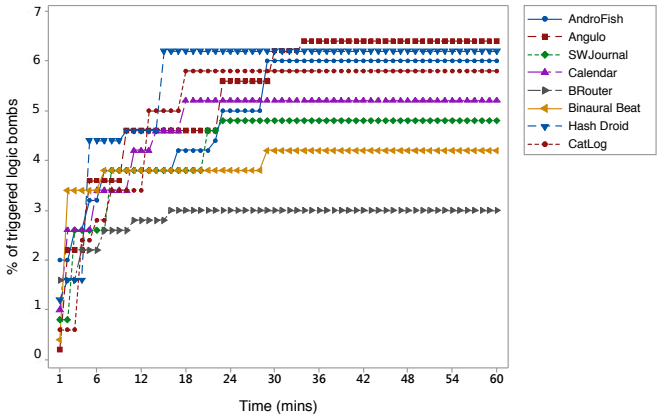


Fig. 6: Number of bombs triggered by Dynodroid in one hour. Each of the eight apps has a line representing the percentage of triggered bombs.

TABLE 5: Execution time overhead. For each app, we use BOMBDROID to generate two protected apps based on different types of payload inserted.

App	T_a (sec)	Payload	T_b (sec)	Overhead (%)
AndroFish	124	*type I*	126	1.6
		type I and *II*	127	2.4
Angulo	125	*type I*	127	1.6
		type I and *II*	129	3.2
SWJournal	115	*type I*	118	2.6
		type I and *II*	118	2.6
Calendar	148	*type I*	151	2.0
		type I and *II*	153	3.3
BRouter	132	*type I*	135	2.3
		type I and *II*	135	2.3
Binaural Beat	163	*type I*	166	1.9
		type I and *II*	167	2.5
Hash Droid	155	*type I*	158	1.9
		type I and *II*	160	3.2
CatLog	131	*type I*	134	2.3
		type I and *II*	135	3.1

9.4 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.

BOMBDROID implements two repackaging detection methods (see Section 5), including detecting repackaged apps (*type I*), and detecting repackaged apps with illegal code modification (*type II*). The side effects vary based on the number of logic bombs inserted and the types of the detection code (payload) carried in the bombs.

Thus, for each app, we use BOMBDROID to generate two protected apps: one is inserted with the *type I* payload only, and another one is inserted with both *type I* (50%) and *type II* (50%) payload.

False positives. As the response code injects difficult-to-debug 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. The results show that there were no false positives.

Code size change. For the apps inserted with the *type I* payload only, the code size change ranges from 8% to 13% and averages 9.7% among all the apps. For the apps inserted with both the *type I* and *type II* payload, the code size change ranges from 8.2% to 13.7% and averages 9.9% among all the apps. The *type I* payload is slightly larger than the *type II* payload.

Execution time overhead. 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 execution time overheads, which are 3.3% at most. It can be seen that the overhead is very small; moreover, the *type II* payload is a little bit expensive compared to the *type I* payload. 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.

9.5 User Study

For each original app, we use BOMBDROID to generate two protected apps: one is inserted with the *type I* payload only, and another one is inserted with both *type I* (50%) and *type II* (50%) payload. Next, for each protected app, we modify its code (e.g., inserting some simple code that prints a random generated string), and then repackage it. Through this, we have two repackaged apps for each original app, and 16 repackaged apps totally.

We next asked four human testers to play the repackaged apps. Each tester plays four randomly assigned repackaged apps, and each app is played for two hours. They are free to choose any emulator configurations (e.g., device types, SDK versions, and CPU/ABI, etc.) to install the apps. After that, they report their feedback on playing these repackaged apps.

All of them report that it is beneficial to have more fine-grained detection, i.e., to detect whether or not the repackaged app's code has been modified. They mentioned that if they were only alerted that an app had been repackaged, they might not uninstall the app immediately as some app even repackaged may not hurt their privacy; for example, attackers may simply repackage an app to insert ads or make profits without inserting malicious code to steal sensitive information or launch attacks. However, if they were informed that the app's code had been modified, they would promptly take down the app to protect themselves. Therefore, it is very useful and beneficial to not only detect whether an app has been repackaged, but also identify whether the repackaged app's code has been modified, such that victim users can be informed clearly whether their devices contain such high risk apps.

They also mention that it would be better if the detection can further detect whether the code modification indeed involves malicious code, instead of only notifying the app code has been modified. Many malware detection techniques can be adopted for this purpose [64], [65], [66], [67].

