AppInk: Watermarking Android Apps for Repackaging Deterrence

Wu Zhou [†], Xinwen Zhang [‡], Xuxian Jiang [†]
[†]North Carolina State University [‡]Huawei Research Center
{wzhou2, xjiang4}@ncsu.edu xinwen.zhang@huawei.com

ABSTRACT

With increased popularity and wide adoption of smartphones and mobile devices, recent years have seen a new burgeoning economy model centered around mobile apps. However, app repackaging, among many other threats, brings tremendous risk to the ecosystem, including app developers, app market operators, and end users. To mitigate such threat, we propose and develop a watermarking mechanism for Android apps. First, towards automatic watermark embedding and extraction, we introduce the novel concept of *manifest app*, which is a companion of a target Android app under protection. We then design and develop a tool named *AppInk*, which takes the source code of an app as input to automatically generate a new app with a transparently-embedded watermark and the associated manifest app. The manifest app can be later used to reliably recognize embedded watermark with zero user intervention. To demonstrate the effectiveness of AppInk in preventing app repackaging, we analyze its robustness in defending against distortive, subtractive, and additive attacks, and then evaluate its resistance against two open source repackaging tools. Our results show that AppInk is easy to use, effective in defending against current known repackaging threats on Android platform, and introduces small performance overhead to end users.

Categories and Subject Descriptors K.6.5 [**Management of Computing and Information Systems**]: Security and protection – *Invasive software*

General Terms Security; Algorithms

Keywords: Mobile Application; App Repackaging; Software Watermarking; App Protection; Smartphone Security

1. INTRODUCTION

With the unprecedented adoption of smartphones in consumer and enterprise users, a large number (and a wide variety) of mobile applications (apps) have been developed and installed to extend the capability and horizon of mobile devices. These apps in return foster an emerging app-centric business model and drive innovations across personal, social, and enterprise fields. At the same time, the wild proliferation of mobile apps has introduced serious risks to the stakeholders in the ecosystem. Particularly, app repackaging has been considered as a major threat to both app developers and end users. Through app repackaging, malicious users can breach the revenue stream and intellectual property of original app authors, and plant malicious backdoors or payloads to infect unsuspecting mobile users. Recent studies have shown that app repackaging is a real threat to both official and third-party Android markets [52,53], and regarded as one of the most common mechanisms leveraged by Android malware to spread in the wild [54]. Investigation has also shown that app repackaging presents serious vulnerability to mobile banking [27].

Facing the prevalent risks brought by app repackaging, we are in a desperate need of a reliable, efficient, and easy-to-use mechanism to detect repackaged apps and prevent their propagation. Android package obfuscation tools such as Proguard [29] and DexGuard [25] have been provided by Google and other companies to confuse attackers when they are in the process of repackaging an app. However obfuscation can only increase the difficulty of reverse engineering Android apps, and cannot stop determined attackers from achieving their purposes through manual analysis, laborious experiments, and strong persistence. To defend against Android app piracy and repackaging, Google has introduced a tool library named license verification library to protect app developers from having their apps stolen by third parties [17]. Other app market operators (such as Amazon and Verizon) also provide their own digital right management (DRM) options for app developers which can be applied to prevent apps from being copied and pirated [41]. However, these mechanisms are hard to deploy correctly [13] and also easy to crack [30–32]. Recently, researchers have introduced various techniques to detect repackaged apps on a large scale [8,16,42,52,53]. However, these mechanisms usually cannot detect app repackaging online and in real-time. For these reasons, repackaged apps have usually been widely distributed before being detected.

To overcome the above weaknesses, we propose to embed software watermarks dynamically into the running state of an Android app to represent the author or developer's ownership. For verification, an authorized party can extract the embedded watermark by running the app with a specific input in a dedicated environment, e.g., a customized emulator. When the extracted watermark matches the one provided by the developer, the verifying party (e.g., an arbitrator) can confirm the ownership of the original developer even when the app is repackaged by another publisher. The proposed watermarking mechanism should be resistant to manipulation by common static and automated attacks, therefore making it hard for an attacker to remove the original embedded watermark or embed his own watermark.

A desirable solution needs to meet two requirements: watermark embedding should be readily integrated into current app development practice, and watermark extraction should be convenient for the authorized verifying party to perform. To fulfill these two requirements, we introduce the concept of *manifest app*, which is a companion app to the original app under protection. Basically the manifest app encapsulates a specific input to drive the watermark-protected app automatically, and thus eliminates the user interventions needed in traditional watermark extraction. Based on the manifest app, we design and implement a practical tool named *AppInk*, which consists of four components: watermarking code generation, automatic manifest app generation, watermark embedding, and watermark extraction. By seamlessly integrating these four components, AppInk presents an effective app protection solution for both developers and other authorized verifying parties. Specifically, to leverage AppInk to protect her own app, a developer applies the first three components of AppInk to the app's source code to generate two apps: the watermark-protected app to be released to the public; and the manifest app which is presented on demand to an authorized party to verify the originality of the watermark-protected app. Upon request, the verifying party will run the manifest app in the watermark extractor (the fourth component of AppInk), which automatically launches the app under review and extracts the originally embedded watermark.

Two typical scenarios can leverage this process to detect unauthorized repackaged apps and prevent their propagation. The first scenario has a central authority (e.g., the app market operator) to review each submitted app to verify its originality before accepting it for publication. For that purpose, each app publisher submits the manifest app along with the app under review (publishers who cannot submit a companion manifest app are most likely not original developers). The app market operator then runs the watermark extracting algorithm, using the manifest app provided by the publisher to drive the app under review.

The second scenario has a third-party arbitrator, who inspects the evidence of app ownership to resolve dispute upon request. In this scenario, when an app author suspects that one app is a repackaged version of her own, she can run the suspect app inside the watermark recognizer, using her own manifest app to feed input. If the watermark extracted from the suspect app is the same as the watermark from her own app, she can submit this as evidence to prove that the suspect app is a repackaged version of her own app. Within this scenario, AppInk provides an effective mechanism to prevent the propagation of repackaged apps across different app markets.

To demonstrate the effectiveness of AppInk in deterring app repackaging, we analyze its robustness against general watermark-targeted attacks, including distortive, subtractive, and additive attacks. We also study its resistance against two open source repackaging tools (Proguard [29] and ADAM [51]). Our results show that AppInk is effective in defending against common automatic repackaging attacks. Our performance evaluation indicates that an embedded watermark introduces only a small overhead for end users.

In summary, this paper makes the following contributions::

- First, we design a complete dynamic graph based watermarking mechanism for Android apps, which can be used to detect and deter further propagation of repackaged apps. To the best of our knowledge, it is the first watermarking mechanism for Android apps.
- Second, we introduce the concept of manifest app and design a series of automatic processes to make the watermarking mechanism integratable into current app development practice and also conveniently deployable for arbitrators.
- Third, we implement a prototype tool named AppInk and evaluate it against two open source repackaging tools, and demonstrate that it is effective in defending against commonly available automatic attacks.

The rest of this paper is organized as follows. We present the paper overview in Section 2, describe the AppInk design in Section 3, and present its prototype implementation in Section 4. After that, we present the robustness analysis of AppInk and evaluate it against real world repackaging attacks in Section 5, and discuss the system's limitations and suggest possible improvements in Section 6. Lastly, we describe related work in Section 7 and conclude this paper in Section 8.

2. OVERVIEW

2.1 Problem Statement

App repackaging refers to disassembling one app, making some changes (to the code, data, or simply the signing key inside the original app), and rebuilding the modified components into a new app. As a technical method, it can be used for benign purposes. For example, ADAM [51] uses app repackaging to tweak malware samples for the purpose of stress testing various Android antivirus tools. Aurasium [50] uses app repackaging to intercept an app's interaction with its underlying OS, aiming to enforce user-specified security policies for the app. However, app repackaging is more commonly used for surreptitious and malicious purposes. For example, greedy publishers use app repackaging to replace existing in-app advertisements or embed new ones to steal advertisement revenues [8, 52, 53]. Malicious attackers use app repackaging [42, 53, 54] to plant malicious backdoors or payloads into benign apps.

