Tackling runtime-based obfuscation in Android with TIRO

Michelle Y. Wong and David Lie
University of Toronto

Abstract

Obfuscation is used in malware to hide malicious activity from manual or automatic program analysis. On the Android platform, malware has had a history of using obfuscation techniques such as Java reflection, code packing and value encryption. However, more recent malware has turned to employing obfuscation that subverts the integrity of the Android runtime (ART or Dalvik), a technique we call *runtime-based obfuscation*. Once subverted, the runtime no longer follows the normally expected rules of code execution and method invocation, raising the difficulty of deobfuscating and analyzing malware that use these techniques.

In this work, we propose TIRO, a deobfuscation framework for Android using an approach of **T**arget-**I**nstrument-**R**un-**O**bserve. TIRO provides a unified framework that can deobfuscate malware that use a combination of traditional obfuscation and newer runtime-based obfuscation techniques. We evaluate and use TIRO on a dataset of modern Android malware samples and find that TIRO can automatically detect and reverse language-based and runtime-based obfuscation. We also evaluate TIRO on a corpus of 2000 malware samples from VirusTotal and find that runtime-based obfuscation techniques are present in 80% of the samples, demonstrating that runtime-based obfuscation is a significant tool employed by Android malware authors today.

1 Introduction

There are currently an estimated 2.8 million applications on the Google Play store, with thousands being added and many more existing applications being updated daily. A large market with many users naturally draws attackers who create and distribute malicious applications (i.e. malware) for fun and profit. While dynamic analyses [10, 27, 28, 34] can be used to detect and analyze malware, anti-malware tools often use static analysis as well for efficiency and greater code coverage [1, 2, 12]. As a result, malware authors have increasingly turned to obfuscation to hide their actions and confuse both static and dynamic analysis tools. The presence of obfuscation does not indicate malicious intent in and of itself, as many legitimate applications employ code obfuscation to protect intellectual property. However, because of its prevalence among malware, it is crucial that malware analyzers have the ability to deobfuscate Android applications in order to determine if an application is indeed malicious or not.

There exist a variety of obfuscation techniques on the Android platform. Many common techniques, such as Java reflection, value encryption, dynamically decrypting and loading code, and calling native methods have been identified and discussed in the literature [11,22,26]. These techniques have a common property in that they exploit facilities provided by the Java programming language, which is the main development language for Android applications, and thus we call these *language-based obfuscation* techniques. In contrast, malware authors may eschew Java and execute entirely in native code, obfuscating with techniques seen in x86 malware [3, 8, 17, 20, 24]. We call this technique *full-native code obfuscation*.

In this paper, we identify a third option—obfuscation techniques that subvert ART, the Android RunTime, which we call *runtime-based obfuscation* techniques. These techniques subtly alter the way method invocations are resolved and code is executed. Runtime-based obfuscation has advantages over both language-based and full-native code obfuscation. While language-based obfuscation techniques have to occur immediately before the obfuscated code is called, runtime-based obfuscation techniques can occur in one place and alter code execution in a seemingly unrelated part of the application. This significantly raises the difficulty of deobfuscating code, as code execution no longer follows expected conventions and analysis can no longer be performed piece-

meal on an application, but must examine the entire application as a whole. Compared to full-native code obfuscation, runtime-based obfuscation allows a malware developer to still use the convenient Java-based API libraries provided by the framework. Malware that use native code obfuscation will either have to use language- or runtime-based obfuscation to hide its Android API use, or risk compatibility loss if it tries to access APIs directly. Our study of obfuscated malware suggests that authors almost universally employ language- and runtime-based methods to hide their use of Android APIs in Java.

To study both language- and runtime-based obfuscation in Android malware, we propose TIRO, a tool that can handle both types of obfuscation techniques within a single deobfuscation framework. TIRO is an acronym for the automated approach taken to defeat obfuscation — **T**arget-**I**nstrument-**R**un-**O**bserve. TIRO first analyzes the application code to target locations where obfuscation may occur, and applies instrumentation either in the application or runtime to monitor for obfuscation and collect run-time information. TIRO then runs the application with specially generated inputs that will trigger the instrumentation. Finally, TIRO observes the results of running the instrumented application to determine whether obfuscation occurred and if so, produce the deobfuscated code. TIRO performs these steps iteratively until it can no longer detect any new obfuscation. This iterative mechanism enables it to work on a variety of obfuscated applications and techniques.

TIRO's hybrid static-dynamic design is rooted in an integration with IntelliDroid [31], which implements targeted dynamic execution for Android applications. TIRO uses this targeting to drive its dynamic analysis to locations of obfuscation, saving it from having to execute unrelated parts of the application. However, IntelliDroid uses static analysis and is susceptible to language-based and runtime-based obfuscation, which can make its analysis incomplete. By using an iterative design that feeds dynamic information back into static analysis for deobfuscation, TIRO can incrementally increase the completeness of this targeting, which further improves its deobfuscation capabilities. In this synergistic combination, IntelliDroid improves TIRO's efficiency by targeting its dynamic analysis toward obfuscation code and TIRO improves IntelliDroid's completeness by incorporating deobfuscated information back into its targeting. Successive iterations allow each to refine the results of the other.

We make three main contributions in this paper:

1. We identify and describe a family of runtime-based obfuscation techniques in ART, including DEX file hooking, class modification, ArtMethod hooking, method entry-point hooking and instruction hooking/overwriting.
2. We present the design and implementation of TIRO, a framework for Android-based deobfuscation that can handle both language-based and runtime-based obfuscation techniques.
3. We evaluate TIRO on a corpus of 34 modern malware samples provided by the Android Malware team at Google. We also run TIRO on 2000 obfuscated malware samples downloaded from VirusTotal to measure the prevalence of various runtime-based obfuscation techniques in the wild and find that 80% use a form of runtime-based obfuscation.

We begin by providing background on the Android runtime and classical language-based obfuscation techniques in Section 2. We then introduce and explain runtime-based obfuscation techniques in Section 3. We present TIRO, a deobfuscation framework that can handle both language- and runtime-based obfuscation in Section 4 and provide implementation details in Section 5. We present an analysis of obfuscated Android malware in Section 6 and show how TIRO can deobfuscate these applications. We analyze our findings and our limitations in Section 7. Related work is discussed in Section 8. Finally, we conclude in Section 9.

2 Background

Android applications are implemented in Java, compiled into DEX bytecode, and executed in either the Dalvik Virtual Machine or the Android Runtime (ART).[1] The Dalvik VM, used in Android versions prior to 4.4, interprets the DEX bytecode and uses just-in-time (JIT) compilation for frequently executed code segments. ART, a separate runtime introduced in Android 4.4 and set as the default in Android 5.0, adds ahead-of-time (AOT) compilation (using the `dex2oat` tool) to a DEX interpreter. Starting in Android 7.0, ART also includes profile-based smart compilation that uses a mixture of interpretation, JIT, and AOT compilation to boost application performance.

We briefly discuss traditional language-based obfuscation and full-native code obfuscation techniques:

Reflection. Java provides the ability to dynamically instantiate and invoke methods using reflection. Because the target of reflected method invocations is only known at run-time, this frustrates static analysis and can make the targets of these calls unresolvable (e.g. by using an encrypted string), thus hiding call edges and data accesses.

Value encryption. Key values and strings in an application can be encrypted so they are not visible to static analysis. When executed, code in the application decrypts

[1] https://source.android.com/devices/tech/dalvik/

the values, allowing the application to use the plain text at run-time. Value encryption is often combined with reflection to hide the names of classes or methods targeted by reflected calls.

Dynamic loading. Code located outside the main application package (APK) can be executed through dynamic code loading. This is often used in packed applications, where the hidden code is stored as an encrypted binary file within the APK package and decrypted when the application is launched. The decrypted code is stored in a temporary file and loaded into the runtime through the use of the dynamic loading APIs in the `dalvik.system.DexClassLoader` and `dalvik.system.DexFile` classes. Normally, the temporary files holding the decrypted bytecode are deleted after the loading process to further hide or obfuscate it from analysis. In some cases, the invocation to the dynamic loading API may be obfuscated by performing the invocation reflectively or in native code, using multiple layers of obfuscation to increase the difficulty of analysis.