10 RELATED WORK

10.1 Malware Obfuscation

Obfuscation is a semantics-preserving transformation to hinder figuring out the original form of the resulting code [68], [69]. It has been widely used by malware to evade detection [70], [71], [72], [73], [74], [75], [76], [77], [78], [79], [80], [81], [82], [83], [84]. Malware obfuscation techniques include encryption, polymorphism and metamorphism [85]. (a) An encrypted malware typically consists of a decryptor and an encrypted main body [72], [75], [86]. Malware uses a different key to make the encrypted main body unique; the decryptor recovers the main body at runtime. However, the problem is that the decryptor remains the same, making it possible to detect malware based on the decryptor. (b) Polymorphic malware is capable of creating distinct *decryptors* using obfuscation, such as junk code insertion, instruction reordering, and register reassignment, etc. [71], [73], [74], [83], [87]. However, the main body, after decryption, can still be used for detection. (c) Metamorphism malware makes use of various obfuscations, such as control-flow/data-flow obfuscation techniques, to evolve its *main body* into new generations, and thus is difficult to be detected even its main body appears after decryption [82], [84]. We also encrypt the logic bombs with varying keys, but the keys are not embedded in the app code. Moreover, the strengthen of our technique is not based on how well the logic bombs are concealed; instead, we do not intentionally conceal the locations of logic bombs, yet still achieve resilience to various evasion attacks.

10.2 Repackaging Detection

The app repackaging problem has drawn efforts from both industry and academia. Different app repackaging detection techniques use different features and methods for comparing the code between a large number of apps to detect repackaging [3], [13], [15], [16], [88], [88], [89], [90], [91], [92], [93], [94], [95], [96], [97], [98], [99]. For example, Zhou et al. propose DroidMOSS, which uses hashing of app instruction sequence to detect repackaging [15]. Potharaju et al. uses program syntactic fingerprints to detect plagiarized applications under different levels of obfuscations [16]. Chen et al. use the program dependency graph as features to detect repackaging [14]. `AppInk` [100] and `DroidMarking` [101] inject watermarking into apps so that a trusted party with the knowledge of watermarking can help detect repackaging. Crussell et al. propose `AnDarwin` which is a scalable approach based on clustering to detecting similar Android apps using their semantic information [90]. Repackaging detection techniques based on code similarity comparison can be evaded by code obfuscations. Most of them rely on a centralized effort to detect repackaging, and may be imprecise when handling obfuscated apps.

BOMBDROID implements decentralized repackaging detection, which adds repackaging detection into apps, such that it becomes an inherent capacity of apps and does not rely on a third party.. SSN [18] attempts to build repackaging detection into the app code, but as detailed in Section 2.2, it is vulnerable to a variety of attacks. BOMBDROID implements the first resilient decentralized repackaging detection.

10.3 Tamper-proofing Techniques

There are roughly three categories of tamper-proofing techniques for preventing illegal code modification.

Self-Checksumming Based Approaches. Many tamper-proofing techniques are based on computing checksums of code segments [43], [102], [103], [104], [105], [106], [107]. Chang et al. define small pieces of code called *guards*, to compute checksums over code fragments [43]; disabling the protection requires all the guards to be disabled, making it non-trivial for attackers. Horne et al. [108] extend this technique and utilize *testers* and *correctors* that redundantly test for changes in the executable code as it is running and report modifications. Tsang et al. implement a large number of lightweight protection units to protect critical regions of a program from being modified [102]; this protection scheme supports non-deterministic execution of functions, resulting in different execution paths and nondeterministic tamper responses. Jakubowski et al. present software integrity checking expressions, which are program predicates, to dynamically check whether a program is in a valid state [104]. Jakubowski et al. further propose a scheme to transform programs into tamper-tolerant versions that use self-correcting operation as a response against attacks [103]; it chops a program into blocks, which are duplicated, individualized, and rearranged. Our technique does not rely on code scanning.

Oblivious Hashing Based Approaches. Chen et al. propose oblivious hashing that implicitly computes a hash value based on the execution of the code to verify the runtime behavior of the software [109]. Chen et al. then propose a tamper-proofing software technology for stack-machine based languages, such as Java, by improving oblivious hashing [110]. It inserts hash instructions into basic blocks at the bytecode level, to monitor the top of the stack and check whether the program running has been tampered with or not. It is unknown how this technique can detect mobile app repackaging that does not tamper with the code.

Code Encryption and Decryption Based Approaches. Aucsmith proposes an approach utilizing cryptographic methods to decrypt and encrypt code blocks before and after each execution round [23]. The decryption and encryption procedures are controlled by an integrity verification kernel, which communicates with other code segments to create an interlocking trust model. Wang et al. propose a dynamic integrity verification mechanism to prevent modification of software [25]. The mechanism utilizes multi-blocking encryption technique to encrypt and decrypt code at runtime. Cappaert et al. also propose an approach which enciphers code at runtime, relying on other code as key information [111]; this way, any tampering will cause the code to be decrypted with a wrong key and produce incorrect code. Our technique also employs cryptography, but it does not rely on a modified kernel or code scanning, and keys to decrypt the code are derived from the program inputs.

