2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

Repackage-proofing Android Apps

Lannan Luo, Yu Fu, Dinghao Wu, Sencun Zhu, and Peng Liu
The Pennsylvania State University
University Park, PA 16802
Email: {{lzl144,yuf123,dwu,pliu}@ist,szhu@cse}.psu.edu

***Abstract*—App repackaging has become a severe threat to the Android ecosystem. While various protection techniques, such as watermarking and repackaging detection, have been proposed, a defense that stops repackaged apps from working on user devices, i.e., *repackage-proofing*, is missing. We propose a technique that builds a reliable and stealthy repackage-proofing capability into Android apps. A large number of detection nodes are inserted into the original app without incurring much overhead; each is woven into the surrounding code to blur itself. Once repackaging is detected, a response node injects a failure in the form of delayed malfunctions, making it difficult to trace back. The response nodes and detection nodes form high-degree connections and communicate through stealthy communication channels, such that upon detection several of the many response nodes are selected stochastically to take actions, which further obfuscates and enhances the protection. We have built a prototype. The evaluation shows that the technique is effective and efficient.**

Keywords*—*Android apps; repackaging; tamper-proofing; obfuscation

I. Introduction

The explosive growth of the Android markets over the past few years has led to a booming app economy [45]. However, Android app piracy is rampant. Take game apps as an example, a recent report showed a 95% piracy rate for Android games [16]. Among other forms of piracy, app repackaging is especially notorious, because it does not only cause financial loss to honest developers, but also threatens the overall app ecosystem and users. Attackers may repackage an app under their own names to earn the app purchase profit or change the app's ad library causing the ad profit to go to attackers [14], [6], [27], [15]. Furthermore, repackaging has become one of the main forms of propagating malware. Previous research showed that 86% of more than 1200 Android malware families repackaged legitimate app to include malicious payloads [59]. A popular app may be repackaged by attackers with malicious payload injected to steal user information, send premium SMS text messages stealthily, or purchase apps without the victim user's awareness [14], [27], [58], [51], [9].

As Android app repackaging is prevalent and dangerous, tremendous efforts have been made to address the problem in recent years. Repackaging detection [14], [24], [8], [58], [42], e.g., based on code similarity comparison, is performed by authorities like Google Play, which, however, can be easily evaded by various obfuscations; besides, depending on the effect and timeliness of the detection, a repackaged app may have been widely distributed before it is detected and removed from the markets. Code obfuscation by legitimate authors is used to increase the difficulty of reverse engineering [13], [35], [36], so that it may take attackers more effort to inject malicious code. Watermarking can be used to prove the ownership of a repackaged app when disputes occur [45], [57]. However, none of the existing techniques thwarts threats caused by repackaged apps once they are installed on user devices. A defense that prevents repackaged apps from working on user devices, which we call *repackage-proofing*, is needed.

Repackage-proofing can be classified as a type of tamper-proofing (specialized in tackling app repackaging). Various tamper-proofing techniques exist. For example, Aucsmith [3] proposed a cryptography-based approach, which breaks up a binary program into individually encrypted segments. The protected program is executed by jumping to a chain of temporarily decrypted segments. The technique requires decryption and jumps; while they are possible in Android bytecode, they cannot be done in a stealthy way since such operations have to go through calls to a class loader due the code-loading mechanism in Android. Many tamper-proofing techniques are based on computing checksums of code segments [6]. They have similar issues when applied to protecting Android bytecode, since the code checking operation has to involve a call to a custom class loader, and thus can be easily found and bypassed. Instead of computing code checksums, Chen et al. [10] proposed to calculate a hash value based on the actual execution trace of the code and compare with a stored value. However, the approach requires pre-computation of expected hash values under all possible inputs; thus, it can only be applied to relatively simple functions that produce deterministic hash values. So far few existing tamper-proofing techniques are applicable to dealing with Android app repackaging. To the best of our knowledge, there is no study in the open literature that investigates repackage-proofing—tamper-proofing that *prevents repackaged apps from working on user devices.*

Tamper-proofing of type-safe distribution formats such as bytecode in Android apps is more challenging than tamper-proofing native code [13]. Operations as simple as code reading, which otherwise can be blended with the original assembly code seamlessly, involve calls to a class loader in Android. Thus, how to insert tampering detection into bytecode stealthily is challenging. Furthermore, as in all tamper-proofing implementations, it requires a careful design to hide the response code that injects failures upon tamper detection, as an attacker can leverage debugging to locate the response code once a failure is noticed. While the principle of delaying the effect of an injected failure can be shared, how to achieve it in Android apps needs fresh ideas. In addition to hiding the statically inserted code and dynamic failure generation operations, how to deliver an efficient tamper-proofing implementation compatible with the current Android system is a practical consideration as well as a challenge. We identify these as the main challenges of designing an effective repackage-proofing technique resilient to evasion attacks.

We propose a repackage-proofing technique, named *Stochastic Stealthy Network* (SSN), that overcomes these challenges. It builds the capability of repackage-proofing into apps, such that repackaged apps cannot run successfully on user devices.

978-1-4673-8891-7/16 $31.00 © 2016 IEEE
DOI 10.1109/DSN.2016.56

IEEE computer society

Our insight is that a unique identification of the app author and an immutable value bound with the app installation is the public key contained in the certificate of an app package. So, instead of calculating code checksums or hashes of execution traces, our detection code detects repackaging by comparing the public key hard-coded and hidden in the code against the one contained in the app certificate.

Given an app, SSN inserts a large number of detection nodes into the original code without incurring a high overhead. To achieve stealthiness, each detection node is obfuscated and then woven into the surrounding code, such that the inserted detection code and original code blur together without identifiable boundaries. Upon detection of repackaging, rather than triggering a failure instantly, the detection node stealthily transmits the detection result to a response node via a stealthy communication channel, and the latter delays the failure for a random period of time. As a result, the failure point is apart from the response node, which makes it difficult for attackers to trace back to the response node. In addition, many detection nodes and response nodes form a large network with high-degree connections. Each time repackaging is detected, a response node is picked stochastically to take actions. There are a variety of response nodes, which means that, given the same input, two consecutive executions of a repackaged app may end up with different failures injected by different response nodes, resulting in a more difficult debugging scenario for attackers. Through these strategies, our technique constructs a stochastic and stealthy network of repackage-proofing that is resilient to evasion attacks.

We have implemented a prototype of SSN, which can be applied by legitimate developers during compile time to build repackage-proofing into their apps. We evaluated SSN on 600 Android apps. The evaluation results show that the protection provided by SSN is effective to defeat repackaging, resilient to various evasion attacks, and incurs a very small overhead.

We made the following contributions.

- To the best of our knowledge, this is the first work reported in the open literature that prevents repackaged apps from working on user devices without relying on authorities.
- We identify the main challenges in designing a repackage-proofing technique, and propose SSN that overcomes those challenges. Unlike conventional tamper-proofing techniques, SSN leverages unique characteristics of Android apps to construct effective and stealthy protection.
- We have implemented a prototype of SSN compatible with the Android platform. The evaluation shows that SSN is effective, efficient, and resilient to many evading attacks.

II. Overview

A. Problem Statement

Android app developers produce apps and sell them in the form of Android application packages (APK) files. Both legitimate users and attackers can download the APK files, and we consider that attackers try to repackage apps and re-release them. An attacker may modify the code, such as the in-app billing and ads components, to earn financial profit, insert malicious payload to infringe upon user privacy, or simply repackage the app without making changes for fun and fame. Repackage-proofing, as a type of tamper-proofing, aims to detect repackaging and prevent repackaged apps from working properly on user devices.