Because it is relatively easy to reverse engineer Android apps (which are mainly written in Java), app repackaging has been identified as a widespread practice in current diversified app distribution channels [8, 16, 23, 42, 53]. As a result, it not only brings a lot of damages to app authors (in terms of losing their monetary income and intellectual property), but also causes tremendous risks to the large community of mobile users and affects the burgeoning innovative app economy. As concrete examples, severe vulnerabilities have been found in mobile banking apps through app repackaging, and serious doubt is cast on mobile banking security and feasibility in general [27]. More recently, researchers have identified that about 10% of Android apps available in popular third-party markets are repackaged [53]. The latest investigation from the industry [48] has reported that most of the popular mobile apps are beset by app repackaging threat: 92 of the top 100 paid apps for Apple iOS, and all of top 100 paid apps for Android were found to be hacked.

Facing the widespread propagation of the app repackaging threat, effective security defenses are seriously lagging behind. The current industrial practices are either too weak to deter determined attackers from conducting repackaging attacks, or too complex to be deployed properly. For example, app developers are encouraged to use obfuscation to protect their apps, but the introduced confusion is usually not strong enough to prevent determined attackers from achieving their goals [29]. App market operators (e.g., Google) have provided license verification or DRM service to apps submitted to their stores, but automatic repackaging tools can work around them easily [30–32]. Recently, researchers have begun to tackle this problem, but most of the proposed solutions so far focus on feasible mechanisms to detect repackaged apps after their propagation [8,16,42,53]. Considering the wide and severe impact of app repackaging, an effective and robust mechanism is urgently demanded to efficiently prevent and deter app repackaging in the first place.

2.2 Software Watermarking

Software watermarking has been studied extensively to defend against the piracy of desktop software [4, 6, 35, 37, 39]. Since the processes of mobile app repackaging and desktop software piracy are similar, we believe that software watermarking can be a promising technique in deterring app repackaging. In general, watermarking software involves two steps: first a watermark, typically a number or a message string known only to the author or publisher, is embedded into the target software in a specific way such that it does not affect the running behaviors of the original app and is difficult to remove without modifying the original app semantic; then, a recognition technique is used to extract the original watermark from the software. The matched watermarks verify that this software package belongs to the original developer or publisher. Depending on how the watermark is embedded and extracted, there are static and dynamic watermarking methods. Static watermarking embeds the watermark into the code or data of a package and extracts the watermark without executing the code; dynamic watermarking embeds the watermark into the execution state of the target software, and extracts the watermark during runtime.

Regardless of which method is used, it is desirable that a watermark embedded in a software package should be robust to various well-known attacking techniques, especially distortive attacks – to apply semantic preserving transformation on the watermarked code to modify the embedded watermark, subtractive attacks – to remove complete or partial watermark, and additive attacks – to add attacker's own watermark and confuse the arbitrator on resolving ownership dispute. Typically, dynamic watermarking has a stronger resistance against these attacks than static watermarking, and it is used in our approach to embed and verify the ownership of a specific Android app package.

2.3 Challenges of Watermarking Android Apps

There are several key challenges to incorporate dynamic watermarking into current Android app development practice and make it easily deployable by arbitrating parties.

Firstly, the state of the art dynamic Java watermarking techniques need extensive intervention from developers to embed a watermark. For example, SandMark [5] requires developer to manually annotate source code to indicate where watermark *can* be inserted, and to manually give input to drive the software when embedding a watermark. These manual interventions make it cumbersome to apply this technique in real practice and hard to be made right.

Secondly, it is desirable to have automatic watermark recognition so that it can handle thousands of apps with little human effort in an online and realtime manner. To recover embedded watermarks, SandMark leverages programmable Java Debug Interface [24] to access memory objects on the heap in order to infer object reference relationships. However, there is no known programmable debugging interface available on Android. Even worse, manually providing input is required for watermark extraction phase as well. Obviously, this cannot scale to handle the large number of watermark recognition requests for thousands of Android apps submitted to current app markets on a daily basis.

Thirdly, Android apps, although mainly developed in Java language, have significant difference from desktop Java software. For example, Android apps depend more heavily on event-driven mechanisms and the underlying execution environment to work correctly. Unlike legacy Java applications that have a single entry point named `main`, Android apps in general have multiple entry points.

2.4 Solution Overview

To overcome these challenges, we introduce an entity named *manifest app*, which is a companion app for a target app under protection. Basically, the manifest app encapsulates a sequence of input event to drive the watermark-protected app automatically, and thus eliminates the user intervention needed in traditional watermark extraction. Based on the manifest app, we design and implement a practical tool named AppInk to automatically generate the manifest app, embed the watermark, and execute the dynamic watermark extraction with zero user intervention. As an input during watermark embedding, the manifest app encodes the event sequences and accordingly indicates the event handlers of the target app where watermarking code segments can be inserted, which addresses the first challenge. As an input for watermark extracting, the manifest app automatically launches the original app and feeds the input event sequences to it, which triggers all the inserted watermarking code segments and thus recovers the watermark object embedded. By doing so, we effectively overcome the second challenge. Based on the insight that each event in Android platform is uniquely mapped to a well-known system API, we propose a conservative method to automatically generate an event flow model for Android app, and leverage model-based test generation to automatically create a suboptimal manifest app for watermarking purpose, thus resolving the third challenge.

3. APPINK DESIGN

Figure 1 depicts the overall AppInk architecture. At the app developer (left) side, AppInk consists of three components: manifest app generation, watermark code generation, and watermark embedding based on source code instrumentation. The input of the manifest app generation component is the source code of the target app including its resource files. The watermark code generation takes a watermark object (e.g., a number or a string) and outputs watermarking code segments. The watermark embedding component takes the manifest app and the code segments as inputs, and generates a watermarked Android package that can be released to app markets. At the arbitrator (right) side, the watermark recognizer takes the inputs of the released Android package and the manifest app from the app developer side, and extracts the embedded watermark.

Watermarking code generation: Given a watermark value specified by app developer, this component encodes the watermark value into a special graph structure and transforms the graph into watermarking code. In order to improve the stealthiness of the embedded watermark, AppInk splits the watermarking code into a variety of segments, each of which will be inserted into different locations of the original app's source code. The execution states of these code segments collaboratively present specific object reference relationships and thus can be leveraged to reveal the original watermark value. Section 3.1 explains the detailed design of this component.

Manifest app generation: The main function of the manifest app is to feed pre-determined user inputs to the app under review, which trigger the executions of embedded code segments and thus recover

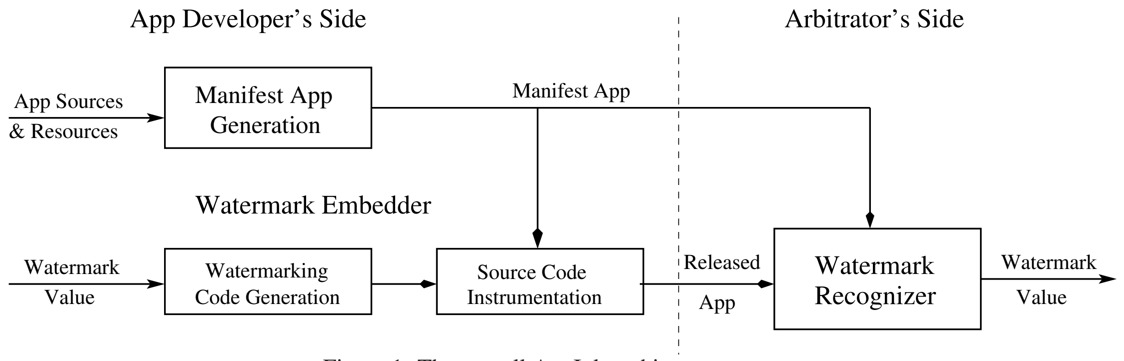


Figure 1: The overall AppInk architecture.

the watermark value with the help of the watermark recognizer. To ease the burden of writing manifest apps by developers, AppInk leverages the event-driven nature of Android apps and the latest model-based test case generation to automatically generate these input events and makes this process totally transparent to app developers. Section 3.2 presents the detailed design of this component.