10.4 Logic Bombs

Various approaches have been proposed for discovering trigger-based behaviors [30], [31], [32], [33], [36], [37], [112], [113], [114], [115], [116]. Some techniques identify trigger-based behavior by *exploring paths during execution*. Moser et al. propose a system to explore multiple execution paths of Window executables [29]. The system uses QEMU and dynamic tainting to identify conditions whose outcome depends on certain inputs, and attempts execution on both branches by solving the path constraints. Symbolic execution has been widely applied to deriving predicates leading to specific execution paths. Crandall et al. propose an approach to detect time bombs in Windows binaries by varying time in a virtual machine and using symbolic execution to identify conditions depending on time [32]. `Bitscope` uses static analysis and symbolic execution to understand the behavior of malware binaries [30]. `MineSweeper` utilizes binary instrumentation and mixed concrete and symbolic execution for detecting trigger-based behavior [31]. By combining symbolic execution, path predicate reconstruction, and control-dependency analysis, `TriggerScope` can identify time-, location-, and SMS-related triggers in apps [33]. Our obfuscation on trigger conditions makes the path constraints unresolvable. `HSOMINER` [116] 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 [36]. Wilhelm and Chiueh propose a forced sampled execution approach that forces execution along different paths [37]. As BOMBDROID applies encryption on payloads, it is infeasible to directly execute payload without discovering the key used for decryption.

Crane et al. propose the concept of "booby traps" [117], which are only triggered to take effect when attacks are detected; e.g., sending forged results to remote attackers. The defense keeps dormant during normal executions. While sharing the spirit of trigger-based defenses, they assume attackers cannot tamper with the enhanced software, so the problems we are solving are very different.

11 DISCUSSION

11.1 Preventing Abuse of Logic Bombs

From the perspective of the app store, the use of logic bombs certainly increases the difficulty of security analysis of apps. This may inspire malware authors to adopt the technique in their malicious apps to fool malware scanning and analysis by the app store. We propose that, for each app submitted to the app store, the app authors should submit two versions of the app, including an original version and the one enhanced using BOMBDROID along with the keys, such that the app store can easily verify the equivalence of the two versions, and then proceed to scan and analyze the original version as usual. Thus, this simple method prevents malware authors from abusing the proposed technique, and it only requires very little extra effort from app stores.

11.2 User-side Repackaging and Tampering Detection

When users use repackaged apps, they may experience pop-up warnings, slowdown, and other negative user experiences.

We consider this acceptable, as causing negative user experiences is a common practice for handling pirated software in the industry [118].

Second, we make use of user devices to detect repackaging and tampering, and software by many vendors like Microsoft and Oracle also checks on the user side whether the software is genuine or registered properly. Thus, this should not raise an ethical concern.

11.3 Other Possible Attacks

During the analysis, whenever an outer trigger condition is evaluated, the attacker may trace back to the source of the variable used in that condition through backward data flow analysis. If it is from an array of literals or the return value of a system call, then the set of values taken by the variable is largely reduced, which enables a more efficient *dictionary attack*. However, if the data dependency is hard to determine or the source is some input data from users, such attacks are infeasible. Since we can construct artificial qualified conditions as trigger conditions, we can use data flow analysis and select program variables derived from input data to be used in the conditions. Thus, although such attacks are a threat to some of the existing qualified conditions, they can be defeated by using artificial qualified conditions. In addition, it is worth noticing that the encrypted branch code of a logic bomb essentially has obfuscated the data flow; thus, it is unknown how to perform backward data flow analysis that involves multiple bombs in a path.

The attacker may run a repackaged app many times and clone all the *executed* code. Specifically, given a logic bomb, if it is triggered we assume the attacker can remove it from the copied code; otherwise, the payload of the bomb is not copied since it is not executed. The attacker may be able to obtain a workable app without bombs eventually. This attack may work if the attacker can achieve a high code coverage. However, achieving a high code coverage has been a well known challenge. For example, the initialization of an app may check the user environments and takes different paths under different environments. Without a decent code coverage, the partially cloned app will cause program crashes and other unexpected errors on the user side.

11.4 Limitations

It is widely recognized that any software-based protection can be bypassed as long as attackers are determined enough and willing to spend time and effort. This is also true for BOMBDROID. We assume that 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 admit that determined attackers can crack the keys of bombs via, e.g., 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 app code can help reduce the number of possible values and assist attackers to guess the keys.