On the Android platform, an app, no matter it is legitimate or repackaged, has to be digitally signed with a certificate before it is released. Each certificate contains a unique public and private key pair. As the private key of the certificate is held by the developer which is unknown to the public, an attacker who wants to repackage an app has to choose a new certificate to sign the repackaged app. Thus, the public key contained in a certificate, which is a unique identification of an app developer, can be leveraged to determine whether an app has been repackaged by an attacker. Specifically, our repackage-proofing technique monitors the change of the public key to detect repackaging, and prevents any repackaged app from working properly on user devices. As a consequence, few users are willing to purchase and play the repackaged app, which limits its propagation as well as harms. Other responses upon detection are possible, for example, notifying legitimate developers and authorities of the detection by emails and removing repackaged apps from markets, which is not explored in our current implementation but discussed in Section VIII.

B. Threat Model

We assume attackers can get the APK file of an app; however, they cannot obtain the source code and the private key, which is reasonable in practice. Moreover, it is possible that end users are in collusion with attackers; for example, a user may run a custom firmware that always generates the original public key when running a repackaged app. We do not consider such collusion attacks, and aim at protecting legitimate users.

Attackers may launch evasion attacks to bypass or nullify the repackage-proofing protection built into apps. There are at least three types of evasion attacks. First, attackers can utilize static analysis techniques, such as text searching, pattern matching, static taint analysis, to find the injected protection code and make any code transformations necessary aiming to disable/remove the protection code. Second, attackers can also perform dynamic analysis, such as dynamic monitoring, dynamic taint analysis, debugging, etc., to execute and examine the app (bytecode and native binary), line by line, to identify the injected protection code and disable/remove the code. Third, in order to assist other evasion attacks and facilitate debugging, an attacker may control the execution to provide the original public key to bypass the detection, similar to the idea of replay attacks in networks. We aim to address these evasion attacks.

Attackers may carefully infer the program semantics and re-write parts of an app to bypass specific protection. Moreover, they may be willing to sacrifice certain functionalities of the original app and re-publish it. We handle both selective re-writing and functionality-pruning attacks.

It is generally believed that a program protected by any software-based approach can eventually be cracked as long as a determined attacker is willing to spend time and effort, which is also true with our protection. However, we assume attackers are interested in repackaging an app only if it is cost-effective, for example, when the cost of repackaging is less than that of developing the app from scratch.

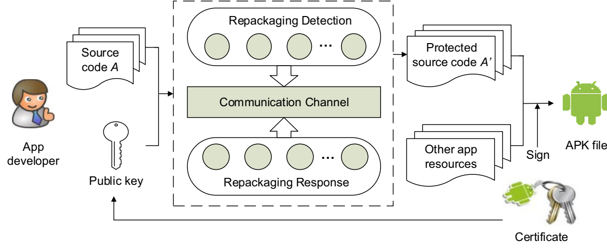


Fig. 1. The SSN architecture and deployment scheme.

C. Architecture

We propose a repackage-proofing technique called Stochastic Stealthy Network (SSN), which builds reliable and stealthy protection into apps without relying on authorities. Fig. 1 shows its architecture and deployment scheme.

As shown in the dashed rectangle of Fig. 1, SSN is comprised of three functional parts: *Repackaging Detection*, *Repackaging Response*, and *Communication Channel*. An app developer passes the app's source code A and the public key as inputs to SSN. Multiple distinct detection and response nodes are then inserted into the original code for repackaging detection and response. The communication channel is between the detection and response nodes to construct a highly connected network, so that whenever a detection node detects repackaging, one of multiple response nodes takes actions stochastically in order to confuse attackers. Finally, the resultant source code A' is generated as output and packaged with other app resources, which are then signed with the original certificate to produce an APK file for publication. In the following three sections, we will present the design and implementation details of the three functional parts: repackaging detection, repackaging response and the communication channel, respectively.

III. Repackaging Detection

This section presents the repackaging detection part, including how to construct and inject detection nodes.

A. Detection Node Construction

1) Construction Process: Repackaging detection consists of multiple detection nodes; each checks the public key. Specifically, a detection node extracts the public key from an app during runtime and compares it against the original one to detect whether or not the app has been repackaged. In the following, we refer to K_r as the public key extracted during runtime, and K_o as the original public key. Once a detection node detects that K_r is different from K_o, it determines that the app has been repackaged, and transmits the detection result to response nodes via a communication channel (presented in Section V).

K_o is provided by the legitimate developer who uses SSN. The next question is how to retrieve K_r at runtime. We leverage `PackageManager`, which is a class that (1) installs, uninstalls, and upgrades apps, and (2) stores and retrieves various kinds of application information. If an app being installed has passed the signature verification PackageManager parses its APK file to read the application information and then stores the information in three files under the `/data/system` folder. One of the three files is `packages.xml`, which contains package names, permissions, public keys, code paths, etc., of the installed apps. We make use of PackageManager to extract K_r from this file.

```
1  CertificateFactory ctFty =
       CertificateFactory.getInstance("X509");
2  //Get the X509certificate
3  X509Certificate cert =
       (X509Certificate)ctFty.generateCertificate(new
       ByteArrayInputStream(sig));
4  //Get the public key
5  String pubKey = cert.getPublicKey().toString();
6  //Extract a substring from the public key
7  String keySub = pubKey.substring(1,4);
8  //Compare with substring of the original key
9  if (!keySub.equals("elc")) {
10  //Repackaging is detected.
11  SenderCommunicationChannel();
12 }
```

Fig. 2. A code snippet of a detection node.

We construct detection nodes based on multiple predefined distinct "polymorphic" templates which implement the same detection functionality but look different. For example, reflection is used to call functions. The function names (`getPublicKey` and `generateCertificate`, etc.) are generated in different ways in different templates: a variety of substrings are produced and concatenated to form function names, which are passed to reflection methods to invoke the corresponding functions. This way, attackers cannot identify the detection nodes by statically searching function names. Moreover, different code obfuscation techniques, such as instruction reordering, variable renaming, dummy code injection, and opaque predicate insertion, are also combined and applied to the detection nodes to make them difficult to reverse-engineer and improve their stealthiness [12], [37], [32], [43], [38].

2) An Example: Figure 2 shows an example of the code snippet in a detection node with reflection calls omitted. The detection node first obtains a `CertificateFactory` object with the specified type "X509" (Line 1). It then retrieves an `X509Certificate` object (Line 3). Next, it calls `getPublicKey` to get the runtime public key `pubKey` (Line 5). Note that `pubKey` is extracted from `packages.xml`. Then the detection node extracts a substring `keySub` from `pubKey` (Line 7), and checks whether `keySub` equals to the hard-coded substring of the original public key with the same index range (Line 9). Sometimes, the two substrings are applied on the same transformation function (e.g., transform each one to a hash code, etc.), and then the equivalence of the two resulting values are compared. Here, we denote the substring of the original public key as K_o^{sub}. Different detection nodes compare different substrings with varied lengths of the public key. Comparing an individual character is insufficient, as different public keys may happen to have the same character at a given index. Neither do we compare the entire public key, since it is conspicuous to attackers (K_o is a long string). Finally, if `keySub` is not equal with K_o^{sub}, the detection node determines the app has been repackaged, and stealthily transmits the detection result via a communication channel by calling `SenderCommunicationChannel` (Line 11). How to transmit the detection result will be presented in Section V.