Source code instrumentation: By parsing the files in the manifest app, this component first identifies encoded user input events, and then determines their corresponding event handlers based on the source code of the original app. Next, it inserts the watermark code segments under the path of these identified event handlers. After that, AppInk packages the modified app source into a released app, which is for both public publication and arbitrating purposes, and the manifest app into another executable package, which is not released to the public but rather dedicated to arbitrator use. Section 3.3 presents the detailed design of this process.

Watermark recognizer: This component is a modified Android emulator on x86 [19], which is invoked by a shell script. The script first installs both the Android app and the manifest app in the emulator, then starts the manifest app which feeds a sequence of input events to the Android app, and then calls the extended Dalvik Virtual Machine (DVM) [10] to export all object reference information in the runtime heap. From this information, AppInk uses a special pattern to match potential watermarking structure. If such a watermarking structure is identified, a reversed process of the watermarking code generation is invoked to recover the embedded watermark value. Section 3.4 illustrates the design of this component.

3.1 Watermarking Code Generation

Different from static watermarking which embeds a secret watermark object (e.g., a numerical value or a message string) into the code or data section of a target application, dynamic watermarking embeds a watermark object into special structures that present themselves only in the runtime of the target application. AppInk adopts graph-based data structure, which is hard to be reversed by attackers due to the inherent difficulty of analyzing point-to relationship in graphs [14,45].

There are different ways to use graph to encode a watermark object. AppInk uses permutation graph, which adopts a special graph structure to encode a permutation mapping to the watermark object. As depicted in Figure 2, the graph includes 5 nodes, each of which has two outgoing edges, one in solid line and one in dotted line. Through the solid line edges, the graph forms a cycle. If we can further identify one unique node (e.g., the one which is referenced by any other object outside the figure), a specific order is defined. Suppose that only the first node is referenced by another object not in the figure, we can assign number 0 to this node, which

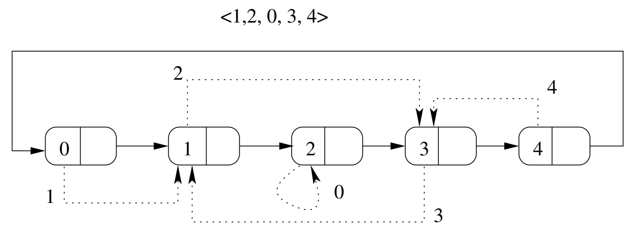


Figure 2: Example permutation graph. This graph encodes numerical value of 116 as a watermark.

```
1  class WatermarkNode {
2    WatermarkNode solid;
3    WatermarkNode dotted;
4    ......
5  }
6  ......
7  node0 = new WatermarkNode();
8  ......
9  node4 = new WatermarkNode();
10
11 node0.solid  = node1;    // Points to next node
12 node0.dotted = node1;    // Encode number 1
13 ......
14 node4.solid  = node0;
15 node4.dotted = node3;    // Encode number 4
```

Figure 3: Watermarking code for the permutation graph in Figure 2.

is called root node, and number 1 to 4 for the other four along the circle. A dotted line edge of a node is then associated with a number counting the distance from this node to its target node along the solid line edges. For example, the dotted outgoing edge from `node0` to `node1` encodes a number of 1 since the distance from `node0` to `node1` is 1 along the solid line edges. Similarly the dotted outgoing edge from `node3` to `node1` encodes a number of 3. In Figure 2, the 5 dotted edges encodes the numbers of 1, 2, 0, 3, and 4 respectively, which is a perfect form of permutation. According to the permutation-to-number algorithm in [28], `<1, 2, 0, 3, 4>` is mapped to `116`, which is the watermark value encoded by this graph.

In Java language, the permutation graph depicted in Figure 2 can be represented by a doubly linked list, as shown by the skeleton code in Figure 3. The class `WatermakrNode` (lines 1 to 5) represents the node in the permutation graph. The following initialization code (lines 7 to 9) creates five instance nodes, and the later code (lines 11 to 15) defines the object reference relationship, from which we can reconstruct the permutation graph at runtime. As commented in the list, the member field `solid` points to the next node in the list, and all these fields form a cycle. The member

```
 1 public class TestAndroidCalculator
 2   extends ActivityInstrumentationTestCase2<Main> {
 3     protected void setUp() { ... }
 4     protected void tearDown() { ... }
 5     public void testEventSequence() {
 6       enterText(0, "10");
 7       enterText(1, "20");
 8       clickOnButton("Multiply");
 9     }
10 }
```

Figure 4: Example manifest app based on Robotium.

field `dotted` encodes the permutation distance for each node in the permutation graph, which jointly encodes the watermark value specified by the app developer.

When the above code is executed on Android platform, memory space will be allocated for each instance object (`node0` to `node4`) at lines 7 to 9. At lines 11 to 15, the member fields are assigned, which results in the establishment of the object reference relationship among these `WatermarkNode` instances. Through analyzing the runtime heap, this object reference relationship can be extracted and decoded to recover the original watermark value. Section 3.4 presents the details of this process in the watermark recognizer.

Because linked structures are very commonly used in Java applications, it is hard to distinguish these watermarking code from other code. The *stealthy* nature of these graph structures, combined with the inherent difficulty of point-to analysis in graph [14, 45], makes it very challenging for attackers to succeed in reverse engineering the watermarked code. (More detailed analysis on the robustness of this technique is presented in Section 5.1.) To further improve the stealthiness of the watermarking code, AppInk splits the watermarking code into a number of segments and inserts them into a variety of places in an app. This is especially helpful when the watermark value is large and thus has to be represented by a large number of code segments. Section 3.3 presents more details of this technique with the help of manifest app.

3.2 Manifest App Generation

3.2.1 Manifest App Based on Robotium

Working as a companion app to drive the execution of a released app inside watermark recognizer, a manifest app functions in similar way as test cases. However, unlike common unit tests which only provide component specific tests [21] and special Android UI/application exerciser [22] which sends random stream of events to apps under test, a manifest app needs programmable event delivery within the entire target app, so that watermarking code can be scattered to different places, thus achieving better stealthiness. For this purpose, AppInk generates manifest app based on Robotium [46], which extends Android app instrumentation framework and provides precise UI element locations and event delivery. Figure 4 shows an example of the Robotium test case.

In Figure 4, the method `setUp` starts the main activity of the app under test, `tearDown` clears the execution environment and stops its execution, and `testEventSequence` (lines 5 to 9) sends specific sequence of events to the app. To automatically generate manifest app and later extract watermark by the recognizer, AppInk needs to decide a proper input sequence and fill them into the method `testEventSequence`. Specifically, AppInk has two requirements for the input sequence: its execution must deterministically trigger the watermarking code segments, and it must be diversified enough so that the watermarking code can be scattered into a large enough scope. AppInk leverages the event-driven nature of Android apps and model-based automatic test case generation to achieve these purposes.

3.2.2 Manifest App Generation

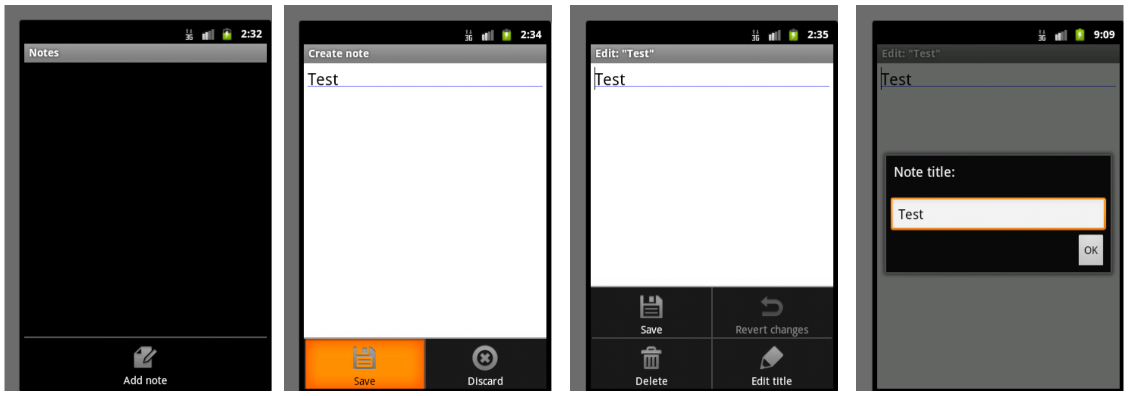
Different from desktop applications, the control flow of Android apps depends heavily on the diversified Android events, including user generated events (e.g., key presses and screen taps) and system generated events (e.g., short messages received, incoming phone calls, and various sensor events). Each event is handled by a well-defined Android API. For example, a menu item click is handled by the method `onOptionsItemSelected` in the corresponding activity, a button press is handled by the `onClick` method of the listener object registered for the button, and a short message received event is handled by the method `onReceive` of the activity registered for `SMS_RECEIVED` intent. By issuing these events in a well-defined order, the app under test invokes these event handlers in order, and responds in a deterministic manner.

