# I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications

Benjamin Davis*, Ben Sanders*†, Armen Khodaverdian* and Hao Chen*
*University of California, Davis
{bendavis, bmsanders, aekhodaverdian, chen}@ucdavis.edu
†Sandia National Labs, Livermore, CA
bsander@sandia.gov

***Abstract*—Mobile applications are a major force behind the explosive growth of mobile devices. While they greatly extend the functionality of mobile devices, they also raise security and privacy concerns, especially when they have not gone through a rigorous review process. To protect users from untrusted and potentially malicious applications, we design and implement a rewriting framework for embedding In-App Reference Monitors (I-ARM) into Android applications. The framework user identifies a set of security-sensitive API methods and specifies their security policies, which may be tailored to each application. Then, our framework automatically rewrites the Dalvik bytecode of the application, where it interposes on all the invocations of these API methods to implement the desired security policies. We have implemented a prototype of the rewriting framework and evaluated it on compatibility, functionality, and performance in time and size overhead. We showcase example security policies that this rewriting framework supports.**

***Keywords*-Mobile applications; Reference monitor; Rewriting; Security policy**

## I. Introduction

Smartphones and tablets are becoming ubiquitous, fueled by the explosive growth of mobile applications. For example, Google's Android Market has topped the $200K$, $300K$, and $400K$ available apps milestones in April, August, and December 2011, respectively [1]. Since most of these applications are written by third-party developers and have not gone through rigorous review, they may violate the user's security or privacy expectations. For example, they may exfiltrate the user's confidential or private information or modify critical data. Even though Android allows the user to review and approve the permissions requested by each application, few users are able or willing to understand the implications of each permission completely. Moreover, unless the user agrees to grant all the requested permissions, Android will refuse to install the application.

To overcome the above limitations, we propose a rewriting framework to embed *In-App Reference Monitors* (I-ARM) in Android applications. Unlike traditional x86 applications, most Android application code runs on the Dalvik Virtual Machine (DVM) and is structured and unambiguous. This is in stark contrast to x86 instructions where the meaning of a particular section of binary code depends on the context of the running application. We build a general rewriting framework for Dalvik code in Android applications. Our framework implements a user's desired security policy by interposing on method invocation in an application. Our approach requires no modification of the Android framework and no application source code.

### A. Alternative design choices

Our goal is to enforce flexible security policies on Android applications. We chose the approach of embedding reference monitors in applications by rewriting the Dalvik bytecode directly. We considered but rejected three alternative approaches:

*Modifying the Android Platform:* One could modify the Android framework to specify and enforce flexible security policies [2]. This approach has a few benefits, such as no need to modify individual applications and being able to incorporate system information into the policy enforcement that is inaccessible to the applications.

However, this approach has a number of major drawbacks. First, it would require building custom firmware and platform code, which is tedious, and deploying them, which generally requires rooting the device and voiding the user's warranty. Second, maintaining the modified platform is challenging, especially for different devices and different versions of the platform. Upgrading to a newer version of Android requires reimplementing the custom modifications into the new platform source, and rebuilding and deploying to the device. Third, this approach may not be flexible enough, as applications are limited to the security policies that are supported by the modified Android framework. Furthermore, the build and deployment concerns make it tedious for a user to add or update new security policies over time.

*Rewriting via Java bytecode:* The Android developer writes her application in Java, compiles it into Java bytecode, and finally to the Dalvik bytecode. Since the Java VM (JVM) has existed much longer than Dalvik (DVM), research and tools on rewriting Java bytecode are much more mature, e.g., the ASM framework [3]. To take advantage of these

existing tools, we were tempted to first convert the Dalvik bytecode to Java bytecode, then rewrite the Java bytecode using existing tools, and finally convert the rewritten Java bytecode back to Dalvik bytecode.

However, there are several important differences between JVM and DVM, the most glaring one being that JVM operates on a stack, whereas DVM uses virtual registers to manipulate local state. Several tools, such as dex2jar [4] and ded [5], attempt to convert Dalvik bytecode back to Java bytecode. Because some information from the Java bytecode is lost when being converted to Dalvik, these tools have to infer the missing details based on the surrounding context, but the inference is sometimes wrong (as described by Reynaud et al. [6]). Even though these errors may not prevent static analysis on the converted Java bytecode, in our experience they often lead to invalid Java bytecode or later invalid Dalvik bytecode. In other words, after we converted an application's Dalvik bytecode to Java bytecode (e.g. using dex2jar) and then back to Dalvik bytecode, the resulting application sometimes failed to run. To avoid these errors, we decide to rewrite Dalvik bytecode directly.

