Leveraging Information Asymmetry to Transform Android Apps into Self-Defending Code against Repackaging Attacks

Kai Chen, *Member, IEEE,* Yingjun Zhang,and Peng Liu, *Member, IEEE*

Abstract—By simply adding malicious code or advertisements in legitimate smartphone apps, attackers could benefit a lot from repackaging. The existing license protection mechanisms can be easily subverted by repackaged apps. A major defense is to detect. However, detection requires finding at least two "similar" apps simultaneously. We propose a *self-defending* approach: let a repackaged app automatically expose itself. However, it is very challenging to achieve this goal. *If developers and smartphones/users do not share any secret, attackers' app repackaging studio would be able to do whatever legitimate smartphones/users are able to do.* We find that there exists a unique information asymmetry between developers and attackers. Leveraging this asymmetry, our new SDC (self-defending code) approach encrypts parts of the app code at compile time and dynamically decrypts the ciphertext code at runtime. Different from previous work, the key is derived from both the information asymmetry and the app's checksum. Once the app is repackaged, the changed checksum will let the app run abnormally, further exposing the repackaging. The information asymmetry protects the key from being attacked. We build a smartphone anti-repackaging system prototype. To the best of our knowledge, this is the first work that lets repackaged apps automatically malfunction while having none effect on a benign app's function.

Index Terms—Android, repackaged apps, self-defending code, checksum.

1 Introduction

From PC era to smartphone era, a big change from security point of view is that repackaging plays a dominant role in malware spreading. According to a recent study [1], 1083 out of 1260 malware samples (or 86.0%) repackaged legitimate apps to insert malicious payloads, indicating that mobile malware prefers to use repackaged apps as the vehicle for propagation. How to deal with the repackaging problem is at the center of smartphone security research.

1.1 Existing Measures

Category 1. License protection mechanisms (including APK file size, MD5 hash/checksum, and signatures). One representative mechanism is the licensing service [2] offered by Google Play. By querying Google Play using the License Verification Library (LVL) at run time, apps can obtain the licensing status for the current user, then decide whether further use is allowed. However, the apps protected by LVL can still be repackaged. AntiLVL [3] aims to subvert standard license protection methods (e.g., the Android LVL) by decompiling apps with baksmali [4] and rewriting those methods to always return successfully.

Category 2. App review process. Google Play has the application approval process for automatically reviewing apps before publishing them. However, this process is mainly based on the content guidelines enforcement and malware detection [5], and this process can usually be easily circumvented [6]. Manual check does not work either due to large number of apps in the market.

- *K. Chen is with State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences; and also with the University of Chinese Academy of Sciences, Beijing, 100195, China. E-mail: chenkai@iie.ac.cn*
- *Y. Zhang is with with Institute of Software, Chinese Academy of Sciences, Beijing, 100190, China.*
- *P. Liu is with College of IST, Penn State University, PA 16801, U.S.A.*

Category 3. Obfuscation. Obfuscation techniques such as Proguard [7] and DexGuard [8] could confuse attackers in the process of repackaging. However, the obfuscation techniques can only increase the difficulty of reverse engineering an app, but not stop attackers from repackaging (e.g., inserting malicious code).

Category 4. Detection techniques (including apps comparison [9], [10], [11] and watermarking [12]). Managers of Android markets compare the apps in their markets and/or other markets to detect repackaged apps. The detection requires finding at least two "similar" apps simultaneously. Although detection plays a very important role in understanding the landscape of app repackaging, bridging the gap between detection and removal of repackaged apps from users' smartphones is costly and time-consuming. Due to false positives and false negatives of automatic detection techniques, human-in-the-loop investigation is often needed.

1.2 Our Defense Goal

Instead of further improving the effectiveness of repackaging detection, we seek a *self-defending* approach. In this way, we could have the following benefits.

- No need to compare apps. It means there is no need to collect apps from Android markets for comprehensive comparison. Also, all the problems of comparison (e.g., cannot compare obfuscated code, and human-in-the-loop) can be avoided.
- No false positive.
- No gap between detection on markets and removal of repackaged apps on users' smartphones.

In particular, we aim to let repackaged apps themselves expose their "fake" code authenticity. Our defense goal is as follows: due to our self-defending design, within a certain period of running time (whenever a repackaged app is used), the repackaged apps will run abnormally (e.g., crash) with very high likelihood. Our design supports users to check the reason of the abnormal run. If

the reason is due to repackaging, users will gradually stop using the apps. Also, users can notify markets like Google Play about the repackaged apps. Then the market may blacklist the app and even close the account holding the app, which further gradually enables the markets to self-heal from repackaging.

1.3 Our Approach and Main Contributions

We realize that it is very challenging to achieve this goal. *If developers and smartphones/users do not share any secret, attackers' app code repackaging studio would be able to do whatever legitimate smartphones/users are able to do.* That is, attackers will be able to take advantage of a *white-box* app running environment. Facing such a powerful and daunting repackaging studio, we identify the following prerequisites which must be met by any effective "Letting repackaged apps expose themselves" approaches.

Prerequisite 1 (existence of information asymmetry) Not all information contained in the source code can be collected by the attacker's repackaging studio with *cost* lower than redeveloping the target app. Here, we assume the attacker only knows the compiled code, not source code. We also assume that if repackaging costs more than (professional) redeveloping, the attacker would rather redevelop the target app.

Prerequisite 2 (ability to leverage information asymmetry) The information asymmetry can be leveraged to let repackaged apps run abnormally (e.g., crash). Otherwise, attackers can still repackage the app.

Prerequisite 3 (non-harming) Information asymmetry should not harm the legitimate apps.

Problem statement. *How to satisfy the three prerequisites while achieving the self-defending goal of "letting repackaged apps expose themselves" on Android?* Note that our goal is not to prevent all the malicious activities from happening before the crash point.

Our approach. We found that the self-defending code (SDC) mechanism can result in *information asymmetry*. This property says that after the SDC transformation, the code contained in a set of branches (e.g., `if` statements) is no longer in cleartext. Instead, the code is encrypted using a one-way function. Moreover, the keys used to encrypt the code are partially derived from the constants in the original app code, which are further concealed after the SDC transformation. Nevertheless, we found that leveraging this information asymmetry directly to protect Android apps would be defeated by Android specific counter attacks. Accordingly, our approach takes the first steps to effectively transform Android apps into self-defending code not only against repackaging attacks, but also against the related counter attacks.

Our effort results in a novel mechanism denoted *SDC and Twin-SDC*. As an Android specific mechanism, it leverages source code information asymmetry and successfully transforms (the code contained in) Android apps into self-defending code. When such code is not repackaged, the code will always run correctly as if the app were not transformed. However, when such code is repackaged by the attacker, it will automatically let the app run abnormally within a certain time window. SDC utilizes the app's checksum as part of the key to encrypt a piece of code. Once the app is repackaged, the checksum will be changed, which will let the decryption fail to recover the original code. Running the wrongly decrypted code will automatically let the app run abnormally, which can be caught by our approach to ensure the repackaging. In addition, because the decryption key is also partially derived from the information asymmetry, the attacker cannot figure out the key.

Specifically, we design two schemes. The first one serves manufactures who can customize the Android system. By adding an instruction into Dalvik to perform the decryption operation, the scheme can be supported from system level. Note that the smartphone does not need to be rooted. Apps not having SDC can also run normally as usual on the smartphone without calling the newly added decryption operation. The second scheme does not modify Dalvik or the Android system, only using native code to perform the decryption operation which maximizes the compatibility. Both the two schemes are opaque to developers, which means no additional effort will be imposed on developers.

The main contributions are as follows.

- To the best of our knowledge, our anti-repackaging system is the first work that transforms the code contained in Android apps into self-defending code.
- We designed and implemented two schemes of the SDC system. Both can work in the real world. One scheme works with no modification on Dalvik or the Android system.
- Twin-SDC is Android specific and very different from the techniques in the literature of encryption-based code obfuscation.
- We evaluated the system. After over 500 hours testing (including 480 hours "robot testing", 20 hours "manual testing" and another 20 hours "manual testing" by 10 different human testers), we demonstrate that our approach is resilient to replacement attack. It has a minimum performance overhead.

2 Self-Defending Code and Scheme I

We first show the motivation and design of *self-defending code* (SDC). Then we give the first scheme to overcome the challenges of current integrity checking techniques.

2.1 Motivation

To let repackaged apps automatically expose themselves, one idea is to add the code for integrity checking. However, current integrity checking techniques face the following two challenges. Firstly, all the code is exposed to attackers. Attackers could spend money to buy the apps and become legitimate users. Thus, all the functionalities are exposed to attackers (including the code for integrity checking). Secondly, code encryption without leveraging information asymmetry would not work. A developer may encrypt some code in the program to make it difficult for attackers to read or modify, and conceal the keys inside apps. When the app including all the code is exposed to attackers, the keys can be located.

To address these two challenges, we need to discover a kind of asymmetric information which is known to developers but not attackers (Prerequisite 1). By deriving the key from this asymmetric information, we could encrypt some code inside an app to make it unreadable to attackers (Prerequisite 2). Since this information is unknown to attackers, they cannot find the keys inside an app.

Adversary Model. We assume that the adversary has enough experience to analyze a given app. Particularly, the attacker can build a running environment (e.g., a special emulator) to run the app and observe the execution of any code running in the environment. We also assume that the attacker can spend money to buy the license of the app, which means that he has the ability to explore any

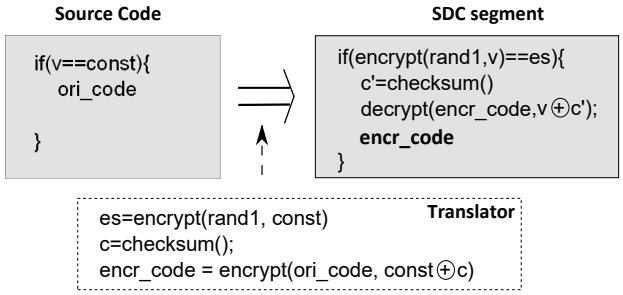

Fig. 1. The design of SDC.

functionalities in the app as a normal legitimate user. He can also repackage the app in whatever way he likes. On the other hand, we assume the attacker is reasonable. He cannot spend unlimited time and money to analyze an app. Whenever he finds that the cost of analyzing/repackaging a given app exceeds the benefits (e.g., revenues) that he can gain, he will stop analyzing/repackaging the app. Considering that the least cost is to create an account for uploading apps in a market (e.g., $25 in Google Play), the attacker should let his repackaged apps impact as many users as possible. So we do not consider an attacker who only expects a repackaged app to run once or to target a very limited number of users.