11.5 Future Work

We regard this novel logic bomb based repackaging detection approach as a *malware-inspired* defense. There are many techniques that have been used by malware, such as data-flow/control-flow obfuscation (which can make it is more difficult to analyze the app code) and in-memory code rewriting. We plan to explore how the techniques that are widely used in malware can be used for repackaging detection. For example, we plan to apply *custom packers* [119] and data-flow obfuscations [120] to the logic bombs.

The use of the logic bomb structure certainly increases the difficulty of security analysis of Android apps from the perspective of the app store. This may inspire malware authors to adopt the technique in their malicious apps. We are working on an effective scheme that allows the app store to check the submitted apps, such that the proposed technique will not become new arms of attackers. A simplest design would be to require app authors to submit two versions of the app: an original app and one with the processing of BOMBDROID, such that the app store can easily verify the equivalence of the two apps, and then proceed to analyze the security of the original app.

12 CONCLUSION

Application repackaging and tampering detection is an important problem for protecting the IP rights of legitimate developers and the security and privacy of app users. Building repackaging and tampering detection into apps brings many advantages over the centralized scheme. However, the challenge of making the detection code resilient to various adversary analysis was not resolved in prior work.

We proposed a novel use of logic bombs to protect repackaging detection code from attackers. Cryptographically obfuscated bombs are used to construct resilient bombs, while double-trigger bombs achieve fine control of the triggering of bombs. Each bomb includes (1) an outer trigger condition that thwarts automatic path exploration, (2) an environment-sensitive inner trigger condition that renders fuzzing-based hacking ineffective, and (3) code weaving and encryption that increase the difficulty of code understanding and rewriting. We have comprehensively analyzed and evaluated BOMBDROID. The analysis and evaluation results show that the technique is efficient, effective and resilient. We expect that our approach might benefit numerous honest Android application developers.

ACKNOWLEDGMENTS

Dr. Lannan Luo was supported by NSF CNS-1850278, NSF CNS-1815144, and the University of South Carolina ASPIRE-I Program. Dr. Qiang Zeng was supported by NSF CNS-1856380. Dr. Xiaojiang Du was supported by NSF CNS-1828363.

REFERENCES

[1] Q. Zeng, L. Luo, Z. Qian, X. Du, and Z. Li, "Resilient decentralized android application repackaging detection using *Logic Bombs*," in *IEEE/ACM International Symposium on Code Generation and Optimization*, 2018.

[2] Pocket Gamer, "Know thy enemy: Using data to push back against app piracy," 2018, https://www.pocketgamer.biz/comment-and-opinion/67583/sponsored-using-data-against-app-piracy/.

[3] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on Android markets," in *ESORICS*, 2012.

[4] H. Huang, S. Zhu, P. Liu, and D. Wu, "A framework for evaluating mobile app repackaging detection algorithms," in *Trust and Trustworthy Computing*, 2013.

[5] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for Android applications," in *USENIX Security*, 2012.

[6] P. Faruki, V. Ganmoor, V. Laxmi, M. S. Gaur, and A. Bharmal, "Androsimilar: Robust statistical feature signature for Android malware detection," in *SIN*, 2013.

[7] S. Liang and X. Du, "Permission-combination-based scheme for android mobile malware detection," in *Communications (ICC), 2014 IEEE International Conference on*. IEEE, 2014.

[8] L. Wu, X. Du, and J. Wu, "Mobifish: A lightweight anti-phishing scheme for mobile phones," in *Computer Communication and Networks (ICCCN), 2014 23rd International Conference on*. IEEE, 2014.

[9] L. Wu, X. Du, and X. Fu, "Security threats to mobile multimedia applications: Camera-based attacks on mobile phones," *IEEE Communications Magazine*, vol. 52, no. 3, 2014.

[10] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *S&P*, 2012.

[11] Y. Zhang, "Kemoge: Another mobile malicious adware infecting over 20 countries," 2015, https://www.fireeye.com/blog/threat-research/2015/10/kemoge_another_mobi.html.

[12] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "ViewDroid: Towards obfuscation-resilient mobile application repackaging detection," in *WiSec*, 2014.

[13] 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 *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2012, pp. 62–81.

[14] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on Android markets," in *ICSE*, 2014.

[15] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *CODASPY*, 2012.

[16] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang, "Plagiarizing smartphone applications: attack strategies and defense techniques," in *In Engineering Secure Software and Systems*, 2012.

[17] M. Lindorfer, S. Volanis, A. Sisto, M. Neugschwandtner, E. Athanasopoulos, F. Maggi, C. Platzer, S. Zanero, and S. Ioannidis, "Andradar: fast discovery of android applications in alternative markets," in *DIMVA*, 2014.

[18] L. Luo, Y. Fu, D. Wu, S. Zhu, and P. Liu, "Repackage-proofing android apps," in *DSN*, 2016.