B. Detection Node Injection

1) Candidate Methods: Once the detection nodes are constructed, we next automatically inject them into the app. To achieve it, we first need to determine the candidate methods

Algorithm 1 Detection Node Injection

```
    D: a random subset of the constructed detection nodes
    G: the CFG of a candidate method

 1: function DetectionNodesInjection(D, G)
 2:     while D is not empty do
 3:         b ← SelectBlock(G) // randomly select a block from G
 4:         S ← FindDominators(b, G) // store dominators into S
 5:         Insert(d, S) // insert d into S
 6:         d ← deq(D)
 7:         n ← GetLineofCode(d) // get # of lines of code in d
 8:         m ← GetNumofBlocks(S) // get # of blocks in S
 9:         k ← RandomNum(1, min(n, m))
10:         C ← SplitDetectionNode(k, d)
11:         B ← SelectBlocks(k, S)
12:         for each c_i in C and each b_i in B do
13:             Inject c_i into b_i
14:         end for
15:     end while
16: end function
```

into which the detection nodes should be injected. We require the candidate methods should not be *hot* methods, for example, those keep running in the background. If a detection node is injected into a hot method, it will be executed over and again, incurring a high overhead. Instead, we consider relatively *cold* methods which are invoked for a few times during an execution of the app. There are a variety of methods satisfying the condition, for example, those invoked at initialization, exit, or phase transitions.

To assist selection of candidate methods, we profile the apps using Monkey [41] and Traceview [49]. Monkey is a tool that can generate pseudo-random streams of keystrokes, touches, and gestures, and Traceview a profiling tool that can log the execution trace. We first use Monkey to generate the pseudo-random stream of 100,000 user events and feed these events to the app. Simultaneously, we use Traceview to monitor the app at runtime to log the execution trace. The log contains the information about the invocation number of each method. We consider the methods which are called more than 50,000 times as "hot", and exclude them. The rest methods are used as candidate methods.

2) Injection Algorithm: To inject a detection node, we do not inject it as a whole; instead, we split it into several parts and inject each part separately in order to achieve better stealthiness.

Algorithm 1 shows the pseudo-code for detection node injection. Given the Control Flow Graph (CFG) of a candidate method and a randomly selected subset D of the constructed detection nodes, the function `DetectionNodesInjection` inserts each detection node in D to G. The algorithm weaves code of each detection node into the candidate method, so that detection nodes are difficult to locate and identify by attackers. To inject a detection node, it first randomly selects a basic block b from G (Line 3). Then all the dominators of d are found and are stored along with d into a set S (Line 4 and 5). In a CFG, a basic block a dominates a basic block b if every path from the entry block to b must go through a [34]. Next d is split into k parts and stored in C, and k basic blocks are selected from S and stored in B; k is a random integer between 1 and $min(n, m)$, where n is the number of lines of code in d, and m the number of basic blocks in S (Line 7 to 11); note that d is mainly a straight-line code sequence except for a few conditional basic blocks, each of which is considered as a single *line*. Finally, each part in C is sequentially inserted into B (Line 12 and 13) following the dominance order of these parts in the CFG.

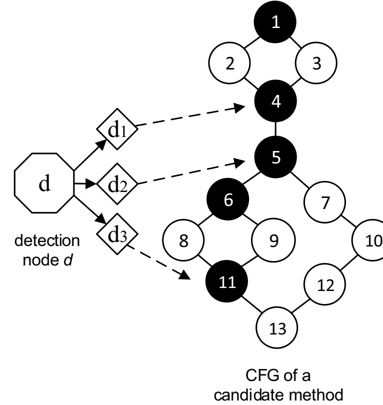


Fig. 3. An example of detection node injection.

The injection method guarantees that the execution order of the instructions in each detection node remains after injection. In addition, as we use different names for variables in different detection methods, it does not cause problems even when parts of multiple detection nodes are inserted into one basic block of a candidate method.

Figure 3 shows, as an example, how to inject a detection node into a candidate method. In the CFG of the candidate method, we first randomly select a basic block; assume it is node 11. Then we adopt the classic *Lengauer-Tarjan* algorithm [34], an efficient dominance algorithm, to find all the dominators of node 11: node 1, 4, 5, and 6. The five nodes (node 1, 4, 5, 6, and 11) are stored in a set S, and the detection node d is split into three parts: d_1, d_2, and d_3. The number of the split parts is a random integer not greater than the number of instructions in d and the number of nodes in S. We then randomly select three nodes from S; assume they are node 4, 5, and 11. Finally, we inject d_1, d_2 and d_3 into the three nodes, respectively.

IV. Repackaging Response

This section presents how to construct and inject response nodes with design and implementation details.

A. Response Node Construction

1) Stealthy-modification Mechanism: A few response strategies for tamper-proofing have been proposed [48], [13]. The response is usually conspicuous in the form of, for example, program crashes and infinite loops, which are very unusual behaviors. Such that attackers can quickly locate the injected response, and then try to trace back to the response injection code.

Instead of stopping apps from working, we propose to inject responses in the form of *delayed logical malfunctions*: (1) after a response is injected, it takes effect after some delay; (2) the response is in the form of logical malfunctions, so that when attackers notice it, little trace is left behind. Combined with stochastic responses (Section IV-C), delayed logical malfunctions make debugging and evasion attacks on the attacker side much more difficult.

We give some examples of responses in the form of delayed logical malfunctions. In the *OpenSudoku* game, assume a response has been injected, so that when a user selects a puzzle, instead of rendering the selected puzzle, some other puzzle is presented to the user. In this example, the response shows itself as some logical bug. Alternatively, the response may turn the app in some disorder, e.g., showing unrecognizable text, rendering a huge button, or supplying a textbox too small to type in. These effects accumulates very negative user experiences, so that few benign users continue using the app. On the other hand, the response is not evident in the beginning and it even does not show itself until some delay after it is injected; thus, when attackers decide to launch debugging, it is difficult to reproduce the problem and locate the response injection code.

To implement delayed logical malfunctions, we propose *stealthy-modification methods*, which stealthily modify particular features of an app so as to cause logical malfunctions. We consider two types of stealthy-modification methods. The first one is to modify integer variables; for example, modifying the Intent value to disturb inter-activity communication, modifying the attributes of Button, TextView, EditText objects, such as the size, visibility, and inputType, modifying the operands in arithmetic operations. The second type is to modify string variables, for example, appending a random string to it.

The modifications may result in program crashes instead of intended delayed logical malfunctions. For instance, modifying the array index may lead to buffer overflow exceptions and hence crashes. we can check against crashes, for example, by avoiding overflows in the previous example, but we choose not to do so in order to simplify the response nodes. Our goal is to achieve the delayed logical malfunctions in most situations rather than all.

2) Construction Process: To automatically construct the response nodes, we search the app's code to find all of the candidate variables which our stealthy-modification methods can apply to. We consider all class member variables (i.e., member fields) as candidate variables.

For each candidate variable, a stealthy-modification method is applied to constructing the response node. We then apply code obfuscation techniques on the constructed response nodes to make them difficult to reverse-engineer.

3) An Example: The following is the code snippet of a response node with respect to a candidate variable v.

```
1 //Infer repackaging detection result
2 bool flag = ReceiverCommunicationChannel();
3 //if flag is true, repackaging is detected.
4 if (flag) {
5   v += 3; //3 is a random number
6 }
```

As shown in this code snippet, the response node first monitors the communication channel to infer the detection result (Line 2; how to infer the detection result is presented in Section V). If repackaging is detected (Line 4), the response node modifies the candidate variable v by adding a random number to it (Line 5).

One detail is that when applying modifications to member variables of a class and inserting the corresponding response node to another class, the modifications are subject to the scope of the variables and availability of the assistant methods. For example, the attribute declared in the `Button` class are all member variables; they can only be modified through `set*` methods when methods of other classes are to modify them.

Algorithm 2 Response Nodes Injection

G: a candidate method

```
 1: function RESPONSENODESINJECTION(G)
 2:     α ← ReferencedCandidateVariables(G)
 3:     for each α_i in α do
 4:         λ ← FindJavaClass(α_i)
 5:         site ← FindInjectionSite(α_i, λ)
 6:         Injection(α_i, site)
 7:     end for
 8: end function

 9: function INJECTION(γ, Γ)
10:     (name, type) ← FindNameType(γ)
11:     r ← ConstructResponseNode(name, type)
12:     RandomInject(r, Γ)
13: end function
```

B. Response Node Injection

To automatically inject the response nodes, we first determine where to inject them. Again, we use relative cold methods as candidate methods (Section III-B1). Note that each response node is quite small and simple; thus, unlike injection detection nodes, we inject a response node as a whole rather than splitting it into multiple parts.