2.2 Self-Defending Code

Our basic idea of SDC design is adapted from [13] with modifications. SDC is a special form of self-modifying code. Different from ordinary self-modifying code, we use the decryption to restore the original code instead of modification. Different from [13], we use the checksum of an app's code as part of the key. Figure 1 shows how to generate SDC. The left box is the source code before the SDC transformation. The right box is the transformation result. The bottom box tells what is done by the *translator* during the transformation. The transformation result is also a piece of code, which we call a *SDC segment*. Each SDC segment includes an *entry door* and a *decryption operation*.

An entry door is a boolean-valued expression (*boolean-expression*) of an `if` statement. It is in the form of `encrypt(rand1, v)==es`. "`rand1`" is a randomly generated string by the translator. `v` is part of the app state (e.g., a variable) at the entry door. `es=encrypt(rand1, const)`, which is also computed by the translator. As shown in the "Source Code" box in Figure 1, `const` is the constant value used in the condition of the `if` statement. `const` is an essential part of the app logic. It is not a value manipulated by the translator or security officer. In runtime, when the entry door is satisfied (note that the door will not be "opened" unless the current value of variable v is exactly the constant value used in the original source code), we say the SDC segment is *triggered*.

The decryption operation combines `v` and the checksum `c'` of the app (not including `encr_code`) in memory to decrypt `encr_code`. `encr_code=encrypt(ori_code,const⊕c)`, which is computed by the translator. It is the ciphertext code of the executable of the original source code `ori_code`. `c` is the checksum of the app (including manifest file, resources and code except `encr_code`) computed by the translator. We exclude `encr_code` because we cannot let it change the checksum (which is part of the key). Section 2.4 shows how to protect the integrity of `encr_code`. Particularly, the algorithm to compute the checksum is MD5 which is also popularly used in many previous studies.

In the running process, when the SDC segment is triggered, the checksum of the app will be computed. The original app with the correct checksum will let the app run normally as usual. However, a repackaged app with a different checksum will let the decryption fail to recover the original checksum, which will further let the app run abnormally. We also design SDC Auditor (Section 2.4) to check whether the abnormal run (e.g., crash) is caused by repackaging or other reasons (e.g., the crash caused by the original app). If due to repackaging, users would stop using the repackaged app and can further report it to Google Play which will further take actions to remove the app and even close the account.

SDC is one-time use. Once a SDC is triggered, the secret key (i.e., `const`) is exposed to attackers, which allows an experienced attacker to revise the code inside the SDC segment. To counterattack this kind of repackaging, our idea is to find a good number of SDC segments in an app. Since different SDC segments use different keys, there are still many other SDC segments protecting the app even if a few SDC segments are compromised by the attacker. Then the question is how many SDC segments could be found in an app. Fortunately we find that SDC segments could be located anywhere in an app (Section 2.5). They are created from specific types of branches (i.e., `if` statements) which meet the two conditions: (1) use equality test (such as `==` operator, `startswith()`, `ends-with()` or `equals()` routines); (2) one of the equality test operands is a constant numerical/string value. We find hundreds of such branches in most Android apps (Section 5.1), which benefits developers deploying the self-defending code. We also find that the second SDC segment could be triggered very soon after the first SDC segment is triggered (based on our evaluation in Section 5.2 and Section 5.5), which means that the time left for the attacker to make revenues is very limited. Furthermore, once any user finds that the app is repackaged, she or he can alert Google Play, and Google may blacklist the app and even close the account holding the app. Since the account costs an attacker $25 [14], based on our adversary model, a reasonable attacker could not afford to continuously lose such accounts.

2.3 Why SDC satisfy the three prerequisites?

Satisfying Prerequisite 1 (Information asymmetry): Note that the key (i.e., `const`) is not in the transformation results. It shows that, the original developer knows both `const` of the original branch condition and the enclosed branch code (`ori_code`). In contrast, an attacker does not know this information at all.

Satisfying Prerequisite 2 (Leverage information asymmetry): If the apps is repackaged, the dynamically computed checksum `c'` is no longer the original one. So the decrypted `encr_code` would be a bunch of random bytes, which will make the app automatically crash. Our SDC Auditor (Section 2.5) can report that the crash is caused by repackaging.

Satisfying Prerequisite 3 (Non-harming): When the boolean-expression is met, SDC dynamically computes the checksum `c'`. If the app is not repackaged, `c'=c`. `encr_code` will be correctly decrypted and executed.

2.4 Protecting encr_code

An SDC segment cannot protect its own `encr_code`. So attacks could be performed on it (e.g., wrap it with malicious code). To protect `encr_code`, our idea is to ask each SDC segment to check the integrity of `encr_code` in several randomly selected others. The watcher code is in ciphertext code. Even if the

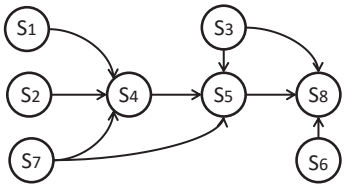

Fig. 2. An example of protecting SDC segments.

attacker can wrap all the `encr_code` blocks, the watcher code will still get executed and then detect the existence of malicious wrapping. Here the weakness of the attacker is that he cannot afford to delete or replace `encr_code`, because if so he may lose certain essential app functionality. Note that the watcher code keeps the checksums of both the protected `encr_code` and its decrypted code. In this way, no matter the protected `encr_code` is decrypted or not (depends on whether the corresponding SDC segment is executed), the watcher can guarantee that the protected code is not tampered.

Suppose there are n SDC segments (referred to as a set $S = \{s_1, ..., s_n\}$) in an app. We first randomly select an SDC segment (e.g., s_i) from S as a watcher to check the integrity of randomly selected m SDC segments (referred to as S') from the other $n-1$ ones (i.e., $S - \{s_i\}$). Then we repeat this process to randomly choose a watcher SDC segment S_j from $S - S' - \{s_i\}$ to watch m newly selected SDC segments (referred to as S'') from $S - \{s_j\}$. We keep choosing a watcher SDC segment from $S - S' - S'' - \{s_i, s_j\}$ and continue this process until there is no element for us to choose. Figure 2 shows an example. Three SDC segments s_1, s_2, s_7 are watched by s_4; s_5 watches s_3, s_4, s_7; and s_8 watches s_3, s_5, s_6. Note that one SDC segment could also be watched by several SDC segments. As long as a watcher SDC segment is not compromised, detection is guaranteed when it is triggered.

In runtime, the execution of SDC segments does not need to follow the order in which the watcher SDC segments are selected. As mentioned above, a watcher keeps the checksums of both the protected `encr_code` and its decrypted code. For example, in Figure 2, s_1 can be executed either before or after the execution of s_4. If s_1 is executed before s_4, the `encr_code` in s_1 will be decrypted. When s_4 is executed afterward, it calculates the checksum of the decrypted `encr_code` and compares it with the stored checksum (computed at compile time). Having the same checksum indicates that `encr_code` is not tampered. On the contrary, if s_4 is executed before s_1, `encr_code` in s_1 will not be decrypted. s_4 calculates the checksum of `encr_code` and compares it with the stored checksum for verification. In this way, the execution of SDC segments does not need to follow the order in which the watcher SDC segments are selected.

2.5 Scheme I

We design two schemes towards "Letting repackaged apps automatically expose themselves". The first one serves manufactures who can customize the Android system. The second scheme does not modify Dalvik or the Android system, only using native code to perform the decryption operation which maximizes the compatibility. Both the two schemes do not require the smartphone to be rooted. Apps without SDC can run as usual on the smartphone. We elaborate the first scheme as follows and the second one in Section 3.3.

Scheme I is essentially an Android-specific implementation of the SDC design in Figure 1. Figure 3 shows the overall deployment model which consists of three major components: SDC Generator, a modified Dalvik which supports a new instruction for decrypting ciphertext code, and SDC Auditor. SDC Generator is used by developers. It is fed with source code and outputs DEX bytecode with SDC. It is packaged together with resources in an APK package (i.e., an app). Then developers submit the app to Android markets for users to download. In the end, the app will be executed as usual by Dalvik in users' smartphone. Manufactories customize Dalvik to let it support decrypting ciphertext code. In this way, SDC will be executed through the new instruction in Dalvik. When a repackaged app is executed, the triggered (i.e., decrypted) SDC will make the app run abnormally in a certain period of running time. Note that the smartphone is not rooted. We also design an approach that does not need to modify Dalvik in Scheme II (Section 3.3). SDC Auditor is an app developed by us to check whether a crash is caused by repackaging. If so, users can report to markets such as Google Play. SDC Auditor can be downloaded from Android markets. The details of the three components are demonstrated below.

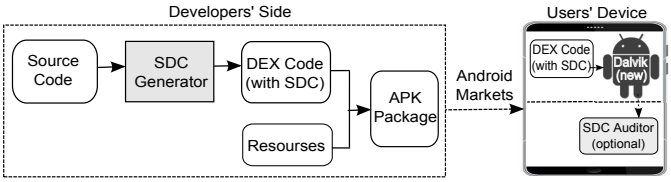

Fig. 3. The overall deployment model of Scheme I.

2.5.1 SDC Generator

The SDC generator embeds SDC segments into an app program. It is fed with an app source code as input and takes four steps to output a DEX file with SDC (Figure 4). The first two steps aim to add a good number of SDC segments (in source code) so that attackers cannot remove all of them with a low cost. The other two steps embed SDC segments into Android apps.

Step 1: Finding Candidate `if` Statements. We refer to the `if` statement in Figure 1 as the *standard* form `if` statement, which meets two requirements: 1) The boolean-expression should be in the form of "`v == const`". 2) The `const` should be a non-zero numerical/string value. However, in the real world, `if` statements are very diverse (not in the "standard" form). To handle the diversity, we identify four types of `if` statements and transform them to the standard form. We refer to the standard form `if` statements and the four types of `if` statements as *already-in `if` statements*. The corresponding SDC is referred to as *already-in SDC* (A-SDC for short).