For the purpose of automatic test generation, model based methods have been well studied for event-driven software in general, and actively investigated for Android apps in particular [26, 34, 38, 47]. But to use it in AppInk, we need to have an app's event flow model as input. One option is to ask developers to provide the model. But this puts an extra burden on them, and it is also prone to error. Another option is to infer the event flow model through reverse engineering. We note that generating a complete app model is hard, but unlike common test case generation (whose task is to exhaustively generate test sets to cover as many paths as possible), AppInk only needs one test case if it can trigger as large set of code segments as possible to achieve stealthy watermark embedding. Therefore we only need to have a partial model for the app event flow.

AppInk uses static method to infer a partial event flow model for Android apps in a conservative but safe way. This is achieved through parsing app source files, including `AndroidManifest.xml`, UI layout, Java and other resource files. The generated model is fed to an existing model-based test generator for Android [11] to generate a set of test cases, from which we pick the test input that covers most code segments.

We now use an example app to describe how AppInk generates the event flow model. Figure 5 shows the user interface elements and relevant events for the app `NotePad`. The first screen (5a) pops right after the app starts. By analyzing the layout and Java source files, we infer that the event `Add note` is handled by method `onOptionsItemSelected` in file `NotesList.java`. The second screen (5b) shows the UI elements for the action of adding a note, including a text input box and two menu items (`Save` and `Discard`), whose event handlers are the method `onOptionsItemSelected` in file `NoteEditor.java`. The third screen (5c) shows the UI elements for editing a note, including a test input and three menu items (`Save`, `Delete`, and `Edit Title`), each of which has its own handler. The last screen (5d) shows the UI elements for editing title, including a test input and one button (`Ok`), whose handler is the `onClick` method in file `TitleEditor.java`.

At this point, each screen shows only individual events. Through analyzing the handler for the event of clicking `Add note` (method `onOptionsItemSelected` in file `NotesList.java`), AppInk determines that it starts an activity with the intent of `ACTION_INSERT`, which is found later to be defined in file `NoteEditor.java` by parsing file `AndroidManifest.xml`. So the event `Add Note` connects screens 5a and 5b. Through



(a) Main Activity - NotesList (b) Note Editor - Create Note (c) Note Editor - Edit Note (d) Note Editor - Edit Title

Figure 5: User interface elements in app `NotePad`.

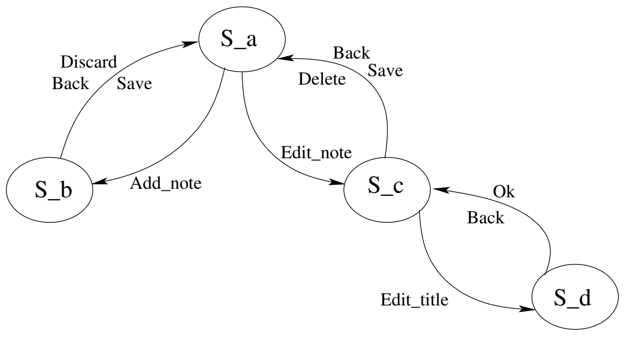


Figure 6: Event flow graph for `NotePad`.

```
public void testEventSequence() {
  clickOnMenuItem("Add note");
  enterText(0, "Test");
  clickOnMenuItem("Save");
  goBackToActivity("NotesList");
  clickInList(1);
  clickOnMenuItem("Edit Title");
  clickOnButton("Ok");
  clickInList(1);
  clickOnMenuItem("Delete");
}
```

Figure 7: Skeleton code to drive `NotePad`.

a similar analysis, we determine that the event of clicking list item connects screens 5a and 5c, and the event of clicking `Edit title` connects screens 5c and 5d. Furthermore, there is a back button below the display screen of the phone, which connects the current screen and the one before it. Having these connecting events, AppInk generates the event flow model as depicted in Figure 6.

After feeding the above event flow graph into M[agi]C [11] — a test input generator tool, we obtain a test case with the skeleton shown in Figure 7. This skeleton encodes the event sequence of `Add note`, `Enter text`, `Save`, `Back`, `Click list item 1`, `Edit Title`, `Ok`, `Click list item 1`, and `Delete`. This sequence covers all the activity classes in the app, thus presenting an optimal test for watermarking purpose.

3.3 Source Code Instrumentation

Having the manifest app source, together with the original app source and the generated watermarking code, AppInk uses source code instrumentation to perform the watermark embedding. The choice of source code instrumentation is reasonable since AppInk is used by app developers, who already have the app source code at hand. This also helps integrate AppInk with the well-established app development environment for Android. We note that AppInk can be supported by bytecode level instrumentation as well.

Source code instrumentation uses three steps to embed developer-provided watermarks. First, AppInk fetches all control events (including clicking button, menu and list items) from the manifest app, each of which is mapped to a single event handler in the original app (such as `onOptionsItemSelected` and `onClick`). Next, AppInk splits the watermarking code into the same number of code segments as the number of the event handlers, and generates a configuration file to record the one-to-one mapping from the watermarking code segments to the event handlers. Last, AppInk parses the source code of the original app, generates its abstract syntax tree, identifies nodes for the event handlers, and inserts the watermarking code segments into their corresponding event handlers.

After the above instrumentation, AppInk automatically builds and generates an executable app package and signs it [20], which can be used for public release. The manifest app is built into another executable package, which is not released to public. Instead it will be submitted upon request to the arbitrator for the verification purpose. All these steps are serialized with an automated script, which is seamlessly integrated into the app-building process.

3.4 Watermark Recognizer

The watermark recognizer takes both apps as input: a released app for reviewing and the associated manifest app as the driver. The core part of the recognizer (Figure 8) is an extended Dalvik virtual machine (DVM), which is the execution engine for Android app code and maintains the runtime heap. Unlike traditional watermarking tool for Java that uses the Java debug interface to access object reference information and reconstruct the watermarking object, AppInk leverages the customized DVM to fetch object reference information from the runtime heap directly. Further, with the help of the manifest app generated in Section 3.2, the watermark recognizer enables automatic watermark extraction without any user intervention, which is highly desirable for scalable handling of a large number of apps submitted to app markets.

Just like Java virtual machine executing Java code by interpreting bytecode, DVM executes Dalvik bytecode, which is the main body of Android apps. Therefore it has access to all the needed information for watermark extraction purpose. Particularly, DVM

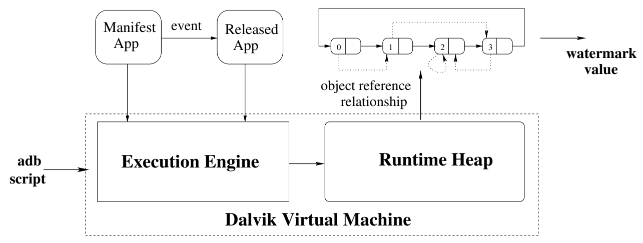


Figure 8: Work flow of watermark recognizer.

```
1 adb install -r releasedApp.apk
2 adb install -r appTest.apk
3 adb shell am instrument -w InstrumentTestRunner
4
5 pid=`adb shell ps|grep appName| awk '{print \$2}'`
6 ### Send USR2 signal to trigger GC
7 adb shell kill -10 $pid
8 adb logcat -d | grep $pid > $pid.log
9 java appink.wmGraphRecognizer $pid.log
```

Figure 9: Shell script to drive watermark extraction.

manages memory space for Android apps, and maintains relevant information for memory reclaim (garbage collection). All object reference information is maintained, so AppInk only needs to extend the garbage collector to record and export this information, among which a later module will search for the watermarking graph. The identified watermarking graph is then decoded to recover its corresponding watermark object and to verify whether it is the same as what the author claims.