Native methods. Java applications may use the Java Native Interface (JNI) to invoke native methods in the application. When used for obfuscation, malicious behavior and method invocations can be performed in native code. Unlike Java or DEX bytecode, native code contains no symbol information—variables are mapped to registers and many symbols are just addresses. Thus, static analysis of native code yields significantly less useful results and the inclusion of native code in an application can hide malicious activity or sensitive API invocations from an analyzer.

Full-native code obfuscation. Because Android applications can execute code natively, it would also be possible to implement an entire Android application in native code and utilize native code obfuscation techniques. Native code obfuscation has a long history on x86 desktop systems, and can be extremely resistant to analysis [3]. The primary drawback to this approach is that access to Android APIs, which can reveal the user's location and give access to various databases containing the user's contacts, calendar and browsing history, can only be reliably accessed via API stubs in the Java framework library provided by the OS. On one hand, calling APIs from Java code without language- or runtime-based obfuscation would expose the APIs calls to standard Android application analysis [2, 12]. On the other hand, calling these APIs from native code requires the application to correctly guess the Binder message format that the services on the Android system are using. Because the ecosystem of Android is very fragmented,[2] this poses a challenge for malware that wishes to avoid executing Java code. As a result, applications that use native code obfuscation still need obfuscation for Java code if they want to be able to make Android API calls reliably.

[2] https://developer.android.com/about/dashboards/index.html

3 Runtime-based obfuscation

Before we describe runtime-based obfuscation, we first describe how code is loaded and executed in the ART runtime. Figure 1 illustrates three major steps in loading and invoking code. First, A shows how DEX bytecode must be identified and loaded from disk into the runtime. Second, B is triggered when a class is instantiated by the application and shows how the corresponding bytecode within the DEX file is found and incorporated into runtime state. Finally, C shows how virtual methods are dynamically resolved via a virtual method table (vtable) and execution is directed to the target method code. We describe these steps in more detail below.

3.1 DEX file and class loading

In Stage A, DEX files are loaded from disk into memory, a process that involves instantiating Java and native objects to represent the loaded DEX file. The Java `java.lang.DexFile` object is returned to the application if it uses the `DexFile.loadDex()` API; in normal cases, this object is passed to a class loader so that ART can later load classes from the new DEX bytecode.

The class loading process, Stage B, is triggered when a class is first requested (e.g. when it is first instantiated). The class linker within ART searches the loaded DEX files (in the order of loading) until it finds a class definition entry (`class_def_item`) matching the requested class name. The associated class data is parsed from the DEX file, now loaded in memory, and a `Class` object is used to represent this class in ART. In addition, data for class members are also parsed, and `ArtField` or `ArtMethod` objects created to represent them. To handle polymorphism, a vtable is stored for each class and used to resolve virtual method invocations efficiently. The table is initially populated by pointers to `ArtMethod` instances from the superclass (i.e. inherited methods). For overridden methods, their entries in the table are replaced with pointers to the `ArtMethod` instances for the current loaded class.

3.2 Code execution

When a non-static virtual invocation is made, marked by Stage C, the target method must be resolved. The resolution begins by determining the receiver object's type, which references a `Class` object. The method specified in the invocation is used to index into the vtable of this class, thereby obtaining the target `ArtMethod` object to

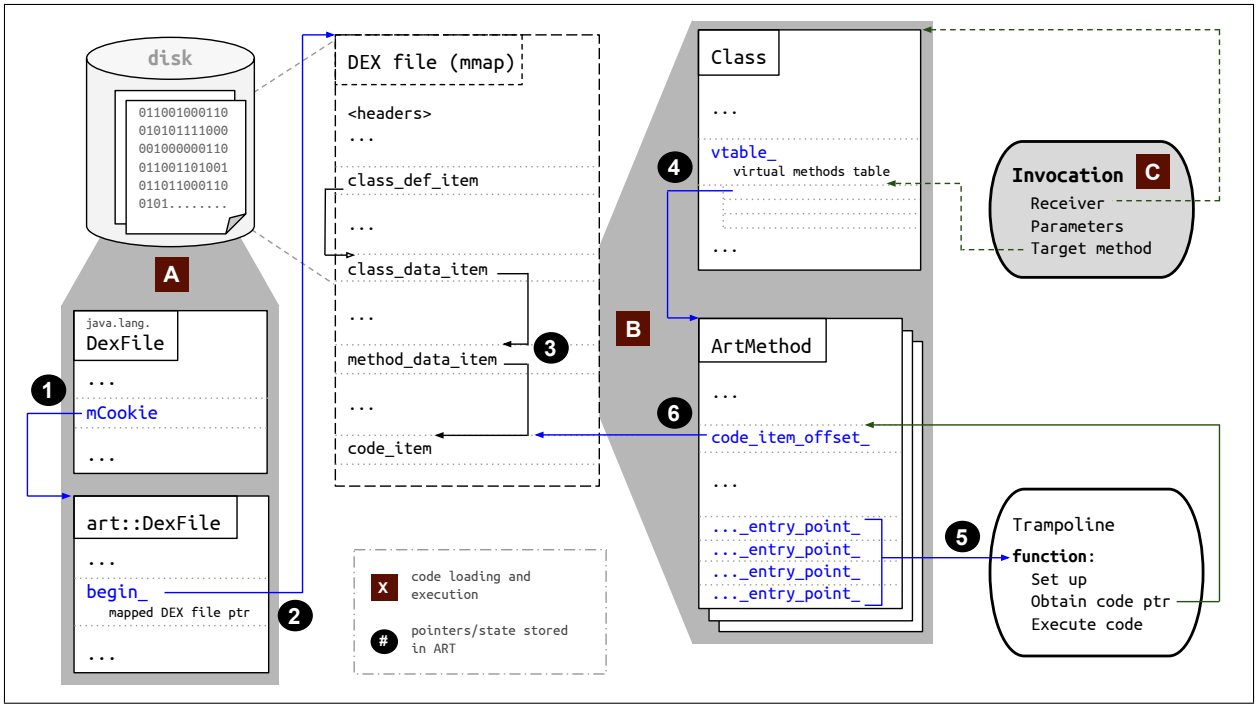

Figure 1: ART state for code loading and execution

invoke (see (4) in Figure 1). The actual invocation procedure depends on the method type (e.g. Java or native) and the current runtime environment (e.g. interpreter or compiled mode). A set of entry-points are stored with the `ArtMethod` to handle each case (see (5)); each is essentially a function pointer/trampoline that performs any necessary set-up, obtains and executes the method's DEX or OAT code, and performs clean-up. While Figure 1 shows only how the DEX code pointer is retrieved for a method (see (6)), OAT code pointers for compiled code are obtained in an analogous way.

3.3 Obfuscation techniques

Runtime-based obfuscation redirects method invocations by subverting runtime state at a number of points during the code loading and execution process outlined above. Because runtime-based obfuscation works by modifying the state of the runtime, it must acquire the addresses of the runtime objects it needs to modify, which is normally done using reflection, and modify them using native code invoked via JNI (since Java memory management would prevent code in Java from modifying ART runtime objects). In total, our analysis with TIRO has identified six different techniques used by malware to obfuscate the targets of method invocations. In Figure 1, (1) - (3) indicates runtime state that can be modified to hijack the code loading process such that the state is initialized with unexpected data (with respect to the input provided to the runtime from the application). (4) - (6) indicates runtime state that can be subverted to alter the code that a method invocation resolves to. We describe these techniques in more detail below:

(1) (2) **DEX file hooking.** When loading a DEX file, the `dalvik.system.DexFile` class is used in Java code to identify the loaded file; however, the bulk of the actual loading is performed by native code in the runtime, using a complementary native `art::DexFile` class. To reconcile the Java class with its native counterpart, the `DexFile::mCookie` Java field stores pointers to the associated native `art::DexFile` instances that represent this DEX file. When classes are loaded later, this Java field is used to access the corresponding native `art::DexFile` instance, which holds a pointer to the memory address where the DEX file has been loaded/mapped. Obfuscation techniques can use reflection to access the private `mCookie` field and redirect it to another `art::DexFile` object, switching an apparently benign DEX file with one that contains malicious code. In most cases, the malicious DEX file is loaded using non-API methods and classes within native code, or is dynamically generated in memory, further hiding its existence.