Algorithm 2 shows the pseudo-code for injecting the response nodes. Given a candidate method *G*, the function first finds candidate variables referenced by the method (line 2). Then for each candidate variable α_i, it determines a *proper* injection site (line 5). In order to cause a delay between the response injection site and the point where a response is evident, we define a method that does not reference the candidate variable as its proper injection site; such that, after a member variable is assigned with some problematic value by a response node, the variable is not used until the method containing this response node returns and another method references it. In this way, it creates extra difficulty for attackers to locate the response node when the malfunction is evident. If there exist multiple proper injection sites, a random one is selected. Then the response node is constructed and injected to the selected injection site (line 6).

C. Stochastic Responses

To improve the resilience to evasion attacks, we integrate the *stochastic* response mechanism based on nondeterministic programming. Nondeterministic programming is a programming strategy that does not explicitly specify which further action would be executed at a certain point, called *the choice point*; instead, it allows the program to make a choice among a number of alternatives based on a *selective method* at runtime [11]. In our case, each response node is a choice point, and the alternatives of a response node include whether or not the candidate variable is to be modified, and, if it is to be modified, the possible values that can be assigned to it.

Several selective methods have been proposed [11]. For example, one can generate a random number, based on which one of the alternatives is selected; or read an uninitialized variable and use its value to determine an alternative; or use the race condition on concurrent threads, or an unordered iteration,

or memory address, or time stamps, etc. We adopt the first one as our selective method.

V. Communication Channel

This section presents the system design of the communication channel. We first present the requirements for the communication channel in our system, and then propose a new type of communication mediums.

A. Stealth Communication Channel

To support the information transmission between detection and response nodes, we implement several different communication channels. Specifically, we divide all of the detection and response nodes into several sets, and implement a different communication channel for each set; this way, the detection and response nodes within the same set communicate based on the same communication channel.

We do not implement different communication channels for each pair of detection and response nodes. Because an app is an event-driven application, it is difficult to pre-determine whether both of detection and response nodes of the same pair will execute at an execution path. If not, there will be no response even if repackaging is detected, which will decrease the probability of successfully carrying out responses. We neither attempt to implement the same channel for all pairs of detection and response nodes, in order to improve the resilience.

We aim to implement a reliable stealthy communication channel that is difficult to detect and prevent. Since if the channel involves transmission errors, an app will still incur malfunctions even though it has not been repackaged. There are a few stealthy (covert) channels for Android apps proposed in the literature [22], [18], [54]. The first one is based on vibration settings; when they are changed, the system sends a notification to interested apps. The second one is based on volume settings; it is similar to vibration settings. The third one is based on the mobile phone screen; the changes on the screen will trigger a notification to interested apps. The forth one is based on Intent; the transmitted data is encoded into an additional field of an Intent. We do not adopt them for our system. For the first three channels, they incur some noise when a user himself performs some actions on the settings which cannot be avoided. In addition, they are not stealthy in the context of apps (few app modifies these settings); thus they can be easily identified by attackers. For the last one, the number of detection and response nodes that can be injected is restricted by Intent, which limits the feasibility of our system.

B. Communication Medium

We propose a new communication medium which can be used to implement a reliable stealthy channel that meets our requirements. The communication medium is a resource (a *final* variable) in the R class. The R class is a Java class, automatically generated by *aapt* (the Android Asset Packaging Tool), containing the resource IDs for all the resources in the *res* directory. For example, *R.drawable.icon* is a resource; it has a resource ID which is a static integer and can be used to retrieve the icon resource. We randomly pick several resources in the R class as our communication mediums, each of which is used to implement a reliable stealthy channel. We modify their resource IDs if repackaging is detected. Because a final variable cannot be modified once it has been assigned, we use Reflection to achieve it. Since there are usually many places in the original app that access the resources, the injected code are stealthy in the context of the app. Moreover, when DVM (Dalvik Virtual Machine) complies the source code to generate the Dalvik bytecode, it replaces the references to the resources everywhere in the code with the actual values, namely, the resource IDs; thus the modification on the resource IDs during runtime will not change the semantics of the app.

Because our communication channel relies on Reflection, it should be stealthy in apps that involves enough Reflection operations. In fact, Reflection is widely used in Android apps [21], [33], which is also confirmed in our evaluation. However, there are some apps in which Reflection are scarce. For these apps, one possible attack is to search all Reflection in the app and then disable/remove these Reflection one by one. To counter this attack, we generate *decoy* Reflection. Specifically, we deliberately select a few callee methods from the original code and transform them to utilize Reflection to call them; such generated Reflection are called *decoy* Reflection. In this way, attackers cannot distinguish the Reflection used in the communication channels from the *decoy* Reflection. Moreover, we apply code obfuscation techniques on the communication channels to make them difficult to reverse-engineer and improve the stealthiness. Furthermore, we adopt opaque predicate to add many instances of Reflection, which confuses attackers and increases their uncertainty as to whether or not they have successfully disabled/removed the communication channels, and thus improve the resistance of our system.

VI. Security Analysis

A. Static Analysis

One important issue is that if detection and response nodes contain fixed code sequences, they may be easily identified and removed by attackers through statically scanning (e.g., using pattern matching and text searching techniques). For detection nodes, because we construct them based on different predefined "polymorphic" templates and apply code obfuscation techniques, the resulted detection nodes are distinct. Moreover, we split each detection node into several parts and weave these parts into the original app code. In this way, it is quite difficult to identify the detection nodes. With regard to response nodes, because different response nodes target different variables, these response nodes are distinct. In addition, we also apply code obfuscation techniques on them, making them more distinct from each other and improving their stealthiness.

B. Dynamic Analysis

Attackers can also perform dynamic analysis to identify the detection and response nodes. For the detection nodes, if an attacker executes and observes the protected app in its dynamic environment, he may be able to identify the detection nodes and recognize the hard-coded public key substrings. After that, the attacker may replace the original public key substrings with his own one. However, since we inject multiple detection nodes in terms of the execution paths into the app, the attacker must iterate this attack for many times to visit all or most of the execution paths. Although automated test input generators can be leveraged to generate the possible inputs (e.g., Randoop [44], Robotium [46]), or directly analyze specific method calls (e.g., Brahmastra [4]), it still requires attackers to inspect the code to pinpoint the detection nodes and make any code modifications necessary so as to disable/remove them. Moreover, because

each detection node is obfuscated and then woven into the surrounding code without identifiable boundaries, it greatly increases the investment cost for attackers.

With respect to the response nodes, because our stealthy-modification mechanism causes the delayed logical malfunctions which leave little noticeable traces behind, it is very difficult to find the failure points (where the malfunctions explicit), let alone to trace back to the response nodes. Moreover, to analyze an app, attackers have to feed it with the same input many times and hopefully can observe the same output. However, because of our stochastic response mechanism, they are faced with different delayed malfunctions given the same input, rendering it a more complicated scenario to investigate.

Attackers may adopt the taint analysis to taint the communication mediums once identified, or *packages.xml* containing the public key. However, because we utilize string operations to manipulate the mediums and file name at runtime and transform them to Reflection for accessing the resources, the communication channel and protection nodes are difficult to be identified by current taint analysis. We examined taint analysis attacks in our evaluation Section VII-D3.