- Multi-variables. This form looks like `if(exp(`v_1`,...,`v_n`)==const)` and the expression `exp` consists of more than one variable. We use a new variable `v` to replace the expression: "`v=exp(`v_1`,...,`v_n`); if(v==const)`".
- Not-equal. `v`$\neq$`const` is used as the boolean-expression. To handle it, we change "$\neq$" to "==" and switch the two branches. If there is no `else` part in the `if` statement, we add one and randomly generate two effect-canceling statements (e.g., add a random number to a variable and subtract the number) in it.
- Multi-conditions. Logic conjunctions (e.g., logical "or" and logical "and") combine several boolean-expressions to a non-standard form. This form can be split into several `if` statements as shown in Figure 5.

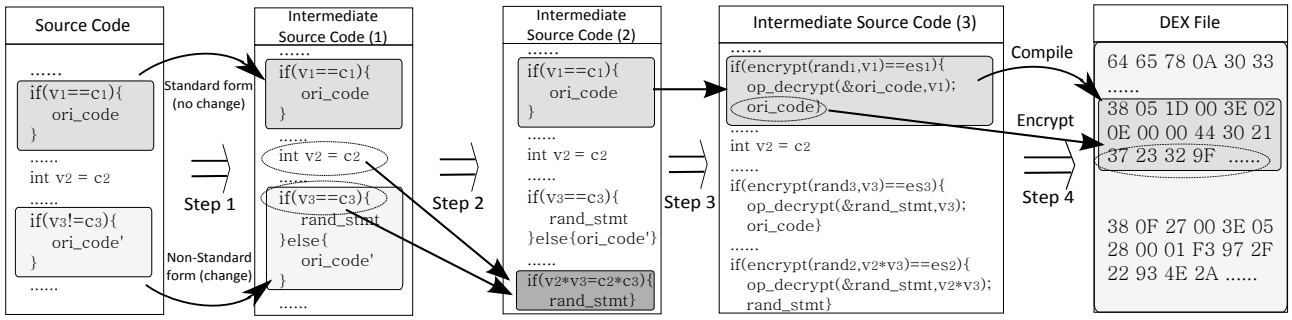

Fig. 4. SDC Generator takes four steps to embed SDC segments into an app program.

- `Switch` statement. `Switch` statement is not even an `if` statement. To handle it, we transform each `case` statement into an `if` statement.

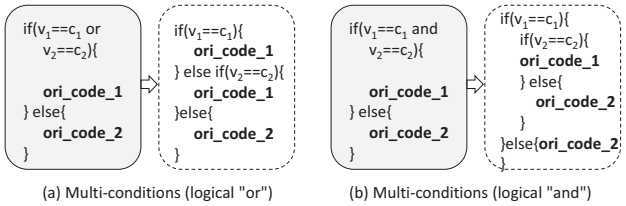

Fig. 5. Transformation for multi-conditions `if` statements.

In this way, we can find a good number of A-SDCs (around 100 for each app on average as shown in Table 2). Table 1 gives detailed examples of the conditions. We find the conditions are diverse and difficult for attackers to guess. In addition, developers can actually extend the length of strings/numbers in some of the conditions (e.g., command, score, file tag) to make it even harder for attackers to guess. Also note that, we statically analyze the app code line-by-line in sequence and seek for suitable if statements for inserting SDC segments. This process does not need to perform traditionally data-flow or control-flow analysis. In this way, we can guarantee the path coverage.

Category	Examples
Handle a received command	if(c.equals("TakePhoto"))
Give bonus if a score is achieved	if(score==1,000,000)
Parse a file	if(tag.equals("LastStatus"))
Schemes	if(scheme.equals("poplaper://"))
File name	if(fname.equals("userconfig.cfg"))
Domain name	if(domain.equals("www.mydomain.com"))
Position on screen	if(x==355 and y==1258)

TABLE 1
Examples of constants in `if` statements

Step 2: Generating Manipulated `if` Statements. If the already-in `if` statements do not provide sufficient number of SDC segments (due to "one-time use"), we find that we can add as many manipulated `if` statements as needed by the resiliency requirements from users. They are in the form of "`if(v==c) {statements}`". `v` is a variable and `c` is a constant. `statements` are randomly generated effect-canceling statements which do not change the semantics of the original apps. This form (after encryption) looks like the already-in `if` statements so that attackers cannot easily identify and remove them. The corresponding SDC is referred to as *manipulated SDC* (M-SDC for short).

In designing a M-SDC (choosing suitable `v` and `c`), on one hand, we should let it be triggered when a particular condition is met. On the other hand, the condition cannot be easily met (low-hanging-fruit). Our insight is to use the variables and constants already in an app to form M-SDC. Based on the insight, we choose v and c by leveraging assignment statement and `if` statements which use non-equal operators ($<, \leq, >, \geq$).

Specifically, we first scan the app code line-by-line and look for constant-to-variable assignment statements (e.g., $v = c$). One idea is to simply add a new statement `if(`$v == c$`)` after the assignment. However, the `if` statement will always be triggered, making it not suitable for adding SDC segment. To handle this problem, we relocate the `if` statement to other methods in which the variable v is used. Our insight is that the value assigned to v may also appear in other places where v is used, which makes it possible to trigger the newly added `if` statement. Considering some experienced attackers could enumerate the constants in the app as the keys to decrypt the manipulated SDC segments, we further combine variables and constants in different constant-to-variable assignment statements. For example, two statements $v_1 = c_1$ and $v_2 = c_2$ can generate an `if` statement `if(`$v_1 * v_2 == c_1 * c_2$`)` which is then inserted into other methods that use both v_1 and v_2. The multiplication operation can also be replaced by others. We also reuse already-in `if` statements and move the statement to other methods that utilize the same variable, since the encrypted values are hard for attackers to enumerate.

Secondly, we look for the `if` statements which use non-equal operators ($<, \leq, >, \geq$) and create new `if` statements according to them. For example, when we find "`if(`$money > 1,000,000$`)`", we randomly choose a constant $c > 1,000,000$ (e.g., $12,327,985$) and add a new `if` statement `if(`$money == c$`)` to two places in the intermediate source code. One place is just before the original `if` statement. The other place is in other methods where $money$ is used. In this way, this condition can be triggered by some specific users, but difficult for a single attacker to trigger in a short-time run.

Note that M-SDC segments do not contain any functionality of an app, which means that attackers can safely remove them after knowing their locations inside the app. However, what benefits us is that M-SDC and A-SDC look very similar (encrypted code after an `if` statement). Comparing the length of M-SDC and A-SDC does not work either since the code inside M-SDC can be arbitrary by padding with junk code. As a result, before triggering them, attackers cannot distinguish them, which further stops attackers from removing or bypassing them.

In our evaluation, we found that we could generate a good number of M-SDCs. Similar to A-SDCs, M-SDCs are triggered when some specific conditions are met. From the evaluation results (Section 5.2), on average 2 M-SDCs are triggered in the first 24 hours by robots or in the first two hours by human.

Step 3: Embedding SDC Segments. SDC generator transforms each candidate `if` statement to a SDC segment. Ac-

cording to our design in Figure 4, SDC generator first transforms the condition v_1=c_1 in the branch to an entry door `encrypt(rand1,v1)==es1`. Then SDC generator adds a statement `op_decrypt(&ori_code,v1)` before `ori_code`. `op_decrypt` is designed to execute the SDC segment. The first parameter is the offset of `ori_code` in DEX file, which is used for the modified Dalvik to locate the ciphertext code. The other one is the dynamic value of v_1.

Step 4: Compilation and Encryption. At last, the SDC generator compiles the intermediate source code to the DEX bytecode. Then, for each SDC segment, the SDC generator encrypts the bytecode of `ori_code` (in A-SDCs) or `rand_stmt` (in M-SDCs). As shown in Figure 1, the key includes two parts: the constant in the `if` statement and the checksum. To protect the whole app, the checksum is calculated over all the bytecode (except for the ciphertext code) and the library code (including both the bytecode libraries and the native code libraries). Resources can also be protected if needed. After encryption, SDC Generator modifies the DEX file: it uses the ciphertext code to replace `ori_code` or `rand_stmt`. At this time, the DEX bytecode is fully generated and is ready for packaging.

2.5.2 Modified Dalvik

We modify the original Dalvik to support the execution of SDC, which is designed for manufactories who want to support identifying repackaged apps. If a SDC segment gets executed (i.e., the boolean-expression is satisfied), it will compute the checksum of the app code (excluding ciphertext code). We use the checksum as part of the key to decrypt the ciphertext code contained in SDC segments. Then the modified Dalvik executes the decrypted code. When an app is repackaged, the incorrect checksum will make the decrypted code no longer the original one. In fact, the decryption will result in wrongly formatted code (not really executable). Executing the wrongly formatted code will automatically crash the app.

New Opcode. According to the design of SDC, the modified Dalvik needs two parameters for decrypting: the offset of the ciphertext code (e.g., `encr_code`) in DEX file and the value of the variable in the entry door (e.g., v_1 in Figure 4). We design a new opcode called "`OP_DECRYPT`" to pass these two parameters to the modified Dalvik. For example, in the "Intermediate Source Code (3)" box in Figure 4, the statement "`op_decrypt(&ori_code,` v_1`)`" is compiled to the newly designed opcode "`OP_DECRYPT`". `&ori_code` is the offset of the ciphertext code. v_1 is the variable in the entry door.

To support the new opcode `OP_DECRYPT`, we modify Dalvik to perform the real work of self-defending including the checksum computation and decryption. When the modified Dalvik executes `OP_DECRYPT`, it first gets the two parameters of the opcode (offset of the ciphertext code and the value of v_1 in the entry door). Then it computes the memory address of ciphertext code by adding the offset and the memory address of DEX file. Secondly, it computes the checksum c' of the app (except for ciphertext code) in memory. Thirdly, it decrypts the ciphertext code by using $v_1 \oplus c'$ as the key. If the app is repackaged, c' does not equal c, which will fail the decryption. Fourthly, it replaces the ciphertext code in memory with the decrypted code. In this way, when the SDC is executed again, Dalvik does not need to do the decryption for the second time. At last, it sets up the program counter and runs the decrypted code.