Similarly, instead of modifying the `mCookie` field, the obfuscation code can also modify the `begin_` field within the `art::DexFile` native class and redirect it to another DEX file. However, this approach can be more brittle since the obfuscation code must make assump-

tions about the location of the `begin_` field within the object.

③ **Class data overwriting.** Obfuscation code can also directly modify the contents of the memory-mapped DEX file to alter the code to be executed. DEX files follow a predetermined layout that separates class declarations, class data, field data, and method data.[3] Both the class data pointer (`class_data_item`), which determines where information for a class is stored, and method data pointer (`method_data_item`), which determines where information is stored for a method, are prime targets for such modification. Modifying the class data pointer allows the obfuscation code to replace the class definition with a different class while modifying the method definition allows the obfuscation code to change the location of the code implementing a method. This can be done en masse or in a piecemeal fashion, where each class or method is modified immediately before it is first used. We note that there are no bounds checks on the pointers, so while class and method pointers normally point to definitions and code within the DEX file, obfuscation code is free to change them to point to objects (including dynamically created ones) anywhere in the application's address space.

Class declarations (`class_def_item`) are not normally modified by obfuscation code since this top level object is often read and cached into an in-memory data structure for fast lookup. If the obfuscation code misses the small window where the DEX file is loaded but this data structure has not yet been populated, any modifications to the class declarations will not take effect in the runtime.

④ **ArtMethod hooking.** After the receiving class of an invocation is determined, the target method is found by indexing into the class's vtable. Obfuscation code can obtain a handle to a `Class` object using reflection and determine the offset at which the vtable is stored. By modifying entries in this table, the target `ArtMethod` object for an invocation can be hooked so that a different method is retrieved and executed. The target method that is actually executed must be an `ArtMethod` object, which might have been dynamically generated by the obfuscation code or loaded previously from a DEX file. In the latter case, the use of virtual method hooking is to hide the invocation and have malicious code appear to be dead. The feasibility of this type of modification for obfuscation was established in [6].

⑤ **Method entry-point hooking.** Once the target `ArtMethod` object has been determined for an invocation, the method is executed by invoking one of its entry-points, which are mere function pointers. Similar to `Class` objects, reflection via the JNI can be used to obtain the Java `Method` object and through this, the obfuscation code can determine the location of the corresponding `ArtMethod` object, which is a wrapper/abstraction around the method. By modifying and hooking the values of these entry-points, it can change the code that is executed when the method is invoked.

Although the new entry-point code can be arbitrary native code, there exists a number of method hooking libraries [18, 19, 35] that allow an application developer to specify pairs of hooked and target methods in Java. They use method entry-point hooking so that a generic look-up method is executed when the hooked methods are invoked. This look-up method determines the registered target method for a hooked method invocation and executes it.

⑥ **Instruction hooking and overwriting.** The final stage in the method invocation process is to retrieve the DEX or OAT code pointers for a method and execute the instructions; this is performed by the method's entry-points. These code pointers are stored and retrieved from the `ArtMethod` object. Instruction hooking can be achieved by modifying this pointer such that a different set of instructions is referenced and executed when the method is invoked. Alternatively, instruction overwriting can be achieved by accessing the memory referenced by this pointer and performing in-place modification of the code—this normally requires the original instruction array to be padded with NOPs (or other irrelevant instructions) to ensure sufficient room for the newly modified code. While the invocation target does not change, the obfuscation code can essentially execute a completely different method than what was first loaded into the runtime. The modification of a method's instructions can occur before or after class loading, since the runtime links directly to the instruction array in `ArtMethod` objects. It is even possible to overwrite the instructions multiple times such that a different set of instructions is executed every time the method is invoked.

4 TIRO: A hybrid iterative deobfuscator

To address language-based and runtime-based obfuscation techniques, we describe TIRO, a deobfuscator that handles both types of obfuscation. At a high level, TIRO combines static and dynamic techniques in an iterative fashion to detect and handle modern obfuscation techniques in Android applications. The input to TIRO is an APK file that might be distributed or submitted to an application marketplace. The output is a set of deobfuscated information (such as statically unresolvable run-time values, dynamically loaded code, etc.) that can be passed into existing security analysis tools to increase

[3] https://source.android.com/devices/tech/dalvik/dex-format

their coverage, or used by a human analyst to better understand the behaviors of an Android application.

The main design of TIRO is an iterative loop that incrementally deobfuscates applications in four steps:

Target: We use static analysis to target locations where obfuscation is likely to occur. For language-based obfuscation, these are invocations to the methods used for the obfuscation (e.g. reflection APIs with non-constant target strings). For runtime-based obfuscation, we target native code invocations as these are necessary to modify the state of the ART runtime.

Instrument: We statically instrument the application and the ART runtime to monitor for language-based and runtime-based obfuscation, respectively. This instrumentation reports the dynamic information necessary for deobfuscation.

Run: We execute the obfuscated code dynamically and trigger the application to deobfuscate/unpack and execute the code.

Observe: We observe and collect the deobfuscated information reported by the instrumentation during dynamic analysis. If TIRO discovers that the deobfuscation reveals more obfuscated code, it iterates through the above steps on the new code until it has executed all targeted locations that could contain obfuscation.

TIRO's iterative process allows for deobfuscation of multiple layers or forms of obfuscation used by an application, since the deobfuscation of one form may reveal further obfuscation. This is motivated by our findings that obfuscated code often combines several obfuscation techniques and that deobfuscated code often itself contains code that has been obfuscated with a different technique. For instance, an application that dynamically modifies DEX bytecode in memory often uses reflection to obtain classes and invoke methods in the obfuscated code. Without supporting both forms of obfuscation, either the deobfuscated reflection target is useless without the bytecode for the target method, or the extracted obfuscated code appears dead since the only invocation into it is reflective.

4.1 Targeting obfuscation

A fundamental part of TIRO's framework is the ability to both detect potential obfuscation (i.e. targeting) and to perform deobfuscation (i.e. observation). Without targeting, TIRO would need to instrument and observe all program paths, which could be infinite in number. Targeting enables TIRO to only instrument and observe the program paths that are involved in deobfuscating or unpacking obfuscated code. For this reason, we build the static analysis portion of TIRO on top of IntelliDroid [31], a tool for targeted execution of Android applications. Given a list of targets (i.e. locations in the code), IntelliDroid automatically extracts call paths to these targets and generates constraints on the inputs that trigger these paths. An associated dynamic client solves these constraints at run-time, assembles the input object from the solved values, and injects the input objects to trigger the paths. Using IntelliDroid, TIRO specifies locations of obfuscation as targets. While recent Android obfuscators generally automatically unpack application code at startup (and thus require no special inputs), an added benefit of targeting is that we can use IntelliDroid to generate inputs to trigger paths in future obfuscated code that may only unpack sections of code under specific circumstances [25].

For language-based obfuscation, obfuscation locations are visible in static analysis and the targets provided to IntelliDroid are invocations to reflection APIs, dynamic loading APIs, and native methods. For runtime-based obfuscation, while the obfuscated code is executed in the runtime (i.e. in Java/DEX bytecode), the actual obfuscation is done in native code as described in Section 3.3. IntelliDroid is currently unable to target locations inside native code. As a result, we instead target all Java entry-points into application-provided native code, such as invocations to native methods and to native code loading APIs (e.g. `System.load()`, which calls the `JNI_OnLoad` function in the loaded native library). While this is an over-approximation, targeting native code will ensure that any runtime-based obfuscation can be detected in the instrumentation phase.