C. Replay Attacks

The last issue is replay attacks. We assume that attackers know we use the public key to detect repackaging. Because the public key is distributed with an app's apk file (contained in the selfsigned X.509 certificate "META-INF/CERT.RSA") for signature verification, an attacker can easily obtain it. However, because he does not know the private key, he has to choose a new certificate to sign the repackaged app. To bypass the repackaging checking of SSN, the attacker's goal is to provide the original public key to the detection nodes whenever the detection nodes ask for K_r. Because K_r is extracted from *packages.xml*, one possible way is to substitute the public key stored in *packages.xml* with the original one. However, this does not work. Because *packages.xml* is owned by the *system* user and *system* group, it cannot be modified by any app process, which means that once the public key is stored in *packages.xml*, it is protected by the Android system and cannot be modified by other processes.

Another possible way is similar to the man-in-the-middle attack. It refers to an attacker attempts to hijack an essential function, e.g., *getPublicKey*, so that whenever it is called, the original public key will be returned. However, when an app is launched, Zygote (an Android system service who launch all apps) creates a clone of itself and initializes a DVM which preloads the required system classes before the app is started; this way, the system classes are already loaded and linked by Zygote. Thus the attacker cannot override the path of the system class so as to load his own class to substitute the system class. In addition, because Android runs on Linux whose kernel implements a strategy called Copy-On-Write (COW), no memory is actually copied and the memory is shared and marked as copy-on-write, meaning that the system classes are read-only and not writable; thus, the attacker either cannot overwrite the system class after it is loaded.

A more advanced way is to *hijack vtable*. A vtable is a virtual function table, pointed by a virtual table pointer (*vfptr*). A vtable consists of several virtual function pointers. An attacker can either overwrite a vfptr to point to an attacker-crafted vtable, or overwrite a vtable's contents to manipulate virtual function pointers, causing further virtual function calls to be hijacked [55]. For instance, an attacker can swap the pointer of *getPublicKey* with his own defined method, such that whenever *getPublicKey* is called, the attacker's method will be executed. Our work does not deal with such attacks, as it is out of the scope of this work. Many techniques have been proposed to address this problem in the literature and can be adopted [55], [31], [23], [40].

TABLE I. FEASIBILITY ANALYSIS RESULTS

Category	Avg LOC	% of Apps use Reflection	Avg # of Detection nodes	Avg # of Response nodes
Development	5,684	66.7%	368	232
Office	6,268	40%	406	285
Multimedia	10,080	73.3%	658	344
Game	6,900	43.3%	455	276
Internet	15,170	56.7%	988	546
Security	10,609	36.7%	685	423
Reading	14,625	46.7%	946	471
Navigation	13,938	50%	887	483
Phone&SMS	13,391	50%	859	503
Science&Education	9,800	43.3%	623	443

VII. EVALUATION

A. Experimental Settings

We evaluated our tool SSN based on the following four aspects: feasibility, effectiveness, resilience, and side effects. We randomly collected 600 apps in 10 categories from F-Droid [20], which is an catalogue of open source Android apps. Each category contains 60 apps. We conducted our experiments on a Nexus 4 ARM emulator with Android 4.1 system. SSN is flexible to work on different devices with other Android versions. We choose to use emulator in our experiments as it is flexible to configure and does not affect our results and conclusions compared with using real devices.

B. Feasibility

We begin by studying the feasibility of SSN. We seek to understand: (1) the percentage of the apps using Reflection, and (2) whether SSN can inject enough detection nodes and response nodes into the apps. We show the results in Table I.

In Table I, there are five columns, including the average lines of code (LOC), the percentage of apps using Reflection, the average numbers of detection nodes and response nodes injected, with respect to the 10 categories of apps. Note that the LOC only includes the Java source code of each app.

For each app, the number of the response nodes is determined by the number of the candidate variables, and the number of the detection nodes is decided by the number of basic blocks in the candidate methods; specifically, for a candidate method has n basic blocks, we injected $\lceil n/5 \rceil$ detection nodes. Since the detection nodes are mutually independent, the more detection nodes are injected, the better resiliency of the repackage-proofing protection has; however, on the other hand, it results in more runtime overhead to the protected apps. Thus there is a trade-off between the number of injected nodes, the resiliency and the performance overhead. In our experiments, we chose to inject $\lceil n/5 \rceil$ detection nodes to a candidate method with n basic blocks, to achieve good resiliency and small overhead. However, one can easily modify the number for other choices based on the demand. For example, if he wants to protect a real-time app, he may choose to inject fewer nodes to attain smaller overhead. All the detection and response nodes for each app were divided into eight sets; each set corresponded

to an independent communication channel. The number of sets can also be configured based on the demand.

Table I also reveals that almost 51% of the 600 apps in our data set use Reflection. For some apps that do not involve Reflection, we generated *decoy* Reflection by selectively picking a few callee methods and transforming them to use Reflection to call them. For all the 600 apps, we also adopted opaque predicate to add many instances of spurious Reflection to confuse attackers. Moreover, we used proGuard [43], Shilef4J [47], and yGuard [53] to obfuscate the code, to make it difficult to reverse-engineer and improve the resistance.

From the results, we can conclude that (1) Reflection is widely used in Android apps, which is also demonstrated in previous work [21], [33], and (2) SSN can inject many detection nodes and response nodes into apps, and has good feasibility.

C. Effectiveness

We then evaluated the effectiveness of SSN. We want to understand: (1) whether or not SSN can prevent the repackaged apps from running successfully on user devices, and (2) the percentage of the repackaged apps that are deterred from working properly.

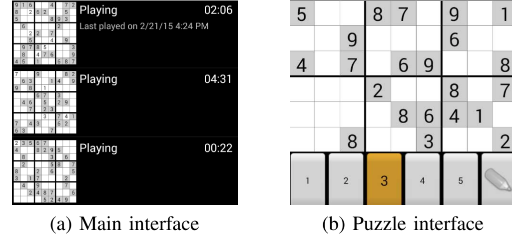
For each app, we first utilized SSN to build a repackage-proofing protection, and signed it using the original certificate to generate the apk file. These apk files are the protected apps. Next, we used Apktool [2] to repackage these protected apps and resigned it using a new certificate. After that, we obtained a set of repackaged apps. We installed these repackaged apps on our Nexus emulator, and tested each of them.

The results show that all of the repackaged apps could not run successfully on the device. Some repackaged apps crashed once they were launched, while some others could be launched but incurred various malfunctions and worked abnormally. Therefore, SSN successfully deterred all of the repackaged apps from working properly. We also evaluated the impact of the injected protection code on the functionality of the protected apps, which is presented in Section VII-E.

D. Resiliency

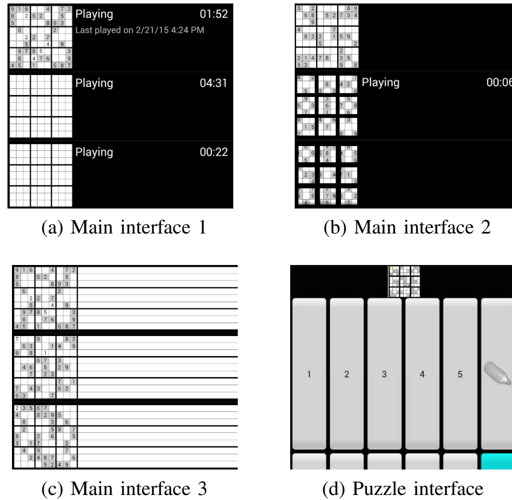
1) Resiliency to Static Analysis: Attackers may statically analyze the protected app to identify the detection nodes and response nodes. Note that our assumption is that attackers can only get the apk files, but cannot obtain the source code, which is reasonable in practice. Moreover, we also assume that the end users are legitimate ones and will not modify the apk files.