*Placing the reference monitor in another application:* Our approach embeds the reference monitor in the untrusted application directly. Alternatively, one could embed the reference monitor in another application. The advantage of this approach is that the user could then remove all the permissions from the untrusted application. Instead, the untrusted application delegates all API calls that require permissions to the reference monitor application. Even though this approach has the desirable *fail-safe default* property, it suffers from several disadvantages:

- *Complexity*. Implementing API call delegation correctly is difficult, especially the stateful calls. For example, when a call takes a parameter that refers to a context local to the application (e.g. database connection or file handle), the delegation must map the corresponding contexts between the untrusted and reference monitor applications, which is difficult and error prone.
- *Performance*. Cross-application API calls are expensive. When they occur often, they degrade the application performance.
- *Anti-least privilege*. Since the reference monitor must be able to perform delegated access from all possible applications, it would need all the permissions. Therefore, a bug in the reference monitor application might allow an untrusted application to invoke a privileged API call (via delegation) that its original permission would never allow.

Therefore, we chose the approach of embedding the reference monitor inside the untrusted application directly.

## II. Design

Our framework for building reference monitors into Android apps is built on the ability to identify and rewrite method calls, as method calls are the primary way most Android apps interact with the underlying platform of the device. In this section we describe the design of our framework for intercepting and altering the behavior of method calls. Our goal is to have a system flexible enough to add whatever functionality is required to achieve the user's security goals.

Our approach involves rewriting the applications themselves, leaving the underlying platform unmodified. Users can easily add different behavior to each app individually, and deploying a rewritten app is as easy as installing any other app. Identifying new methods of interest and adding new custom behavior does not require a change in the underlying platform. Also, apps that have not been modified do not pay a performance penalty for changes users require for other apps.

### A. Changing Method Behavior

Before rewriting an application using our system, the user must identify the methods on which they wish to interpose. We call these methods of interest the *target methods*. Users identify target methods by the full method signature, including the types and package and class containing the method. In Dalvik bytecode methods are identified uniquely so with full method signatures there is no danger of confusing two methods with the same name. The methods of interest greatly depend on the goal of the rewriting, though tools such as static analysis may be valuable ways to determine methods to target. For example, Stowaway [7] used static analysis on the Android platform to map all Android methods that required Android permissions. One could use a method listing like this as a basis for adding a flexible, fine-grained access controls into Android applications without requiring any platform modifications.

In addition to identifying the methods the user wishes to interpose on, the user must also specify the new behavior they wish the app to execute in place of the original method invocation. We designed our system so that the user can specify their custom behavior by writing Java code. This gives the user the power to write arbitrarily complex behavior without having to consider the details of the underlying bytecode. Our system automatically compiles the custom behavior code into Dalvik bytecode and adds it to the app during the rewriting phase.

### B. Adding Custom Behavior

The user must specify custom behavior for each target method. We place this custom behavior into separate methods inside classes in our own package that we add to the rewritten application. In our rewriting phase, we identify all method calls in the app that match the signatures specified by the user. We change these method calls to invoke our methods instead of the original calls.

We take this approach instead of adding the custom behavior to the apps inline for two main reasons. First, it

limits the size of the code we add to an app during rewriting. Adding new behavior inline around each individual method invocation means that the code size increases linearly with the number of places a method is invoked. Instead, in our approach the amount of code we add to an app scales linearly only with respect to the number of different methods signatures that we wish to intercept. Second, this approach greatly simplifies the rewriting process. The Dalvik VM is a register-based architecture, and many operations can only operate on a subset of registers (e.g. the first 16 registers) [8]. By simply changing the method specified in the invoke operation we can achieve our goal by making minor modifications to the bytecode, without requiring a detailed analysis of register allocation and use.

Note that as long as our custom method returns an object of the correct type, there is no requirement that we actually invoke the original target method. For example, the custom behavior in our new methods could return fake sensor data without ever accessing the actual sensor on the device.

### C. Types of Method Invocation

In Dalvik code there are three main types of methods that can be invoked: constructors, static methods, and instance methods. In each of these cases we create new static methods that contain the desired custom behavior as specified by the user. We call the static methods we add *stub methods*. We describe the semantics of our stub methods and how we convert existing method calls to use them in Section III.