4.2 Instrumenting obfuscation locations

Once all of the target obfuscation locations have been identified, TIRO instruments the application and the ART runtime such that any detected obfuscation is reported and deobfuscated values/code are extracted. For language-based obfuscation, TIRO instruments application code since that is where the actual obfuscation occurs. The instrumented code is inserted immediately before the target locations and the instrumentation reports the values of unresolved variables to `logcat`, Android's logging facility. A separate process monitors the log and keeps a record of the dynamic information reported. For example, to deobfuscate a statically unresolvable reflection invocation, the parameters to the invocation are logged (as well as the exact location where invocation occurs, to disambiguate between multiple uses of reflection). To deobfuscate dynamic loading, part of the instrumentation will store the loaded code in a TIRO-specific device location and report this location in the log. Native code transitions are also de-

obfuscated by instrumenting calls from Java into native code and Java methods that can be called from native code. This allows TIRO to create control-flow connections of the type: `Java caller` → `[native code]` → `Java callee`, which helps shed light into what actions are being taken in the native code of an application, even though TIRO does not perform native code analysis.

For runtime-based obfuscation, TIRO instruments the ART runtime. Since the result of this modification is the execution of unexpected code on a method invocation, one approach might be to record the code that was loaded into the runtime for a given method and check whether this code has been modified at the time of invocation. However, this poses a catch-22 situation: to detect the obfuscation, TIRO would have to target the obfuscated method but with runtime-based obfuscation, the obfuscation code could modify any class or method in the program. It would be impractical to target every method in the program. Instead, we use the fact that runtime-based obfuscation must rely on native code to do the actual state modification. As a result, to detect runtime-based obfuscation, TIRO instruments transitions between native to Java and Java to native code to detect whether runtime state has been modified while the application was executing native code.

The runtime state monitored is specific to the objects used to load and execute code, as described in Section 3.3. For example, to detect DEX file hooking, TIRO finds and monitors the `DexFile::mCookie` and `art::DexFile::begin_` fields of all instantiated objects for changes before and after native code execution. If modifications are detected, TIRO reports the call path which triggered the modification, the element(s) that were modified and affected by the modification, and if possible, the code that is actually executed as a result of the runtime-based obfuscation. In some cases, there are legitimate reasons why runtime state may change between initial code loading and code execution (e.g. lazy linking or JIT compilation). We detect these and eliminate these cases from TIRO's detection of runtime-based obfuscation.

Checking all runtime state for modifications can be expensive as there can be many classes and methods to check. To reduce this cost we: (1) only monitor runtime state used in the code loading and execution process, and that are retrievable via the dynamic loading or reflection APIs (i.e. state stored within `DexFile`, `Class`, and `Method` objects); (2) only monitor the objects for methods and classes used by the application, as determined by reachability analysis during TIRO's static phase. This process relies on TIRO's iterative design, since the reachability analysis and subsequent monitoring becomes more complete as the application becomes progressively deobfuscated in later iterations.

4.3 Running obfuscated code

TIRO substitutes the original application with its instrumented code and uses IntelliDroid's targeting capabilities to compute and inject the appropriate inputs to run the instrumented obfuscation locations. However, doing this on obfuscated code raises an additional challenge—many instances of obfuscated applications also contain integrity checks that check for tampering of application code and refuse to run if instrumentation is detected. We found that the most robust method for circumventing these checks is to return (i.e. spoof) the original code when classes are accessed by the application and return instrumented code when accessed by the runtime for execution. To avoid conflicts with any runtime state modification that may be performed by obfuscation code, TIRO checks if any state modifications target instrumented code and if so, TIRO aborts execution of the instrumented code and allows the modifications to be performed on the original application code instead. In the next iteration, after extracting the modified code, the previously obfuscated code will be instrumented and executed.

4.4 Observing deobfuscated results

TIRO observes how the application either resolves and runs sections of code (to defeat language-based obfuscation), or how the application's obfuscation code modifies the runtime state (for runtime-based obfuscation). The results of this observation and the information provided by TIRO's instrumentation are reported to the user for deobfuscation of the application.

The iterative approach taken by TIRO also relies on these observed results to incrementally deobfuscate layers of obfuscated code. For obfuscation that hides or confuses invocation targets (e.g. reflection, native method invocations, method hooking), TIRO's instrumentation reports the caller method, the invocation site, and the actual method that is executed. This information is used in the next iteration to generate a synthetic edge in the static call graph that represents the newly discovered execution flow. Often, this turns apparently dead code into reachable code and TIRO will target this code on the next iteration. For obfuscation that executes dynamically loaded code (e.g. dynamic loading, DEX file hooking, etc.), TIRO's instrumentation extracts the code that is actually executed into an *extraction file*, and a process monitoring TIRO's instrumentation log pulls this file from the device. The extracted code is then included in the static analysis in the following iteration. An example of how TIRO iteratively deobfuscates code from the *dexprotector* packer is given in Appendix A.

5 Implementation

We implemented the static and dynamic portions of TIRO on top of IntelliDroid [31] and added the ART instrumentation that deobfuscates runtime-based obfuscation.

5.1 AOSP modifications

The modifications to AOSP are located within the ART runtime code (`art/runtime` and `libcore/libart`). We have implemented these changes on three different versions of AOSP: 4.4 (KitKat), 5.1 (Lollipop), and 6.0 (Marshmallow) due to the portability issues of the DEX file hooking technique, which is performed by most of the malware in our datasets. In order to access the private `DexFile::mCookie` field for DEX file hooking, applications must use reflection or JNI, but the `mCookie` field type has changed from an `int` in 4.4, to a `long` in 5.0, and finally to an `Object` in 6.0. These changes and other conventions that the malware relies upon (such as private method signatures and locations of installed APKs) result in crashes when the applications are not executed on their intended Android version.

5.2 Extending IntelliDroid

TIRO uses IntelliDroid's [31] static analysis to target likely locations of obfuscation and its dynamic client to compute and inject inputs that trigger these locations. The deobfuscated information extracted by TIRO is incorporated into the static analysis prior to the call graph generation phase and the code instrumentation is performed after the extraction of targeted paths and constraints. To enable support for ART, which was introduced in Android 4.4, we have ported IntelliDroid from Android 4.3 to Android 6.0. In addition, we have ported IntelliDroid to use the Soot [29] static analysis framework, which provides direct support for instrumentation of DEX bytecode via the smali/dexpler [14] library. Previously, IntelliDroid used the WALA analysis framework, which does not have a backend for DEX bytecode. While instrumentation could have been achieved by using WALA with Java-to-DEX conversion tools [7, 21], we found that malicious applications and packers often use very esoteric aspects of the bytecode specification that are not always supported by conversion tools.

5.3 Soot modifications

To incorporate deobfuscated values back into the static portion of TIRO, we made several modifications to Soot [29]. Most of these changes were in the call graph generation code, where we tag locations at which deobfuscated values were obtained and add special edges to the call graph representing dynamically resolved/deobfuscated invocations. Other deobfuscated values/variables are tagged in the intermediate representation and can be accessed in the post-call-graph-generation phases of Soot.

Some obfuscated applications are armored to prevent parsing by frameworks such as Soot. For example, there were several instances of unparseable, invalid instructions in methods that appear to be dead code. While this code is never executed, a static analysis pass would still attempt to parse these instructions, resulting in errors that halt the analysis. In cases where a class definition or method implementation is malformed (which often occurs for applications performing DEX bytecode modification), we skip these classes/methods and do not produce an instrumented version. If the bytecode is modified at run-time, TIRO will extract them and instrument them in the following iteration.