For other opcodes, the modified Dalvik executes them as usual. In other words, the modification on Dalvik is only triggered by the new opcode `OP_DECRYPT`. The apps with or without SDC can both run well on the modified Dalvik. Thus, this design does not impact the compatibility.

2.5.3 SDC Auditor

Some users may want to check whether a crash is caused by repackaging or other reasons (e.g., the crash caused by the original app). To achieve this goal, we let the native code audit the decryption operations. In particular, after a piece of code is decrypted, the native code logs the timestamp and the decrypted code to a file. Then we supply an app called SDC Auditor. Users could download this app from Android markets and install it in their own devices. SDC Auditor can look into the audit file and check whether the decrypted code is wrongly formatted. If so, SDC Auditor will report to users that the crash is caused by repackaging. After that, users can submit the report to markets such as Google Play through a secure way. When other users know the app is repackaged, they will not use it any more. Also Google could clean up the app and even the account holding the app. Note that each account costs an attacker $25 [14]. Continuously losing such account may impose big cost for attackers.

2.6 Attack Model and Security Analysis

Type 1 (Bypassing SDC): Can attackers bypass the checksum-used-as-a-key code decryption (e.g., flipping the conditions in the branch)? No. If the entry door is bypassed, the decryption will fail and SDC will not correctly run. Neither can attackers bypass the whole SDC since `ori_code` needs to be executed. Otherwise, the legitimate functionalities built inside the if statement can no longer be used, which will further let the app run abnormally.

Type 2 (Decrypting SDC): Dramatically different from previous work which stores the keys inside a program [15], SDC is key management free. One part of the key is the runtime value of the variable in the entry door (`v` in Figure 1). The other part is the checksum. Both these two parts do not ask to store any information inside the app. Thus, there is no key escrow and zero key management overhead. Attackers cannot find the keys in the app, either.

Type 3 (Inserting code before any SDC gets triggered): Specifically, attackers could add malicious code at the start point of an app. This kind of attack suffers from three problems as follows. (1) The injected code could be caught by market managers. During the app review process which is widely deployed in the real world Android markets [5], the market manager can automatically detect malware. Especially, app markets such as Google Play perform the security check by running the app for 15 minutes in a sandbox and observing suspicious behaviors, which is very likely to expose the malicious code at the start point of an app. (2) The main functionalities of the repackaged app could no longer be leveraged by the attacker. Even if the inserted code passes the review process and gets executed, it must complete the attack goal before the execution of the app hits any SDC segment. To avoid hitting any SDC segment, the attack is afraid of letting the repackaged app provide any main functionality, because SDC segments could be everywhere. Since attack goals are usually achieved through the main functionalities of the repackaged app, the attacker must give up many if not most attack goals. (3) The revenues gained in this attack could be very low. Since the repackaged app can be detected whenever any SDC segment is executed, the time left for the attack

is very limited. Although the attacker might gain some revenues from the injected code (e.g., steal users location), the attack may neither continue for long nor impact an enough number of users, since the repackaged app will be "quickly" found and removed from app markets.

Type 4 (Replacing SDC): Attackers could dynamically run the app. When a SDC segment is triggered and decrypted by itself, attackers dump the decrypted code. Then they could repackage the app by replacing the ciphertext code with the decrypted code. In this way, they do not need to worry about the checksum. Typically, there are three kinds of attacks. (1) Using symbolic execution [16] or concolic execution [17] to compute a specific sequence of inputs (i.e., events) which lets an app reach a particular line of code. However, it is very difficult for these approaches to handle nonlinear constraints [18]. For the encryption in the entry door boolean-expression, it is even more difficult to get such a sequence. Attackers may try to guess the constants through symbolic execution. However, some constants (e.g., command in Table 1) controlled by developers are almost impossible to guess. (2) Using robots (e.g., the Monkey [19], a popular stress-testing tool). However, based on our evaluation (Section 5.2), robots can only trigger very limited number of low-hanging-fruit SDC segments. (3) Manually triggering. Still, based on the evaluation results (Section 5.2), limited number of SDC segments could be triggered. Even if an attacker removes all the SDC segments triggered by "him", there still exists low-hanging-fruit SDC segments for "other users", which could be triggered in several minutes (Section 5.5).

3 Twin-SDC and Scheme II

3.1 Motivation

For the smartphones only using original Dalvik without providing any support for decrypting ciphertext code, we design Scheme II. Since the Dalvik bytecode cannot directly use any memory address for computing the checksum of an app, Scheme II leverages native code. One straightforward idea is to use Android native code (i.e., ARM instructions) through Java Native Interface (JNI). In this way, when a SDC segment is triggered, the native code can get the memory address of ciphertext code and decrypt it. In short, all the code executed in Dalvik is packaged in a native library. However, the library is not fully protected by Android system, and we discovered a novel attack to circumvent this straightforward idea. Scheme II is indeed motivated by this new attack.

Circumventing attack. An attacker could pre-compute the checksum of an original app before it is repackaged. Then in the repackaged app, he lets `checksum()` in Figure 4 always return the pre-computed checksum. If `checksum()` in the native library is not fully protected by Android system, attackers could modify it and let SDC decrypt `encr_code` correctly. Note that this attack cannot happen in Scheme I because the attacker cannot modify the code inside Dalvik. He might add his own native code for decryption. But manufactories can let Dalvik check the existence of any native code that modifies app bytecode. If such native code exists, the modified Dalvik can directly alert users about the repackaged app. To face this attack, we do provide an approach which does not need to rely on Dalvik or Android system. Inspired by the idea of SDC, our idea is to put both the checksum computation and decryption operation in the ciphertext code.

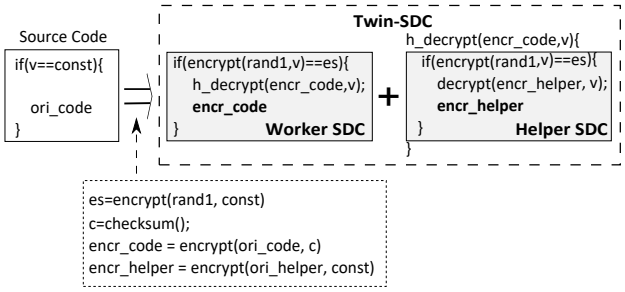

Fig. 6. The design of twin-SDC. The twin-SDC mechanism includes a worker SDC and a helper SDC.

3.2 Twin-SDC

Motivated by this attack, we propose a twin-SDC approach (Figure 6) including a *worker SDC* and a *helper SDC*. The code for decrypting `encr_code` is encrypted inside helper SDC, which is decrypted only when worker SDC is triggered. In this way, attackers cannot touch the decryption code. Also, we let the checksum computation in one SDC segment differ from others. That is, the computation in each SDC will get a different value of checksum. So attackers cannot pre-compute the checksum.

Worker SDC. The worker SDC is inside the Dalvik bytecode part in Android apps. The entry door in the worker SDC is the same as that in Scheme I. The difference is that the worker SDC does not use `const` as the key to encrypt `ori_code`. We give a detailed design of the decryption operation as follows.

The decryption operation calls function `h_decrypt()` to decrypt `encr_code`. `encr_code = encrypt(ori_code, c)`, which is computed by the translator. `c` is the checksum of the app (excluding ciphertext code and some randomly selected bytecode). For different worker SDC segments, `c` is different. After transformation, `c` is removed from the transformed code. In this way, attackers can neither pre-compute `c` nor get it from other decrypted worker SDC segments to perform circumventing attack.

```
h_decrypt(encr_code, v){
    if(encrypt(rand1,v)==es){
        c'=checksum();
        decrypt(encr_code, c')
    }
}
```
ori_helper

Fig. 7. The function `h_decrypt` before the helper SDC is applied. After the helper SDC is applied, `h_decrypt` is transformed to the one in Figure 6. `ori_helper` is the code in the box.

Helper SDC. The helper SDC is inside the native code part in Android apps, called from the worker SDC. It aims to protect the code for decrypting `encr_code`, and further to prevent circumventing attack. For simplicity, the code for decrypting `encr_code` is encrypted using the same key in the worker SDC. Therefore, only when the worker SDC is triggered, the helper SDC can be triggered. Figure 7 shows the code of the help SDC before encryption. It is included in function `h_decrypt()`. `ori_helper` is its main part which regenerates the key (i.e., the checksum `c'` of the app in memory excluding both ciphertext code and the same bytecode chosen in the compile time) and decrypts `encr_code` in the worker SDC. After the decryption, `encr_code` will be executed.

Addressing the circumventing attack. In twin-SDC design, the checksum computation and decryption are all in `encr_helper`. Also, the checksum computed in one SDC segment is different from those in other SDC segments. Attackers cannot pre-compute the checksum and replace the computed checksum in the ciphertext code `encr_helper`. This raises the bar for attackers to repackage apps.

3.3 Scheme II

Scheme II is based on Scheme I. The overall deployment model of Scheme II (Figure 8) is similar to that of Scheme I (Figure 4). The difference is that we use a Runtime Supporter (in the form of Android native code) instead of modifying Dalvik to decrypt ciphertext code. Below gives the detailed design.

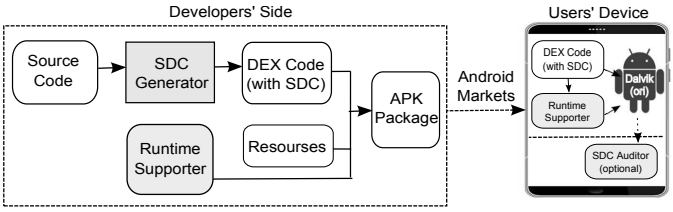

Fig. 8. The overall deployment model of Scheme II. No modification on Dalvik or Android system kernel.

SDC Generator. For the first two steps, SDC Generator in Scheme II works in the same way as that in Scheme I. In the third step, the only difference is that SDC Generator calls function `h_decrypt()` instead of `op_decrypt()`. In the fourth step, SDC Generator embeds SDC segments for both DEX bytecode (in Worker SDC) and native code (in Helper SDC). For the bytecode side, SDC Generator encrypts the `ori_code` using `c` as the key. `c` is the checksum of both the bytecode (except for `encr_code` and some randomly selected bytecode) and the library code (except for `encr_helper`). For the native code side, SDC Generator compiles the `h_decrypt()` function and encrypts `ori_helper` using the constant in `if` statement as the key. For the `checksum()` function inside `ori_helper`, SDC Generator configures it not to compute the randomly selected bytecode.