We use an additional technique to interpose on calls to methods in classes that can be extended by the developer of the original application. If a user wishes to interpose on a method call in some (non-final)[1] class `A` in the Android platform, in addition to the corresponding stub methods we generate a *wedge class* `B` that extends `A`. This wedge class includes a *wedge method* corresponding to the target method. We identify all classes in the application that extend `A` and modify them to instead extend our wedge class `B`. In this way, all developer classes that originally extended and called the target method in `A` pass through our wedge method instead, where we include the user's custom behavior. This also simplifies our rewriting process by reducing the need to analyze all classes in the developer's hierarchy for methods that call or override methods in the parent class. We describe wedge classes in more detail in Section III.

In addition to normal method invocation, developers can execute code dynamically using reflection. However, these calls to the reflection library are normal method invocations that we can intercept using our approaches described above. That is, we can statically determine where the app invokes a method in the reflection API, even if we may not be able to predict the values that will be passed as parameters to these methods. Since we can statically identify calls to the reflection API, we can replace these method invocations with calls to our own methods that include code to inspect the arguments at runtime.

Android developers may also attempt to run bytecode that has not been rewritten by using a ClassLoader to inject code at runtime. In these situations, we can still analyze the app for method calls that initiate the ClassLoader by specifying the appropriate signatures. Again, by writing signatures for these methods we can interpose on these calls to detect their use and insert our own code to perform runtime checking.

[1]We do not need wedge classes for final classes, as developers cannot extend final classes.

### D. Other Code Execution

Android applications may also call libraries of native code. While analyzing and rewriting native code requires techniques beyond the scope of this paper, we can identify calls from Dalvik bytecode to native code. In our current implementation, we abort the rewriting process and notify the user when we detect native code.

## III. Implementation

In order to rewrite an Android application, users of our prototype must specify the method calls they wish to interpose on, and define the augmented behavior they want to add to the application. Our prototype system takes in a list of fully-qualified method signatures, which contain the package, class, and method name as well as the types of the parameters and the return value. We call the methods the user wishes to intercept the "target" methods. We use this information to automatically generate "stub" and "wedge" code (as described in Section II) as needed. Then we analyze the Dalvik bytecode and identify all calls to specified methods, and rewrite the bytecode to call our stub and wedge methods instead.

In addition to specifying the method calls on which to interpose, the user of our tool must also define the desired behavior for these modified methods. For example, this added behavior could be logging when a method is called, or dynamically generating fake sensor data to return to an application. We discuss some types of behavior we can add in Section VI.

In this section, we describe exactly how we generate our stub and wedge code, and how we perform the rewriting on the Dalvik bytecode itself to redirect target methods to our new methods.

### A. Generating Stub and Wedge Code

We create a new method containing the augmented behavior we want to execute for every target method specified by the user. When an application would normally have called a target method, an app rewritten using our system calls our new method instead.

Stub and wedge methods are placed into classes that mirror the hierarchy of their corresponding target methods.

We pick a random prefix *pkgprefix* that is not already part of any package names in the Android platform or application. Then, if the user wanted to interpose on the `openStream` method in the `java.net.URL` class then our framework creates the stub method `openStream` in the class `pkgprefix.java.net.URL`. Similarly, we put wedge classes into the *pkgprefix.wedge* package, mirroring the target method's class name and package hierarchy. We also declare all of our methods to throw the same exceptions as the original methods on which we are interposing.

Our prototype allows users to specify a template for stub and wedge methods, allowing simple specification for classes of behavior. If users make use of this template method (e.g. having all stub methods call some same "`doCheck`" method) then these stub and wedge methods will be very small.

Note that although the stub classes we create have the same name as the classes of the associated target methods, our stub classes do *not* extend the original classes (in fact, they cannot if the methods are final). Using the same name simplifies the organization of our code generation, and in the Dalvik bytecode classes are uniquely identified in a way that there will be no ambiguities about classes with the same name in different packages.

*1) Static Methods:* To generate stub methods (and wedge methods for non-final methods) for a given static target method, we simply duplicate the original method's Java signature. For example, to interpose on the on the `sqrt` method in `java.lang.Math` then our stub method `public static double sqrt(double var0)` in the `pkgprefix.java.lang.Math` class. If desired, we can compute the original value from within our method directly (e.g. `java.lang.Math.sqrt(var0)`).

*2) Instance Methods:* We generate a static stub method for each target instance method. For an instance method that takes $n$ arguments, we create a static stub method that takes $n+1$ arguments, where the first argument in our stub method is the instance object in the original app. For example, for the instance method `setContentView(int)` in the `Activity` class we generate a stub method with the signature `public static void setContentView(android.app.Activity var0, int var1)`. Since we're receiving the instance value as the first parameter, we can call the original method inside our static stub (e.g. `var0.setContentView(var1)`).