6 Evaluation

To evaluate TIRO's accuracy, we acquired a labeled dataset of 34 malware samples, each obfuscated by one of 22 different Android obfuscation tools. This dataset was provided by the Android Malware team at Google and were transferred to us in two batches: one in March 2017 and another in October 2017. The samples in the dataset were chosen for their use of advanced obfuscation capabilities and difficulty of analysis, and attention was made to ensure that they represent a wide range of state-of-the-art obfuscators. Each sample was manually confirmed as malware and classified by a security analyst from Google, independent of our own analysis using TIRO. To evaluate TIRO's accuracy, we shared the results of TIRO's analysis with Google and they confirmed or denied our findings on the samples.

In our evaluation, the static portion of TIRO was executed on an Intel i7-3770 (3.40GHz) machine with 32 GB of memory, 24 GB of which were provided to the static analysis JVM. The dynamic portion was executed on a Nexus 5 device running TIRO's instrumented versions of Android 4.4, Android 5.1, and Android 6.0.

We begin by evaluating TIRO's accuracy, as well as detailing the findings made by TIRO on the labeled dataset. Then, to measure the use of obfuscation on malware in the wild, we apply TIRO to 2000 obfuscated malware samples from VirusTotal [30]. Finally, we present an analysis of TIRO's performance.

6.1 General findings

Table 1 summarizes our findings after running TIRO on the labeled dataset. The table lists the name of the obfuscator, the number of samples from that obfuscator, the

Table 1: Deobfuscation results

		Obfuscation								TIRO	Sensitive APIs	
		Language-based			Runtime-based							
Sample	#	Reflection	Dynamic loading	Native code	DEX file hooking	Class data overwriting	ArtMethod hooking	Instruction hooking	Instruction overwriting	Iterations	Before	After
aliprotect	2	•	(n)	•	•	•				3	0	44
apkprotect	1	•	(d)	•						2	8	52
appguard	1	•		•	•					2	0	5
appsolid	1	•	(n)	•						2	0	82
baiduprotect	1	•	(n)	•	•	•				2	1	2
bangcle	1	•	(n)	•						2	1	4
dexguard	3	•								2	0	4
dexprotector	3	•	(r)	•						4	0	80
dxshield	2	•	(n)	•	•					2	3	25
ijiamipacker	2	•	(n)	•	•	•	•	•	•	2	1	93
liapp	1	•	(n)	•						2	4	90
naga	1	•	(n)	•	•					2	2	2
naga_pha	1	•	(n)	•	•	•	•	•	•	2	0	6
nqprotect	1	•	(d)	•						2	1	12
qihoopacker	3	•	(n)	•	•					2	3	217
secshell	2	•	(r) (n)	•	•	•				2	200	287
secneo	1	•	(n)	•						3	0	12
sqlpacker	2	•	(d)	•						2	1	31
tencentpacker	2	•	(n)	•	•					3	3	504
unicomsdk	2	•	(d)	•						2	226	227
wjshell	1	•	(d)	•	•					2	8	13

(d) Direct dynamic loading invocation (r) Dynamic loading invoked via reflection (n) Dynamic loading invoked in native code

obfuscation techniques found by TIRO and the number of iterations TIRO used to fully deobfuscate the sample. We also show the number of sensitive APIs that are statically visible before and after TIRO's deobfuscation. For obfuscation tools where there was more than one sample, the table shows the results for the sample with the most sensitive behaviors detected.

After sharing our results with the Google Android Malware team, we confirmed that TIRO successfully found and deobfuscated the known obfuscated code in the applications, with the exception of the two samples packed with *unicomsdk*, and was able to reach and analyze the original applications (i.e. the bytecode for the underlying application before it was obfuscated or packed). On closer analysis, we found TIRO failed on the *unicomsdk* samples because while TIRO does trigger call paths that invoke dynamic loading, the obfuscation code tries to retrieve bytecode from a network server that is no longer active. Our comparison also showed that TIRO did not have any false positives on the dataset—in no case did TIRO mistake legitimate state modification performed by ART for an attempt to perform runtime-based obfuscation by the application.

We make several general observations about the results. First, all of the malware samples employed basic language-based obfuscation such as reflection and native code usage, while roughly 53% (18/34) of the samples also employed the more advanced runtime-based obfuscation techniques. We note that none of the samples in this set employed method entry-point hooking, perhaps owing to their age as these samples are older than those used in our VirusTotal analysis described in Section 6.3. In addition, all used between 2-4 layers of obfuscation, requiring multiple iterations by TIRO. These findings demonstrate the utility of TIRO's iterative design and ability to simultaneously handle multiple types of obfuscation.

Second, many of the obfuscators employed tactics to make analysis difficult. For example, 21 of the 34 samples included code integrity checks that TIRO's code

spoofing was able to circumvent. In addition, a common post-loading step in most of the samples was the deletion of the decrypted code file after it had been loaded. This made it marginally more difficult to retrieve the code, since the unpacked DEX file was unavailable after it was loaded; however, since TIRO extracts DEX code from memory during the loading process, this did not impact its deobfuscation capabilities.

Finally, in all cases, the obfuscation was used to hide calls to sensitive APIs in Java, which were used to perform malicious activity. The number of sensitive APIs shown in Table 1 are the number of API calls found by static analysis before and after running TIRO, where the set of sensitive APIs were obtained from FlowDroid's [2] collection of sources and sinks. On average, TIRO's iterative deobfuscation resulted in over 30 new hidden sensitive API uses detected in each sample. The new sensitive behaviors detected after TIRO's iterative deobfuscation included well-known malware behaviors such as premium SMS abuse and access to sensitive data, including location information and device identifiers.

6.2 Sample-specific findings

We now describe in detail some of the interesting behaviors and obfuscation techniques TIRO uncovered:

aliprotect: During TIRO's first iteration, we found that the APK file contained only one class (`StubApplication`) that set up and unpacked the application's code. Static analysis found only one case of reflection to instrument and one direct native method invocation via `System.load()`. During dynamic analysis, we found that the sample used DEX file hooking to load the main application code dynamically. After loading, the obfuscated DEX file was also overwritten prior to class loading to change the bytecode defining the application's main activity. When extracting the modified DEX bytecode, TIRO found that some of the class data pointers referred to locations outside the DEX code buffer (i.e. outside the DEX file). The application stored code in separate memory locations and, via pointer arithmetic, modified the DEX class pointers to refer to those locations. In the second iteration, static analysis showed that most of the methods in the obfuscated (and now extracted) DEX file were empty—when invoked, they would throw a run-time exception. These empty methods and classes appeared to be decoys and were never actually executed by the application. The methods and classes that were executed had undergone DEX bytecode modification, and TIRO successfully extracted the new non-empty implementations.

apkprotect: In the first iteration, TIRO found several classes in the APK file, none of which were the components declared in the manifest. In the dynamic phase, instrumentation of dynamic loading and reflection retrieved the dynamically loaded code and deobfuscated the reflection targets. From the run-time information gathered, TIRO reported that a number of class objects were requested via reflection, but only one was instantiated via a reflected call to the constructor method.

In the second iteration, TIRO found that only the class that was instantiated was actually present in the dynamically loaded code. Further analysis showed that the application performed a trial-and-error form of class loading, where it looped through class names `app.plg_v#.Plugin` (with `#` a sequentially increasing integer) until it found a class object that could actually be retrieved and instantiated. This form of class loading would have introduced a great deal of imprecision in static analysis since the class name was unknown and obscured by the loop logic; however, with the dynamic information retrieved by TIRO, the static analysis in the subsequent iterations was able to precisely identify the loaded and executed class. During the static phase, TIRO also found two methods within the dynamically loaded code that contained invalid instructions and were unparseable. These methods did not appear to be invoked but attempting to load them without patching Soot resulted in crashes stemming from parsing errors.

baiduprotect / naga / naga_pha: These samples used DEX file hooking to load code dynamically but they would also modify the hooked DEX file multiple times in their execution. Each modification would change the data for one class but also invalidated header values in another; therefore, after the DEX bytecode modification process had begun, no single snapshot of the DEX code memory buffer would result in a valid DEX file. Since TIRO retrieves modified code in a piecemeal fashion as the modification is detected for each class (rather than taking a single snapshot of the buffer), it was able to handle the multiple code modifications and the subsequent mangling of class metadata.

dexprotector: This sample highlights how TIRO deobfuscates multiple layers of obfuscation and is described in Appendix A. It used a combination of reflection to invoke dynamic loading APIs (`DexFile.loadClass()`) and to invoke methods in the dynamically loaded code. The loaded code included another call to `DexFile.loadDex()` for a second layer of dynamic loading that unpacked the main activity. Further iterations deobfuscated the reflected and native method invocations that formed most of the application's call graph.

ijiamipacker: When first installing this APK, the `dex2oat` tool reported a number of verification errors in most of the classes. TIRO's static analysis had similar

results but within the parseable classes, it detected instances of reflection, native methods, and dynamic loading. The dynamic phase showed that some of the classes with DEX verification errors were executed without error due to dynamic modification of the classes' bytecode. Furthermore, the methods were modified one at a time as they were loaded by the class loader, which was achieved by hooking a method within the class loader. In the second iteration, TIRO was able to analyze the extracted bytecode for the now-parseable classes and instrumented new cases of reflection.