Runtime Supporter. To use native code, we should first let the native code find the memory address of ciphertext code in DEX. Different from Dalvik, native code cannot directly get the memory address of DEX code so that it is difficult to locate the ciphertext code in memory. To solve this problem, our idea is to give every SDC segment a unique mark (ID number) and locate the SDC segment by searching the mark in memory.

In detail, when a SDC segment is triggered, it first calls the native code (`h_decrypt()`) through JNI and passes the mark of ciphertext code (`encr_helper`) and `v` to the native code. Then, the native code checks whether `v` could open the entry door of the helper SDC. If the entry door is opened, `h_decrypt()` searches for the mark to locate the ciphertext code `encr_helper` and decrypts it using `v` as the key. After decryption, the decrypted code (i.e., `ori_helper` replaces `encr_helper`) and gets executed. Thirdly, it computes checksum c' using the same way as that in SDC Generator. Then it uses c' as part of the key to decrypt the ciphertext code. Fourthly, it overwrites the ciphertext code in memory with the decrypted code. Since we use a block cipher algorithm (AES), the decrypted code has exactly the same length as the original bytecode (no buffer overflow will occur). After the JNI call returns, the decrypted code will be executed. We let the decryption operation overwrite the ciphertext code in memory with the decrypted code. In this way, the decryption operation for the SDC is only performed once no matter how many times the entry door is satisfied.

4 Implementation

We implemented the smartphone anti-repackaging system based on Scheme II. The system includes about 4,000 lines of Python code, 1,000 lines of Java code and 1,000 lines of C++ code.

SDC Generator. For the first three steps (Figure 4) of SDC Generator, Python code is used to generate the intermediate source code. In Step 1 and 2, SDC Generator transforms non-standard form `if` statements to the standard form. Then it generates manipulated `if` statements. Note that it does not place an `if` statement in a loop in the same function (for decreasing overhead). In Step 3, the encryption algorithm is AES [20]. In Step 4, we modify the DEX compiler (i.e., the standard compiler "dx" from Android SDK) to generate SDC segments. It compiles the intermediate source code to Dalvik bytecode. Then it views `ori_code` as zero and computes the checksum according to different schemes. At last, it uses the checksum as part of the key to generate ciphertext code. Since some SDC segments can be easily triggered by attackers, in current implementation, we remove those triggered in the first 15 minutes in manual testing. This time duration is very common for a developer to test apps.

Cache mechanism: For each SDC segment, we use a cache (the size is one) to store the value of v and the result of whether the entry door is satisfied. If the value of (new-coming) v is the same as the cached value, then the encryption is not needed. Otherwise, we perform the encryption and update the cache. If the entry door is satisfied, we lock the cache to make it unable to be updated. In later execution, as long as the value of v is different from the value in the cache, the entry door cannot be satisfied. In this way, we decrease the number of encryption operations in the entry door, which also decreases the overhead.

Modified Dalvik (Scheme I). In Scheme I, we modify the Dalvik virtual machine to support SDC decryption and execution. We first extend "config-armv7-a" file in the "vm/mterp" directory to add a declaration of the opcode `OP_DECRYPT`. In the "vm/mterp/c" directory, we add a new file "OP_DECRYPT.cpp". This .cpp does the real checksum computation and decryption for the new opcode. At last, we need to bind the opcode "0x3E" with the declaration of `OP_DECRYPT` in the source files (e.g., "InstrUtils.cpp", "DexVerify.cpp" and "Liveness.cpp").

Runtime Supporter (Scheme II). The Runtime Supporter is implemented using C++ code as a native library. The `h_decrypt()` function is called through JNI. One parameter is a unique mark which is used for locating the ciphertext code.

Deployment. For Scheme I, Dalvik code needs to be modified. This scheme is suitable for vendors who can customize the Android system (e.g., Google, Samsung, HTC, LG and Sony). As we know, some vendors have customized the Android system to better protect users. They also hope the security customization could attract more users who care about their security and privacy. Note that this deployment option does not need approval from Google. To deploy Scheme I in Android system, the vendors only need to add the declaration and implementation of the opcode OP_DECRYPT in "config-armv7-a" file and "OP_DECRYPT.cpp" (see the implementation above), or simply use our complied version of Dalvik. For Scheme II, there is no

Category	# of apps	# of B-C*	# of B-S
Games	8757	95	37
System Tools	3108	85	41
Weather	338	45	27
Web Browsers	68	76	44
Music	683	75	29
Security Tools	120	127	56
Entertainment	337	92	44
Services	3237	73	49
Photography	347	79	41
Education	957	42	47
Shopping	368	122	65
Social Networking	284	128	77
Finance	291	86	44
Communication	302	147	82
News	803	112	132

TABLE 2
The evaluation on feasibility. The dataset includes 20,000 apps in 15 categories. (*# of B-C: number of branches per app (numerical constant); # of B-S: number of branches per app (string constant))

modification of Dalvik or the Android system. Only a runtime supporter needs to be used inside Android apps. We support a fully automated tool for developers who only need to run our tool on their Android apps at compile time. The whole process, including insertion of SDC segments and the runtime library, is less than 10 seconds.

5 Evaluation

We evaluated the feasibility, attack resilience, the performance overhead (responsiveness degradation), vulnerable time window and the diversity (on an Android 4.3 for Scheme I and Android 7.1 for Scheme II).

5.1 Feasibility

We start the evaluation from understanding the feasibility of the system. In particular, we seek to study whether there are sufficient candidate branches in Android apps for embedding SDC. We consider a dataset from a popular Android market – Anzhi [21], including 20,000 apps in 15 categories.

To better understand the crypto strength of the keys SDC segments, we classify all the candidate branches into two groups based on the type of constant values: numerical constant and string constant. Table 2 shows the categories, the number of apps in each category, the number of branches (per app) that have numerical constants, and the corresponding number for string constants. From the table, we found most apps have a decent number of constants for embedding SDC. Except for categories "Weather" and "Education", each app on average has over 100 candidate branches. For some categories (e.g., News), we found a relatively large number of candidate branches for embedding SDC. In four categories (e.g., "News"), each app has over 200 candidate branches. Also note that developers can extend the length of some constants (in Table 1), which makes it even difficult for attackers.

5.2 Resilience to Replacement Attack

According to the security analysis in Section 2.6, static analysis does not really impact the protection of SDC, even if techniques like symbolic execution are used (cannot handle nonlinear constraints in SDC's entry doors). In contrast, replacement attack is a main threat to our approach. From attackers' point of view, on one hand, they would not spend more resources (time, money) on triggering and removing all SDC segments in an app than redeveloping the app or repackaging other apps without SDC. They hope that (most of) the users could run the repackaged app without crashing for a sufficiently long period of time to achieve their goal (e.g., ad revenue stealing). On the other hand, once a user finds a repackaged app, she can alert Google Play. Google could clean up the app and even the account holding the app. Since the account costs an attacker $25, he could not afford to continuously lose such accounts. So our evaluation focuses on how long it takes a repackaged app to crash. In general, there are two main attack strategies.

Attack Strategy I: Robot Only. As a cheap and efficient approach, robots are widely used to test Android apps. For example, the Monkey [19] is a popular stress-testing tool that generates pseudo-random streams of user events such as clicks. Attackers could use robots to attack. They may also customize the robots to follow the control flow graphs or leverage symbolic/concolic execution to help the robots execute though a specific path. But this does not really help attackers since these approaches need to solve the non-linear constraints in entry doors. We use the Monkey as the robot. Specifically, we first use it to test the target apps for 24 hours and see the number of triggered SDC segments. Then we mimic attackers to remove the triggered SDC segments (replace them with decrypted code) and use the Monkey to test for another 24 hours. Note that removing the triggered ones will help attackers. We set the seed of the Monkey to a random value to generate different sequences of events (using "-s" option).

Figure 9 shows the results of OpenSudoku [22] in the first 24-hour test. OpenSudoku is a popular open source sudoku game with 90 puzzles in 3 difficulty levels. We add 37 A-SDCs and 200 M-SDCs to it. From Figure 9, 1 A-SDC and 4 M-SDCs are triggered. We remove the "1+4" triggered SDC segments and perform another 24-hour test. Only one SDC segment is triggered (Figure 10), which indicates the resilience of our approach. In day 2, the attack's effectiveness dramatically drops. Attackers might remove the triggered SDC segments. However, untriggered SDC segments still exist and will expose the repackaged apps. According to the diversity test (Section 5.5), even if the attacker remove all the low-hanging-fruit SDC segments "for him", there still exists low-hanging-fruit SDC segments "for other users". This means the repackaged app after removing all the triggered SDC segments (in 48 hours) could also be exposed very soon (in several minutes) when used by other users. Why did this happen? We found four reasons from the code.

- Some SDC segments need human intelligence to trigger. For example, a SDC segment uses `if(state==SudokuGame.GAME_STATE_COMPLETED)`, which is triggered only after a sudoku puzzle is successfully solved (very difficult for robots).
- Some SDC segments depends on users. For example, only that a specific high score is achieved can trigger `if(score==1,000,000)`.
- Some SDC segments need correct inputs to trigger. For example, a SDC segment uses `if(lastTag.equals("game"))` to parse an XML file. When the `game` tag does not exist, the SDC segment will not be triggered. A randomly chosen file may not have the tag.
- Some SDC segments depend on hardware and software environment. For example, a SDC segment uses `if(display.getWidth()*display.getHeight()==76800)`. It can only be triggered when a smartphone has a specific resolution.

We also do similar tests on 9 other Android apps. To make

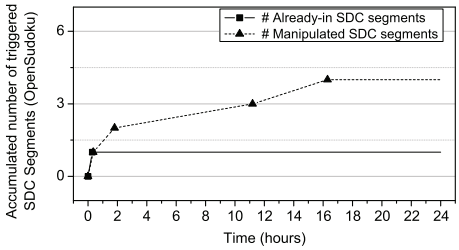

Fig. 9. First 24-hour robot testing on OpenSudoku. More SDC segments are triggered in the first hour.