As we described in Section II we also generate wedge classes for all classes containing non-final methods. These wedge classes extend the original class, so if the user wants to intercept `Activity->setContentView(int)` then we create class `pkgprefix.wedge.android.app.Activity` that extends `android.app.Activity`. In this wedge class we have a method with the same signature as the original method (e.g. `public void setContentView(int var0)` where we add code to add our desired functionality. We can make the original method call on the instance with `super.setContentView(var0)`.

*3) Constructors:* For final constructors that users want to intercept, we create static "factory" stub methods that create the object of an appropriate type and return it back to the application. For example, to interpose on the constructor that takes a `String` for the final class `java.net.URL`, we create a method in our stub `URL` class with the signature `public static java.net.URL cons_java_net_URL(java.lang.String var0) throws java.net.MalformedURLException`. In this method we perform whatever behavior the user desires and return a `URL` object.

*4) Injecting Stub and Wedge Classes:* After generating the Java code for our stub and wedge classes, we compile them into Dalvik bytecode. We add this new bytecode to the application we are rewriting. All of our new classes are in the new *pkgprefix* package hierarchy we created, none of our fully qualified package and class names will interfere with existing classes.

### B. Method Call Transformations

After creating the necessary stub and wedge classes, we must identify all calls to the target methods in the original application, and rewrite the bytecode to call our methods instead.

*1) Static Methods:* In Dalvik bytecode, static methods are called via the `invoke-static` (and `invoke-static/range`) instructions. These instructions take a variable number of registers – one for each parameter to the static method. Because our static methods take the same number and types of arguments as the original static methods, we simply change the method referenced in the original instruction to our associated stub method.

*2) Instance Methods:* Instance methods are invoked with the `invoke-virtual` and related instructions. These instructions include a reference to the method to be invoked (including a reference to the associated class and package). These instructions also include a variable number of registers. The first register contains a reference to the instance object on which the method should be called. The remaining registers referenced in the instruction point to the values that are passed as parameters to the method itself.

We rewrite these instructions by replacing the instruction opcode with the `invoke-static` opcode of the same form (e.g. `invoke-virtual/range` becomes `invoke-static/range`). We change the reference to the target to method into a reference to our stub method. We do not need to make any changes to the registers associated with the instruction. When invoking a static method, the first

register is now passed as the first parameter to the method, which matches the parameters our stub method expects.

*3) Constructors:* Normally, apps construct new instances by calling the appropriate constructor with the `invoke-direct` instruction. The first register in instructions of this type is the destination where the reference to the new instance will be stored after the object is created. The remaining registers are the parameters passed to the constructor.

We rewrite these instructions by replacing the opcode with the `invoke-static` opcode of the same form, with the method reference pointing to our corresponding stub "factory" method. We include all but the first register from the original instruction in the `invoke-static` call. Inside our stub method we receive the parameters passed to the original constructor, which can be used as needed. We create an instance of the appropriate type and return it. After our `invoke-static` call we also insert an instruction to move the instance returned from our stub method into the register expecting the new instance.

### C. Modifying Class Hierarchy to Inject Wedge Classes

After we have created our wedge classes, we identify classes in the app code that extend from the parents of our wedge classes. In each of these classes, we modify the class to extend our wedge class instead of the original class in the Android platform. We also modify all method invocation calls in these app classes to point to our wedge class instead of the original parent class. In this way, the rewritten developer's class cannot call the methods in the Android platform classes without passing through the methods we have added to catch intercept calls.

### D. Prototype Implementation Details and Scope

Java's classloader and reflection capabilities still rely on normal method invocation that we can detect and interpose on using our framework. That is, we can statically determine all points where reflection is invoked, even though we might not know the parameters passed to these methods until runtime. In theory, we could interpose on reflection calls and do dynamic introspection of the arguments passed to reflection methods, and determine how to execute each call at runtime. Note that this requires a recursive solution, as a crafty developer may attempt to call reflection methods via reflection. In our current implementation we only detect and prevent the use of reflection to block this avenue of executing unanalyzed code.

Analyzing and rewriting native code requires different techniques that are beyond the scope of this paper. We detect the use of native code in our prototype, but do not attempt to modify the behavior or native code included with Android apps.

We use smali/baksmali to parse and assemble our dex bytecode [9].

## IV. Applications

A flexible application-level method rewriting framework can be used to add a wide variety of useful functionality to applications without requiring changes to the underlying platform. In this section we describe some ways our rewriting-based reference monitor framework may be applied.

### A. Improved Fine-Grained Access Control

The current Android permission model offers a user only a limited amount of flexibility and control over how apps access the resources on their device. If a user wants to install an app then she must grant the application all of the permissions that it requests. It can be difficult to determine what an app will do just from this permission list alone. Once the user agrees and installs the app then they have no control over how the app behaves as longs as it does not perform operations requiring permissions that it did not request.

