Reptor: Enabling API Virtualization on Android for Platform Openness

Taeyeon Ki, Alexander Simeonov, Bhavika Pravin Jain, Chang Min Park,
Keshav Sharma, Karthik Dantu, Steven Y. Ko, Lukasz Ziarek
Department of Computer Science and Engineering
University at Buffalo, The State University of New York
{tki, agsimeon, bhavikap, cpark22, keshavsh, kdantu, stevko, lziarek}@buffalo.edu

ABSTRACT

This paper proposes a new technique that enables open innovation in mobile platforms. Our technique allows third-party developers to modify, instrument, or extend platform API calls and deploy their modifications seamlessly. The uniqueness of our technique is that it enables modifications *completely at the app layer* without requiring any platform-level changes. This allows practical openness—third parties can easily distribute their modifications for a platform without the need to update the entire platform. To demonstrate the benefits of our technique, we have developed a prototype on Android called *Reptor* and used it to instrument real-world apps with novel functionality. Our evaluation in realistic scenarios shows that Reptor has little overhead in performance and energy, and only modest overhead in memory usage that ranges from 0.6% to 10% for the observed worst cases.

Keywords

API virtualization; Android app instrumentation; Android platform instrumentation; platform openness

1. INTRODUCTION

Openness drives innovation. There are numerous examples in computer systems where openness has led to an explosion of new advances. Perhaps the most recent example is Software-Defined Networking (SDN) [23], which has unlocked the control interface of commodity routers and switches. This has enabled third-party developers to easily write and deploy software controllers for networking hardware. Many other examples exist [8, 26, 29, 5, 9], where openness has enabled realization of novel ideas.

By openness we broadly mean two properties—(1) *extensibility*, i.e., the ability for third-party developers to *modify, instrument, or extend* an existing platform, and (2) *deployability*, i.e., the ability for end users to *easily deploy* third-party platform modifications. Previously, extensibility on desktop platforms has been studied extensively at all layers, i.e., the kernel (e.g., Kprobes [8]), system libraries (e.g., Detours [26]), and the application layer (e.g., AspectJ [29]). It has also been studied across different subsystems, e.g., FUSE [5] for filesystems and NetFilter [9] for networking. On the other hand, deployability for end users has not been studied in detail as it is less of a concern in desktop environments. End users have low-level access (such as shell or root access) which allows them to easily download and deploy third-party platform modifications. For example, FUSE allows an end user to download a new filesystem such as SSHFS [12] and deploy it easily.

MobiSys '17, June 19–23, 2017, Niagara Falls, NY, USA.

DOI: http://dx.doi.org/10.1145/3081333.3081341

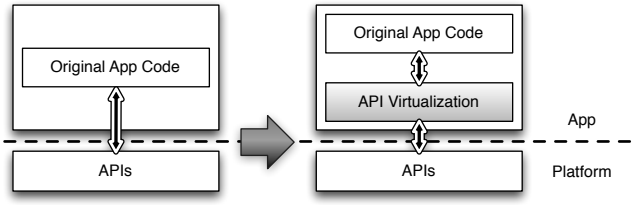

Figure 1: App Transformation with API Virtualization

However, this is not the case in modern mobile platforms. Deployability is severely restricted, and regular end users cannot easily use any third-party platform modifications. The default interaction available between a regular user and an off-the-shelf mobile platform is installing and using apps downloaded from online stores (such as Google Play). Platform modifications typically require a vendor-controlled update mechanism (such as Google's Over-The-Air), low-level system access to overwrite system files, or an ability to recompile and install a new platform image—none of which is realistic to expect for a regular user.

Given this constraint, deploying third-party platform modifications on a mobile platform requires an *app-layer* approach, where all platform modifications are contained *in an app* and those modifications are embedded in the app *statically*. This way, we can achieve both extensibility and deployability—we can use existing app distribution mechanisms (such as Google Play) to disseminate third-party platform modifications, without requiring any other form of access to user devices. This app-layer approach has an additional benefit that per-app customizations are possible, i.e., different apps can use different platform modifications on a single mobile device.