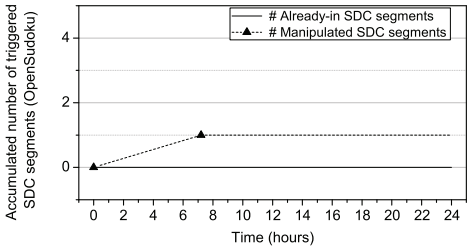

Fig. 10. Second 24-hour robot testing on OpenSudoku (after removing those triggered in the first 24 hours).

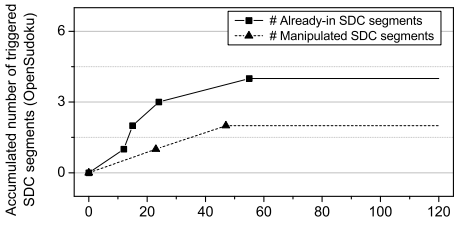

Fig. 11. The number of triggered SDC segments in OpenSudoku by manual testing.

the 10 apps representative, we choose the apps by considering the LOC (lines-of-code). Specifically, we measure the histogram of 30,000 apps in Google Play regarding LOC. Accordingly, we choose the number of apps corresponding to the histogram. Table 3 shows the results. From the table, in the first 24 hours, the number of triggered SDC segments ranges from 1 to 6 depending on the different apps. In the second 24 hours, only three out of ten apps have one SDC segment triggered. No app has more than one SDC segment triggered. Actually, some SDC segments can never be triggered by robots.

Attack Strategy II: Human Only. An attacker could manually play the apps as a human tester and try to trigger as many functionalities as possible so that more SDC segments could be decrypted by human intelligence. We mimic what a human tester could do. Figure 11 shows the results for OpenSudoku. Compared with robot testing, manual testing triggers 3 more A-SDCs in the first 60 minutes. None of them are triggered by the robot. They depend on the solving process which needs human intelligence to solve Sudoku puzzles. The results on 9 other Android apps are similar (Table 3).

Will long-time manual testing be useful? More manual testing might trigger more SDC segments. However, some SDC segments, even popular ones, may not be easily triggered. For example, an extra bonus in an app (e.g., AngryBirds) is only triggered when some special conditions are met (e.g., finishing a puzzle in a time limit). These conditions are not easy to meet and are not necessary for the tester to get into the next game level. So, in most cases, the tester will ignore them. Furthermore, developers would probably likely to add bonuses to protect an app (it will not bother users).

Robots+Human. Attackers could manually play an app and send the intermediate results (e.g., the state of a solved Sudoku puzzle) to robots, hoping to trigger more SDC segments. However, it is difficult for robots to understand and use the results. Moreover, in many cases, different results need to be processed differently. It is hard to give a robot a general result-handling capability. For example, an algorithm for Sudoku puzzles may not be suitable for AngryBirds. Compared with designing such an algorithm, the attacker may be more likely to repackage an app that does not contain any SDC segment.

5.3 Performance

We measure the runtime overhead (responsiveness degradation) caused by the SDC segments, the compilation time overhead and the sizes of new APK files. The latter two are negligible. Here, we only show the performance overhead of Scheme II, which is higher than that of Scheme I. To measure the runtime overhead, we need an automatic testing approach which should be repeatable and traverse as many execution paths as possible. So we use Monkey to generate 10,000 events and run each app (with and without SDC segments) to measure the overhead. In most cases, it is lower than 2%. For few cases, it reaches 3.5%. Figure 12 shows the results. The x-axis shows 10 apps with different numbers of SDC segments (50, 100, 150 and 200). The y-axis shows the overhead in percentage. For example, the first bar shows that the overhead for "RandomMusic" with 50 SDC segments is less than 1%. Interestingly, we find that the increase of overhead is not proportional to the number of SDC segments. When the number of SDC segments doubles from 100 to 200, the overhead increases about 80% (better than linear).

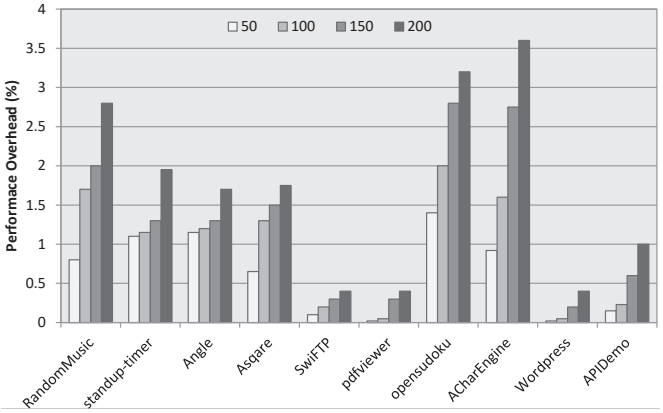

Fig. 12. The performance overhead.

Why the overhead is not high? The overhead mainly comes from two sources: the encryption at the entry door and the decryption process inside the SDC. 1) The number of encryption operations is very few compared with other operations in an execution path. Thus, the time spent on the encryption is minimum. 2) The cache mechanism decreases the number of encryption operations. 3) The decryption inside SDC is only performed once when the entry door is satisfied. 4) SDCs are not placed in a loop in the same function, which benefits the performance.

5.4 Vulnerable Time Window

Many malicious apps [23] need long life time to steal sufficient private information (e.g., users' locations everyday) or get the stable advertising revenue. Before a repackaged app is detected, users are vulnerable. We refer to the time period as the *vulnerable time window*.

As for the attackers, they try to extend the vulnerable window for longer attack time. As for the defenders, they want the vulnerable window to be short for quicker detection of repackaging. So the game theoretic perspective is as follows. On one hand, attackers can hire more human testers to trigger and remove more SDC segments (i.e., perform replacement attack). On the other hand, the compiler can add more (manipulated) SDC segments to ensure a short vulnerable window. From the cost-benefit angle, it seems this game is in favor of defender because adding SDC segments is with low cost (Figure 12), but removing them will cost orders

Apps	LOC	# SDC in apps		Robot testing (Attack strategy I) # Triggered SDC (1st 24H)		# Triggered SDC (2nd 24H)		Manual testing (Attack strategy II) # Triggered SDC (1st 60 mins)		# Triggered SDC (2nd 60 mins)	
		A-SDC	M-SDC	A-SDC	M-SDC	A-SDC	M-SDC	A-SDC	M-SDC	A-SDC	M-SDC
RandomMusic	1692	4	200	1	3	0	0	3	0	0	1
standup-timer	2525	3	200	1	5	0	1	2	2	0	1
Angle	5550	4	200	2	2	0	0	2	0	0	1
Asqare	6153	14	200	3	0	0	0	3	0	0	1
SwiFTP	6685	20	200	0	1	0	0	2	0	1	0
Pdfview	6722	21	200	2	1	0	0	3	1	2	0
OpenSudoku	10391	37	200	1	4	0	1	4	2	0	0
AChartEngine	13655	31	200	1	3	1	0	3	1	1	0
Wordpress	28320	257	200	3	3	0	0	6	5	1	0
ApiDemos	44493	72	1000	3	2	0	0	2	0	1	1

A-SDC: Already-in SDC segments; M-SDC: Manipulated SDC segments.

TABLE 3
The results of robot testing and manual testing on 10 Android apps.

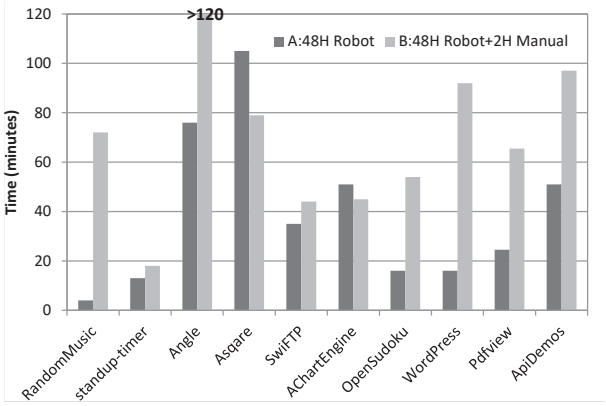

Fig. 13. The vulnerable time window. Bar A shows the time windows after 48-hour robot testing. Bar B shows the time windows after both robot testing and 120-minute manual testing.

of magnitude more. Also, considering that the repackaged app could be used by millions of users while the attacker can only hire limited number of testers (due to limited resources), as long as a SDC segment is triggered on the smartphone of one user, she or he could report to Google Play, and Google may further blacklist the app and even close the account ($25). This also shows that the game is in favor of defenders.

Based on Attack strategy I, we do robot testing for 48 hours. Then we play the apps and measure when the next SDC segment is triggered. Figure 13(Bar A) shows that all the apps will crash within 2 hours. Based on Attack strategy II, we keep tracking the timeline after robot testing (48 hours) and manual testing (120 minutes) in Figure 13(Bar B). The time windows for most of the apps are extended (compared with Bar A). The main reason is that we remove the SDC segments triggered in manual testing (difficult for robots to trigger).

5.5 Relation between SDC Triggering and Diversity

By diversity, we mean that different runs of the same app will result in different sets of triggered SDC segments. If diversity exists, our approach will be more resilient. We focus on the diversity of the initial states in apps and the diversity of different users. We use OpenSudoku as an example to illustrate. For other apps, the results are similar.

Table 4 shows the results. The first row shows 10 different users (with different genders and ages). To avoid the influence by other users, we reinstalled OpenSudoku before a new user played. They were asked to play two specified puzzles and three random puzzles (about 120 minutes). To clearly show what SDC segments

SDC \ User	A	B	C	D	E	F	G	H	I	J
1	●●●●●	●●●●●	●●●●●	●●●●●	●●●●●	●●●●●	●●●●●	●●●●●	●●●●●	●●●●●
2	● ● ●	●●●●	●● ●	●● ●●	● ●●●	●●	●●●	● ●●●	●● ●●	● ● ●
3	●●●	●	● ●	●● ●	●●	●●●●	●●	●	●●●●	●●
4	●●	●●						● ●	●● ●	
5						●				
6						●				
7								●●		

TABLE 4
The results of diversity test.

are triggered, we give each SDC segment an ID number (first column). In each cell, we show the five plays from left to right. A dot means a specific SDC segment is triggered in a corresponding play. For example, User A triggered $No.2$ SDC segment in his first, third and fifth play.

