Droidmarking: Resilient Software Watermarking for Impeding Android Application Repackaging

Chuangang Ren *
Department of CSE
The Pennsylvania State University
University Park, PA 16802
cyr5126@cse.psu.edu

Kai Chen *
State Key Laboratory of Information Security, Institute of Information Engineering
Chinese Academy of Sciences
Beijing, China
chenkai010@gmail.com

Peng Liu
College of IST
The Pennsylvania State University
University Park, PA 16802
pliu@ist.psu.edu

ABSTRACT

Software plagiarism in Android markets (app repackaging) is raising serious concerns about the health of the Android ecosystem. Existing app repackaging detection techniques fall short in detection efficiency and in resilience to circumventing attacks; this allows repackaged apps to be widely propagated and causes extensive damages before being detected. To overcome these difficulties and instantly thwart app repackaging threats, we devise a new dynamic software watermarking technique - *Droidmarking* - for Android apps that combines the efforts of all stakeholders and achieves the following three goals: (1) copyright ownership assertion for developers, (2) real-time app repackaging detection on user devices, and (3) resilience to evading attacks. Distinct from existing watermarking techniques, the watermarks in Droidmarking are *non-stealthy*, which means that watermark locations are not intentionally concealed, yet still are impervious to evading attacks. This property effectively enables normal users to recover and verify watermark copyright information without requiring a confidential watermark recognizer. Droidmarking is based on a primitive called *self-decrypting code* (SDC). Our evaluations show that Droidmarking is a feasible and robust technique to effectively impede app repackaging with relatively small performance overhead.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Invasive software,*

Keywords

Software Watermarking; Android; App Repackaging

*Chuangang Ren and Kai Chen are co-first authors and contact authors.

ASE'14, September 15-19, 2014, Vasteras, Sweden.

http://dx.doi.org/10.1145/2642937.2642977.

1. INTRODUCTION

The past years have witnessed the unprecedented popularity of Android devices and the booming of Android application markets. Software plagiarism in Android markets, however, which is also known as *app repackaging*, is posing prevalent and severe threats to the health of the Android ecosystem. For instance, App repackaging has caused intellectual property infringement and enormous advertising revenue losses for original app developers [15]. A significant portion of Android malwares use repackaged apps as a "free" vehicle to carry malicious payloads, which have posed serious risks to user privacy and financial security [31, 32]. While there have been considerable recent efforts from both academia and industry to impede the app repackaging acts [12, 30, 29, 11, 28, 5, 16], the existing methods have limitations that largely undermine their effectiveness in practice.

Most existing app repackaging detection approaches are "offline" solutions, performing centralized repackaging detection regularly or on purpose by authorities like Android markets (e.g., similarity mining from a large basis of app samples [12, 30, 29, 11, 16]). The primary drawback of these "offline" solutions is that they allow for repackaged apps to be widely distributed (depending on how often repackaging detection is performed) and to cause extensive harm to users and original app developers before they are detected. Moreover, most existing repackaging detection approaches can be easily evaded by various obfuscation techniques as illustrated by [17], which further encourages repackaged app developers (*attackers*) to submit more repackaged apps to the markets, hoping to make as much illegal profits as possible before they are detected and removed from the markets.

To this end, we have a pressing need for a new robust anti-repackaging scheme that can combine the efforts of other major stakeholders - users and app developers - and can instantly thwart the distribution of repackaged apps before massive damages are caused. We thus have the following design goals for the new anti-repackaging scheme:

1. Accurately and quickly detect app repackaging threats with help from users and original developers
2. Provide undeniable and solid proof of copyright ownership to protect app developers
3. Offer strong resilience to various evading attacks

Toward all these goals, we are intrigued by traditional watermarking techniques, which are widely employed by printing industries to fight against piracy acts. They provide three noteworthy properties: (1) copyright assertion, (2) real-time copyright verification by users and (3) resilience to evading attacks. These properties are highly desirable for those who want to overcome the difficulties in impeding app repackaging in today's Android market.

Software watermarking is not new. By using a custom tool, namely *watermark embedder*, a software developer can insert a copyright notice into a program. An concerned authorized party can later recover and verify the software's copyright ownership using a tool called *watermark recognizer*, which keeps some form of secrets (e.g., password, special input) to reverse the watermarking embedding process and recover the copyright notice from the program.

To prevent attackers removing the watermarks, existing software watermarking approaches rely heavily on the stealth of watermark (i.e., embed watermarks in a way to make them locally indistinguishable from the carrier code). However, this "stealthy" design principle causes two consequences that prevent existing software watermarking techniques from achieving the properties we desire in traditional watermarking. First, the watermark recognizer has to be kept confidential from the public and be held by a small number of authorized parties (e.g., developer or trusted authority) as it contains the secret to disclose the locations of watermarks in a program. As a result, there is no way for normal users to verify watermarks in real-time like they can on, for example, a book page. Second, it is difficult to ensure absolute "stealth" as we will see in Section 2. Therefore, simply deploying existing software watermarking techniques on Android apps would fail to achieve our goals.

It is exactly these challenges that this paper seeks to address by presenting a new "non-stealthy" software watermarking scheme. Contrary to the design criteria of existing software watermarking approaches, our design is based on the insight that "non-stealthy", i.e. do not conceal the locations of watermarks on purpose, is an appealing property for software watermarking due to two advantages: (1) when the locations of watermarks in a program is open to the public, normal users (and also attackers) are free to recover watermarks without confidential tools needed. Of course, users still need assistance from certain tools to recover watermarks, but these tools contain no secret for watermark recovery, and, thus are no longer exclusive to authorized parties. (2) It circumvents the design challenge of "how to make watermarks stealthy", which is difficult based on the experience of previous work [10, 6]. On the other hand, "non-stealthy" makes watermarks vulnerable to evading attacks. Our solution is to make the resilience of our watermarking rely on "*inter-dependency*" between the embedded watermarks and the carrier program. It means that not only do watermarks depend upon a carrier program to reside, but watermarks also are embedded in a way that the correct and complete functionality of the original program also relies on the integrity of original watermarks. Simply de-coupling the two would result in miss- or non-functional code.

Based on these new criteria, our watermarking scheme - *Droidmarking* - lets a developer embed watermarks into an app and later allows a normal user who installs this app on his/her device to quickly recover and verify the developer's watermark from the app by simply playing the app for a short period of time without any confidential watermark recognizer needed. Based on the recovered watermark information, a Droidmarking facility on a user device can automatically detect potential repackaging threat and immediately report it to user and/or the Android market. Droidmarking is based on a primitive called *Self-decrypting Code* (SDC) segment, which is first introduced by Sharif et al. and referred to as *conditional code obfuscation* in their work [24]. A SDC segment is an encrypted code block (containing both the original functional code and embedded watermark code snippet) in a special type of branch in the program. The encryption key comes from the constant value in the special branch condition. The branch condition is re-written in a semantic equivalent way that this particular constant/key is removed from the static code and can only be dynamically recovered upon branch condition satisfaction at runtime. By this means, SDC helps build the "inter-dependency" between watermark and original carrier code by combining and encrypting the two into a single SDC segment. Furthermore, by creating a large number of watermark-carrying SDC segments, it is difficult for attackers to de-couple all embedded watermarks from the original functional code without significant efforts. We summarize our contributions below:

- We propose a new "non-stealthy" software watermarking approach that allows users to freely recover and verify watermarks without requiring a confidential watermark recognizer yet is still resilient to de-watermarking attacks.
- Our "non-stealthy" watermarking is based on a primitive called self-decrypting code (SDC). We implement our watermarking approach to a prototype - Droidmarking - for Android applications.
- We systematically evaluate the efficacy of Droidmarking on its feasibility, resilience and performance.