This observation brings us to the central idea of this paper—enabling openness through *API virtualization*. We introduce a shim layer between an app and its platform (Figure 1), where we allow third parties to customize platform API call behavior and add new functionality. We then statically inject this virtualization layer into an app for ease of deployment, leveraging bytecode instrumentation. Example use cases could be automatically augmenting local storage read/write calls to enable seamless cloud backup [20], improving energy efficiency for always-on sensing [36], or seamless HTTP-to-HTTPS migration and vendor-specific library switching as we describe in Section 4.

To concretely demonstrate the benefits of our idea, we have developed a prototype called *Reptor* on Android. The primary challenge we address is *correctly enforcing* an app to use our API virtualization layer. The exact description of this challenge entails an involved discussion, which we defer to Sections 2 and 3. Briefly, the challenge stems from the constraint that all platform modifications should occur in an app, coupled with the need to comply with a rich set of features in Android and its default language, Java, such as sub-typing, polymorphism, and callbacks. To the best of our knowledge, this challenge of correct enforcement has not been addressed in the literature. As we detail in Section 2, previous instrumentation systems (e.g., SIF [24] and RetroSkeleton [21]) as well as systems that leverage instrumentation (e.g., AppInsight [33], VanarSena [32], and MobileHub [36]) do not provide techniques for correct enforcement.

We make three contributions. First, we show that an app-layer approach to platform openness is *feasible* by designing and implementing a mechanism that enables it on Android (Section 3). Second, we show that our approach is *useful* by implementing three use cases as well as discussing how our approach could improve previously-proposed systems (Section 4). Third, we show that our approach is *practical* by evaluating our prototype with our own micro-benchmark apps as well as 1,200 real apps downloaded from Google Play (Section 5). Our prototype has correctly processed about 234.8M lines of code to produce the results presented in this paper, and shows modest overhead—for real apps we see almost no difference in performance, 0.6% to 10% increase in memory usage for the observed worst cases, and no statistically significant overhead in energy usage.

2. API VIRTUALIZATION CHALLENGES

Our goal in enabling API virtualization on Android is to allow third-party developers to provide *alternative platform API implementations with novel functionality*. For example, if a third-party developer has an implementation for a novel gesture recognition algorithm, we want to be able to provide that as an alternative to the default gesture recognition provided by Android.

We target Android platform APIs for extensibility because all functionality of Android is provided by the platform APIs for an app—they provide traditional OS service APIs (e.g., file system operations), APIs to access system resources (e.g., sensors), sensor data analytics APIs (e.g., gesture recognition APIs), high-level app support APIs (e.g., HTTP APIs), and convenient data structure APIs from standard Java (e.g., hash maps). Conceptually, they are analogous to the system call interface for a traditional OS. Thus, allowing alternate implementations for the platform APIs enables extensibility of the platform.

However, when designing an extensibility mechanism for Android, we need to be conscious about deployability as discussed in Section 1. Many existing techniques cannot be adopted easily because they fall short in this regard. Examples include library overloading using `LD_PRELOAD`, Java class loaders, syscall library hooks, Java VM hooks, proxies [28], AspectJ [29], etc. These techniques require either low-level access or modifications to the underlying platform, resulting in limited deployability for Android.

Thus, we devise an *app-layer* approach that provides both extensibility and deployability where (1) we *statically inject* an alternative third-party API implementation into an app, and (2) we make sure that the app *correctly* uses the injected API implementation instead of the original API implementation from Android. Consequently, there are two challenges we need to address: (1) injecting a third-party platform API implementation into an existing app, and (2) correctly enforcing the app to use the injected code. Among these, the first challenge (code injection) is a solved problem and can be done by Java bytecode rewriting [37, 21, 24].

```
1  public class MyStream extends FileOutputStream {
2    public void write(int b) {
3      // Overriding
4    }
5  }
6  public class Main {
7    public void main(String args[]){
8      MyStream obj1 = new MyStream("/tmp/test");
9      mainWrite(obj1, 10);
10   }
11   public void mainWrite(
12                FileOutputStream obj, int b) {
13     obj.write(b);
14   }
15 }
```

Figure 2: Sub-Typing Example

While code injection can easily be solved, correctly enforcing the use of injected code is not. To illustrate this challenge, suppose an app calls `FileOutputStream.write()`, a platform API method. Also suppose that we want to replace it with a third-party `write()` implementation that augments the original `write()`. For example, it could perform a local write as well as a remote write for seamless cloud backup.