More specifically, DVM uses mark-and-sweep algorithm [49] to execute the task of garbage collection, which scans all allocated objects and their member fields (a reference relationship forms between an object and its member fields) and determines if any object is not needed any more and thus can be reclaimed. We develop a module in DVM to record these object reference relationships in the scanning phase and export them into a log file. From these reference relationships, a reverse process of that in the watermarking code generation (cf. Section 3.1) is applied to recover the watermark value.

Towards automatic operations, we create a shell script based on Android debug bridge [18] to link all these steps, as shown in Figure 9. The script first installs both apps – the released app and the manifest app (lines 1 and 2), and then starts the manifest app through the instrumentation command of Android activity manager (line 3). This will feed the event sequence to the released app in a specified order. The command at line 5 gets the process identifier of the running DVM, and line 7 sends a `SIGUSR2` signal to trigger the object reference recording module inside the extended DVM. The commands at lines 8 and 9 fetch these recorded messages, search for reference relationship pattern among them, and try to extract the embedded watermark.

4. IMPLEMENTATION

We have implemented an AppInk prototype on Ubuntu 10.04. The watermarking code generation component is implemented in Java, which accepts an integer or a string as input and outputs its corresponding watermarking code. In the manifest app generation component, the parsing of Java source files is based on ANTLR [40] – a language parser generator. More concretely, we input a Java language grammar [15] into ANTLR, which generates a Java AST (abstract syntax tree) parser. By iterating through the AST, AppInk can locate the nodes for all event handlers, and identify the connecting events for different UI states. The parsing of `AndroidManifest.xml`, UI layouts, and resource files is written in Python. Another Python script glues the output results from these parsing modules, generates the event flow graph, feeds it to the test case generator named M[agi]C, and picks up the test case which has the largest coverage of watermarking code.

The source code instrumentation component includes three steps. It first parses the manifest app source to identify all these event handlers to insert the watermarking code, and then splits the watermarking code into segments with the same number as that of events to be delivered. Last it extends the Java source parser generated in the manifest app generation component to insert the watermarking code segments into the execution path for their corresponding event handlers. To automate the watermark extraction on the arbitrating side, this component also generates the shell script to drive watermark extraction (as presented in Figure 9). Basically, this script only needs relevant information for a released app and its manifest app, which is readily available after the completion of the first three components.

Having this watermark extraction script at hand, the watermark recognizer needs two modules to achieve the final watermark recognition task. The first module implements the extended DVM to record and export object reference information when the app receives a `SIGUSR2` signal. This is achieved by modifying the garbage collector code (in C language) in DVM and rebuilding the Android open source project. The second module is written in Java, which searches through these object reference relationships to match any potential watermarking graph, and decodes the graph to recover the corresponding watermark value.

5. ANALYSIS AND EVALUATION

While AppInk aims to embed strong ownership verification mechanism into Android apps, attackers always strive to defeat the protections in any way they can think of. In this section, we first analyze the robustness of AppInk against three common attacks toward watermarking, namely distortive, subtractive, and additive attacks. We then evaluate it against two open source repackaging tools to demonstrate its effectiveness. Finally we evaluate the runtime performance overhead for watermarked apps.

5.1 Robustness Analysis

AppInk adopts dynamic graph based watermarking as its key technique to defend against app repackaging. Therefore its robustness depends heavily on that of dynamic graph watermarking, which is highly resistant against distortive attacks, subtractive attacks, and additive attacks according to our analysis.

Distortive attacks: This type of attacks applies semantic-preserving transformations on target apps, trying to make it hard or impossible to extract original watermarks from the modified apps. Many static watermarking mechanisms are highly susceptible to distortive attacks since they leverage the code or data syntax to encode the watermark, which is very sensitive to semantic-preserving transformations. Dynamic watermarking, however, never depends on any syntax structure in application code, but instead encodes watermark object into the execution state of the application. Furthermore, the semantic of runtime graph data structures is usually hard to analyze without executing it in real environment, because of the inherent difficulty in analyzing point-to relationships [14,45]. These factors make it very hard for any static transformation to change these graph structures without changing the application semantics. For these reasons, most semantic-preserving transformations cannot affect the execution states of apps, and theoretically dynamic graph watermarking is resistant to distortive attacks. To further confirm AppInk's robustness in this aspect, we evaluate AppInk against a series of semantic-preserving transformations available in two open source tools [1], and report our results later in this section.

Subtractive attacks: This type of attacks tries to remove watermarking relevant code segments in an application, and usually needs manual analysis to identify the location of these code segments in the first place. The dynamic graph based watermarking mechanism adopted in AppInk makes it relatively easier to defend against subtractive attacks. First, since the arbitrators are in general trustworthy, we can assume the manifest app is kept as secret. So although the watermarking mechanism used in AppInk can become public knowledge, the secrecy of manifest app provides one layer of protection for the watermarking code segments. Second, the data structures used in AppInk are commonly used in normal Java applications, which makes it hard to separate these watermarking code segments from other functional code.

What is more, we can also leverage the inherent difficulty of alias analysis [14, 45] to add another layer of protection against subtractive attacks. Since the runtime graph data structures in AppInk have reference relationships among themselves, an app developer can easily know the correct reference relationships among the inserted graphic nodes. Instead, without this pre-knowledge, attackers have great difficulty in identifying these reference relationships through reverse engineering. Therefore we can create bogus dependency relationships between the original code and the newly inserted code [7]. The attempt to remove or modify the watermarking graph code segments will have high probability to damage the original application logic, making it useless after repackaging.

Additive attacks: This type of attacks tries to add another watermark on a watermarked app, with the assumption that attackers somehow understand and implement the same watermark embedding algorithm as presented in AppInk. In the defense model applied in centralized app markets (cf. Section 1), there is an inherent timing gap between the submission of the original app and the repackaged app. That is, the original author always submits her AppInk-protected app before an attacker succeeds in executing an additive attack on that app. Therefore when the attacker submits his repackaged app to the app market, the app market can detect that there is an earlier app which has the same functionality but has a different watermark extracted. The operator can then launch another watermark extracting session on the app under review, using the manifest app for the earlier app as the watermark extraction driver. If the same watermark is extracted as the earlier published app, it is derived that the second is a repackaged one. However, this does not prevent the attacker from downloading an app from one store and publishing in another one where the original app has not been published yet.

In the postmortem arbitrary model, when an app author suspects that one app is a repackaged version of her own, she can apply the watermark recognizer with her own manifest app on the suspected app. If the extracted watermark is identical with the one from her own app, she can submit this as evidence to prove that the suspected app is a repackaged version of her own one. In case that an attacker can somehow embed his own watermark into the AppInk-protected app and generate his own manifest app, two watermarks can be extracted by using their corresponding manifest apps. Under this confusion, the original author can present another evidence to show that her watermark is original, and the other is additional. The evidence is that her manifest app can extract the same watermark from both her original app and the repackaged app, but the attacker's manifest app can only extract his own watermark from the repackaged app.

[1]They can be used for app repackaging as well.

Next we evaluate the AppInk's robustness against real repackaging tools. With our best effort we do not find any available tool for subtractive and additive attacks. Therefore our evaluation is for distortive attacks only.

5.2 Evaluation with Repackaging Tools

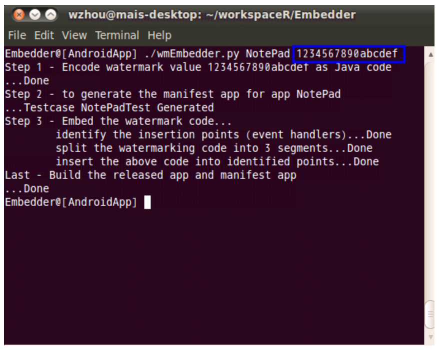
We evaluate AppInk against two open-source tools, which apply semantic-preserving transformations on Android app code and therefore can simulate the aforementioned distortive attacks. We have five Android apps under evaluation, one named `AndroidCalculator` from Robotium, and the other four from Android SDK samples (including `ContactManager`, `NotePad`, `HoneycombGallery`, and `SearchableDictionary`). We first apply AppInk to embed watermarks on these apps, and then apply the available transformations present in the above tools to the watermarked apps. Last we feed the modified apps to the AppInk watermark recognizer to see if the originally embedded watermarks can be extracted.