2. WATERMARKING BACKGROUND

2.1 Traditional Watermarking

Traditional watermarking embeds a watermark message into the carrier media to declare copyright ownership. Take watermarking in printing industry for instance, a valid watermark (usually an image or a pattern in paper) in a printed book carries unique authorship and publisher information that discloses precise copyright ownership of the book. A concerned reader can readily verify these information immediately when getting a hard copy. An effective watermarking technique is designed to make sure that (1) watermark is readily verifiable by people who possess the object, and (2) it is infeasible to copy or counterfeit a watermark. The first objective allows users be able to easily verify the authenticity and originality of the object by themselves immediately. The second objective protects the integrity of watermarks against forgery attempts. As a result, there is no easy way for an attacker to forge a watermarked object without being noticed by concerned users who always verify the watermark first.

2.2 Software Watermarking

Software watermarking [6] does not prevent software copying but instead discourages software piracy by embedding copyright information into software code and allows one to prove copyright ownership when plagiarism has occurred.

Software watermarking embeds a watermark W into a program P, such that W can later be reliably located and recognized from P, even after P is subject to code transformations. Software watermarking comes with two flavors - static and dynamic. Static watermarks are stored in the code or data of the executable, whereas dynamic watermarks are usually built at runtime and contained in the dynamic state of the program, e.g. data structure, execution trace, etc. Software watermarking must be able to defend against two types of attacks in general - *steal attack* and *change attack*. In a steal attack, attackers wish to disguise plagiarized software as original by copying/forging the original watermark, making users believe that their copy comes from a trusted original software vendor. In a change attack, attackers attempt to disable the original watermark by changing the program. More specifically, change attacks fall into the following three categories: (1) A *subtractive attack* tries to completely or partially remove the watermarks from the program. (2) A *distortive attack* applies a sequence of code transformations, attempting to obstruct the recovery of original watermarks. (3) An *additive attack* add attacker's own watermarks to the software and claim attackers' ownership of the software. In practice, attackers can opt for a combination of the above attacks. On the defense side, one natural step towards defeating subtractive and distortive attacks in existing work [23, 18, 7, 13] is to conceal the watermark in a stealthy way, such that it is difficult to locate/remove/distort all watermarks in the program, or the quality of the distorted program is significantly degraded such that it is no longer of any value to the attacker.

However, the principle of stealth is inherently flawed and in turn causes two consequences. First, the watermark recovery tool - watermark recognizer - that contains some forms of secret information (e.g., password, special inputs) used for watermark recovery is kept confidential from the public. It means that unlike traditional watermarking, existing software watermarking approaches cannot enable normal users to perform watermark verification; instead, it solely relies on a small number of authorities who have access to the confidential watermark recognizer. Although software watermarking techniques have been developed for many years, in practice, no such authorities actually exist. We believe that Android markets do not have strong enough incentives to become such a liable authority. Nevertheless, the originality of Android apps is a serious concern for developers and users. Second, previous experience in watermarking design has shown us that it is non-trivial to ensure absolute stealth[19, 26]. Once a program resides in an attackers' machine, attackers can use any conceivable techniques to (approximately) locate the watermarks. For example, it is well-known that static watermarking is highly susceptible to semantics preserving transformation attacks. Dynamic watermarking, while yielding a certain degree of resilience to semantics preserving transformation attacks, relies on the stealth of runtime states (e.g., bogus graph data structure [23]). Once a subset of these runtime states are detected and scrambled by the attacker, the watermark is ruined and rendered useless.

3. OVERVIEW

To overcome the limitations of existing watermarking methods, Droidmarking is designed to be a "non-stealthy" watermarking technique; in other words, the purpose is not to hide watermarks. In fact, it is fairly easy to locate the watermarks in an Android app program. To defend against watermarking evading attacks, Droidmarking is based on a primitive called Self-Decrypting Code (SDC), which makes it difficult to de-couple the two without significant efforts and costs for attackers.

3.1 Problem Statement and Assumptions

In watermark embedding, our scheme takes as input an app program A and the original developer's watermark instance w, and outputs a watermarked app program A' such that A' is semantically equivalent to A, i.e. $A' = E(A, w)$, where $E(\cdot, \cdot)$ is the watermark embedding function. In the watermark recognition phase, a set of watermark instances (may consists of multiple w s), denoted by Σ , is recovered from program A' and collected by a watermark recognizer, i.e. $\Sigma = D(A', \alpha)$. α represents a series of user inputs (and interactions) to app A'. $D(\cdot, \cdot)$ is watermark extraction function called by A' in response to user inputs α. Let $T(A')$ denotes the app program after code transformations performed by an attacker, and Σ' denote the set of watermark instances recovered from $T(A')$, i.e. $\Sigma' = D(T(A'), \alpha)$. We say the original watermark instance w is successfully recovered even subject to code transformations, iff:

$$\Sigma' \neq \phi \textbf{ and } w \in \Sigma'.$$

ϕ denotes an empty set. In other words, successful watermark recovery ensures that at least one watermark instance w from original developer be recovered and collected by the watermark recognizer. In addition, our goal also requires w to be successfully recovered within an anticipated short period of time t. Otherwise, we call it "failure to recover" original watermark w.

Now we consider a set of watermarks $\hat{\Sigma}$ recovered from an unknown app $\hat{A}$, i.e. $\hat{\Sigma} = D(\hat{A}, \alpha)$, a *watermark conflict* occurs if: $\exists w_1, w_2 \in \hat{\Sigma}$ and $w_1 \neq w_2$. A watermark conflict unquestionably indicates that $\hat{A}$ includes different copyright information and hence is a repackaged app.

We assume that an original app developer has the app source code. Developers are concerned about their intellectual property and financial interests and, thus, are motivated to apply protection schemes against potential piracy acts. Android users, however, are divided into two classes with respect to their care and caution to the originality of the apps: *concerned users* and *causal users*. Concerned users are prudent about their privacy and cyber security and are willing to build trust with newly installed apps first, whereas casual users do not bother to care. Droidmarking draws the joint efforts from original developers, concerned users and Android markets to defend against app repackaging threats.