We also found that this sample suppressed log messages after a certain point in the unpacking process before the main activity was loaded. Since TIRO's feedback system of relaying dynamic information to static analysis depends on instrumented log messages, this initially posed a problem for deobfuscation. Fortunately, this sample did not suppress error logs, so TIRO was modified to write to the error log as well. A more robust approach would be to implement a custom deobfuscation log that only TIRO can access and control.

qihoopacker: In addition to the DEX file hooking obfuscation that this sample employed, we found that it also invoked `art::RegisterNativeMethods()` to redefine the native method `DexFile.getClassNameList()`. This is a form of native method hooking, where the native function attached to a method is swapped for another. The hooked method `getClassNameList()` does not actually play a part in the class loading process nor was it used by the application; however, it is useful for code analysis as it returns a list of loaded classes and its redefinition made such interactive analysis more difficult.

For completeness, we also found two publicly available method hooking libraries: Legend [18] and YAHFA [19], and used these to create our own application obfuscated with method hooking. For both libraries, TIRO detected the hooked methods, which contained modified method entry-point pointers. These pointers were redirected to custom trampoline/bridge code that resolved the hooked invocation and invoked the target method specified by the developer. TIRO heuristically reported the method objects retrieved by the application that were likely to serve as target methods for this hooking, and in the following iterations, correctly constructed call edges between the hooked and target methods.

6.3 Evaluation on VirusTotal dataset

We also use TIRO to measure the types of obfuscation used by malware in the wild. We searched VirusTotal for malware tagged as obfuscated or packed, and downloaded 2000 randomly selected samples that were submitted throughout the month of January 2018. When TIRO was run on this dataset, it exceeded the 3 hour timeout on the static analysis phase for four of the samples and ran out of memory on two others. Of the remaining samples, all proceeded to instrumentation and analysis by TIRO's dynamic phase. Table 2 shows the breakdown of the types of obfuscation found by TIRO.

Table 2: Obfuscation in 2000 recent VirusTotal samples

Language-based		**Runtime-based**	
Reflection	58.5 %	DEX file hooking	64.0 %
Dynamic loading	79.9 %	Class data overwriting	0.7 %
Direct	52.2 %	ArtMethod hooking	0.5 %
Reflected	0.1 %	Method entry hooking	0.3 %
Native	49.2 %	Instruction hooking	33.7 %
Native code	96.8 %	Instruction overwriting	0.1 %

On this dataset, a larger proportion (80%) of these applications used runtime-based obfuscation techniques, compared to 53% on the labeled dataset. In addition, usage of all types of runtime-based obfuscation were observed, including method entry-point hooking. While this dataset is larger, we speculate that these differences and the broader use of runtime-based techniques likely owe more to the fact that the malware in this dataset are more recent than those in the previous labeled dataset.

The most frequent form of runtime-based obfuscation found was DEX file hooking, which is likely due to the ease of implementing the state modification (i.e. the `DexFile::mCookie` field) required for the obfuscation. Likewise, use of instruction hooking was also prominent, since the obfuscation required changing just the DEX code pointer (and possibly the compiled OAT code pointer) in `ArtMethod` objects. Techniques that require overwriting larger regions of memory or more precise determination of a location to modify (e.g. modifying a vtable entry for ArtMethod hooking) were much less common. This may be due to the implementation effort of these techniques, which require greater knowledge of the runtime objects being modified to ensure that any overwriting maintains the expected layout of these objects and preserves the stability of the runtime. However, we do see instances of these techniques in recent malware, and the overall frequency of runtime-based obfuscation techniques in our dataset is likely in response to advances in analyses that can deal with the simpler and more well-known language-based techniques.

6.4 Performance

We evaluate the performance of the static and dynamic phases in TIRO separately. The run time of the static component increases as iterations find and deobfuscate more code to analyze. In the first iteration of the static

component (where the analysis is only targeting obfuscation locations in the original APK file), the average static analysis time for the samples in Table 1 is 4.3 minutes. However, after the last iteration, the static component takes an average of 12.2 minutes across our dataset.

TIRO's instrumentation also incurs overhead in its dynamic phase. Since the majority of obfuscation occurs in the application launch phase (i.e. when the application unpacks its main activity and other components), we compare the launch time of the application when running in TIRO against the launch time in an unmodified version of AOSP. On average, there is a 3.3× slowdown, with all of the applications launching in under 11 seconds. The majority of this overhead is due to the checking of ART runtime state before and after native code is executed. While this is a noticeable performance impact, we note that TIRO is meant for analysis and not production usage; thus, while the slowdown is large, applications still launch and run in a reasonable amount of time. To further reduce performance overhead, we believe that we can optimize TIRO's monitoring using hardware support. Currently, a full check is performed of all tracked runtime state on every native-to-Java transition. By manipulating memory protections or dirty bits in the hardware page table to identify modified pages, and tracking which objects are stored on those pages, TIRO can reduce the number of objects it must check for modifications.

7 Discussion

From our analysis of obfuscation in recent Android malware, we identify and classify a type of runtime-based obfuscation that differs from obfuscation seen in previous work on x86 and Java. The use of a runtime introduces another technique of hiding code that we show is already in use in Android malware.

7.1 Bypassing the runtime

Unlike language-based obfuscation where the application abuses Java language features, runtime-based obfuscation requires modifying runtime data, which must be done using native code. A natural question is whether runtime-based obfuscation is a stepping stone toward full-native code obfuscation. Static analysis of native code is more imprecise and most existing static malware analyzers for Android are limited to Java bytecode, so a full native code application would make them ineffective. We argue that runtime-based obfuscation is not superseded by full native code but is a complementary technique.

In runtime-based obfuscation, native code is used to modify the runtime state but the execution inevitably returns to Java code after the modifications have been performed. This highlights the main difference between the two forms of obfuscation: in runtime-based obfuscation, the actual malicious behavior can be implemented in Java. Whether this is useful to the malware developer is dependent on the type of malicious activity they wish to execute on a victim's device and how they want to implement it. Many state-of-the-art obfuscators are commercial tools that add wrapper classes to an application to pack them into an obfuscated APK and unpack them when the application is launched. Runtime-based obfuscation allows for complex obfuscation while still allowing the users of these commercial tools to implement their code in Java, which may be preferable due to ease of development. Reusing the existing runtime on Android makes it easier for commercial obfuscation tools to reliably support all forms of Android applications.

In addition, system services are normally accessed through their RPC interface, which would require a transition back into the runtime and would be detected by TIRO's monitoring of native-to-Java transitions. To avoid any Java code (i.e. a true fully native application), the application would have to access system services by calling the low-level Binder interface or Unix `ioctl`s directly. Since the Binder library is not part of the Android NDK, the application is then sensitive to any changes in implementation in the Binder kernel driver or Android service manager. We believe that this is one of the reasons why language- and runtime-based obfuscation is so prominent on Android despite the long history and effectiveness of native code obfuscation on x86. As a result, for the foreseeable future, language- and runtime-based obfuscation techniques will likely still be relevant techniques for obfuscated code on Android.