ADAM: We first evaluate AppInk against an automatic Android app repackaging tool named ADAM [51], which operates on Android apps (`.apk` files) directly and automatically repackages apps with different code transformation techniques. Figure 10 shows the above three-step evaluation process for the app named `NotePad`. The first snapshot (Figure 10a) shows the embedding session on `NotePad.apk`, with a string of `1234567890abcdef` as the watermark value. It clearly demonstrates the working process of the three AppInk components.

The second snapshot (Figure 10b) shows the repackaging session, where we apply seven semantic-preserving transformations from ADAM on `NotePad.apk`. Among these transformations, three do not modify the app code but change other phases in the app packaging process. For example, *resign* transformation disassembles the app and re-signs the app using attacker's signing key, *rebuild* transformation disassembles the app and re-assembles the components into a new one with the open-source tool named apktool [44], and *zipalign* transformation realigns the locations of different data in the app package in a different and pre-determined way. Four other techniques apply various code obfuscation transformations on the app code, including defunct code insertion, identifier renaming (include packages, classes, methods, and fields), control flow obfuscation, and string encryption. To conduct the evaluation, we apply these seven transformations on the watermarked code with a bash script, and output the repackaged apps into a directory.

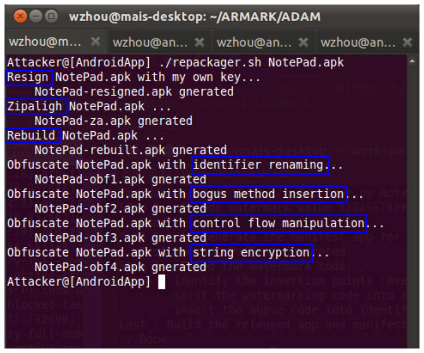
The third snapshot (Figure 10c) shows the recognizing session. We create a script to feed each of these seven repackaged apps into the watermark recognizer, and check the extracted watermark value. The first attempt shows that AppInk recognizes the watermark value of `1234567890abcdef` correctly from six of these repackaged apps. The app repackaged with *identifier renaming* obfuscation fails the first attempt. Further analysis shows that ADAM incorrectly renames one Android API method, which results in the incorrect execution of the repackaged app. After fixing this bug in ADAM, AppInk recognizes the correct watermark from all these repackaged apps (Figure 10c shows the result after fixing the ADAM bug).

Proguard: Our second evaluation is against a popular Android app obfuscation tool named Proguard [29]. Different from ADAM


wzhou@mais-desktop: ~/workspaceR/Embedder
File Edit View Terminal Help

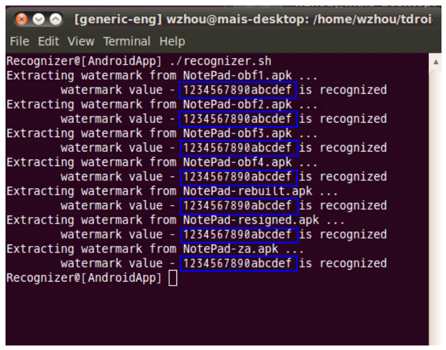
```
Embedder@[AndroidApp] ./wmEmbedder.py NotePad 1234567890abcdef
Step 1 - Encode watermark value 1234567890abcdef as Java code
...Done
Step 2 - to generate the manifest app for app NotePad
...Testcase NotePadTest Generated
Step 3 - Embed the watermark code...
        identify the insertion points (event handlers)...Done
        split the watermarking code into 3 segments...Done
        insert the above code into identified points...Done
Last - Build the released app and manifest app
...Done
Embedder@[AndroidApp]
```

(a) Snapshot of AppInk watermark embedding



```
Attacker@[AndroidApp] ./repackager.sh NotePad.apk
Resign NotePad.apk with my own key...
    NotePad-resigned.apk gnerated
Zipaligh NotePad.apk ...
    NotePad-za.apk gnerated
Rebuild NotePad.apk ...
    NotePad-rebuilt.apk gnerated
Obfuscate NotePad.apk with identifier renaming...
    NotePad-obf1.apk gnerated
Obfuscate NotePad.apk with bogus method insertion...
    NotePad-obf2.apk gnerated
Obfuscate NotePad.apk with control flow manipulation...
    NotePad-obf3.apk gnerated
Obfuscate NotePad.apk with string encryption...
    NotePad-obf4.apk gnerated
Attacker@[AndroidApp]
```

(b) Snapshot of ADAM repackaging


[generic-eng] wzhou@mais-desktop: /home/wzhou/tdroi
File Edit View Terminal Help

```
Recognizer@[AndroidApp] ./recognizer.sh
Extracting watermark from NotePad-obf1.apk ...
        watermark value - 1234567890abcdef is recognized
Extracting watermark from NotePad-obf2.apk ...
        watermark value - 1234567890abcdef is recognized
Extracting watermark from NotePad-obf3.apk ...
        watermark value - 1234567890abcdef is recognized
Extracting watermark from NotePad-obf4.apk ...
        watermark value - 1234567890abcdef is recognized
Extracting watermark from NotePad-rebuilt.apk ...
        watermark value - 1234567890abcdef is recognized
Extracting watermark from NotePad-resigned.apk ...
        watermark value - 1234567890abcdef is recognized
Extracting watermark from NotePad-za.apk ...
        watermark value - 1234567890abcdef is recognized
Recognizer@[AndroidApp]
```

(c) Snapshot of AppInk watermark recognizing

Figure 10: Snapshots for watermark embedding, app repackaging, and watermark recognizing.

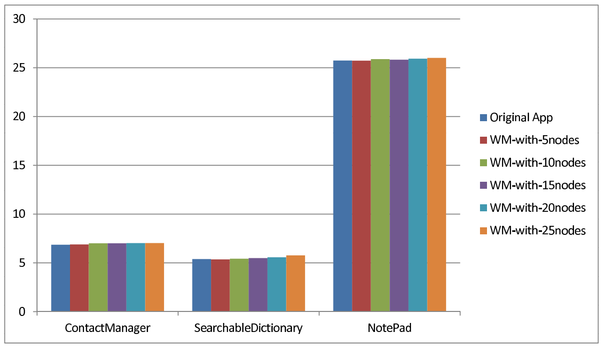


Figure 11: Execution time of watermarked app.

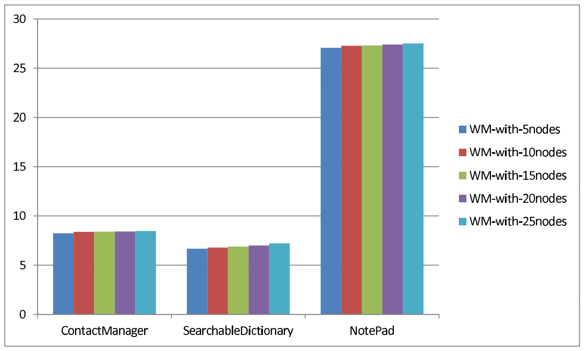


Figure 12: Extraction time of watermarked app.

which works directly on final `.apk` files, Proguard operates on class files generated in Android app building process. To conduct this evaluation, we modify the watermarking embedding process as presented in Section 3, by adding Proguard obfuscation as a post-compilation action into the app building process [2]. With this extra action, the generated class files are optimized and obfuscated first, and then packaged into the final released apps. Last we feed these obfuscated apps into the AppInk recognizer. Our experiments show that AppInk recognizer can extract the correct watermarks embedded into all of these transformed apps successfully.

These two sets of evaluations demonstrate that AppInk has high resistance against currently available repackaging and transformation tools, and thus is very robust against distortive attacks.

5.3 Performance Evaluation

We conduct performance evaluation in two aspects with three different Android apps (`ContactManager`, `SearchableDictionary`, and `NotePad`). As the size of the permutation graph is the main factor to decide the extra code size and thus the final performance, we watermark each of these three apps with five different watermark values, which encode permutation graphs with sizes of 5, 10, 15, 20, and 25, respectively. These values can encode a number from `24` to $1.6*10^{25}$. Our experiments show that even the longest watermark value only introduces trivial performance overhead.