Previous work (such as [2], [10]) add fine-grained control by modifying the underlying platform. We believe our system could be used to build similar controls directly into apps, eliminating the serious build, configuration, and deployment issues involved with custom platforms like these. We could use an approach such as the described in [7] to build a list of permission-sensitive methods. Then we could define custom behavior for each method to check with a dynamic policy to determine how to handle each request. Similar to the systems and operate in custom Android framework, we could allow the user to specify a dynamic policy on how to handle each kind of access to device resources. For example, our stub and wedge methods could check a dynamic policy to determine whether to allow or deny a request. Furthermore, as long as we return objects of the expected type, we just as easily generate fake or anonymized data and return these values in lieu of the original result to provide additional control over user privacy. Also, because our stub and wedge methods receive all of the parameters that were intended to be passed to the target method, we can use this information to make further policy decisions about how to handle each method request. We can use this approach to distinguish between Internet request to different domains, similar to the functionality offered by ACPlib [11].

We note that any policy accessed within an app should be protected from tampering. One easy approach is to simply make the policy information stored in a resource that the rewritten app has read-only access to (e.g. an Android ContentProvider managed by a separate, trusted app).

### B. Implementing "Same Origin Policy" for App/Ad Code

Because our app rewriting includes an analysis of an entire app's codebase, we can provide different behavior for target methods called in different parts of the code. For example, many Android apps include advertising libraries. While these

libraries were written by a different party, normally Android does not distinguish between code in ad libraries and code for the rest of an app.

The "same origin policy" is a security policy in web browsers that prevents a script from one origin from accessing document content from another origin. We can use our rewriting framework to implement a sort of "same origin policy" for Android applications, where we distinguish between advertising code and normal application code.

For example, imagine a contact management app that also shows advertisements retrieved from the web. A user may wish to give the main portion of the app access to their contacts, but might not be comfortable giving their contacts to the ad library. Also, the user may wish to support the developers by permitting advertisements retrieved from the web, but not want the main portion of the app to access the web after accessing their contacts. Users of our system could detect repackaged library code during the rewriting phase using techniques such as those suggested by Zhou et al. [12]. We could add separate stub and wedge methods for each kind of behavior during the rewriting phase. Alternatively, we could simply have our stub and wedge methods inspect the call-stack at runtime to determine what part of the code called a target method. Of course, it is difficult to remove all possible side-channels of communication between different portions of an application, but our system would allow a user far more control over their apps than they have in Android's current design.

### C. General-Purpose Instrumentation

Our app rewriting framework allows a user to gain valuable insight into the behavior of apps on their device. Aside from building a reference monitor, another use case of our system is that it can allow instrumentation of applications. Since our system can detect the call sites of target methods it can, for example, log every time a target method gets called. As a result, this could provide a useful tool for determining an application's behavior by analyzing how often and which permissions the applications exercises. By adding logging to key method calls our system allows a user to add simple profiling capabilities to any app. Users can also gain insight into how applications use device resources like GPS, SMS, accelerometer, and network access.

## V. Evaluation

We performed several tests to demonstrate our approach is feasible for rewriting real-world Android applications.

### A. Compatibility

We randomly selected thirty free applications from the top 100 free apps on the official Android Market. For each of these applications, we interposed on the static method `java.lang.Math.sqrt`, common Java instance methods `java.net.URL.openStream` and `java.lang.StringBuilder.append`, Java reflection method `java.lang.reflect.Method.invoke`, Android instance method `android.app.Activity.setContentView`, and a constructor for `java.lang.String`.

| app name | # wedge | # stub |
|---|---|---|
| com.amazon.kindle | 46 | 62 |
| com.bayview.tapfish | 120 | 42 |
| com.creativemobile.DragRacing | 193 | 37 |
| com.droidhen.irunner | 179 | 1 |
| com.espn.score_center | 117 | 6 |
| com.facebook.orca | 90 | 243 |
| com.google.android.apps.maps | 151 | 325 |
| com.google.android.apps.translate | 276 | 24 |
| com.google.android.googlequicksearchbox | 157 | 13 |
| com.google.android.youtube | 24 | 67 |
| com.justwink | 169 | 14 |
| com.magmamobile.game.BubbleBlastValentine | 384 | 14 |
| com.mobilityware.solitaire | 394 | 53 |
| com.myxer.android | 191 | 31 |
| com.oovoo | 106 | 41 |
| com.pandora.android | 33 | 94 |
| com.rechild.advancedtaskkiller | 187 | 1 |
| com.scannerradio | 639 | 17 |
| com.shootbubble.bubbledexlue | 134 | 10 |
| com.socialnmobile.dictapps.notepad.color.note | 254 | 18 |
| com.stylem.wallpapers | 219 | 5 |
| com.twitter.android | 57 | 146 |
| com.weather.Weather | 141 | 95 |
| com.yahoo.mobile.client.android.im | 215 | 75 |
| com.yahoo.mobile.client.android.mail | 224 | 93 |
| com.zynga.hanging | 379 | 66 |
| com.zynga.livepoker | 54 | 10 |
| com.zynga.words | 375 | 57 |
| mk.g6.crackyourscreen | 222 | 6 |
| net.lucky.star.mrtm | 306 | 12 |