At first glance, it may seem that all we need to do is *call replacement* that consists of two steps—(1) search every occurrence of `obj.write()`, where `obj` is of type `FileOutputStream`, and (2) replace it with a call to the third party's `write()` implementation. This, however, can easily result in incorrect behavior as shown in Figure 2.

In the code, there are two classes, `MyStream` and `Main`. `MyStream` extends a platform class `FileOutputStream` and overrides `write()`. `Main` initializes a `MyStream` object (in line 8) and calls `mainWrite()` (in line 9). `mainWrite()`, in turn, calls `write()` to perform a file write (in line 13).

Note that `mainWrite()` (line 11) takes `FileOutputStream` as a parameter, and not `MyStream`. In `Main.main()`, `MyStream` is passed as an argument to `mainWrite()` (line 9). This is possible because of sub-typing, i.e., `MyStream` can be cast to `FileOutputStream`.

Now suppose that we want to extend the example code to use a third-party implementation of `FileOutputStream.write()` by searching and replacing objects of type `FileOutputStream` on which we call the `write()` method (i.e., using the call replacement approach as described earlier). In the example code, there is only one such place, line 13. However, replacing this call is *incorrect*; at run time, line 9 passes `MyStream` to `mainWrite()`, and replacing line 13 means replacing `MyStream.write()`, not `FileOutputStream.write()`. We note that this is *not* an isolated problem; in general, it is not difficult to construct various examples where call replacement produces incorrect behavior.

Unfortunately, previous systems such as SIF [24] and RetroSkeleton [21] only allow call replacement and cannot enforce an app to use a third-party API implementation correctly. Moreover, many recent systems, e.g., AppInsight [33], VanarSena [32], and MobileHub [36], instrument API calls to enable selective logging, fuzzy testing, and sensor call replacement, respectively. Their instrumentation requires correct replacement of API calls but only uses call replacement that does not produce correct program behavior.

In Section 7, we further discuss previous techniques and their limitations. In general, prior work falls into two broad categories, each with a major limitation—techniques with limited deployabil-

```
1  public class NewFOS {
2    public void write(int b) {
3      // A reimplementation goes here.
4    }
5  }
6  public class MyStream extends NewFOS {
7    public void write(int b) {
8      // Overriding
9    }
10 }
11 public class Main {
12   public void main(String args[]) {
13     MyStream obj1 = new MyStream("/tmp/test");
14     mainWrite(obj1, 10);
15   }
16   public void mainWrite(NewFOS obj, int b) {
17     obj.write(b);
18   }
19 }
```

Figure 3: Class Replacement Example

ity (e.g., LD_PRELOAD, Java class loaders, Java VM hooks, proxies [28], AspectJ [29], etc.), and techniques that only support call replacement (e.g., SIF, RetroSkeleton, etc.).

3. API VIRTUALIZATION ON ANDROID

The above discussion reveals that using method calls as the unit of replacement is fundamentally incorrect, as there are many class-level features that determine method-level behavior. Consequently, our approach uses a more natural unit of replacement—*classes*. This means that (1) a third party writes a *replacement class* for a platform API class that has a new implementation of the platform class, and (2) our approach replaces all uses of the platform class in an app with this replacement class. Effectively, our API virtualization layer becomes a set of replacement classes that modify or reimplement Android platform APIs, and are injected into an app. Since a class is a natural abstraction for modular implementation, it is also well-suited as a programming interface for third-party developers.

Figure 3 extends the previous example and illustrates our use of a replacement class. In the example, there is a new class `NewFOS` that replaces `FileOutputStream` and reimplements `write()`. It is injected into the app's bytecode (we show the source for illustrative purposes). In the app classes (`MyStream` and `Main`), `NewFOS` is used instead of the original class, `FileOutputStream`. This is shown in line 6 and line 16, which are the only places where `FileOutputStream` is used in the original app code. Essentially, `NewFOS` replaces all uses of `FileOutputStream` in the app. This correctly solves the sub-typing problem described earlier as line 17 uses `MyStream`, not `NewFOS`.