First, we measure how much extra time is required to execute these watermarked apps, which affects the user experience of an end mobile user when running these AppInk protected apps on their devices. To reduce the undecidability of human input and also exercise all these watermarking code, we use the manifest apps to drive these apps in a normal Android emulator. Figure 11 shows the times in seconds to finish each watermarked app. Please note that for each app, the first column shows the time to execute the original (un-watermarked) app. The small differences between the five watermarked apps and the original apps show that AppInk causes very small runtime overhead (2.4% at most in our evaluation).

Second, we measure how much time is required to recognize a watermark. This is the time that an arbitrating party needs to verify an app's originality. For that purpose, we feed these watermarked apps into the extended Android emulator as presented in Section 3.4, and measure how much time elapses when AppInk recognizes the watermark values. Figure 12 shows the measurement results in seconds. As shown, it takes 7 seconds to 28 seconds to verify these apps. Additionally, a longer watermark value in general requires more time to be recognized, but the difference is small. Compared with the data in Figure 11, we find that most time for watermark recognition is spent on the app execution itself. The time differences for the same watermarked apps in these two figures show the execution time dedicated to the watermark extraction, is from 1.3 seconds to 1.6 seconds. With this scale of time requirement, AppInk's watermark recognition can be deployed at current largest app market to handle thousands of app submissions every day.

[2]Concretely, we add a new Proguard configuration file and Proguard action into one ant [20] building script.

6. DISCUSSION

Our prototype implementation and evaluation have demonstrated the effectiveness of AppInk for preventing the propagation of repackaged Android apps and deployable capability for general Android app development practice. In this section, we examine possible limitations in the current prototype and discuss future improvements.

First, AppInk uses a conservative model-based test generation algorithm to generate manifest apps, which may not be the optimum for watermarking purpose. One possible enhancement is to investigate the latest automatic test case generation methods that have been studied by researchers in software engineering field. For example, the concolic execution based and GUI ripping based techniques [1, 2, 33] are actively investigated in software engineering community to enable automatic generation of high-coverage test inputs for Android apps. We plan to study these methods to see if they can be leveraged by AppInk for watermarking purpose.

Second, our AppInk prototype supports user input events only, which are the primary driver for app functionality, but ignores possible discrete system events, such as short messages received, incoming phone calls, and various sensor events. We plan to study the working mechanisms of all these events and explore ways to incorporate them into AppInk.

7. RELATED WORK

Software watermarking: Static watermarking embeds watermarks into the code or data of applications [35, 37], which usually involves syntax transformation and is vulnerable to semantic-preserving transformations. A variety of dynamic watermarking mechanisms have been proposed to overcome these attacks, including graph based [6, 39], thread based [36], and path based watermarking [4]. AppInk does not claim any novel contribution in this aspect. We instead leverage existing dynamic graph based watermark to improve Android app's capability in preventing and defending against common app repackaging attacks.

Java software protection: Android apps are mainly written in Java. Due to its high-level expressiveness, Java code is relatively easier to be decompiled and reversed [43] than native code. To protect software written in Java, various solutions are pursued since its inception. One popular solution is to apply different levels of obfuscation to Java code, such as code or data layout obfuscation, control flow obfuscation, and string encryption [3, 9, 12]. Watermarking is also used to prove the ownership of Java code and to discourage Java software piracy [6,35]. SandMark [5] is one popular research platform to study how well different obfuscation and watermarking mechanisms work in protecting Java software.

Android app protection: To protect Android apps from piracy and foster the healthy development of Android app economy, Google has introduced several mechanisms. For example, Google recommends developers to leverage ProGuard [29] to optimize and obfuscate apps. Google also provides licensing verification library [17] to query a server to verify if an app running on a mobile device has been properly downloaded from Google's app market. There are also attempts from other parties in this aspect. For example, Amazon and Verizon have introduced their own digital right management solutions for Android apps available in their app markets [41]. These mechanisms can increase the difficulty of reverse engineering Android apps, but are not strong enough to deter determined attackers from repackaging through more laborious manual analysis. For example, open source tools are available to automatically crack these protections [30–32].

More recently, there are a series of work studying the app repackaging problem in Android platform [8, 16, 42, 52, 53]. Different from the dynamic watermarking mechanism proposed by AppInk, all these systems attack the app repackaging problem from the point of view of measuring the apps similarity. DroidMOSS [53] uses fuzzy hashing to speed up the pair-wise similarity comparison at the opcode level. Potharaju et al. [42] compares each app pair using different syntactic fingerprinting schemes, and can handle different levels of obfuscation used by the attacker. Juxtapp [16] collects static code features and represents them as bit vectors to improve the efficiency of pairwise comparison. It also supports incremental update and distributed analysis. DNADroid [8] uses program dependency graph (PDG) to characterize Android app and compares PDGs between methods in app pairs, showing resistance to several evasion techniques. By proposing a new distance metric design and an associated nearest neighbor search algorithm, PiggyApp [52] overcomes the scalability limitation from the pairwise comparison as presented in previous systems, and achieves a better scalability with $O(n * logn)$ complexity. None of these similarity based methods need modification to the released apps, but their results can only indicate possible repackaging, the final decision of which is left for human review. AppInk, although requiring some modification from the app developer side, does not have this drawback. The successful extraction of the author's watermark from the suspected app clearly derives the fact that it is repackaged. What is more, AppInk automates the work at both the app developer and arbitrator sides, enables the online and realtime detection of repackaged apps, and thus provides stronger deterrence towards app repackaging threat.

8. CONCLUSION

App repackaging is a serious threat to Android ecosystem including app developers, app store operators, and end users. To prevent the propagation of unauthorized repackaged apps, we propose to adopt a dynamic graph based watermarking mechanism and discusses two scenarios where this mechanism is mostly useful. To make the watermarking mechanism readily integratable into current app development practice and conveniently deployable by relevant parties, we introduce the concept of manifest app, which is a companion app for an Android app under protection. We then design and implement a tool named AppInk to generate manifest apps, embed watermarks to apps, and extract watermarks without any user intervention. Our robustness analysis and practical evaluation against currently available open source tools demonstrate that AppInk is effective in defending against common automatic repackaging threats while introducing trivial performance overhead.

Acknowledgements The authors would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. We also want to thank Yajin Zhou, Michael Grace, Chiachih Wu, Minh Q. Tran, Lei Wu and Kunal Patel for the helpful discussion. This work was supported in part by the US National Science Foundation (NSF) under Grants 0855297, 0855036, 0910767, and 0952640. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

9. REFERENCES

[1] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using GUI Ripping for Automated Testing of Android Applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM.

[2] Saswat Anand, Mayur Naik, Hongseok Yang, and Mary Jean Harrold. Automated Concolic Testing of Smartphone Apps.

In *ACM International Symposium on Foundations of Software Engineering*, FSE, 2012.

[3] Jien-Tsai Chan and Wuu Yang. Advanced Obfuscation Techniques for Java Bytecode. *J. Syst. Softw.*, 71(1-2):1–10, April 2004.

[4] C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn, and M. Stepp. Dynamic Path-based Software Watermarking. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 107–118, New York, NY, USA, 2004. ACM.

[5] Christian Collberg, Ginger Myles, and Andrew Huntwork. SandMark - A Tool for Software Protection Research. IEEE Security and Privacy, Vol. 1, Num. 4, July/August 2003.

[6] Christian Collberg and Clark Thomborson. Software Watermarking: Models and Dynamic Embeddings. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 311–324, New York, NY, USA, 1999. ACM.

[7] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 184–196, New York, NY, USA, 1998. ACM.

[8] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the Clones: Detecting Cloned Applications on Android Markets. In *17th European Symposium on Research in Computer Security*, ESORICS 2012, September 2012.

[9] Mila Dalla Preda and Roberto Giacobazzi. Control Code Obfuscation by Abstract Interpretation. In *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, SEFM '05, pages 301–310, Washington, DC, USA, 2005. IEEE Computer Society.

[10] DalvikVM.com. Dalvik Virtual Machine - Brief Overview of the Dalvik Virtual Machine and Its Insights. `http://www.dalvikvm.com/`. Online; accessed at Nov 30, 2012.

[11] FITTEST. M[agi]C Tool: M*C Test Generation Tool. `http://selab.fbk.eu/magic/`. Online; accessed at Dec 1, 2012.