To analyze an app, the attacker has to first utilize a disassembler (e.g., *apktool*, *baksmali*) to disassemble the apk file and generate the smali files containing Dalvik bytecode. In the following, we assume that he has obtained the smail files. Then the attacker may statically scan the smail files for some essential functions (e.g., *getPublicKey*, *getCertificate*, etc.). However, because we use string manipulation and Reflection to hide the signatures of these functions, it prevents the attacker from finding them. The attacker may also scan the smali files for similar code segments if he has identified one node. However, because the constructed nodes are distinct and are applied on code obfuscation, the resulted nodes are difficult to be identified on the basis of another one. One more sophisticated attack is to adopt taint analysis to taint the communication mediums or *packages.xml*, so that whenever they are accessed, the location



(a) Main interface (b) Puzzle interface

Fig. 4. Examples of OpenSudoku when it works normally.



(a) Main interface 1 (b) Main interface 2

(c) Main interface 3 (d) Puzzle interface

Fig. 5. Examples of OpenSudoku when malfunctions occur.

of the nodes could be identified. The related evaluation results are presented in Section VII-D3.

2) Resiliency to Dynamic Analysis: An alternative attack is to dynamically execute and monitor the app to identify the injected nodes. To identify the detection nodes, one option is to combine debugging tools and automated test input generators to execute the app and observe its behaviour, and then pause the app to start debugging once a failure is noticed. A more advanced way is to hook some essential functions (e.g., *getPublicKey*) to learn when the hooked functions are activated so as to identify the code regions for the detection nodes. We conducted three user studies in the following. Attackers may also adopt dynamic taint analysis to taint the communication mediums or *packages.xml*. We present the related evaluation results in Section VII-D3.

We now consider how attackers identify the response nodes. Utilizing taint analysis is one option. Another option is to find the failure points, and then trace back to the response nodes. However, because our stealthy-modification mechanism causes the delayed logical malfunctions which leave little noticeable trace behind, it is very difficult to identify the failure points, let along to trace back to the response nodes.

Examples of stochastic logical malfunctions. Fig. 4 and Fig. 5 show the screenshots of OpenSudoku when it works normally and abnormally, respectively. From Fig. 5, we can see that when OpenSudoku is repackaged, it only incurs logical malfunctions; specifically, Fig. 5(a), (b) and (c) show the screenshots when OpenSudoku is subjected to stochastic responses; different malfunctions are incurred at different times. Besides these, there are some other stochastic malfunctions. For instance, when an attacker selects a puzzle, a different one may

TABLE II. SURVIVING PROTECTION NODES OF EACH APP.

App	LOC	Total # of Det. nodes	Total # of Res. nodes	% of Surviving Det. nodes	% of Surviving Res. nodes
CatLog	5667	378	231	82.3%	89.6%
PhotoGallery	2011	137	90	78.9%	76.3%
OpenSudoku	6079	404	253	81.4%	80.1%
CEToolbox	2422	168	194	75.0%	82.9%

be presented at a different time; or when an attacker selects a cell in the puzzle interface and types a number into the cell, a different number may be typed into an arbitrary cell at a different time. These stochastic malfunctions greatly frustrate attackers and maximize the workload. Other apps suffered from similar or different kinds of stochastic malfunctions. Due to limited space, we do not present them here.

Case studies. We now seek to understand how quickly an attacker can identify a node, and how many nodes can be identified in a given time, based on simple dynamic analysis. In principle, it depends on many factors (e.g., the attacker's skill, the complexity of an app, the automated tools available, etc.) and cannot be easily quantified.

We conducted three case studies. For each app, we generated its protected apk by SSN. To begin, we asked three human players to play these apks to confirm that the functionalities of these apps were still held. Then we provided these apks to another three human testers who did not know our certificate but know SSN's techniques, to analyze the apks separately for twenty-four hours and try to break the repackage-proofing protection built into the apps. The three human testers have three, two, and one years of Android development experiences, respectively. They are skilled in the debugging tools such as DDMS [17] and test input generators such as Monkey, Randoop [44], Robotium [46] and Brahmastra [4]. After twenty-four hours, they repackaged these apps using their own certificate (a new one) to generate the repackaged apps. Finally, we asked the previous three human player once again to play these repackaged apps for one hour; if the app works properly for one hour, we consider it has been successfully repackaged. Note that we do not require every nodes have to be disabled/removed to achieve successful repackaging. Instead, we claim that *an app is successfully repackaged as long as it can work properly for one hour*.

Table II shows the results of four sample apps, CatLog, PhotoGallery, OpenSudoku, and CEToolbox, from the category of Development, Multimedia, Game, and Science&Education, respectively. Other apps have shown similar results and are not included here. In Table II, the percentage of surviving nodes is the average percentage among the three human testers. We can see that even after twenty-four hours, almost four-fifths of the detection and response nodes are still not being identified. Moreover, none of the apps was successfully repackaged reported by the three players.

We then asked the three testers for their experience. They indicated that at first they tried to statically scan the apps but few useful information was obtained. They then had to dynamically execute the apps to observe their behaviour with the help of the debugging tools and test input generators; but unfortunately, they encountered different malfunctions each time. Moreover, the malfunctions caused the apps to work abnormally and they did not know where the malfunctions started. Thus they had to inspect the bytecode carefully using the debugging tools and try to understand the work flow of the apps which cost them a lot of time. When they had identified a detection node or response node, they tried to figure out the final variable that was used as the communication medium. They then adopted the taint analysis tools (e.g., TaintDroid [19]) to taint it and hoped to find the detection and response nodes easily; however, this did not work. Next, they set a watch window in the debugging tool to watch this variable at runtime to detect its modification; however, because the final variable was declared in the R class which was out of the scope chain for debugging, the watch window did not work. Therefore, they had to set multiple conditional exceptions in the code which would suspend the execution once the final variable was modified, and then gradually shorten the searching range. The entire process was very laborious, error-prone and time-consuming.

During their analysis, we also recorded the number of the identified nodes for every two hours. We found that the numbers were decreased during the twenty-four hours, indicating that the difficulty in detecting nodes was increasing. It is intuitive as the more nodes existed, the easier for triggering one, causing more nodes being executed in a certain time, and vise verse. Thus, we infer that less nodes would be detected after the twenty-four hours analysis.

Advanced dynamic analysis attacks. Attackers may perform more advanced dynamic analysis attacks. For instance, they may utilize *ptrace* to control the execution of a process, but to carry out it requires sufficient knowledge of low-level programming. Moreover, the rationale is that only one process can attach to a target process at the same time. To prevent attackers using ptrace, we can let the app attaches to itself at runtime; this way, ptrace cannot attach to it [56], [52]. Attackers may leverage existing binary instrumentation tools such as Adbi [1] and DECAF [25] to monitor some essential functions (e.g., *getPublicKey*), and use automated test input generators such as Monkey and Brahmastra to automatically execute the app, so that when these functions are activated, the locations of the corresponding nodes can be logged. However, some patches on these tools are needed to achieve this goal; in addition, most automated test input generators can only cover partial execution paths of an app [39], [45], [51]. Thus, the attacker still needs to analyze many traces and inspect the code manually to uncover the other nodes.

3) Resiliency to Taint Analysis: Attackers can also adopt taint analysis to taint the communication mediums or *packages.xml*, to discover the injected nodes.

Communication mediums. To apply taint analysis, the attacker must identify the final variables used as the communication mediums. We assume he finds one through some ways. For static taint analysis, since we adopt string manipulation on the final variable's name at runtime and transform that to Reflection for accessing the variable, it is difficult to detect the data flow between the sources and sinks (the places where the source is accessed), given the string is unknown statically. Moreover, static taint propagation usually makes the program end up with many false tainted values, and leads to high computation overheads and large memory consumptions. Thus, static taint analysis is impractical to counterattack our technique.