We assume that attackers can use any conceivable static or dynamic analysis tools to examine the compiled app program (bytecode and native binary), line-by-line, and make any code transformations necessary aiming to remove original watermarks, render watermarks infeasible to be recovered, or even add attackers' own watermarks to the app. A determined attacker may also carefully infer the program semantics and re-write part of the app program, or in the extreme cases, the whole app program. However, the cost of re-writing the app may outweigh the value of the repackaged app itself. We assume cash/interests-driven attackers are interested in repackaging an app only if the cost of repackaging is less than the value it adds. In that case, an attacker may

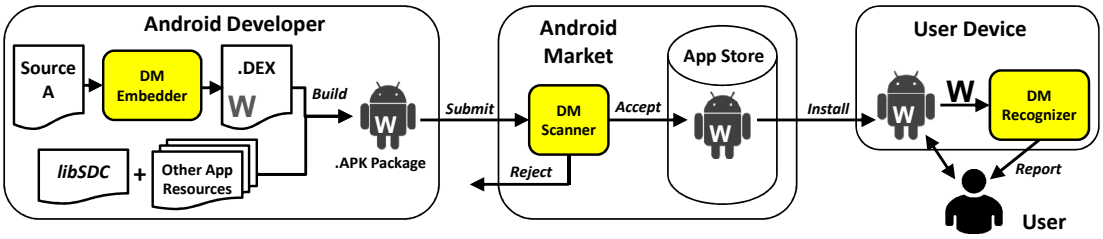


Figure 1: Deployment model of Droidmarking system

(a)
```
if ( uri.startswith("smtp") ) {
   Code
}
```

(b)
```
if ( weight == 75 && height > 180 ) {
   Code
}
```

Figure 2: Examples of candidate branches.

even be willing to sacrifice certain functionality and value of the original app as long as the cost-efficacy still holds. The plagiarized app can be distributed by the attacker either in the same market as the original app, or across different markets.

3.2 Architecture

Figure 1 shows the overall deployment model and architecture of the Droidmarking system. Droidmarking is comprised of three major components (as highlighted in Figure 1) for the three stakeholders, respectively: Droidmark embedder (*DM embedder*) used by developers, Droidmark recognizer (*DM Recognizer*) installed on user devices, and Droidmark scanner (*DM Scanner*) employed by Android market. An app developer feeds app source code A as input to DM embedder. DM embedder embeds watermarks into the code and finally outputs a dex file (executable for Dalvik virtual machine on Android system). The dex file is then packaged with *libSDC* (a native library used in our watermarking scheme) and other app resources to an apk file for publication (application package). Before publishing the app packages to the app store, Android market conducts a quick static scan to the dex file to insure the valid and correct use of the DM embedder. Those who fail DM scanning are subject to misuse or evading attacks (see Section 5) and are rejected by the market. When users install the app on their devices and use the app, the embedded watermarks are automatically recovered. DM recognizer (residing on user devices) is responsible for harvesting the recovered watermark instances from the app and alerting the user and/or Android market immediately when an repackaging threat is detected. In the next section, we will discuss in detail the design and implementation of these three components, respectively.

4. DESIGN AND IMPLEMENTATION

4.1 Self-decrypting Code

Droidmarking is based on a primitive called *Self-decrypting Code* (SDC), which was first proposed by Sharif et al. [24] for malware obfuscation and has nothing to do with watermarking. A *SDC segment* is created from a special type of branch from the app code. Figure 2 shows two examples of these types of branches in Android apps. The two common properties in the examples are that (1) both examples use equality test (such as == operator, `startswith()`, `endswith()` or `equals()` routines) as one of the conditions to enter the branch; (2) one of the equality test operands is a constant numerical/string value. We call the branches that have these branch conditions *candidate branches*.

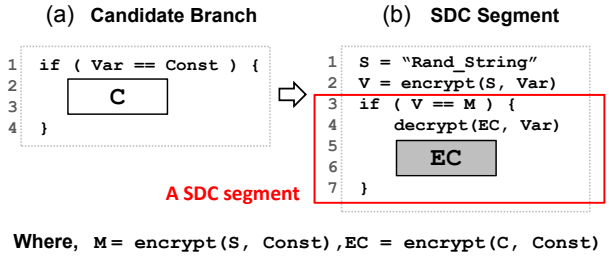


Figure 3: Example of self-decrypting code (SDC) segment.

An SDC segment is created from a candidate branch as illustrated in Figure 3. In this example, the original branch condition `Var==Const` in Figure 3(a) is transformed to an equivalent condition `V==M` in Figure 3(b). The operands `V` and `M` of the equality test are created by performing encryption on the same randomly generated string `S`. One encryption is performed at runtime at line 2, i.e. `V=encrypt(S, Var)`, where `Var` is used as the key for this encryption operation. Another encryption is pre-computed offline, i.e. `M=encrypt(S, Const)`, where `Const` is the key. As a result, `V==M` holds only when `Var==Const`. `Const` is also used as the key to encrypt the original candidate branch code block `C` to a cipher code block `EC`. Note that by using the equivalent condition `V==M` the constant `Const` (also the key to encrypt `C`) has been removed from the code. As shown in Figure 3(b), we define the new branch condition as well as the (cipher) code inside the branch as a SDC segment. During runtime, `EC` is only reached when the equivalent condition is satisfied upon `Var==Const`. When this happens, `Var` is immediately used as the key to decrypt `EC` (line 4). `EC` is then replaced by the resulting decrypted branch code `C` such that the execution of code `C` proceeds normally.

In Droidmarking, we embed watermark code snippets in branch code `C` of all candidate branches and create SDC segments accordingly. By this means, we build the "interdependency" between watermarks and original branch code such that it is infeasible to de-couple the two within a cipher code block unless one first decrypts it.

Knowledge Asymmetry Property of SDC: This property provided by SDC says that, the original developer knows both the constant value (`Const`) of the original branch condition and the enclosed branch code (`C`) that this constant value could "lead" to. In contrast, an attacker does not know this at all until the dynamic execution of the app program

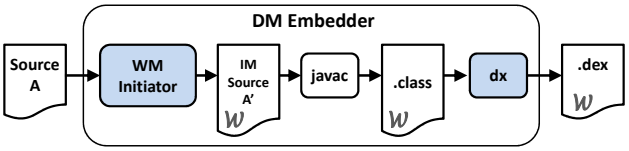


Figure 4: Work flow of Droidmarking Embedder.

"happens to" go into the SDC branch. This property is the fundamental reason why our software watermarks can be "inter-dependent" with original branch code and become non-stealthy yet still resilient to evading attacks.

4.2 DM Embedder

In the watermark embedding phase, the DM embedder helps embed watermarks into an app program for developers. The DM embedder takes app source code as input and takes three steps to output a watermarked dex file: (1) watermark code snippet generation, (2) finding SDC candidate branches, and (3) embedding watermark code snippet into SDC segments. Figure 4 shows the work flow and modules of the DM embedder. Module *WM initiator* handles step 1 and 2 and outputs intermediate (IM) source code. The following compiling process of IM source code is intervened and a modified dx compiler (a tool used for compiling Java bytecode to Dalvik bytecode) takes care of step 3. We now present the design and implementation of these three steps respectively.