Another form of obfuscation may be to embed a natively-implemented interpreter within the application that executes a secret bytecode. This is a complementary technique to runtime-based obfuscation and is also a method of bypassing the ART runtime, since the interpreter would be fully implemented in native code. Similar to full-native code obfuscation, access to system services would be limited and invocations to framework methods would still require execution in the ART runtime and would therefore be deobfuscated by TIRO.

7.2 Other limitations

Part of TIRO's deobfuscation focuses on retrieving DEX bytecode that the application dynamically loads and executes. This implicitly assumes that any manipulation of the DEX bytecode is reflected in the compiled OAT or ODEX code, and vice versa. Obfuscation code may violate this assumption and perform modifications directly on the OAT or ODEX bytecode, bypassing the current

implementation of TIRO. However, in doing this, the obfuscation code forgoes portability across devices, as OAT and ODEX files are device-specific. We did not observe any malware instances that were device-specific in this way. If direct OAT or ODEX modification were to exist, it would be straightforward to enhance TIRO to detect these modifications by monitoring `art::OatFile` objects in the same manner as `art::DexFile` objects.

While we have identified a number of forms of runtime-based obfuscation in Section 3.3, there may be others that TIRO currently does not monitor, providing avenues for newer malware to avoid detection and deobfuscation. However, the framework proposed in TIRO is general enough to accommodate the monitoring of other forms of runtime state as they are identified. A further limitation is that applications can employ x86 obfuscation and hooking techniques to bypass TIRO's monitoring within the ART runtime. While we currently cannot prevent this, due to the shared address space between the application and the runtime environment, future work may explore the separation of application and runtime memory, which would also prevent tampering of runtime state and disable runtime-based obfuscation.

Since TIRO relies on dynamic analysis to report deobfuscated values, full deobfuscation of an application would require executing all of its obfuscation code. Since TIRO was implemented on top of IntelliDroid [31], we rely on it to execute targeted obfuscation locations. However, because its analysis is limited to Java, while it can target native method invocations, it cannot extract execution paths within native code. Since native code is used extensively by obfuscators, we may miss certain paths. In addition, IntelliDroid may not be able to extract all targeted paths and constraints due to static imprecision and complex path constraints in the code; TIRO naturally inherits these limitations. TIRO can be combined with fuzzers if deobfuscation is required in native code or in execution paths with constraints that cannot be solved.

8 Related work

A variety of security and privacy analyzers have been developed for Android, including static [2,12] and dynamic tools [10, 27, 28, 34]. TIRO is a hybrid system similar to [22, 23, 31, 32], which use dynamic information to enhance static analysis. Tools that perform malware classification [1, 12] are often based on application semantics and rely on the ability to determine the actions performed by an application. While they are effective against unobfuscated applications, they cannot handle complex code obfuscation and will likely miss malicious actions that the malware performs. While some tools have been designed with obfuscation resilience in mind [13], they often cannot handle the complex obfuscation techniques used by existing Android packers and malware.

The work that most closely resembles TIRO are existing deobfuscation tools for Android. Some focus only on language-based obfuscation. Harvester [22] uses static code slicing to execute paths leading to specific code locations, such as reflection invocations, and can log deobfuscated values. However, code slices do not always produce realistic executions and it does not handle runtime-based obfuscation. StaDynA [38] uses a hybrid iterative approach similar to TIRO to deobfuscate reflection and retrieve dynamically loaded code. However, it relies on instrumentation of reflection and dynamic loading API invocations. Some Android unpackers, such as DexHunter [36] and Android-unpacker [26], handle certain cases of DEX file and DEX bytecode manipulation, but use special packer-specific values to identify the code that must be extracted. They also do not handle any other form of obfuscation, which makes it difficult to analyze the retrieved code if it is further obfuscated in another way. Others, such as PackerGrind [33] and AppSpear [16] have a more general design but their monitoring for bytecode modification is limited to instrumentation of specific methods they expect obfuscation code to use. While these unpackers identify certain cases of DEX bytecode modification, they do not handle other forms of state modification in the code execution process nor do they address the wider issue of runtime-based obfuscation. DroidUnpack [9] uses full system emulation to dynamically extract packed code. While DroidUnpack can extract dynamically loaded code and decrypted DEX files, they do not discuss or indicate if they can handle runtime-based obfuscation the way TIRO can. DeGuard [4] takes a different approach and uses a statistical model to reverse the name obfuscation performed by the ProGuard [15] tool included with the Android SDK. Since TIRO focuses on the actions taken by an application, we do not deobfuscate class and method names. However, combining the results of TIRO and DeGuard would aid in manual analysis of malware.

TIRO is also similar to deobfuscation tools proposed for general Java applications. TamiFlex [5] deobfuscates reflection by instrumenting the reflection classes loaded by the Java runtime, but does not handle other forms of obfuscation. However, its modification of the class loader in the runtime is similar to the technique used in TIRO to load instrumented application classes. Similarly, Ripple [37] also targets reflection but does so through static resolution, which is less precise. These tools do not address runtime-based obfuscation.

Deobfuscation and unpacking tools also exist for x86 applications. Renovo [17] tracks whether previously written memory regions are being executed and can handle multiple "hidden layers" of packing. Polyunpack [24] checks whether dynamic instruction sequences

match those in its static model of the application and returns new unpacked instruction sequences. Ether [8] presents a transparent malware analysis tool that handles emulator-resistant techniques used by packers to prevent reverse engineering. Omniunpack [20] uses an in-memory malware detector to determine if malicious code is being unpacked and retrieves this code from memory. These techniques are more general than those used in TIRO but would require special support to handle the Android runtime and its code loading processes. By focusing on obfuscation for the Android runtime via language-based and runtime-based deobfuscation, we account for the environment in which Android applications are run and produce effective results that can be integrated with existing Android security tools.

9 Conclusion

In this paper, we identify a family of obfuscation techniques used on the Android platform, which we name *runtime-based obfuscation*. These techniques subvert the integrity of the Android runtime to manipulate the code loading and execution processes and execute malicious code surreptitiously. We propose TIRO, a unified deobfuscation framework for Android applications that can deobfuscate runtime-based obfuscation as well as traditional techniques such as reflection or native method invocation. Through an iterative process of static instrumentation and dynamic information gathering that uses **T**arget, **I**nstrument, **R**un and **O**bserve, we show that TIRO is able to deobfuscate malware that have been packed using state-of-the-art Android obfuscators. We also show that runtime-based obfuscation is prevalent among recent Android malware and that effective security analysis will require deobfuscation of these techniques. Using the deobfuscated application information produced by TIRO, it is possible for existing security analysis tools to achieve more complete analysis and detection of Android malware.

10 Acknowledgments

We would like to thank Mariana D'Angelo, Peter Sun, Ivan Pustogarov, James Zhen Huang, Beom Heyn Kim, Wei Huang, Sukwon Oh, Diego Bravo Velasquez, Vasily Rudchenko, Shirley Yang, and the anonymous reviewers for their suggestions and feedback. We also thank Daniel Bali, Jason Woloz, and Monirul Sharif for sharing their expertise in Android malware and obfuscation. The research in this paper was supported by an NSERC CGS-D scholarship, a Tier 2 Canada Research Chair, and a Google Faculty Research Award.

References

[1] ARP, D., SPREITZENBARTH, M., HUBNER, M., GASCON, H., RIECK, K., AND SIEMENS, C. DREBIN: Effective and explainable detection of Android malware in your pocket. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)* (2014).

[2] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. FlowDroid: precise context, flow, field, object-sensitive and-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), p. 29.

[3] BANESCU, S., COLLBERG, C., GANESH, V., NEWSHAM, Z., AND PRETSCHNER, A. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (2016), ACM, pp. 189–200.

[4] BICHSEL, B., RAYCHEV, V., TSANKOV, P., AND VECHEV, M. Statistical deobfuscation of Android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 343–355.