[19] R. D. Gopal and G. L. Sanders, "Preventive and deterrent controls for software piracy," *Journal of Management Information Systems*, vol. 13, no. 4, pp. 29–47, 1997.

[20] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, p. 20, 2012.

[21] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[22] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," *The Computer Journal*, vol. 52, no. 5, pp. 589–597, 2009.

[23] D. Aucsmith, "Tamper resistant software: An implementation," in *International Workshop on Information Hiding*. Springer, 1996, pp. 317–333.

[24] J. Qiu, B. Yadegari, B. Johannesmeyer, S. Debray, and X. Su, "Identifying and understanding self-checksumming defenses in software," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. ACM, 2015, pp. 207–218.

[25] P. Wang, S.-k. Kang, and K. Kim, "Tamper resistant software through dynamic integrity checking," in *SCIS*. Institute of Electronics, Information and Communication Engineers, 2005, pp. 1–6.

[26] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "Droidfuzzer: Fuzzing the android apps with intent-filter tag," in *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, 2013.

[27] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for Android apps," in *FSE*, 2013.

[28] M. Zalewski, "merican Fuzzy Lop," http://lcamtuf.coredump.cx/afl/.

[29] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *S&P*, 2007.

[30] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin, "BitScope: Automatically dissecting malicious binaries," in *Tech. Rep. CMU-CS-07-133*, 2007.

[31] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," in *Botnet Detection*. Springer, 2008, pp. 65–88.

[32] J. R. Crandall, G. Wassermann, D. A. de Oliveira, Z. Su, S. F. Wu, and F. T. Chong, "Temporal search: Detecting hidden malware timebombs with virtual machines," in *ACM Sigplan Notices*, 2006.

[33] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "TriggerScope: Towards detecting logic bombs in android applications," in *S&P*, 2016.

[34] L. Luo, Q. Zeng, C. Cao, K. Chen, J. Liu, L. Liu, N. Gao, M. Yang, X. Xing, and P. Liu, "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, 2017, pp. 225–238.

[35] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, 2016, pp. 1–16.

[36] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in android applications that feature anti-analysis techniques," in *NDSS*, 2016, pp. 1–15.

[37] J. Wilhelm and T. cker Chiueh, "A forced sampled execution approach to kernel rootkit identification," in *International Workshop on Recent Advances in Intrusion Detection*, 2007.

[38] Kaspersky, "Rooting your Android: Advantages, disadvantages, and snags," 2017, https://www.kaspersky.com/blog/android-root-faq/17135/.

[39] Wikipedia contributors, "Roger Needham," 2019, https://en.wikipedia.org/wiki/Roger_Needham.

[40] ——, "Key derivation function," 2019, https://en.wikipedia.org/wiki/Key_derivation_function.

[41] Z. Brakerski and G. N. Rothblum, "Virtual black-box obfuscation for all circuits via generic graded encoding." in *TCC*, vol. 8349, 2014, pp. 1–25.

[42] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation," in *NDSS*, 2008, pp. 1–13.

[43] H. Chang and M. J. Atallah, "Protecting software code by guards," in *Security and Privacy in Digital Rights Management*, 2002, pp. 160–175.

[44] J. Fridrich, M. Goljan, and R. Du, "Detecting lsb steganography in color, and gray-scale images," *IEEE multimedia*, vol. 8, no. 4, pp. 22–28, 2001.

[45] Xposed, "Xposed Framework," 2019, http://repo.xposed.info/.

[46] C. Mulliner, J. Oberheide, W. Robertson, and E. Kirda, "Patchdroid: Scalable third-party security patches for android devices," in *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 2013, pp. 259–268.

[47] V. Costamagna and C. Zheng, "Artdroid: A virtual-method hooking framework on android art runtime." in *IMPS@ ESSoS*, 2016.

[48] T. Ki, A. Simeonov, B. P. Jain, C. M. Park, K. Sharma, K. Dantu, S. Y. Ko, and L. Ziarek, "Reptor: Enabling api virtualization on android for platform openness," in *MobiSys*, 2017.

[49] F. A. Brandolini, "Hooking java methods and native functions to enhance android applications security," Ph.D. dissertation, University of Bologna, 2016.

[50] W. You, B. Liang, W. Shi, S. Zhu, P. Wang, S. Xie, and X. Zhang, "Reference hijacking: Patching, protecting and analyzing on unmodified and non-rooted android devices," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 959–970.

[51] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. v. Styp-Rekowsky, "Boxify: Full-fledged app sandboxing for stock android," in *Proceedings of Usenix Security*, 2015.

[52] A. Bianchi, Y. Fratantonio, C. Kruegel, and G. Vigna, "Njas: Sandboxing unmodified applications in non-rooted devices running stock android," in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*. ACM, 2015, pp. 27–38.