Table I
Number of method call sites modified to call wedge and stub methods (described in Section II)

Our automated system successfully rewrote all thirty apps to intercept method calls matching our signatures. Our prototype is completely automated and does not require any manual guidance or app-specific hints to perform the rewriting. Table I contains a listing of the apps we tested, along with the number of method call sites that we transformed to instead call our wedge and stub methods.

### B. Functionality

We manually verified that all thirty rewritten apps generated by our automated rewriting framework installed and ran successfully on an Android emulator running Android version 4.0.3 (API level 15). We confirmed by inspection that our wedge and stub code was included in the rewritten app bytecode. We ran and manually exercised the capabilities of each app until the app performed some functionality that involved calling one or more methods specified by our signatures. All of our stub and wedge methods included code to log a message containing information about the method being called to logcat (Android's logging system).

This allowed us to verify that our methods were indeed being called in apps rewritten by our system. We successfully verified that all thirty rewritten apps did call our methods by observing the corresponding messages in logcat.

### C. Performance

We performed a microbenchmark test to evaluate the performance impact of our approach. In our system, rewritten apps call our stub or wedge method every time the app would have normally called a method matching one of the signatures. To evaluate the impact of adding this extra method call, we wrote a simple application that used the `StringBuilder` class to append one million characters together by calling `StringBuilder.append` one million times. Then we used our system to automatically rewrite our test application using our system to interpose on each call to `append`. Following our scheme, our stub method for `append` receives the `StringBuilder` instance and the character to append. Our stub method simply invokes `append` on the incoming `StringBuilder` parameter and returns the result back to the original app.

We performed our microbenchmark test on a HTC Thunderbolt phone running Android 2.3.4, and performed 100 trials with each of the original and rewritten apps. On average, the unmodified app took an average of 680ms to perform one million the append operations, and an average of 790ms to perform the same task after rewriting to interposed on all append method calls. This means that in our microbenchmark, adding an extra method call adds less than 0.2 microseconds to a method call.

In our tests it seems that the overhead of additional method calls for each stub and wedge method will be dominated by the time taken to perform the methods themselves, making our approach practical for all but the most performance-constrained environments. Of course, when users add additional functionality to stub and wedge methods (e.g. checking a dynamic security policy for fine-grained access control) they must consider the performance impact of the code they add.

### D. Size

Mobile devices frequently have storage and bandwidth constraints, so it is important to consider the impact of our system on app size. In our rewriting system we not only modify existing code, but we also add our stub and wedge classes containing our stub and wedge methods. Recall that the number of methods we add increases linearly with the number of method signatures we have, not the number of times the methods are called in the original app.

To test the impact on app size, we wrote signatures for 60 different methods corresponding to popular network, reflection, and Android platform methods. 30 of those methods also required corresponding wedge methods, so our approach requires adding bytecode for a total of 90 new methods to a rewritten app. Each method included its own call to the logcat logging mechanism, with a unique string identifying the method as well as code to perform the original method call and return the result. We used these signatures to rewrite the same thirty apps we selected from the Android Market.

Android applications are stored as APK files, which is a compressed format. Each APK includes a `classes.dex` file containing all of the Dalvik bytecode in Dex format. Adding our stub and wedge classes and method bytecode increased the (uncompressed) classes.dex file size by approximately 13KB in each app. Adding our stub and wedge methods to the classes.dex file resulted in a median increase in size of less than 2% in the thirty apps we examined, which is further minimized when the `classes.dex` file is compressed in the APK.

In our current prototype we do not inspect each app to see if all stub and wedge classes are needed. If we can determine statically that we do not need a stub or wedge method in the rewritten application then we could remove those unneeded methods. This optimization could greatly decrease the impact on application size in situations where there are many signatures for methods that are never called. Of course, if we permit an application to use reflection then it may be impossible to statically identify all unneeded stub and wedge methods.