[5] BODDEN, E., SEWE, A., SINSCHEK, J., OUESLATI, H., AND MEZINI, M. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering* (2011), ACM, pp. 241–250.

[6] COSTAMAGNA, V., AND ZHENG, C. ARTDroid: A virtual-method hooking framework on Android ART runtime. *Proceedings of the 2016 Innovations in Mobile Privacy and Security (IMPS)* (2016), 24–32.

[7] Dex2jar. https://github.com/pxb1988/dex2jar, 2017. Accessed: April 2017.

[8] DINABURG, A., ROYAL, P., SHARIF, M., AND LEE, W. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 51–62.

[9] DUAN, Y., ZHANG, M., BHASKAR, A. V., YIN, H., PAN, X., LI, T., WANG, X., AND WANG, X. Things you may not know about Android (Un)Packers: A systematic study based on whole-system emulation. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)* (2018).

[10] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 2010 Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 2010), pp. 1–6.

[11] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (2011), ACM, pp. 627–638.

[12] FRATANTONIO, Y., BIANCHI, A., ROBERTSON, W., KIRDA, E., KRUEGEL, C., AND VIGNA, G. TriggerScope: Towards detecting logic bombs in Android applications. In *Security and Privacy (SP), 2016 IEEE Symposium on* (2016), IEEE, pp. 377–396.

[13] GARCIA, J., HAMMAD, M., PEDROOD, B., BAGHERI-KHALIGH, A., AND MALEK, S. Obfuscation-resilient, efficient, and accurate detection and family identification of Android malware. *Department of Computer Science, George Mason University, Tech. Rep* (2015).

[14] GRUVER, B. smali. https://github.com/JesusFreke/smali, 2017.

[15] GUARDSQURE. Proguard. https://www.guardsquare.com/en/proguard, 2017.

[16] HU, W., AND GU, D. AppSpear: Bytecode decrypting and dex reassembling for packed Android malware. In *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015. Proceedings* (2015), vol. 9404, Springer, p. 359.

[17] KANG, M. G., POOSANKAM, P., AND YIN, H. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode* (2007), ACM, pp. 46–53.

[18] Legend. https://github.com/asLody/legend, 2017.

[19] LIU, R. Yet another hook framework for art (YAHFA). https://github.com/rk700/YAHFA, 2017.

[20] MARTIGNONI, L., CHRISTODORESCU, M., AND JHA, S. Omniunpack: Fast, generic, and safe unpacking of malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual* (2007), IEEE, pp. 431–441.

[21] OCTEAU, D., JHA, S., AND MCDANIEL, P. Retargeting Android applications to java bytecode. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (2012), ACM, p. 6.

[22] RASTHOFER, S., ARZT, S., MILTENBERGER, M., AND BODDEN, E. Harvesting runtime values in Android applications that feature anti-analysis techniques. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)* (2016).

[23] RASTHOFER, S., ARZT, S., TRILLER, S., AND PRADEL, M. Making malory behave maliciously: Targeted fuzzing of Android execution environments. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on* (2017), IEEE, pp. 300–311.

[24] ROYAL, P., HALPIN, M., DAGON, D., EDMONDS, R., AND LEE, W. PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual* (2006), IEEE, pp. 289–300.

[25] SHARIF, M. I., LANZI, A., GIFFIN, J. T., AND LEE, W. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)* (2008).

[26] STRAZZERE, T. android-unpacker. https://github.com/strazzere/android-unpacker, 2017.

[27] SUN, M., WEI, T., AND LUI, J. TaintART: A practical multi-level information-flow tracking system for Android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 331–342.

[28] TAM, K., KHAN, S. J., FATTORI, A., AND CAVALLARO, L. CopperDroid: Automatic reconstruction of Android malware behaviors. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)* (2015).

[29] VALLÉE-RAI, R., CO, P., GAGNON, E., HENDREN, L., LAM, P., AND SUNDARESAN, V. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research* (1999), CASCON '99, IBM Press, p. 13.

[30] VIRUSTOTAL. Virustotal. https://www.virustotal.com, 2018.

[31] WONG, M. Y., AND LIE, D. IntelliDroid: A targeted input generator for the dynamic analysis of Android malware. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)* (2016).

[32] XIA, M., GONG, L., LYU, Y., QI, Z., AND LIU, X. Effective real-time Android application auditing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (2015), SP '15, IEEE Computer Society.

[33] XUE, L., LUO, X., YU, L., WANG, S., AND WU, D. Adaptive unpacking of Android apps. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on* (2017), IEEE, pp. 358–369.

[34] YAN, L.-K., AND YIN, H. DroidScope: Seamlessly reconstructing the os and dalvik semantic views for dynamic Android malware analysis. In *USENIX security symposium* (2012), pp. 569–584.

[35] ZHANG, A. ZHookLib. https://github.com/cmzy/ZHookLib, 2017.

[36] ZHANG, Y., LUO, X., AND YIN, H. DexHunter: toward extracting hidden code from packed Android applications. In *European Symposium on Research in Computer Security* (2015), Springer, pp. 293–311.

[37] ZHANG, Y., TAN, T., LI, Y., AND XUE, J. Ripple: Reflection analysis for Android apps in incomplete information environments. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ, USA, March 22-24, 2017* (2017), pp. 281–288.

[38] ZHAUNIAROVICH, Y., AHMAD, M., GADYATSKAYA, O., CRISPO, B., AND MASSACCI, F. StaDynA: Addressing the problem of dynamic code updates in the security analysis of Android applications. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy* (2015), ACM, pp. 37–48.

Appendix

A Iterative deobfuscation in TIRO

Most obfuscators and packers use more than one of the obfuscation techniques we have described. For instance, like other API invocations that the malware wishes to hide, dynamic loading invocations may be hidden behind reflection. Deobfuscation in these cases requires multiple iterations to resolve the reflection target and, if the target is used for another form of obfuscation, to resolve the reflected obfuscation API.

As an example, Figure 2 shows how TIRO iteratively applies the T-I-R-O loop to deobfuscate the combination of techniques used by the *dexprotector* packer and to extract a complete application call graph.

Iteration 1: The scope of the static analysis is limited to code in the application's APK file. TIRO finds locations of reflected method invocations and instruments them to determine the reflection targets. The dynamic phase executes the instrumented code and reports the reflection targets. It also finds two dynamically loaded DEX files.

Iteration 2: The static analysis scope is expanded to include code from these two DEX files. This code includes entry-points into the application that were previously unknown. However, the use of reflection in

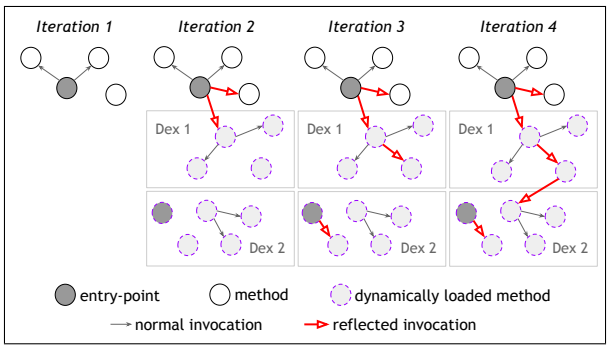

Figure 2: Deobfuscated call graphs produced for an application packed with *dexprotector*

the dynamically loaded code means that the call graph may miss certain invocation edges. TIRO's static analysis adds new instrumentation for any obfuscation (namely, reflection) found in the APK code or dynamically loaded code. The dynamic phase will again execute the instrumented code to find the reflection targets.

Iteration 3: Some reflective call edges are resolved in the static call graph; however, TIRO still sees seemingly-dead code from the second dynamically loaded DEX file. The process is repeated until TIRO encounters no new unresolved obfuscation/reflection.

Iteration 4: The final result is a static call graph that represents all of the code executed by an application and the method invocation relationships. If used alongside a security analysis tool, malicious actions performed by the application can then be discovered by searching the deobfuscated call graph.