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

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

[55] Dashboards, "http://developer.android.com/about/dashboards/index.html," 2017.

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

[57] Apktool, 2017, https://ibotpeaches.github.io/Apktool/.

[58] dex2jar, 2017, https://github.com/pxb1988/dex2jar.

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

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

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

[62] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "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*, 2014.

[63] AndroidHooker, 2016, https://github.com/AndroidHooker.

[64] 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*. ACM, 2012, pp. 281–294.

[65] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "Droidmat: Android malware detection through manifest and api calls tracing," in *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*. IEEE, 2012, pp. 62–69.

[66] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 15–26.

[67] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "DREBIN: Effective and explainable detection of android malware in your pocket." in *Ndss*, vol. 14, 2014, pp. 23–26.

[68] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep. CS-TR-148, 7 1997, http://hdl.handle.net/2292/3491.

[69] A. Balakrishnan and C. Schulze, "Code obfuscation literature survey," http://pages.cs.wisc.edu/~arinib/writeup.pdf, 2005.

[70] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," in *Proceedings of the 12th Conference on USENIX Security Symposium*. USENIX Association, 2003, pp. 169–186.

[71] M. Stamp and W. Wong, "Hunting for metamorphic engines," *Journal in Computer Virology*, vol. 2, no. 3, 2006.

[72] M. Schiffman, "A brief history of malware obfuscation," *Available on: http://blogs. cisco. com/security/a_brief_history_of_malware_obfuscation_ part_1_of_2*, 2010.

[73] E. Konstantinou and S. Wolthusen, "Metamorphic virus: Analysis and detection," *Royal Holloway University of London*, vol. 15, p. 15, 2008.

[74] B. B. Rad, M. Masrom, and S. Ibrahim, "Camouflage in malware: from encryption to metamorphism," *International Journal of Computer Science and Network Security*, vol. 12, no. 8, pp. 74–83, 2012.

[75] X. Jiang and Y. Zhou, "Dissecting android malware: Characterization and evolution," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 95–109.

[76] D. Baysa, R. M. Low, and M. Stamp, "Structural entropy and metamorphic malware," *Journal of computer virology and hacking techniques*, vol. 9, no. 4, pp. 179–192, 2013.

[77] F. Daryabar, A. Dehghantanha, and H. G. Broujerdi, "Investigation of malware defence and detection techniques," *International Journal of Digital Information and Wireless Communications (IJDIWC)*, vol. 1, no. 3, pp. 645–650, 2011.

[78] C. Yan and M. Wu, "An executable file encryption based scheme for malware defense," in *Intelligent Systems and Applications, 2009. ISA 2009. International Workshop on*. IEEE, 2009, pp. 1–5.

[79] K. Kaushal, P. Swadas, and N. Prajapati, "Metamorphic malware detection using statistical analysis," *International Journal of Soft Computing and Engineering (IJSCE)*, vol. 2, no. 3, pp. 49–53, 2012.

[80] L. Martignoni, M. Christodorescu, and S. Jha, "Omniunpack: Fast, generic, and safe unpacking of malware," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. IEEE, 2007, pp. 431–441.

[81] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android security: a survey of issues, malware penetration, and defenses," *IEEE communications surveys & tutorials*, vol. 17, no. 2, pp. 998–1022, 2015.

[82] M. Musale, T. H. Austin, and M. Stamp, "Hunting for metamorphic javascript malware," *Journal of Computer Virology and Hacking Techniques*, vol. 11, no. 2, pp. 89–102, 2015.

[83] S. Cesare, Y. Xiang, and W. Zhou, "Malwise—an effective and efficient classification system for packed and polymorphic malware," *IEEE Transactions on Computers*, vol. 62, no. 6, pp. 1193–1206, 2013.

[84] Q. Zhang and D. S. Reeves, "Metaaware: Identifying metamorphic malware," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. IEEE, 2007, pp. 411–420.

[85] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*. IEEE, 2010, pp. 297–300.

[86] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," *IEEE Security & Privacy*, vol. 5, no. 2, 2007.

[87] S. Cesare and Y. Xiang, "A fast flowgraph based classification system for packed and polymorphic malware on the endhost," in *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*. IEEE, 2010, pp. 721–728.

[88] S. Yue, W. Feng, J. Ma, Y. Jiang, X. Tao, C. Xu, and J. Lu, "Repdroid: an automated tool for android application repackaging detection," in *Proceedings of the 25th International Conference on Program Comprehension*. IEEE Press, 2017.

[89] K. Tian, D. Yao, B. G. Ryder, and G. Tan, "Analysis of code heterogeneity for high-precision classification of repackaged malware," in *Security and Privacy Workshops (SPW), 2016 IEEE*. IEEE, 2016, pp. 262–271.