Findings. 1) Even if the same user plays the same app with different initial states (i.e., puzzles), the sets of triggered SDC segments are various, which means attackers may need to play all the puzzles. 2) The sets are also various between different users. 3) Some SDC segments were only triggered by one user (e.g., $No.7$, which checks whether a specified number appears in a specified position). That is, if an attacker performs replacement attack, it is still very possible that another user will trigger a SDC segment that is never triggered by the attacker. 4) Note that $No.7$ was triggered in the first play of User H, which means that the vulnerable time window could be not necessary longer than the time spent on manual testing. 5) Even for the same puzzle, the sets of triggered SDC segments are various. That is, if attackers perform replacement attack, simply playing each puzzle once is not enough.

5.6 Comparison with Previous Researches

We have compared self-defending code with related previous researches including DroidMoss [9], DNADroid [10] and etc. Details are shown in Table 5. From the table, we find that most of previous researches focus on detection of repackaged apps on markets. That is, to check whether a given Android app is a repackaged one, most of the previous approaches have to find a counterpart app (e.g., the original legitimate app or another repackaged one) that is "similar" to the given one. The similarity could be defined using the amount of shared common code [9], [10], [11], [24], [25], [26] or the number of shared common user interfaces [27], [28]. Once the similarity exceeds a certain threshold, at least one of the two apps is viewed as a repackaged app (sometimes, both of the two apps are repackaged). Based on most of the previous approaches, the more apps are included in the comparison process, the more likely that a repackaged app

Previous Work	Deployment	Counterpart[1]	Modify App	Modify System	Stealthy[2]
DroidMoss [9]	Market	Need	No	No	N/A
DNADroid [10]	Market	Need	No	No	N/A
AdRob [24]	Market	Need	No	No	N/A
Juxtapp [11]	Market	Need	No	No	N/A
PiggyApp [25]	Market	Need	No	No	N/A
ViewDroid [27]	Market	Need	No	No	N/A
Centroid [26]	Market	Need	No	No	N/A
DroidMarking [29]	Phone	No Need	Yes	Yes	No Need
SSN [30]	Phone	No Need	Yes	No	Need
SDC (Scheme I)	Phone	No Need	Yes	Yes	No Need
SDC (Scheme II)	Phone	No Need	Yes	No	No Need

[1] : Whether or not an approach needs to find the counterpart app for detecting its repackaged version. [2] : Whether or not the injected code needs to be stealthy (i.e., conceal itself inside Android apps).

TABLE 5
Comparison with previous researches.

could be caught. As a result, researchers start to use a large number of Android apps (e.g., over one million [28]) and focus on how to achieve an accurate and scalable comparison (e.g., feature hash [11] and centroid [26]) across such a large number of apps.

On the other hand, researches are also developing approaches that do not really need the counterpart app for proving that a given app is a repackaged one. For example, an approach could let a repackaged app expose itself, which is similar to our self-defending code. Related self-exposing studies include DroidMarking [29] and SSN [30]. Both of them inject certain "used only for detection purpose" code into an Android app, which runs on the user's phone together with the original code inside the app. When the injected code finds that the app is not original, an alarm will be triggered. For example, SSN [30] will let the app crash.

Though helpful, the related self-exposing approaches are still quite limited. That is, they either require the Android system to be customized or require the injected code to be concealed from the attacker. In particular, the Android system has to be customized to let Droidmarking [29] work; and SSN [30] lets a repackaged app run abnormally (i.e., crash) without any notice to users (including the attackers) and keep the code for crash to be far from the injected code for repackage checking, hoping an (experienced) attacker cannot trace back to find and remove the injected code. These limitations are adequately addressed by our approach: Scheme II in our approach does not need any change on the original Android system; moreover, our approach leverages a unique information asymmetry between developers and attackers which lets the concealment of injected code become unnecessary.

6 Discussions

Brute-forcing the Keys. Attackers could brute-forcing the keys. However, many keys (i.e., constant numbers and strings) can be extended as long as the developers wish (e.g., commands in Table 1), which makes it very hard for attackers to brute-force. Also, we found: *Logical "and" will help us to extend the length of a key*. Suppose an `if` statement is "`if((`v_1`==`c_1`) and (`v_2`==`c_2`))`" (v_1 and v_2 are 64-bit variables). We could design a one-way function $f(v_1, v_2)$ which extends the two 64-bit variables to a 128-bit variable (using concatenation and transformations). Then, we transform the `if` statement to "`if(`$f(v_1, v_2)$`==`$f(c_1, c_2)$`)`". In this way, the length of the key is doubled. More logical "and" will make the key even longer.

In case the source code does not contain sufficient number of conditions using logical "and", we have three ideas. (A) For M-SDCs, we have already had the logical "and". (B) For A-SDCs, if they have long strings as the keys, we do not need to handle them. For the rest small portion of `if` statements which cannot be controlled by developers, our idea is to create an expression "v_2==c_2" and add it to the `if` statement. To make this idea work, we design another one-way function $g()$ to generate c_2 dynamically. We make $g()$ as the pre-dominator function of the `if` statement so that v_2 equals c_2 at the `if` statement. $g()$ could be protected by obfuscation and/or other SDC segments. So it is difficult for attackers to understand $g()$. Even if a patient attacker might understand it in the end, the cost could be much higher than redeveloping it or repackaging other apps without SDC. (C) Using the same idea, more logical "and" can make the key even longer. With (A), (B) and (C), we raise the bar for attackers to brute-force numerical/short string keys.

Optimization Mode. Dalvik can dynamically optimize the DEX bytecode the first time it is used. This optimization is tied closely to the VM version [31]. It may change the bytecode in memory, which further changes the checksum in memory. Note that this does not impact Scheme I since the modified Dalvik can keep a copy of the un-optimized DEX bytecode in memory only for checksum computation. For scheme II, we let Dalvik run in a non-optimization mode for simplicity. This will not impact the experiment results on resilience, vulnerability time window or diversity. The difference on performance measurements should be negligible. To enable the optimization mode, one simple idea is: For each smartphone model (e.g., Galaxy S4), developers can create a specific DEX based on the particular optimization conducted on that phone (this can be done automatically). Also, learning the rules of optimization and combining them into the SDC Generator could also be an alternative way.

Dalvik and ART. As we know, Android apps are commonly written in Java and compiled to (Dalvik) bytecode for Java virtual machine (in .dex file). It should be translated to machine code before running on a real CPU. Before Android 5.0, such translation is handled by Dalvik, the virtual machine in Android operating system. In newer Android systems, Dalvik is replaced by Android Runtime (ART) for better performance [32]. ART uses the same Dalvik bytecode and accepts the same .dex files. The main difference between the two translators is: Dalvik is based on Just-in-Time compilation, while ART is based on Ahead-of-Time compilation. In detail, Dalvik dynamically translates a piece of Dalvik bytecode into ARM machine code each time when an app runs. On the contrary, ART performs the translation only once when the app is installed on the devices, and stores the machine code in the device's storage for future use. Considering the new runtime ART is compatible with Dalvik bytecode, Scheme II which only injects code into Android apps is actually not impacted. For Scheme I that customizes Dalvik by adding a new opcode "OP_DECRYPT" to dynamically decrypt the ciphertext code in Android apps, we could port Scheme I to ART by customizing ART in a similar way. According to the design of ART, it should statically translate the opcode into machine code which performs the decryption dynamically.

Cost of Attack. According to the security analysis in Section 2.6, replacement attack is a main threat to our approach. We divided the attackers into two types. Type 1: The attacker tries to remove most (but not all) SDC segments in an app, hoping that (most of) the users could run the repackaged app without crashing. However, once a user (any user) finds a repackaged app, she or he can alert Google Play to blacklist the app or even disable the account holding the app which costs the attacker $25, making him not be able to afford continuously losing such accounts. Moreover,

Google could even block the credit card bounded to the account, which makes it difficult for the attacker to create new accounts in the future[1]. When the number of users is below a threshold, the financial benefits of repackaging are usually very limited. The attacker can only gain around $1 if the repackaged app is used by 100 users and each user spends 10 minutes running the app [33].

Type 2: The attacker tries to trigger and remove all SDC segments in an app, targeting to create a "perfect" repackaged app with no SDC segment inside. Based on the evaluation (Section 5.2), the attacker has to spend lots of resources (time, money) on running the app. Also, according to the evaluation of diversity (Section 5.5), triggering the app only by the attacker himself may not be enough. Instead, he should hire diverse users to test the app (e.g., $50/hour for each tester), which would further increase the cost of the attack. Typically, an experienced and patient tester may cost much more since he needs to get familiar with the target app, hoping to trigger some functionalities (with SDC inside) that cannot be triggered by random clicking. As a result, unless the attacker is sure that he will make more money through repackaging than the cost of removing SDC segments, he may prefer to redevelop the app or repackage other apps without SDC.

7 Related Work

Repackage detection and prevention. Some researches detect repackaged apps by comparing one app with others. DroidMOSS [9] uses a fuzzy hashing to detect repackaged apps. DNADroid [10] and AdRob [24] generate and compare program dependency graphs of methods between two apps. Juxtapp [11] uses feature hashing to compute pairwise similarity between apps. PiggyApp [25] extracts semantic features of apps to detect "piggybacked" apps. Previous work on clone detection [34] could help repackage detection. Our work differs from them by letting repackaged apps expose themselves. We do not compare an app with others. DIVILAR [35] transforms a whole Android app into a virtual instruction set (VIS) and designs an execute engine for running these instructions, which makes it difficult for attackers to transform their code into the VIS. Differently, we encrypt a small portion of the app code with a dynamically generated key.