With respect to dynamic taint analysis, we utilized TaintDroid [19] and DECAF [25]. TaintDroid is a dynamic taint tracking and analysis system capable of tracking sensitive data. DECAF is a dynamic binary analysis platform based on QEMU. Assume the communication medium is a final variable *str*. For TaintDroid, we first added *Taint.addTaintString(str, Taint.TAINT_MIC)* at the beginning of the *create* method in the

main activity so that once the app started, *str* would be tainted. Then we inserted the log statements after each detection and response node to check whether or not TaintDroid could detect *str* was accessed. Next we employed Monkey to execute the app for one hour, and checked the log; however, nothing was logged. The reason is that the communication medium is a resource in the R class and is accessed by Reflection based on strings of its name after manipulated. Similar results were obtained from DECAF.

The *packages.xml* file. Attackers may attempt to taint *packages.xml* which contains the public key to reveal the detection nodes. As we use PackageManager to extract the public key, the file name is not explicitly stated in the detection nodes (see the code snippet in Section III-A2). Thus, static taint analysis is ineffectual under such circumstances.

For dynamic analysis by TaintDroid, we first added *Taint.addTaintFile(intFd, Taint.TAINT_MIC)* at the beginning of the *create* method in the main activity, where $intFd$ is a file descriptor of *packages.xml*, and then checked whether TaintDroid could identify *intFd* was accessed. However, TaintDroid failed to do so. The reason is that instead of directly reading the public key from *packages.xml*, we adopt PackageManager. The public key is extracted by *getPublicKey* from *sun.security.x509.X509CertImpl* in JDK that TaintDroid does not taint track.

In this evaluation, we used two state-of-the-art taint analysis tools. However, a better taint analysis may successfully taint both the communication mediums and *packages.xml*, which we recognize is possible theoretically. However, it still requires attackers to inspect the code manually to pinpoint the protection nodes and disable/remove them. As the nodes are obfuscated and woven into the surrounding code without identifiable boundaries, it increases the investment cost for attackers. Furthermore, because multiple nodes are injected in terms of the execution paths, attackers have to iterate this attack as many times to visit sufficient execution paths for insuring the safety of republishing the app. The cost may be high and no longer be attractive for attackers.

E. Side Effects

We further evaluated the side effects of SSN on the apps, from the following two aspects: the impact on the functionality, and the impact on the runtime overhead.

1) Impact on the Functionality: The normal operations of an app should remain intact after protected. To evaluate it, we utilized SSN to build repackage-proofing protection into each app, to generate the protected app. We then asked three human players to play these protected apps for two hours to check whether they could run successfully. All they reported were positive results. Therefore, the protection code injected by SSN has no impact on the functionalities of the apps.

2) Impact on the Runtime Overhead: The application runtime overhead introduced by SSN comes from two major types of computations: (1) the reading public key certificate procedure by PackageManager, and (2) the Reflection operation modifying the value of a final variable. To evaluate the runtime overhead, we utilized Monkey to generate a sequence of 10,000 user events, and fed the same user events to both the apps with and without the protection code embedded three times to measure the average running time. We denote the average running time of the original app and the protected app as T_o and T_p, respectively; the runtime overhead is calculated by $(T_p - T_o)/T_o$. Table III shows the runtime overhead with respect to the four apps. Results with other apps are similar and are not included here due to the space limitation.

TABLE III. RUNTIME OVERHEAD.

App	T_o (sec)	T_p (sec)	App Perf. Overhead
CatLog	185	208	12.4%
PhotoGallery	108	115	6.4%
OpenSudoku	209	227	8.6%
CEToolbox	141	152	7.8%

From Table III, we can see that the runtime overhead has a positive correlation with the number of the detection and response nodes injected (however, CatLog is an exception). For example, the amount of the detection and response nodes injected into PhotoGallery is smaller than those injected into CEToolbox; and the runtime overhead of PhotoGallery is lower than that of CEToolbox.

To understand the reason why the runtime overhead of CatLog is particular higher than others', we examined its source code. CatLog is an app showing a scrolling view of the Android "Logcat" system log, which has multiple reading log procedures that read logs line by line to examine the context of logs (e.g., similar to text searching). The reading log procedures are usually coded as highly repetitive loop statements (e.g., *while((line=read.readLine())!=null)*). If one or more nodes are placed within the highly repetitive loop, they will be executed as many times as the loop iterates, which will greatly sacrifice the performance. Although during the injection, we utilize Traceview to exclude the "hot" methods, we do not exclude the highly repetitive loops. Indeed, in many cases, the nodes do not need to execute over and over again if all they do is to repeat the same self-checking or response action.

To improve the performance, one can opt for a smarter injection strategy. One strategy is to avoid injecting nodes within the performance-sensitive code segments. For instance, one can generate the CFGs of each candidate method, and then exclude the highly repetitive loops so that none of the nodes will be injected into these loops. In this way, one can quantitatively evaluate the trade-off between the runtime overhead and the resilience based on the demand. Another strategy is to declare a boolean variable for each node to make sure that it will execute only once. We adopted the second strategy and generated another set of the protected apps containing exactly the same number of the detection and response nodes except that in this case each node would execute only once. We redid the experiment and calculated the new runtime overhead; the new result for CatLog is reduced to 8.2%.

From the results, we can see that SSN incurs relatively small runtime overhead to the protected apps, and thus is efficient to defend apps against repackaging.

VIII. DISCUSSION

Impacts. SSN is the first work reported in the open literature that prevents repackaged apps from working on user devices without relying on authorities. It builds a complex stochastic stealthy network of defences into apps, such that repackaged apps cannot run successfully on user devices. Unlike repackaging detection techniques based on code similarity comparison that can be easily evaded by various obfuscations, SSN can resist this kind of evasion attacks. No matter whether an app is obfuscated by attackers in order to bypass the code similarity checking, as long as it has been built with the repackage-proofing protection before releasing, the repackaged app cannot run successfully on user devices, which limits its

propagation as well as its harms to the financial security of app developers and user privacy. Thus, repackage-proofing that stops repackaged apps from working is a promising direction in defeating the prevalent repackaging attacks.

Limitations. There are several limitations of SSN. First, although SSN has good resiliency to many evasion attacks (as showed in our evaluation), we do not guarantee that determined attackers cannot disable our protection from an app. For example, the attacker may spend much effort and time monitoring the app's execution to identify the code region for the detection nodes. Or he may leverage a better taint analysis techniques (if available) to taint both the communication mediums and *packages.xml*, to obtain the tainted data flow revealing the injected nodes.

Attackers can inspect the app code manually to pinpoint the nodes and make any code modifications necessary to disable/remove them. Note that attackers do not need to remove every last node to successfully repackage an app. As long as they ensure that the repackaged app can work properly long enough (e.g., one hour, or longer), they tend to be satisfied. Each time after removing a node, attackers need to confirm the functionality of the app is not broken and the node is successfully bypassed.

However, because of the stochastic response mechanism, they cannot be sure. Moreover, we can also selectively inject many spurious nodes to confuse attackers and increase their uncertainty. Furthermore, as multiple nodes are injected in terms of the execution paths, attackers have to iterate this attack as many times to visit sufficient execution paths for guaranteeing the safety of republishing the app. To increase their risks, we can opt for injecting nodes that notify users the app has been repackaged and may be dangerous, into *infrequently executed* paths or methods. This way, even only those nodes are remained, and activated only once or twice a day or month, there is still chance that users are informed of the danger and report it to Google Play, which then removes the repackaged app from all user devices. We leave these as our future work.

Another limitation is that attackers can conduct *hijacking vtable* attacks, which either overwrite a virtual table pointer or manipulate a virtual function pointer, to bypass the repackaging checking. We refer the reader to Section VI-C for details. There exist many techniques addressing this problem in the literature that can be adopted [55], [31], [23], [40].

It is widely recognized that any software-based protection can be bypassed as long as a determined attacker is willing to spend time and effort, which is also true with our protection. We assume attackers are interested in repackaging an app only if it is cost-effective, for example, when the cost of repackaging is less than that of developing the app themselves.