Unfortunately, we cannot rely on simple examples such as Figure 3 to determine whether or not class replacement is the right approach. We must examine all possible uses of a platform class in app code, and make sure that we can correctly replace a platform class with a replacement class provided by a third party. Thus, we start our solution discussion by presenting our classification of platform class uses. We then discuss how we handle the different uses.

3.1 Platform Class Uses on Android

In order to determine all possible uses of a platform class in app code on Android, we have examined every feature described in the Android API documentation [15] as well as the Java specification [7]. We have then classified the uses of platform classes into the categories shown in Table 1. For simplicity of presentation, we use the term *class* to refer to all class-level constructs, i.e., interfaces, abstract classes, and enums, unless otherwise specified.

Our categorization achieves two purposes. First, it concisely summarizes the vast list of platform class uses. Since Java and Android have many features that enable a wide range of platform class uses, it is infeasible for us to discuss every single use in the space given for this paper. Our categorization allows us to capture sufficient details without going into each and every use. Second, our categorization shows our findings that (1) there are a few major challenges we need to address for class replacement, (2) each category represents one such major challenge, and (3) these challenges are inter-related, and it is best to *avoid* solving them individually in isolation. Initially, we came up with point solutions for different categories but had much difficulty integrating them together since our individual solutions ended up being logically incompatible. After some iterations, we have realized that *addressing the categories progressively in the order we present in Table 1* allows us to arrive at a coherent design in the end. The rest of this section discusses this progression.

There are many engineering challenges that we address in the implementation of our design. However, since these only require mechanical solutions, we refrain from discussing those in detail and only briefly mention them when needed. To prove that we have indeed addressed other engineering challenges not discussed in this paper, we take a pragmatic approach where we use 1,200 real apps downloaded from Google Play as test cases. In Sections 4 and 5, we show that these apps work correctly after our transformation, and the total lines of code we have processed is more than 234.8M. In Section 6, we further discuss the experiences and implications of our evaluation strategy with real apps.

3.2 Handling Basic Uses of Platform Classes

We first discuss the most basic uses of platform classes, i.e., (both local and class) variables, parameters, and return types. Together, they pose a challenge for class replacement due to a dependency problem they raise. To illustrate this, consider one method in a platform class, `AtomicFile`:

```
void finishWrite(FileOutputStream str)
```

As shown, `FileOutputStream` is used as a parameter in `finishWrite()`. Using this, the following code shows variable and parameter uses of `FileOutputStream`:

```
FileOutputStream fos = new FileOutputStream(...);
AtomicFile af = new AtomicFile(...);
af.finishWrite(fos);
```

Now suppose that we are replacing `FileOutputStream` with `NewFOS` as done before in Figure 3. The code becomes:

```
NewFOS fos = new NewFOS(...);
AtomicFile af = new AtomicFile(...);
af.finishWrite(fos); // does not work
```

However, this code does not work; `finishWrite()` expects `FileOutputStream`, not `NewFOS`. To make this work, the first thing one could try might be modifying `finishWrite()` to use `NewFOS`. However, we cannot do this—`AtomicFile` is a platform class provided by one of the Android system libraries, which we cannot modify.

Although the above code only shows variable and parameter uses of a platform class, the same problem occurs with return types as well. The broader challenge is that replacing a platform class has a dependency problem, since a platform class we want to replace can

Category	Description	Discussion
Basic	Uses of platform classes in variables, parameters, and return types	Section 3.2
Class hierarchy	Uses of platform classes in inheritance, `super` calls, and type-casting	Section 3.3
Callback	Uses of platform classes in Android lifecycle callbacks, and event (UI, sensor, etc.) callbacks	Section 3.4
Interface return	Uses of platform interfaces returned by platform calls	Section 3.5
Runtime	Dynamic uses of platform classes, i.e., reflection, `synchronized`, reference access, and `instanceof`	Section 3.6
Exception	Uses of platform exception classes	Section 3.6
JNI	Uses of JNI	Section 3.6

Table 1: Platform Class Use Categories

```
1  public class DummyAtomicFile {
2    AtomicFile realObj;
3    // Constructor
4    public DummyAtomicFile() {
5      realObj = new AtomicFile();
6    }
7    // Wraps the original method
8    public void finishWrite(NewFOS str) {
9      Object realStr = str.getRealObj();
10     return realObj.finishWrite(realStr);
11   }
12   // Returns the instance of the original type
13   public Object getRealObj() {
14     return realObj;
15   }
16 }
```

Figure 4: Dummy Class Example

be used as a parameter or return type in other platform classes that we cannot modify.