4.2.1 Watermark Code Snippet Generation

The most important piece of information in a watermark instance is a unique developer identifier. On the Android platform, every installed app must be self-signed by the developer with a certificate whose private key is held by the app developer. The whole purpose of app signing is to allow the Android system to uniquely identify the developer of an app, which is used for authenticating app updates and building trust among apps from the same developer. The public key certificate is distributed with apk file (contained in self-signed X.506 certificate "META-INF/CERT.RSA") for signature verification (signature file "META-INF/CERT.SF") by the Android system upon app installation. To this end, we use public key certificates as the key information for a watermark, as it is uniquely distinguishable for developer identity (either personal or organizational) and easily attainable from the app package.

WM initiator offline generates a watermark code snippet (as shown in Figure 5) that can dynamically create a watermark instance at runtime. As shown in the code snippet, a watermark instance is an intent object (line 1), an Java object developer uses to send messages between app components on the Android system. Multiple pieces of watermark information are stored in the intent object as key-value pairs (line 2 to 7) - "`cer`": developer's pubic key certificate, "`pid`": PID of running app, "`pck`": app package name, "`iss`": certificate issuer (developer), and "`tim`": time of watermark assertion. It is important that the public key certificate be the same one whose private key will be used next for signing the app. The watermark code snippet then explicitly sets an service ("`com.dmrecognizer.dmService`" service in DM recognizer) to receive and handle the intent (line 8-9). This intent/watermark instance is then sent to

```
1  android.content.Intent wm = new android.content.Intent();
2  wm.putExtra("cer", <public key certificate>);
3  int pid = android.os.Process.myPid();
4  wm.putExtra("pid", pid);
5  wm.putExtra("pck", "com.package.appname");
6  wm.putExtra("iss", "CN=Android Debug, O=Android, C=US");
7  wm.putExtra("tim", "04/02/2014 10:06:59");
8  wm.setComponent( new android.content.ComponentName
9       ("com.dmrecognizer", "com.dmrecognizer.dmService"));
10 context.startService(wm);
```

Figure 5: Watermark code snippet.

the DM recognizer by calling `startService()` function (line 10) . We will discuss DM recognizer in more detail in Section 4.3. This watermark code snippet will be embedded into every candidate branch found in Section 4.2.2.

4.2.2 Finding Candidate Branches

In this step, WM initiator aims to find all candidate branches in the app source code that can be used to carry watermark code snippet. WM initiator statically analyzes the source code and determines the following types of `if` candidate branch conditions: (1) equal to operator ==, (2) string comparison routines such as `startswith()`, `endswith()`, and `equals()`, (3) not equal operator `!=`, and (4) multiple simple conditions combined by `&&` and || operators. Type 1 and 2 conditions (we called *simple conditions*) can be directly transformed to SDC segments, as described in Section 4.2.3. Type 3 and 4 conditions must first be transformed to simple conditions.

4.2.3 Watermark Embedding

In this step, the watermark code snippet generated in Section 4.2.1 is embedded into all candidate branches found in Section 4.2.2. These watermark-carrying candidate branches are then transformed to SDC segments.

Watermark code snippet embedding. Figure 6 shows how WM initiator embeds the watermark code snippet into a candidate branch and generates the intermediate (IM) code A'. WM initiator appends watermark code snippet (`WM`) right before the branch code (`C`), and inserts special bogus expressions at line 6 and 10 respectively in Figure 6(b) as the begin and end marker of this combined code block. As we will see, these markers will be used for SDC segment generation and kept in the final executable. As we have already seen in Section 4.1, we replace condition `Var == Const` with semantic equivalent condition `V == M`, where `M = encrypt(S, Const)`, such that the branch code encryption key `Const` in Figure 6(a) is removed from the final executable. For now, the encryption key `Const` is temporarily retained in IM code right before the begin marker and will be removed after being used for code encryption. We also insert `sdc_decrypt()` function call at the beginning of branch code in line 4. `sdc_decrypt()` is a Java function call to our native library `libSDC` via Java Native Interface (JNI). This library is responsible for decrypting the cipher code block once the branch condition is satisfied. So far, WM initiator has finished all its jobs and has generated "watermarked" IM source code, which will be compiled by javac and then passed to our modified dx compiler for SDC segments generation.

SDC segment generation. This is the final stage of DM embedding process, where the transformed candidate branches in IM code are encrypted and transformed by dx

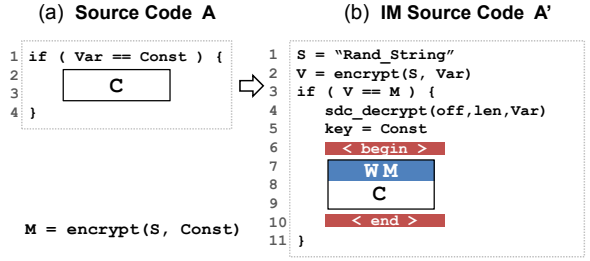


Figure 6: Embedding watermark code snippet into a candidate branch.

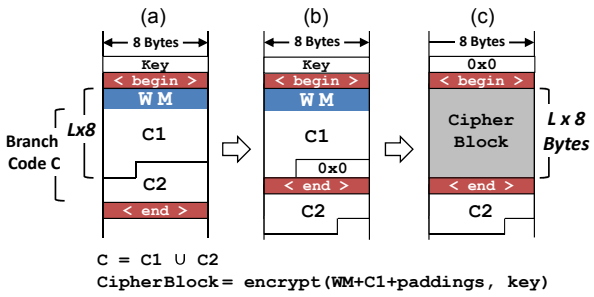


Figure 7: Process of SDC segment generation in Dalvik bytecode segment.

compiler to SDC segments in the output dex executable. dx compiler is a tool used to convert Java class files to a dex file, which contains the code, data and references to all the classes and methods of Java classes. Our modification to dx occurs after it has already generated the Dalvik bytecode segments and before it assembles these segments to an output dex file. Figure 7 shows the layout of the bytecode block in a candidate branch (after compilation from IM source code) observed by dx. It also shows how the branch bytecode `C` is "reshaped", and the beginning part of it (`C1`) together with the watermark code snippet (`WM`) are finally encrypted to `Cipher Block`. For illustration purposes, the code block is depicted with a width of 8 bytes, to be consistent with the block size (8 bytes) of DES block cipher.

In Figure 7(a), we first let dx find the transformed candidate branches by searching for the special bogus instructions indicating `<begin>` and `<end>` markers in all code segments. The code in between the markers is watermark code snippet `WM` followed by original code `C=C1+C2`. It is important to note that, in order to prevent misuse or evading attacks of SDC segments (will be further discussed in Section 4.4 DM scanner), we mandatorily require the resulting cipher code block to have a fixed size, i.e. a constant number L of 8-byte blocks in Figure 7. In one scenario in Figure 7(a) where the size of branch code is larger than $L \times 8$ bytes, we can only encrypt part of the branch code, `C1`, together with watermark code snippet `WM` to the cipher block.