Malware detection and defense. Enck et al. [36] dynamically track taint data and report if sensitive data are sent off the Android phone. DroidAPIMiner [37] and DREBIN [38] extract malware features to classify and capture malware. AndRadar [39] discovers apps similar to a seed of malicious app in alternative markets. AppsPlayground [40] detects malicious functionality in apps. AirBag [41] creates an isolated environment to defend against malware. Lever et al. [42] detect malicious traffic of malware. DroidBarrier [43] detects unauthorized processes dynamically. Alterdroid [44] detects hidden or obfuscated malware components by analyzing the behavioral differences between the original (malicious) app and a number of automatically generated versions of it where modifications have been injected to thwart the execution of suspicious behaviors. SmartDroid [45] can automatically detect the UI-based trigger conditions, which helps to expose sensitive behaviors of Android malware. KARMA [46] protects Android kernel vulnerabilities by filtering malicious inputs to prevent them from reaching the vulnerable code. Different from them, we defend against repackaged apps by a self-code-decryption mechanism. AVRAND [47] introduces a self-encryption and self-decryption routine for defending against code reuse attacks. The main difference between AVRAND and SDC is: AVRAND stores the key in a fixed position of the flash memory while SDC does not store any key.

Watermark and birthmark. A watermark/birthmark is mainly used for identifying the original owner [48] and/or finding similar apps [49]. They are statically stored in software [50], [51] or dynamically computed [12]. In [29], the basic SDC mechanism is deployed to the problem domain of software watermarking. However, our work and [29] have major differences. First, software watermarking focused on detection, but our work focuses on self-defending, not detection. Second, [29] assumes a weaker threat model; as a consequence, the new Twin-SDC mechanism can deal with an advanced threat model.

Integrity checking. Developers use checksums (e.g., compare a dynamically computed checksum with a pre-computed one) to protect software [52], [53]. However, attackers can modify the pre-computed checksum. Different from them, we do not store any checksum. Aucsmith [15] divides software into small pieces and encrypts them for tamper resistance. We generate SDC very differently. In addition, both Scheme I and II do not require smartphones to be rooted, which can prevent attacks utilizing system privileges [54]. Protsenko et al. [55] propose an approach for tamper-proofing using encryption. The decryption key is distributed together with an app while SDC does not store any key inside the app.

Malware behavior hiding. SDC in our system is an adaptation of the malware behavior hiding mechanism in [13]. We have opposite purposes to let a repackaged app expose themselves. Our twin-SDC defeats an important circumventing attack on Dalvik bytecode using checksum-contained key.

8 Conclusion

We propose a self-defending approach to let repackaged apps automatically expose themselves. Two schemes are proposed, one of which requires no modification on Dalvik or the Android system. To the best of our knowledge, this is the first work that lets repackaged apps malfunction while having none effect on a benign app's function.

Acknowledgments

The authors would like to thank editors and anonymous reviewers for their valuable comments. This work was supported in part by National Key Research and Development Program of China (Grant No. 2016QY04W0805), NSFC U1536106, 61728209, 61303248, Youth Innovation Promotion Association CAS and National High Technology Research and Development Program of China (863 Program) (No. 2015AA016006).

References

[1] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *IEEE S&P*, 2012.
[2] G. Play, "Application licensing," http://developer. android.com/google/play/licensing/index.html, 2013.
[3] android cracking, "Antilvl - android license verification library subversion," http://androidcracking.blogspot. com/p/antilvl_01.html, 2013.
[4] smali, "An assembler/disassembler for android's dex format," http://code.google.com/p/smali/, 2013.
[5] Wikipedia, "Google play," http://en.wikipedia.org/wiki/Google_Play, 2015.

1. The attacker may try to buy credit cards in some illegal underground markets. However, these stolen cards can only be used within very short period of time (e.g., 1 month). It is very difficult for the attacker to attract enough users in such a short period of time.

[6] J. Oberheide and C. Miller, "Dissecting the android bouncer," *SummerCon2012, New York*, 2012.
[7] E. Lafortune, "Proguard," http://proguard.source- forge.net, 2013.
[8] S. Inc, "A specialized optimizer and obfuscator for android," http://www.saikoa.com/dexguard, 2013.
[9] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *CODASPY*, 2012.
[10] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," *ESORICS*, pp. 37–54, 2012.
[11] 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*, 2012.
[12] W. Zhou, X. Zhang, and X. Jiang, "Appink: watermarking android apps for repackaging deterrence," in *ASIA CCS*. ACM, 2013, pp. 1–12.
[13] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation." in *NDSS*, 2008.
[14] Google, "Get started with publishing," http://developer.android.com/distribute/googleplay/start.html, 2015.
[15] D. Aucsmith, "Tamper resistant software: An implementation," in *Information Hiding*. Springer, 1996, pp. 317–333.
[16] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *ICSE*. ACM, 2011, pp. 1066–1071.
[17] C. Y. Cho, D. Babic, P. Poosankam, K. Z. Chen, D. Song, and E. X. Wu, "Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery," in *USENIX Security*, 2011.
[18] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, 2013.
[19] Google, "Ui/application exerciser monkey," http://developer.android.com/tools/help /monkey.html, 2013.
[20] Wikipedia, "Advanced encryption standard," http://en. wikipedia.org/wiki/Advanced_Encryption_Standard, 2014.
[21] Anzhi, "Anzhi market," http://www.anzhi.com/, 2013.
[22] Romario, "Sudoku," http://code.google.com/p/opensudoku-android/, 2013.
[23] R. Templeman, Z. Rahman, D. Crandall, and A. Kapadia, "Placeraider: Virtual theft in physical spaces with smartphones," in *NDSS*, Feb. 2013.
[24] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi, "Adrob: Examining the landscape and impact of android application plagiarism," in *MobiSys*, 2013.
[25] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of piggybacked mobile applications," in *CODASPY*, 2013.
[26] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *ICSE*, 2014.
[27] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "View-droid: Towards obfuscation-resilient mobile application repackaging detection," in *Proceedings of the 7th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec 2014). ACM*, 2014.
[28] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, "Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale." in *USENIX Security Symposium*, 2015, pp. 659–674.
[29] C. Ren, K. Chen, and P. Liu, "Droidmarking: resilient software watermarking for impeding android application repackaging," in *ASE*, 2014.
[30] L. Luo, Y. Fu, D. Wu, S. Zhu, and P. Liu, "Repackage-proofing android apps," in *DSN*, 2016.
[31] Netmite, "Dalvik optimization and verification with dexopt," http://www.netmite.com/android/mydroid/ dalvik/docs/dexopt.html.
[32] Google, "Art and dalvik," https://source.android.com/devices/tech/dalvik, 2017.
[33] Quora, "How much does an app developer earn from an app per ad click?" https://www.quora.com/How-much-does-an-app-developer-earn-from-an-app-per-ad-click, 2016.
[34] C. Liu, C. Chen, J. Han, and P. Yu, "Gplag: detection of software plagiarism by program dependence graph analysis," in *SIGKDD*, 2006.
[35] W. Zhou, Z. Wang, Y. Zhou, and X. Jiang, "Divilar: Diversifying intermediate language for anti-repackaging on android platform," in *CODASPY*, 2014.
[36] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones." in *OSDI*, vol. 10, 2010, pp. 1–6.
[37] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *SecureComm*, 2013.
[38] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket," in *NDSS*, 2014.
[39] 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.
[40] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: Automatic security analysis of smartphone applications," in *CODASPY*, 2013.
[41] C. Wu, Y. Zhou, K. Patel, Z. Liang, and X. Jiang, "Airbag: Boosting smartphone resistance to malware infection," in *NDSS*, 2014.
[42] C. Lever, M. Antonakakis, B. Reaves, P. Traynor, and W. Lee, "The core of the matter: Analyzing malicious traffic in cellular carriers," in *NDSS*, 2013.
[43] H. M. Almohri, D. D. Yao, and D. Kafura, "Droidbarrier: know what is executing on your android," in *CODASPY*, 2014, pp. 257–264.
[44] G. Suarez-Tangil, J. E. Tapiador, F. Lombardi, and R. Di Pietro, "Thwarting obfuscated malware via differential fault analysis," *Computer*, vol. 47, no. 6, pp. 24–31, 2014.
[45] 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.
[46] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei, "Adaptive android kernel live patching," in *USENIX Security*, 2017.
[47] S. Pastrana, J. Tapiador, G. Suarez-Tangil, and P. Peris-López, "Avrand: A software-based defense against code reuse attacks for avr embedded devices," in *DIMVA*, 2016.
[48] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation-tools for software protection," *Software Engineering, IEEE Transactions on*, vol. 28, no. 8, pp. 735–746, 2002.
[49] G. Myles and C. Collberg, "K-gram based software birthmarks," in *ACM symposium on Applied computing*, 2005.
[50] S. A. Moskowitz and M. Cooperman, "Method for stega-cipher protection of computer code," Apr. 28 1998, uS Patent 5,745,569.
[51] R. Venkatesan, V. Vazirani, and S. Sinha, "A graph theoretic approach to software watermarking," in *IH*, 2001.
[52] A. Monsifrot and C. Salmon-Legagneur, "Method and apparatus for verifying the integrity of software code during execution and apparatus for generating such software code," Feb. 15 2011, uS Patent App. 12/931,982.
[53] K. D. Morgan, "Tamper proof mutating software," Apr. 4 2013, uS Patent 20,130,086,643.
[54] G. Wurster, P. C. Van Oorschot, and A. Somayaji, "A generic attack on checksumming-based software tamper resistance," in *IEEE S&P*, 2005.
[55] M. Protsenko, S. Kreuter, and T. Müller, "Dynamic self-protection and tamperproofing for android apps using native code," in *Availability, Reliability and Security (ARES), 10th International Conference on*, 2015.

Kai Chen Kai Chen received the B.S. degree from Nanjing University in 2004, and the Ph.D. degree from University of Chinese Academy of Sciences in 2010. He is a Professor in Institute of Information Engineering, Chinese Academy of Sciences and also in the University of Chinese Academy of Sciences. His research interests include Software Security, Security Testing on Smartphone.

Yingjun Zhang Yingjun Zhang received the B.S. degree from Beihang University in 2004, and the Ph.D. degree from University of Chinese Academy of Sciences in 2010. She is an Associate Professor in the Trusted Computing and Information Assurance Laboratory, Institute of Software, Chinese Academy of Sciences. Her research interests include Access Control, Security Testing.

Peng Liu Peng Liu received the B.S. and M.S. degrees from the University of Science and Technology of China, and the Ph.D. degree from George Mason University in 1999. He is a Full Professor of Information Sciences and Technology, founding director of the Center for Cyber-Security, Information Privacy, and Trust, at Penn State University. His research interests are in computer and network security.