[90] J. Crussell, C. Gibler, and H. Chen, "Andarwin: Scalable detection of semantically similar android applications," in *ESORICS*, 2013.

[91] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of "piggybacked" mobile applications," in *CODASPY*, 2013.

[92] L. Malisa, K. Kostiainen, M. Och, and S. Capkun, "Mobile application impersonation detection using dynamic user interface extraction," in *ESORICS*, 2016.

[93] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "Viewdroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. ACM, 2014, pp. 25–36.

[94] W. Hu, J. Tao, X. Ma, W. Zhou, S. Zhao, and T. Han, "Migdroid: Detecting app-repackaging android malware via method invocation graph," in *Computer Communication and Networks (ICCCN), 2014 23rd International Conference on*. IEEE, 2014, pp. 1–7.

[95] H. Wang, Y. Guo, Z. Ma, and X. Chen, "Wukong: a scalable and accurate two-phase approach to android app clone detection," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 71–82.

[96] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged android applications," in *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014, pp. 56–65.

[97] Y. Zhauniarovich, O. Gadyatskaya, B. Crispo, F. La Spina, and E. Moser, "Fsquadra: fast detection of repackaged applications," in *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 2014, pp. 130–145.

[98] C. Yuan, S. Wei, C. Zhou, J. Guo, and H. Xiang, "Scalable and obfuscation-resilient android app repackaging detection based on behavior birthmark," in *Asia-Pacific Software Engineering Conference (APSEC), 2017 24th*. IEEE, 2017, pp. 476–485.

[99] K. Tian, D. Yao, B. G. Ryder, and G. Tan, "Analysis of code heterogeneity for high-precision classification of repackaged malware," in *IEEE Security and Privacy Workshops (SPW)*. IEEE, 2016, pp. 262–271.

[100] W. Zhou, X. Zhang, and X. Jiang, "Appink: watermarking android apps for repackaging deterrence," in *ASIA CCS*, 2013.

[101] C. Ren, K. Chen, and P. Liu, "Droidmarking: Resilient software watermarking for impeding Android application repackaging," in *ASE*, 2014.

[102] H. chung Tsang, M.-C. Lee, and C.-M. Pun, "A robust anti-tamper protection scheme," in *ARES*, 2011.

[103] M. H. Jakubowski, N. Saw, and R. Venkatesan, "Tamper-tolerant software: Modeling and implementation," in *IWSEC*, 2009.

[104] M. H. Jakubowski, P. Naldurg, V. Patankar, and R. Venkatesan, "Software integrity checking expressions (ICEs) for robust tamper detection," in *Information Hiding*, 2007.

[105] J. T. Giffin, M. Christodorescu, and L. Kruger, "Strengthening software self-checksumming via self-modifying code," in *Computer*

Security Applications Conference, 21st Annual. IEEE, 2005, pp. 10–pp.
[106] G. Wurster, P. C. Van Oorschot, and A. Somayaji, "A generic attack on checksumming-based software tamper resistance," in *Security and Privacy, 2005 IEEE Symposium on*. IEEE, 2005, pp. 127–138.
[107] P. A. Cronce, "Method for runtime code integrity validation using code block checksums," 2005, uS Patent 6,880,149.
[108] B. Horne, L. Matheson, C. Sheehan, and R. E. Tarjan, "Dynamic self-checking techniques for improved tamper resistance," in *Proceedings of the 1st ACM Workshop on Digital Rights Management (DRM)*, 2002, pp. 141–159.
[109] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. H. Jakubowski, "Oblivious hashing: A stealthy software integrity verification primitive," in *Information Hiding*, 2002.
[110] H.-Y. Chen, T.-W. Hou, and C.-L. Lin, "Tamper-proofing basis path by using oblivious hashing on Java," in *Information Hiding*, 2007, pp. 9–16.
[111] J. Cappaert, B. Preneel, B. Anckaert, M. Madou, and K. D. Bosschere, "Towards tamper resistant code encryption: Practice and experience," in *ISPEC*, 2008.
[112] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti, "Detecting environment-sensitive malware," in *International Workshop on Recent Advances in Intrusion Detection*, 2011.
[113] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna, "Efficient detection of split personalities in malware," in *NDSS*, 2010, pp. 1–16.
[114] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "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*, 2012.
[115] C. Song, P. Royal, and W. Lee, "Impeding automated malware analysis with environment-sensitive malware." in *HotSec*, 2012.
[116] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin, "Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps," in *NDSS*, 2017, pp. 1–15.
[117] S. Crane, P. Larsen, S. Brunthaler, and M. Franz, "Booby trapping software," in *Proceedings of the 2013 New Security Paradigms Workshop*, ser. NSPW '13, 2013, pp. 95–106.
[118] J. DAVIS, "Eight of the most hilarious anti-piracy measures in video games," 2013, https://www.ign.com/articles/2013/04/29/eight-of-the-most-hilarious-anti-piracy-measures-in-video-games.
[119] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 659–673.
[120] A. Majumdar, S. J. Drape, and C. D. Thomborson, "Slicing obfuscations: design, correctness, and evaluation," in *Proceedings of the 2007 ACM workshop on Digital Rights Management*. ACM, 2007, pp. 70–81.