IX. Related Work

A. Code encryption and decryption based approaches.

Aucsmith [3] proposes an approach utilizing cryptographic methods to decrypt and encrypt code blocks before and after each execution round. However, this method cannot be done in a stealthy way in bytecode and does not scale well because of the time taken by encryption and decryption. Wang et al. [50] propose a dynamic integrity verification mechanism designed to prevent modification of software. The mechanism utilizes multi-blocking encryption technique to encrypt and decrypt code at runtime, which needs no hash value comparison. Cappaert et al. [5] also propose an approach which enciphers code at runtime, relying on other code as key information; this way, any tampering will cause the code to be decrypted with a wrong key and produce incorrect code.

B. Self-checksumming based approaches.

Chang et al. [6] define small pieces of code called *guards*, to compute checksums over code fragments. However, the code checking operation has to involve a call to a custom class loader, and thus can be easily found and bypassed; moreover, the guards are hard to automatically constructed and the maintenance cost is very high. Horne et al. [26] 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. [11] implement a large number of lightweight protection units to protect any critical regions of a program from being modified. This protection scheme supports non-deterministic execution of functions, resulting in different execution paths and nondeterministic tamper responses. Our stochastic response mechanism is inspired by it and has a similar fashion. Jakubowski et al. [29] present software integrity checking expressions, which are program predicates, to dynamically check whether or not a program is in a valid state. Jakubowski et al. [30] further propose a scheme to transform programs into tamper-tolerant versions that use self-correcting operation as a response against attacks; it chops a program into blocks, which are duplicated, individualized, and rearranged.

C. Oblivious hashing based approaches.

Chen et al. [10] propose oblivious hashing that implicitly computes a hash value based on the actual execution of the code to verify the runtime behaviour of the software; however, it requires pre-computation of expected hash values under all possible inputs; thus, it can only be applied to relatively simple functions that produce deterministic hash values. Chen et al. [7] propose a tamper-proofing software technology for stack-machine based languages, such as Java, by improving oblivious hashing. Jacob et al. [28] present an approach which overlaps a program's basic blocks so that they share instruction bytes to improve the tamper-resistance.

One work we know of aiming to improve the stealthiness of the tamper-response mechanism is that of Tan et al. [48]. They introduce a delayed and controlled tamper response technique which makes it difficult to detect tamper responses; the delayed failures are achieved by corrupting a global pointer at well-chosen locations. Although their technique is more tamper-resistant than others that directly cause programs to fail, it still reveals information to attackers for finding tamper responses. Unlike their approach, we attempt to cause the delayed logical malfunctions, such that it is very difficult to find the failure points, as well as trace back to the protection code.

X. Conclusion

To the best of our knowledge, there is no study that investigates repackage-proofing to prevent repackaged apps from working on user devices. In this paper, we conduct a preliminary first-step study on the problem and introduce a repackage-proofing technique called SSN, Stochastic Stealthy Network, which provides a reliable and stealthy protection for Android apps. We have developed a prototype. Our experimental results show that SSN is effective and efficient.

XI. Acknowledgement

This work was supported in part by NSF CCF-1320605, NSF CNS-1422594, NSF CNS-1223710, and ARO W911NF-13-1-0421 (MURI).

References

[1] Adbi, https://github.com/crmulliner/adbi.

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

[3] D. Aucsmith, "Tamper resistant software: An implementation," in *Information Hiding*, 2005.

[4] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall, "Brahmastra: Driving apps to test the security of third-party components," in *USENIX Security*, 2014.

[5] J. Cappaert, B. Preneel, B. Anckaert, M. Madou, and K. D. Bosschere, "Towards tamper resistant code encryption: Practice and experience," in *ISPEC*, 2008.

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

[7] H.-Y. Chen, T.-W. Hou, and C.-L. Lin, "Tamper-proofing basis path by using oblivious hashing on Java," in *Information Hiding*, 2007.

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

[9] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, WeiZou, and P. Liu, "Finding unknown malice in 10 seconds: Mass vetting for new threats at the Google-Play scale," in *USENIX Security*, 2015.

[10] 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.

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

[12] C. Collberg, G. Myles, and A. Huntwork, "SandMark–A tool for software protection research," *IEEE Security and Privacy*, 2003.

[13] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation-tools for software protection," *TSE*, 2002.

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

[15] ——, "Scalable semantics-based detection of similar Android applications," in *ESORICS*, 2013.

[16] C. Davies, http://www.slashgear.com/95-android-game-piracy-experience-highlights-app-theft-challenge-15282064/.

[17] DDMS, http://developer.android.com/tools/debugging/ddms.html.

[18] L. Deshotels, "Inaudible sound as a covert channel in mobile devices," in *USENIX WOOT*, 2014.

[19] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *OSDI*, 2010.

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

[21] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *CCS*, 2011.

[22] W. Gasior and L. Yang., "Exploring covert channel in Android platform," in *CyberSecurity*, 2012.

[23] R. Gawlik and T. Holz, "Towards automated integrity protection of C++ virtual function tables in binary programs," in *ACSAC*, 2014.

[24] 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 *DIMVA*, 2013.

[25] A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin, "Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform," in *ISSTA*, 2014.

[26] 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.

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

[28] M. Jacob, M. H. Jakubowski, and R. Venkatesan, "Towards integral binary execution: Implementing oblivious hashing using overlapped instruction encodings," in *Proceedings of the 2007 ACM Multimedia and Security Workshop*, 2007.

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

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

[31] D. Jang, Z. Tatlock, and S. Lerner, "SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks," in *NDSS*, 2014.

[32] Y.-C. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu, "Value-based program characterization and its application to software plagiarism detection," in *ICSE*, 2011.

[33] H. S. Karlsen, E. R. Wognsen, M. C. Olesen, and R. R. Hansen, "Study, formalisation, and analysis of Dalvik bytecode," in *the 7th Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, 2012.

[34] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," in *TOPLAS*, 1979.

[35] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *CCS*, 2003.

[36] D. Low, "Protecting Java code via code obfuscation," in *Crossroads'98*.

[37] Z. P. Ltd, "Java obfuscator–Zelix KlassMaster," http://www.zelix.com/klassmaster/features.html.

[38] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *FSE*, 2014.

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

[40] M. R. Miller and K. D. Johnson, "Using virtual table protections to prevent the exploitation of object corruption vulnerabilities," 2012, US Patent App. 12/958, 668.

[41] Monkey, http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[42] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang, "Plagiarizing smartphone applications: attack strategies and defense techniques," in *ESSoS*, 2012.

[43] ProGuard, http://proguard.sourceforge.net.

[44] Randoop, https://code.google.com/p/randoop/.

[45] C. Ren, K. Chen, and P. Liu, "Droidmarking: Resilient software watermarking for impeding Android application repackaging," in *ASE'14*.

[46] Robotium, https://code.google.com/p/robotium/.

[47] Shielf4J, http://shield4j.com/.

[48] G. Tan, Y. Chen, and M. H. Jakubowski, "Delayed and controlled failures in tamper-resistant systems," in *Information Hiding*, 2006.

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

[50] P. Wang, S. kyu Kang, and K. Kim, "Tamper resistant software through dynamic integrity checking," in *SCIS*, 2005.

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

[52] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu, "AppSpear: Bytecode decrypting and DEX reassembling for packed android malware," in *Research in Attacks, Intrusions, and Defenses*, 2015.

[53] yGuard, https://www.yworks.com/en/products_yguard_about.html.

[54] M. Yue, W. H. Robinson, L. Watkins, and C. Corbett, "Constructing timing-based covert channels in mobile networks by adjusting cpu frequency," in *HASP*, 2014.

[55] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, "VTint: Protecting virtual function tables' integrity," in *NDSS*, 2015.

[56] Y. Zhang, X. Luo, and H. Yin, "DexHunter: Toward extracting hidden code from packed android applications," in *ESORICS*, 2015.

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

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

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