As shown in Figure 7(b), once `C1` is identified, we pad `WM+C1` with `0x0` (`NOP` ops) to generate a full $L \times 8$ bytes code block (including `WM`, `C1` and padding `0x0`s). The `<end>` marker and `C2` are relocated as shown in the figure. Recall that `Const` is retained in code transformation step and remains in the bytecode before `<begin>` marker (Figure 6(b)). As we proceed in to from Figure 7(b) to Figure 7(c), `Const` is used as the key to encrypt the code block using DES algorithm and the original code block is overwritten by the resulting cipher block (same size with original code block). Note that `<begin>` and `<end>` markers must be retained to bound the cipher code block. Finally, key `Const` is removed from the code. So far, a SDC segment has been generated. Note that L should be at least larger than the watermark code snippet size, otherwise we cannot include any functional code in the SDC. L should not be too large either, to avoid too much wasted space for padding. We choose L to be 16 (128 bytes cipher block) in our prototype.

4.3 DM Recognizer

As we have seen, the watermark code snippets are first executed after the cipher code blocks in SDC segments are decrypted. This is achieved by calling function `sdc_decrypt()` to native library `libSDC` via JNI, which is able to modify bytecode during runtime. `sdc_decrypt()` decrypts cipher code block using the key procured at runtime when the branch condition is satisfied. The decrypted code (including watermark code snippet and original branch code) overrides cipher block so the program proceeds normally: the watermark code snippet is executed first and the watermark instance is created and sent to DM recognizer. For performance reasons, `libSDC` only decrypts a particular cipher code block for once, i.e. when the branch code is decrypted, it will be not be encrypted again.

We implement DM recognizer as a stand-alone application. It is responsible for managing the watermarks received from different apps on the device and for reporting repackaging threats based on the collected watermark information. A repackaged app is determined based on the public key certificate enclosed in watermark instances, if either one of the following two events occurs: (1) Watermark conflict, i.e., different public key certificates are discovered from the watermarks sent from the same app. (2) No watermark conflict, but the public key certificate retrieved from a watermark is different from its counterpart in the self-signed certificate (in file "META-INF/CERT.RSA") of the running app. The first case indicates potential change attack (i.e., attempts to disable original watermarks by changing the code), and the second indicates possible steal attack (i.e., forging original developer's watermark).

4.4 DM Scanner

Before a new app is published to the app store, Android market performs a light-weight static scan to the dex file in the app package to insure the valid and correct use of SDC segments. DM scanner can be used as part of existing vetting process performed by Android markets, such as Bouncer [25]. Those who fail the scanning are subject to misuse or evading attacks and are rejected by the market. In particular, DM scanner is deployed by Android markets to defend against library `libSDC` replacement attack. In other words, attacker's goal is to modify the decryption behavior (invoked by function call `sdc_decrypt()` and performed by native library `libSDC`) such that the watermark code snippet is removed right after the cipher code block is decrypted. Toward this goal, attackers can implement this malicious decryption behavior to be a library namely `libAttacker`.

To defend against this attack, the rules for DM scanner to check are straightforward: (1) DM scanner makes integrity check to both `libSDC` and `sdc_decrypt()` function call site. It prevents library or function call replacement attacks (positioning of SDC segments is ensured by

complying to rule 4 below); (2) DM scanner examines that `sdc_decrypt()` function must be invoked at the very beginning of the candidate branch code. This rule makes sure that no function call to `libAttacker` is added before `sdc_decrypt();` within the branch scope (in which the key to the cipher code block has already been recovered). (3) The cipher code blocks must be guided by `begin` and `end` markers. Any unrecognized instructions outside markers' guide are regarded as violation of this rule. This rule makes sure that all cipher blocks can be precisely located and no "hidden" cipher blocks in dex file can survive the scanning; and (4) Each cipher code block must have a fixed size. This rule is designed to defeat an attack in which an attacker generates an "outer" cipher code block that stealthily incorporates/encrypts both the original SDC segment (which forms nested cipher blocks) and a malicious function call to `libAttacker` (which is "hidden" from DM scanner in the "outer" cipher code block). Specifying a fixed size cipher block prevents attackers from adding encrypted malicious instructions that disable the watermark.

5. SECURITY ANALYSIS

5.1 Steal Attack

In a steal attack, attackers wish to disguise a repackaged app as original by directly copying and forging the original watermarks, making users believe that the app comes from the original trusted app developer. This attack can be readily frustrated by Droidmarking and Android app signing mechanism. Recall that a watermark instance takes the public key certificate as the vital information to identify different developers. In the watermark recognition phase, the DM recognizer detects an app's originality by comparing the public key certificates obtained from the recovered watermarks with its counterpart in the certificate in the app package (in "META-INF/CERT.RSA"), which is used for certificate verification by Android system upon app installation and has to be signed with a legitimate private key. Therefore, unless the attacker can sign the app package with the private key of original developer, leaving the original watermarks intact in the app program is always subject to being flagged as a repackaging threat, even though the app is plagiarized across different Android markets. As a result, attackers are forced to deal with watermarks and resort to more sophisticated change attacks.

5.2 Change Attack

Subtractive Attack: The goal of subtractive attack is to remove watermarks from an app without significantly degrading the value of the app. Since watermarks are embedded in non-stealthy SDC segments (containing cipher block and special begin/end markers) in the app code, it is easy for an attacker to locate all these SDC segments by static analysis. However, blindly removing SDC segments not only removes the watermarks but also the functional code contained in the SDC segments. Considering the large number of SDC segments in a program, this attack approach leads to mis- or non-functional apps, which significantly degrades their value to attackers. Attackers may resort to existing program path exploration techniques, for example forced conditional execution of branch paths or symbolic execution to automatically resolve the keys to the branches in SDC segments. However, all these techniques would fail because it is either impossible to forcibly execute encrypted code or infeasible for these path exploration tools to resolve the branch constraints calculated by non-linear encryption operations. In order to recover the SDC segments, two approaches are promising for attackers: *brute-force attack* and *dynamic analysis*. We will justify the security offered by Droidmarking against these attacks in Section 6.3.

Additive Attack: An additive attack augments the app by inserting the attacker's own watermark, which either completely overrides the original watermark or causes copyright ownership confusion upon watermark conflict, i.e., temporally infeasible to distinguish the copyright violator and original developer solely based on recovered watermark information. In our case, once the cipher block in a SDC segment is cracked by either brute-force or dynamic analysis, an attacker can override the original watermark code snippet with the attacker's own watermark and re-generate the SDC segment (assuming that attacker knows the watermark embedding process and DM scanner rules). As we will see in evaluation, although it is possible to break a subset of SDC segments in an app, it is tricky to crack all the SDC segments. Moreover, there is no copyright ownership confusion upon watermark conflict for Droidmarking, since the app signing mechanism guarantees that a watermark conflict undoubtedly indicates repackaging practice by the app publisher (whose public key certificate is contained in the app package).

Distortive Attack: An effective distortive attack applies a sequence of semantics-preserving code transformations that leads to failure to recover original watermarks. Since watermarks are encrypted with functional code and thus cannot be modified by attackers directly without first decrypting the cipher blocks, attackers may instead focus on modifying the normal decryption process (performed by library `libSDC`) invoked by `sdc_decrypt()`. DM scanner can effectively impede this attack by enforcing stringent rules, as we have discussed in Section 4.4.