[12] Kazuhide Fukushima and Kouichi Sakurai. A Software Fingerprinting Scheme for Java Using Classfiles Obfuscation. In Ki-Joon Chae and Moti Yung, editors, *Information Security Applications*, volume 2908 of *Lecture Notes in Computer Science*, pages 303–316. Springer Berlin Heidelberg, 2004.

[13] Dan Galpin and Trevor Johns. Evading Pirates and Stopping Vampires Using License Verification Library, In-App Billing, and App Engine. `http://www.google.com/events/io/2011/sessions/evading-pirates-and-stopping-vampires-using-license-verification-library-in-app-billing-and-app-engine.html`. Online; accessed at Nov 30, 2012.

[14] Rakesh Ghiya and Laurie J. Hendren. Is It a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-directed Pointers in C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 1–15, New York, NY, USA, 1996. ACM.

[15] Dieter Habelitz. Java 1.5 Grammar for ANTLR v3 That Builds Trees. `http://www.antlr.org/grammar/1207932239307/Java1_5Grammars`. Online; accessed at Nov 30, 2012.

[16] Steve Hanna, Ling Huang, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications. In *9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, DIMVA 2012, July 2012.

[17] Google Inc. Android Application Licensing. `http://developer.android.com/guide/google/play/licensing/index.html`. Online; accessed at Nov 30, 2012.

[18] Google Inc. Android Debug Bridge. `http://developer.android.com/tools/help/adb.html`. Online; accessed at Nov 30, 2012.

[19] Google Inc. Android Emulator. `http://developer.android.com/tools/help/emulator.html`. Online; accessed at Nov 30, 2012.

[20] Google Inc. Building and Running Android App from Command Line. `http://developer.android.com/tools/building/building-cmdline.html`. Online; accessed at Nov 30, 2012.

[21] Google Inc. Testing Fundamental | Android Developers. `http://developer.android.com/tools/testing/testing_android.html`. Online; accessed at Nov 30, 2012.

[22] Google Inc. UI/Application Exerciser Monkey. `http://developer.android.com/tools/help/monkey.html`. Online; accessed at Nov 30, 2012.

[23] Lookout Inc. App Genome Report: February 2011. `https://www.mylookout.com/appgenome/`. Online; accessed at Nov 30, 2012.

[24] Oracle Inc. Java Debug Interface. `http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jdi/`. Online; accessed at Nov 30, 2012.

[25] Saikoa Inc. A Specialized Optimizer and Obfuscator for Android. `http://www.saikoa.com/dexguard`. Online; accessed at Nov 30, 2012.

[26] Yiming Jing, Gail-Joon Ahn, and Hongxin Hu. Model-Based Conformance Testing for Android. In Goichiro Hanaoka and Toshihiro Yamauchi, editors, *Advances in Information and Computer Security*, volume 7631 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin Heidelberg, 2012.

[27] Seolwoo Joo and Changyeon Hwang. Mobile Banking Vulnerability: Android Repackaging Threat. Virus Bulletin, May 2012.

[28] Donald Knuth. *Fundamental Algorithms, Volume 1 of The Art of Computer Programming, Third Edition*. Addison-Wesley, 1997.

[29] Eric Lafortune. ProGuard. `http://proguard.sourceforge.net/`. Online; accessed at Nov 30, 2012.

[30] Lohan+. AntiLVL - Android License Verification Library Subversion. `http://androidcracking.blogspot.com/p/antilvl_01.html`. Online; accessed at Nov 30, 2012.

[31] Lohan+. Cracking Amazon DRM. `http://androidcracking.blogspot.com/2011/04/cracking-amazon-drm.html`. Online; accessed at Nov 30, 2012.

[32] Lohan+. Cracking Verizon's V Cast Apps DRM. `http://androidcracking.blogspot.com/2011/06/`

cracking-verizons-v-cast-apps-drm.html. Online; accessed at Nov 1, 2012.

[33] Riyadh Mahmood, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam Malek, and Angelos Stavrou. A Whitebox Approach for Automated Security Testing of Android Applications on the Cloud. In *Proceedings of the 7th International Workshop on Automation of Software Test*, AST 2012, 2012.

[34] Atif M. Memon. An event-flow model of GUI-based applications for testing: Research Articles. *Softw. Test. Verif. Reliab.*, 17(3):137–157, September 2007.

[35] Akito Monden, Hajimu Iida, Ken-ichi Matsumoto, Koji Torii, and Katsuro Inoue. A Practical Method for Watermarking Java Programs. In *24th International Computer Software and Applications Conference*, COMPSAC '00, pages 191–197, Washington, DC, USA, 2000. IEEE Computer Society.

[36] Jasvir Nagra and Clark Thomborson. Threading Software Watermarks. In *Proceedings of the 6th International Conference on Information Hiding*, IH'04, pages 208–223, Berlin, Heidelberg, 2004. Springer-Verlag.

[37] Jasvir Nagra, Clark Thomborson, and Christian Collberg. A Functional Taxonomy for Software Watermarking. In *Proceedings of the Twenty-fifth Australasian Conference on Computer Science - Volume 4*, ACSC '02, pages 177–186, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[38] Cu D. Nguyen, Alessandro Marchetto, and Paolo Tonella. Combining Model-based and Combinatorial Testing for Effective Test Case Generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 100–110, New York, NY, USA, 2012. ACM.

[39] J. Palsberg, S. Krishnaswamy, Minseok Kwon, D. Ma, Qiuyun Shao, and Y. Zhang. Experience with Software Watermarking. In *Proceedings of the 16th Annual Computer Security Applications Conference*, ACSAC '00, pages 308–, Washington, DC, USA, 2000. IEEE Computer Society.

[40] Terence Parr. ANTLR - ANother Tool for Language Recognition. http://www.antlr.org/. Online; accessed at Nov 30, 2012.

[41] Android Police. [Updated: Amazon Provides Clarifications] Amazon App Store's DRM To Be More Restrictive Than Google's? http://www.androidpolice.com/2011/03/07/amazon-app-stores-drm-to-be-more-restrictive-than-googles/. Online; accessed at Nov 30, 2012.

[42] Rahul Potharaju, Andrew Newell, Cristina Nita-Rotaru, and Xiangyu Zhang. Plagiarizing Smartphone Applications: Attack Strategies and Defense Techniques. In *Proceedings of the 4th International Conference on Engineering Secure Software and Systems*, ESSoS'12, pages 106–120, Berlin, Heidelberg, 2012. Springer-Verlag.

[43] Todd A. Proebsting and Scott A. Watterson. Krakatoa: Decompilation in Java (Does Bytecode Reveal Source?). In *In Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, 1997.

[44] Google Code Project. Android-apktool - Tool for Reengineering Android apk Files. http://code.google.com/p/android-apktool/. Online; accessed at Nov 30, 2012.

[45] G. Ramalingam. The Undecidability of Aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994.

[46] Renas Reda. Robotium. http://code.google.com/p/robotium/. Online; accessed at Nov 30, 2012.

[47] Tommi Takala, Mika Katara, and Julian Harty. Experiences of System-Level Model-Based GUI Testing of an Android Application. In *Proceedings of the 4th IEEE International Conference on Software Testing, Verification, and Validation (ICST 2011)*, pages 377–386, Los Alamitos, CA, USA, March 2011. IEEE Computer Society.

[48] Arxan Technologies. State of Security in the App Economy: Mobile Apps Under Attack. http://www.arxan.com/assets/1/7/state-of-security-app-economy.pdf, 2012.

[49] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Proceedings of the International Workshop on Memory Management*, IWMM '95, pages 1–116, London, UK, UK, 1995. Springer-Verlag.

[50] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 27–27, Berkeley, CA, USA, 2012. USENIX Association.

[51] Min Zheng, Patrick P.C. Lee, and John C.S. Lui. ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-Virus Systems . In *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2012.

[52] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, Scalable Detection of Piggybacked Mobile Applications. In *Proceedings of the 3nd ACM Conference on Data and Application Security and Privacy*, CODASPY '13, February 2013.

[53] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. DroidMOSS: Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy*, CODASPY '12, February 2012.

[54] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, Oakland 2012, May 2012.