## VI. Discussion

In this paper we propose a framework for adding in-app reference monitors to Android applications. The quality of the reference monitor depends on the completeness of our approach. We discuss the two main aspects of completeness.

### A. Policy Completeness

A reference monitor must have a complete policy. Specifically, in our system the user must provide a complete list of Android methods she wishes to detect and rewrite. If the user fails to identify methods in the Android framework that access sensitive resources, then our framework will not know to rewrite these method calls when it sees them in the app. While selecting the correct methods for the policy is important, the methodology for doing so is orthogonal to our work, and highly dependent on the user's security goals. Felt et al. [7] have used static analysis techniques to determine methods in the Android framework that perform permissions-sensitive operations, and these techniques can be adapted to determine methods appropriate for a variety of policies.

### B. Rewriting Completeness

Our reference monitor framework must also rewrite applications completely, in the sense that all calls to target methods must be identified and rewritten. Luckily, in the Dalvik bytecode format there are only a few, well-defined

ways of invoking a method. By enumerating all of the ways methods can be invoked as defined in the official bytecode reference [8], to our knowledge we have covered all cases. The Java reflection API is also invoked from within the app using these mechanisms. Our framework allows users to statically identify and hook these calls to the reflection API like any other method call, and allows them to block or provide runtime analysis as desired. We describe the categories of method calls as they relate to higher-level Java abstractions in Section II-C.

As mentioned earlier, analyzing and rewriting native code is outside of the scope of our paper. However, we do detect calls to native code, and our system refuses to rewrite the application if it includes native code.

## VII. Related Work

### A. Modifying Android Behavior

Many researchers have proposed ways to modify the way that Android applications function on a device by modifying the underlying Android framework [13], [2], [10], [14]. TISSA [2] and Apex [10] are two examples of systems designed to provide fine-grained control over permission-based methods. The drawbacks of these systems are that they require modifying the Android framework. Installing a custom framework on a device may entail voiding the warranty and violating the terms of service associated with the device. One might imagine adding the capabilities these systems provide directly into applications using our system, resulting in apps that can be deployed to any stock Android device. In Section VI we discuss how we might use our system to provide a the functionality these systems provide by rewriting applications without requiring any changes to the underlying Android platform.

### B. Android App Rewriting

Reddy et al. [11] have developed ACPlib/Redexer, which performs Dalvik bytecode rewriting on Android apps with the goal of implementing more fine grained permissions in apps. While our goal was a flexible, general app-rewriting framework, their focus is entirely on enforcing "application-centric security policies." ACPlib is a collection of Android services, which they use to perform specific permission-sensitive API calls. When an Android app makes a permission call, they proxy this request through the appropriate service in ACPlib, which executes the method and returns the result. This allows them to make all sensitive method calls from their service process, to which they assign the Android permissions required to make the API calls.

While this is a reasonable approach for relatively "stateless" requests (e.g. setting the ringtone or toggling the GPS), it is difficult to interpose on other operations that our approach can handle. For example, if we want to modify the behavior of file I/O methods we can easily do so because code running in our stub and wedge methods are still within the same process space as the original application. However, objects tied to resources are difficult to pass between Android applications (i.e. between the app and ACPlib), and it does not appear that ACPlib/Redexer is capable of proxying complex operations like file I/O requests to an ACPlib service.

Reynaud et al. [6] use a specialized rewriting transformation to reveal vulnerabilities in Google In-App Billing. They create a fake Google Market app and rewrite applications to bind to their fake Market app instead of the genuine Market app. While their rewriting goals are limited in scope and rely on conversion to Java code, their work illustrates another application for app rewriting, and supports our belief in the utility of a flexible, general-purpose application-rewriting system.

### C. Java-Based Analysis

Because of the similarities between Dalvik and Java code, there are several tools designed to convert Dalvik code into Java. The tools ded [15] and dex2jar [4] can convert Dalvik bytecode into JVM bytecode. This allows tools designed for Java bytecode analysis to be used on Android applications. However, these tools are only intended for one-way conversion to Java bytecode, and attempts to convert back to Dalvik are reported [6] to frequently result in Android apps that no longer function. It is nontrivial to do the round-trip conversion between Dalvik and Java bytecode, so our approach operates on the Dalvik bytecode without requiring conversion to Java.