6. EVALUATION

We now proceed to the empirical evaluation of the efficacy of Droidmarking in the following facets: feasibility, resilience and performance. We consider a data set of open source Android apps from F-Droid [20], an catalogue of open source apps for the Android platform. We collected totally 228 apps from nine categories as shown in Table 1. Note that the lines of code (LOC) includes the Java source code of both the app and the libraries it includes in the apk package. We conduct our experiments on Samsung Nexus 4 emulator with Android 4.0 system. As proof-of-concept, our implementation of Droidmarking requires Dalvik virtual machine (DVM) to run in non-optimization mode, in order to prevent DVM from making changes to the format of SDC segments by doing bytecode optimization. Droidmarking is applicable to DVM optimization mode and needs DVM to be "Droidmarking-aware",

Category	# of Apps	Avg LOC
Communication	13	36,436
Development	27	10,513
Game	35	6,970
Multimedia	38	13,456
Navigation	37	9,298
Office	44	9,359
Reading	21	15,347
Security	9	5,993
Web Browser	4	9,903

Table 1: Open source Android apps from 9 categories

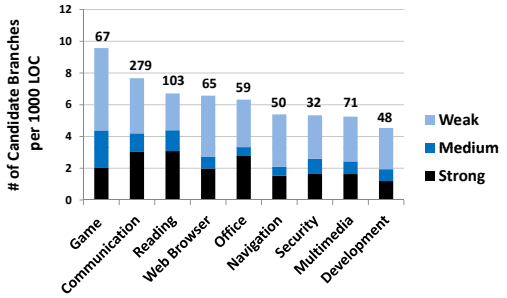
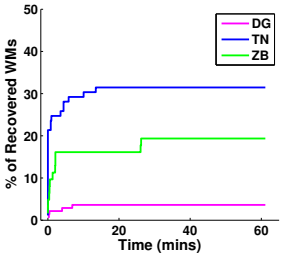


Figure 8: Average # of candidate branches per 1000 LOC (the number above each bar is the average # of candidate branches per app)



(a) % of recovered watermarks

Apps	Total # of Embedded WMs	Avg Time to Recover a WM Inst.(min)
ZB	62	0.91
TN	89	0.18
DG	138	0.14

(b) WM profile of each app

Figure 9: Watermark recovery by human testers. Show three open source apps: Zirco Browser (ZB), Tomdroid Notes (TN), and Dudo Game (DG).

we leave to future work. We choose to use emulator in our experiments as it is flexible to configure and it does not affect our results/conclusions compared with using real devices in our study. Droidmarking is general enough to be applied to apps running on other OS versions and real devices.

6.1 Feasibility

We begin by studying the feasibility of Droidmarking system. In particular, we seek to understand if there are sufficient candidate branches in a Android app program for watermark embedding.

Figure 8 shows the population of candidate branches in the apps from different categories. To even further understand the crypto strength of the keys used for SDC segments, we classify all candidate branches into three groups based on the type of constant values in the branch conditions: (1) weak - boolean or predefined values/strings, (2) medium - large numeric value, and (3) strong - string constant. We find that the density of candidate branches offered by an app source code varies depending on the nature of different categories of apps. For instance, a game app has almost 10 candidate branches per 1000 LOC on average, whereas a development app has less than half of this number. In general, most apps have a decent number of candidate branches either because of high candidate branch density (e.g., games) or large LOC (a communication app has more than $30,000$ LOC and a total of 279 candidate branches on average). Development and security categories, however, have less than 50 candidate branches per app. We also find that weak and medium candidate branches count for roughly two-thirds of the candidate branch population. This is not good news since these constants are vulnerable to brute-force attack (Section 6.3.1).

Key insights: (1) The number of candidate branches varies considerably across different app categories. (2) There are sufficient candidate branches for watermark embedding in most applications.

6.2 How Soon to Recover a Watermark?

In this section, we seek to understand how quickly users can recover a watermark and verify the copyright information. In principle, a quick answer is that, it depends on the frequency and possibility that the program execution can be lead into the SDC segments at runtime.

For clarity of the following presentations, we show the results of three selective sample apps that are representative of different categories of apps having different ranges of candidate populations: Zirco Browser (ZB), Tomdroid Notes (TN) and Dudo Game (DG). Other apps have shown similar statistics and dynamic behaviors in watermark recovery and performance. We embedded watermarks into these apps and let human testers play each app for one hour. Figure 9(a) depicts the percentage of watermarks triggered in a time series. We find that all these apps can only recover less than 40 percent of all watermarks. DG recovers less than 10 percent. We also show in Figure 9(b) the average time to recover one watermark instance (multiple instances may be generated from the same watermark code snippet in an app program). The result is encouraging as it implies that users could expect relatively short waiting time (most likely less than a minute) before receiving and reviewing the first notice of copyright information from DM recognizer.

We also observe that the average time to recover a watermark varies significantly, depending on which part of program logic the SDC segments reside in from different apps. Take Tomdroid Note (TN) for instance, we find that one SDC segment is located inside a `while` loop which searches for a user input string by repeatedly scanning through all note contents. This program logic triggers this particular SDC segment repeatedly once the same string is found during runtime and generates a considerably large number of watermark instances in a short period of time. This is an important observation from the performance point of view (which we will discuss in more detail in Section 6.4), as we could effectively reduce overhead once we know where it comes from.

Key insight: Users could expect a reasonably short period of time to recover a watermark instance.

6.3 Resilience

6.3.1 *Resilience to Static Analysis*

Attackers may statically analyze the app program and launch brute-force attacks to the encryption keys of the cipher code blocks in SDC segments.

More specifically, recall that in watermark embedding phase (Section 4.2.3), DM embedder replaces original branch condition `Var==Const` with an equivalent condition `V==M`, where `V=encrypt(S,Var)` and `M=encrypt(S,Const)`. An attacker can brute force all possibilities of `Var` and determine the key when the resulting cipher text `V` is equal to `M`. Therefore, the strength of a single SDC segment against brute-force attack depends on the bit length l of variable `Var`. Since every SDC segment uses a different random string `S`, it requires 2^l attempts at the maximum to get one key.

As presented in Section 6.1, although a considerable portion of weak and medium candidate branches are vulnerable

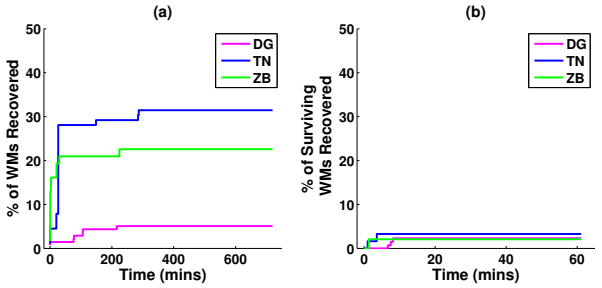


Figure 10: (a) Percentage of watermarks removed by attacker using Monkey. (b) Percentage of surviving watermarks later recovered by human testers.

to brute force attack, considering a large number of lengthy string values being the key in some categories of apps (e.g. communication and reading), brute-forcing all the keys in these apps may still be a formidable task for attackers.