Qiang Zeng is an Assistant Professor at University of South Carolina. He received his bachelor's and Master's degrees from Beihang University, and his Ph.D. degree from the Pennsylvania State University in 2014. His research interests are software and system security.

Lannan Luo is an Assistant Professor at University of South Carolina. She received received her BS degree from Xidian University in 2009, MS degree from the University of Electronic Science and Technology of China in 2012, and PhD degree from the Pennsylvania State University in 2017. Her research interests are software and system security.

Zhiyun Qian is an associate professor at University of California, Riverside. He received his PhD from University of Michigan, Ann Arbor. His research interest is on system and network security, including vulnerability discovery, applied program analysis, system building, Internet security (e.g., TCP/IP), and Android security. His research has resulted in real-world impact on the design and implementation of Linux kernel, Android, macOS, and firewall products. He is a recipient of the NSF CAREER Award, Applied Networking Research Prize 2019, Facebook Internet Defense Prize Finalist 2016.

Xiaojiang Du is a professor in the Department of Computer and Information Sciences at Temple University, Philadelphia, USA. Dr. Du received his B.S. and M.S. degree in electrical engineering from Tsinghua University, Beijing, China in 1996 and 1998, respectively. He received his M.S. and Ph.D. degree in electrical engineering from the University of Maryland College Park in 2002 and 2003, respectively. His research interests are security, wireless networks, and systems. He has authored over 300 journal and conference papers in these areas, as well as a book published by Springer. Dr. Du has been awarded more than $6 million US dollars research grants from the US National Science Foundation (NSF), Army Research Office, Air Force Research Lab, NASA, Qatar, the State of Pennsylvania, and Amazon. He won the best paper award at IEEE GLOBECOM 2014 and the best poster runner-up award at the ACM MobiHoc 2014. He serves on the editorial boards of three international journals. Dr. Du is a Senior Member of IEEE and a Life Member of ACM.

Zhoujun Li received the M.Sc. and Ph.D. degrees in computer science from the National University of Defense Technology, China, in 1984 and 1999, respectively. Since 2001, he has become a Pro- fessor with the School of the Computer, Beihang University. He has published more than 150 papers on international journals, such as TKDE, Information Science, and Information Processing and Management, and international conferences such as SIGKDD, ACL, SIGIR, AAAI, IJCAI, SDM, CIKM, and WSDM. His research interests include the data mining, information retrieval, and database. He is a PC Member of many international conferences, such as SDM2015, CIKM2013, WAIM2012, and PRICAI2012.

Chin-Tser Huang received the BS degree in computer science and information engineering from the National Taiwan University, Taipei, Taiwan in 1993, and the MS and PhD degrees in computer sciences from the University of Texas at Austin in 1998 and 2003, respectively. He joined the faculty at the University of South Carolina in Columbia in 2003 and is now a Professor in the Department of Computer Science and Engineering. His research interests include network security, network protocol design and verification, cloud computing, and distributed systems. He is the director of the Secure Protocol Implementation and Development (SPID) Laboratory at the University of South Carolina. He is the author (along with Mohamed Gouda) of the book Hop Integrity in the Internet, published by Springer in 2005. He is a recipient of the USAF Summer Faculty Fellowship Award and the AFRL Visiting Faculty Research Program Award in 2008-2019. He is a senior member of IEEE and a senior member of ACM.

Csilla Farkas is a Professor in the Department of Computer Science and Engineering and Director of the Center for Information Assurance Engineering at the University of South Carolina. Dr. Farkas' research interests include information security, data inference problem, financial and legal analysis of cyber crime, and security and privacy on the Semantic Web. She is a recipient of the National Science Foundation Career award. The topic of her award is "Semantic Web: Interoperation vs. Security – A New Paradigm of Confidentiality Threats." Dr. Farkas actively participates in international scientific communities as program committee member and reviewer.

Csilla Farkas received her PhD from George Mason University, Fairfax. In her dissertation she studied the inference and aggregation problems in multilevel secure relational databases. She received a MS in computer science from George Mason University and BS degrees in computer science and geology from SZAMALK, Hungary and Eotvos Lorand University, Hungary, respectively.