The rewriting techniques we use in our project is related to similar rewriting work done for the Java platform. Even though the Dalvik VM is register-based while the JVM is stack-based, we can leverage many of the ideas and techniques from these earlier works. We build upon the work by [16], which used both invocation replacements and safe versions of classes, much like our stub methods and wedge classes. We've tailored our work to the Android environment, which has a large number of permissions and sensitive data to control, as well as many of the same concerns from their paper. In [17], Rudys and Wallach perform Java bytecode rewriting to wrap certain pieces of code. In addition, they also implement soft termination to prevent infinite loops, and transactional rollback in order to revert persistent changes by code that has been aborted by their reference monitor. Their approach allows for full instrumentation of all of the bytecode, whereas we have focussed our work on intercepting and instrumenting at the method call level.

Our goal of implementing inline reference monitors directly into applications has been successfully performed by Erlingsson and Schneider [18] for Java applications. Many of the practical advantages of the Java "IRM" design [19] also apply to reference monitors added to applications using our framework. Specifically, by adding security controls inline

it is possible to get great insight into the application being observed. Through careful policy specification, users can apply our framework to leverage the benefits of running their inlined code in the same process space as the rewritten Android application, with full knowledge of the parameters passed to target methods.

## VIII. Conclusion

We have designed and implemented a rewriting framework for embedding reference monitors in Android applications. The framework user identifies a set of security-sensitive API methods and specifies their security policies, which may be tailored to each application. Then, our framework automatically rewrites the Dalvik bytecode in the application, where it interposes on all the invocations of these API methods to implement the desired security policies. We have implemented a prototype of the rewriting framework and evaluated it on compatibility, functionality, and performance in time and size overhead. We showcase example security policies that this rewriting framework supports.

## Acknowledgments

This paper is partially based upon work supported by the National Science Foundation under Grant No. 0644450 and 1018964.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

## References

[1] "Google Android Market Tops 400,000 Applications," http://www.distimo.com/blog/2012_01_google-android-market-tops-400000-applications/.

[2] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh, "Taming Information-Stealing Smartphone Applications (on Android)," *Trust and Trustworthy Computing*, pp. 93–107, 2011.

[3] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: a code manipulation tool to implement adaptable systems," *Adaptable and extensible component systems*, vol. 30, 2002. [Online]. Available: http://asm.ow2.org/current/asm-eng.pdf

[4] *dex2jar: Tools to work with Android .dex and java .class files*. [Online]. Available: http://code.google.com/p/dex2jar/

[5] D. Octeau, W. Enck, and P. McDaniel, "The ded Decompiler," Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, Tech. Rep. NAS-TR-0140-2010, Sep. 2010. [Online]. Available: http://siis.cse.psu.edu/ded/papers/NAS-TR-0140-2010.pdf

[6] D. Reynaud, D. Song, T. Magrino, E. Wu, and R. Shin, "FreeMarket: Shopping for free in Android applications," in *NDSS*, 2012.

[7] A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 627–638.

[8] *Bytecode for the Dalvik VM*, The Android Open Source Project, 2007. [Online]. Available: http://source.android.com/tech/dalvik/dalvik-bytecode.html

[9] B. Gruver, *smali: An assembler/disassembler for Android's dex format*. [Online]. Available: https://code.google.com/p/smali/

[10] M. Nauman, S. Khan, and X. Zhang, "Apex: Extending Android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. ACM, 2010, pp. 328–332.

[11] N. Reddy, J. Jeon, J. Vaughan, T. Millstein, and J. Foster, "Application-centric security policies on unmodified Android," UCLA Computer Science Department, Tech. Rep. 110017, Jul. 2011.

[12] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "DroidMOSS: Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces," *2nd ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2012.

[13] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 639–652. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046780

[14] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "MockDroid: trading privacy for application functionality on smartphones," in *HotMobile*, 2011.

[15] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A Study of Android Application Security," in *Proceedings of the 20th USENIX Security Symposium*, August 2011. [Online]. Available: http://www.enck.org/pubs/enck-sec11.pdf

[16] A. Chander, J. Mitchell, and I. Shin, "Mobile code security by Java bytecode instrumentation," in *DARPA Information Survivability Conference & Exposition II, 2001. DISCEX'01. Proceedings*, vol. 2. IEEE, 2001, pp. 27–40.

[17] A. Rudys and D. Wallach, "Enforcing Java run-time properties using bytecode rewriting," *Software Security Theories and Systems*, pp. 271–276, 2003.

[18] U. Erlingsson and F. Schneider, "IRM enforcement of Java stack inspection," in *Security and Privacy, 2000. S P 2000. Proceedings. 2000 IEEE Symposium on*, 2000, pp. 246 –255.

[19] U. Erlingsson, "The inlined reference monitor approach to security policy enforcement," Ph.D. dissertation, Cornell University, 2003.