6.3.2 *Resilience to Dynamic Analysis*

An alternative attack is to dynamically execute the app. Once the branch condition of a SDC segment is satisfied, the value of the key is recorded and can later be used by an attacker to decrypt the corresponding cipher block in SDC segment. An attacker may even instrument the app and automate the app execution process, hoping that the keys to all cipher code blocks could be recovered automatically. There are many tools that attackers can use for this purpose, e.g., Android app stress testing tool like Monkey [3], or smarter unit test generator like Randoop [2] or Evosuite [1].

Apps	Total # of Surviving WMs	Avg Time to Recover a WM Inst.(min)
ZB	48 (77%)	28.2
TN	61 (68%)	5.1
DG	131 (94%)	20.2

Table 2: Surviving watermarks of each app

To this end, we would like to understand whether users can still be able to recover the original watermarks in a simple dynamic analysis attack. We employ Monkey to generate pseudo-random streams of user events to each app continuously for 12 hours and flag the watermarks recovered as "removed" by attackers. For those *surviving watermarks* that are not "removed", we assume they are kept intact in the program. We then let human testers to manually practice each app for one hour and take down the surviving watermarks recovered during this time period. Figure 10(a) depicts the percentage of original watermarks removed from each app by a attacker in a time series, and Figure 10(b) shows the percentage of surviving watermarks that are later recovered from the "repackaged app" by human testers.

We make several important observations. First, although Monkey is slightly better than human testers, still has difficulty recovering all watermarks, only achieving less than 40 percent as shown in Figure 10(a). Second, Monkey removes most of the original watermarks that the human testers should have recovered when they are practicing the app Third, human testers can still trigger a small subset of surviving watermarks that Monkey does not trigger, even though human testers only run the apps for one hour (compared to 12 hours of testing by Monkey) and feed much slower events to the apps than Monkey does. We re-do the experiments using Monkey but guided by human testers (to avoid redundant testing) for one hour and have found similar results.

By carefully analyzing the surviving watermarks triggered by the humans, we summarize several reasons of the above observations: (1) Some watermark-carrying SDC segments are executed only if they are given the "right" input. Take the text searching functionality of Tomdroid Notes for instance, the SDC segment can be triggered only if the note content contains the exact string of user input. (2) Some SDC segments cannot be easily triggered without human intelligence involved. We find this effect is especially pervasive in game apps like DG, in which certain SDC segments require problem-solving logic to be triggered. (3) Some SDC segments are only reached under very specific software/hardware or environment conditions, which vary considerably case-by-case. For example, the size of the screen and the orientation of the phone affects Zirco Browser's execution logic.

An attacker may resort to smarter unit test generators, such as Randoop or Evosuite (the former uses feed-back directed and the latter uses mutation-driven test generation). However, these techniques cannot benefit from performing code analysis to the encrypted SDC segments, which weakens their capability of test generation for the SDC segments. Moreover, although these widely-used tools and have proved to be capable of finding software defects and achieving high code coverage for non-user-interactive traditional Java programs, the event-driven Android applications have raised unique challenges to the effectiveness of these tools. These tools are not good at (or designed for) mimicking a diversity of user interactions, events, and software/hardware environments on an Android system. For these reasons, it is non-trivial to recover all SDC segments in an Android app even using state-of-the-art unit test generators.

Meanwhile, recall that it is sufficient for Droidmarking to flag an app as repackaged (with zero false-positive rate) as long as one original watermark in an SDC segment is recovered from one user device. Considering the large user base, a repackaged app cannot become fully "safe" from being detected unless the attacker manages to remove (or disable) all original watermarks. While it is possible for a dedicated attacker to achieve this by using a hybrid of attacks (sophisticated static and dynamic analysis) and/or directly removing the un-recovered SDC code segments (losing program functionality), the resulting cost may no longer be attractive for an attacker.

Key insights: (1) Droidmarking provides different levels of defense against brute force attack, depending on the number and types of candidate branch conditions in an app. (2) In a Monkey-based dynamic analysis attack, users are still able to successfully recover original watermarks. For smarter testing tools, the SDC segments in Android apps have raised unique challenges to their effectiveness. (3) To completely remove all watermarks, an attack may resort to a combination of various attacks, which increases the cost of app repackaging.

6.4 Performance

The application performance overhead introduced by Droidmarking comes from two major types of computations: (1) the encryption operation (in bytecode) computing the equivalent branch condition before each SDC segment, and (2) the decryption operation of the cipher code block performed by

libSDC (in native code) when the above condition satisfies for the first time. To evaluate the overhead, we use Monkey to generate the same sequence of 2,000 user events to both the apps with and without watermarks embedded. We measure the run time T and T_w for original and watermarked apps respectively. The overhead is calculated by $(T_w - T)/T \times 100\%$. Table 3 shows the performance overhead of Droidmarking for the three apps. We find that the performance overhead of app DG is relatively low (around 3 percent) although it is a game app and has higher candidate branch density. However, the overheads of ZB and TN are particularly higher, even though the ZB's watermark recovery frequency is the lowest (see Figure9). In fact, this is not surprising. ZB and TN represent a large group of apps, in which a few candidate branches reside in the "popular" loops in these apps, e.g., the program logic of text search in TN app, as we have discussed in Section 6.2. Although these branches are not necessarily entered every time, the branch conditions (encryption) are always calculated. This is the reason why ZB has high overhead in spite of low watermark recovery frequency.

Apps	T w/o WM(sec)	T_w with WM(sec)	App Perf. Overhead
ZB	142	150	5.1%
TN	196	212	8.2%
DG	177	182	2.8%

Table 3: Application performance overhead.

To improve performance, one could opt for a smarter watermark embedding strategy. For instance, in the watermark embedding phase, one could skip those candidate branches in the scopes of loop statements. To be more precise, one can potentially build the program dependency graph of the app and choose to "filter out" those candidate branches in the "popular" execution path of the program. In this way, one can even quantitatively evaluate the trade-off among performance, speed of watermark recovery, and resilience of Droidmarking scheme based on one's demand. We leave this for future work.

Key insight: (1) The overhead is relatively small on some apps but higher on others, mainly depending on whether branch conditions reside in frequent execution paths of the app program. (2) Optimized watermark embedding strategies can be applied to reduce performance overhead.

7. RELATED WORK

Software watermark and birthmark: There has been extensive work on both static [23, 22, 8] and dynamic [6, 19, 18, 7, 13, 8] software watermarking technqiues. Most recently, AppInk [28] has adopted traditional graph-based dynamic watermarking technique and implements it on Android apps. A software birthmark is a characteristic that can uniquely idenfity a program. The characteristic is selected either statically or dynamically from the whole program paths [14] or system call graphs[27]. Droidmarking differs from all existing watermark and birthmark techniques in that our approach is "non-stealthy", allowing normal users to readily verify watermarks without requiring any secret or confidential watermark recognizer. The "inter-dependency" built between watermarks and carrier code insures its resilience to various evading attacks.

Android app repackaging detection: The app repackaging problem has drawn recent efforts from both industry (e.g., Google Android app licensing[5], Amazon DMS [4], etc.) and academia. App repackaging detection techniques have been proposed mainly focusing on similarity mining from a large basis of apps. These techniques differ in the features and methods used for app similarity comparison. For instance, [30, 16] use hashing of app instruction sequence, [21] and [29] use program syntactic and semantic fingerprints respectively, and [9, 12, 11] use program dependency graph as features. These works either employ pair-wise similarity comparison [9, 30, 21, 12] or more efficient algorithms like nearest neighbor searching or clustering [16, 11, 29] to improve scalability. Our work differs from existing techniques in that we devise a new watermarking technique that aims to quickly and accurately detect app repackaging threats and provide solid proof of original copyright ownership of an app.

Malware code obfuscation: Self-decrypting code is first proposed by Sharif et al., referred to as conditional code obfuscation in their work [24]. They apply it to malware programs in order to conceal trigger-based malicious behaviors from state-of-the-art malware analyzers. The fundamental difference in our work is that we use self-decrypting code for designing and implementing a new dynamic watermarking approach, targeting at a totally different problem - Android app repackaging detection. In our design, self-decrypting code is used "non-stealthily" as the primitive of our watermarking scheme. The purpose is not to conceal watermarks but instead to build the "inter-dependency" between watermarks and the carrier code in order to improve its resilience to evading attacks.

8. DISCUSSION AND CONCLUSION

There are several limitations of Droidmarking. First, the types of candidate branches that our watermark can be embedded to is still limited, (i.e., the conditions of candidate branches must be equality test). Operators such as $<$, $>$ are not applicable. Second, the encryption strength relies on the number of possible values and length of a numerical/string constant that is used in a candidate branch. As we have presented in Figure 8, roughly two-thirds of the candidate branch conditions use predefined values, which are in some cases not difficult to be enumerated by attackers. Moreover, for some string constants, understanding the semantics of the branch conditions is helpful for an attacker to guess their values. Take the candidate branch in Figure 2(a) for instance, once an attacker knows that variable url refers to a web resource identifier, it becomes substantially easier than brute force to guess the string constant. Third, the introduction of cipher code blocks may be used by malware writers for code obfuscation although stringent rules are enforced by the DM scanner.

In conclusion, we have designed a new "non-stealthy" dynamic software watermarking technique - Droidmarking - based on SDC for Android apps. We evaluate Droidmarking's efficacy in feasibility, resilience and performance on real-world open source apps. Our results prove that Droidmarking is able to achieve all our goals in impeding app repackaging in today's Android markets.

9. ACKNOWLEDGMENT

We would like to thank anonymous reviewers whose comments help us improve the quality of this paper. This work was supported by ARO W911NF-09-1-0525 (MURI), NSF CCF-1320605, ARO W911NF-13-1-0421 (MURI), NSFC 61100226, NSFC 61170281 and strategic priority research program of CAS (XDA06030600).

10. REFERENCES

[1] Evosuite. http://www.evosuite.org.

[2] Randoop. http://code.google.com/p/randoop.

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

[4] Amazon Appstore Digital Rights Management, 2011. http://www.amazon.com/gp/help/customer/display.html?nodeId=201485660\&_encoding=UTF8\&ref_=mas_help_legacy_legal_doc_page.

[5] Google Play Licensing Service, 2012. http://developer.android.com/google/play/licensing/index.html.

[6] C. Collberg and C. Thomborsen. Software Watermarking: Models and Dynamic Embedding. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)*, 2001.

[7] C. Collberg and E. Carter and S. Debray and A. Huntwork and C. Linn and M. Stepp. Dynamic Path-Based Software Watermarking. In *Proceedings of the ACM Conference on Programming Design and Implementation (PLDI)*, 2004.

[8] C. Collberg and G. Myles and A. Huntwork. SandMark - A Tool for Software Protection Research. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)* , 2003.

[9] K. Chen, P. Liu, and Y. Zhang. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *Proceedings of International Conference on Software Engineering (ICSE)*, 2014.

[10] C. S. Collberg and C. Thomborson. Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection. *IEEE Transactions on Software Engineering*, 28(8):735–746, 2002.

[11] J. Crussel, C. Gibler, and H. Chen. AnDarwin: Scalable Detection of Semantically Similar Android Applications. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2013.

[12] J. Crussell, C. Gibler, and H. Chen. Attack of the Clones: Detecting Cloned Applications on Android Markets. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2012.

[13] G. Myles and C. Collberg. Software Watermarking Through Register Allocation: Implementation, Analysis and Attacks. In *Proceedings of Conference on Information Security and Cryptology (ICISC)*, 2003.

[14] G. Myles and C. Collberg. Detecting Software Theft via Whole Program Path Birthmarks. In *Proceedings of Information Security International Conference (ISC)*, 2004.

[15] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi. AdRob: Examining the Landscape and Impact of Android Application Plagiarism. In *Proceedings of International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2013.

[16] S. Hanna, L. Huang, S. Li, C. Chen, and D. Song. Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications. In *Proceedings of DIMVA*, 2012.

[17] H. Huang, S. Zhu, P. Liu, and D. Wu. A Framework for Evaluating Mobile App Repackaging Detection Algorithms. In *Proceedings of International Conference on Trust and Trustworthy Computing (TRUST)*, 2013.

[18] J. Nagra and C. Thomborson and C. Collberg. Threading Software Watermarks. In *Proceedings of Information Hiding (IH)*, 2004.

[19] J. Palsburg and S. Krishnaswamy and M. Kwon and D. Ma and Q. Shao. Experience with Software Watermarking. In *Proceedings of Annual Computer Security Applications Conference (ACSAC)*, 2000.

[20] F-Droid, Free and Open Source Software Apps for Android, 2014. https://f-droid.org/.

[21] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang. Plagiarizing Smartphone Applications: Attack Strategies and Defense Techniques. In *Proceedings of ESSoS*, 2012.

[22] R. L. Davidson and N. Myhrvold. Method and System for Generating and Auditing a Signature for a Computer Program. US Patent 5559884 A.

[23] R. Venkatesan and V. Vazirani and S. Sinha. A Graph Theoretic Approach to Software Watermarking. In *Proceedings of Information Hiding (IH)*, 2001.

[24] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2008.

[25] Android and Security, 2012. http://googlemobile.blogspot.com/2012/02/android-and-security.html.

[26] William Feng Zhu. *Concepts and Techniques in Software Watermarking and Obfuscation*. Ph.d. thesis.

[27] X. Wang and Y. Jhi and S. Zhu and P. Liu. Behavior Based Software Theft Detection. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2009.

[28] W. Zhou, X. Zhang, and X. Jiang. AppInk: Watermarking Android Apps for Repackaging Deterrence. In *Proceedings of ACS Symposium on Information, Computer and Communications Security (ASIACCS)*, 2013.

[29] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, Scalable Detection of "Piggybacked" Mobile Applications. In *Proceedings of ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[30] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces. In *Proceedings of ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2012.

[31] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2012.

[32] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2012.