Solution—Dummy Classes: We solve the above problem by creating an additional set of replacement classes called *dummy* classes. Using them, we replace the target platform class we want to replace as well as other platform classes that use the target platform class as parameters and/or return types. This is inevitable as we cannot modify other platform classes.

Essentially, a dummy class is a simple wrapper around a platform class and uses replacement classes as parameters and return types. Figure 4 shows an example dummy class that replaces `AtomicFile`. As shown in line 8, `finishWrite()` is a simple wrapper over the original, but uses `NewFOS` instead of `FileOutputStream`. All other methods also simply wrap the original methods. For this wrapping to work, it holds a reference to an instance of the original class (line 2).

The following code snippet shows how the dependency problem illustrated above is solved with dummy classes:

```
NewFOS fos = new NewFOS(...);
DummyAtomicFile af = new DummyAtomicFile(...);
af.finishWrite(fos);
```

It works correctly as expected.

A dummy class (such as `DummyAtomicFile`) and an actual replacement class (such as `NewFOS`) share the same code structure in our design. They all wrap original methods and hold a reference to an instance of the original type. The only difference is that an actual replacement class contains third-party's reimplementation for some or all of the methods; if a third party does not reimplement a method, it simply calls its corresponding original method.

Now, it is not difficult to see that there is a rippling effect—replacing `AtomicFile` with `DummyAtomicFile` affects other platform classes that use `AtomicFile` as parameters or return types, just as `NewFOS` affects `AtomicFile`. That is, if we use a dummy class as described above, it leads us to create even more dummy classes.

Our initial idea was to track this rippling effect and determine the exact set of platform classes that need dummy classes, in a hope that we would minimize the number of new classes we create. However, we observed that most real apps require us to create dummy classes for almost the entire set of platform classes. Thus, we have decided to just create a dummy class for *each and every* platform class. Effectively, this means that our API virtualization layer is a layer that consists of dummy and actual replacement classes.

This approach may seem to incur potentially large overhead. However, compilers are very good at eliminating dead code as well as inlining small methods—our implementation in fact performs basic dead code elimination. Further, using dummy classes paves the way for the rest of the refinements, i.e., if we use dummy classes, most of the other refinements logically follow. The runtime overhead of these dummy classes is modest in terms of latency, energy, and memory, which we show in Section 5. Nevertheless, there is an inevitable static cost of code size increase, which we also show in Section 5.

For the simplicity of presentation, we will simply use "a replacement class" to refer to either a dummy or an actual replacement class in the rest of the paper.

3.3 Handling Class Hierarchy Uses

Our second category is platform class uses related to Java's class hierarchy. This comes second in our category list for a reason, since addressing the previous category gives us a natural starting point of addressing this category.

In Java, every class is part of one large class hierarchy, where `java.lang.Object` is the root. This enables many convenient features of object-orientation: (1) *inheritance*—a class inherits all its parent class's methods and fields without defining or implementing them, (2) *super calls*—even if a class overrides a parent class's method, it can still call the parent class's implementation of the method by calling it with the keyword `super`, and (3) *type casting*—a class can be cast to any of the classes up in the hierarchy and vice versa. A platform class can be used in all these cases.

Ideally, a replacement class for a platform class should be a *complete* replacement in the class hierarchy as well, seamlessly enabling all features of the hierarchy such as inheritance, super calls, and type casting. Otherwise, we need to devise ad-hoc solutions for point cases, which is not a principled way of solving the problem at hand.

Unfortunately simply replacing a platform class does not achieve complete replacement in the class hierarchy. To illustrate this, suppose we have a replacement class `NewStream` that has a reimplementation of `OutputStream`, a platform class. Figure 5 shows the original class hierarchy on the left. Now, consider the following code snippet:

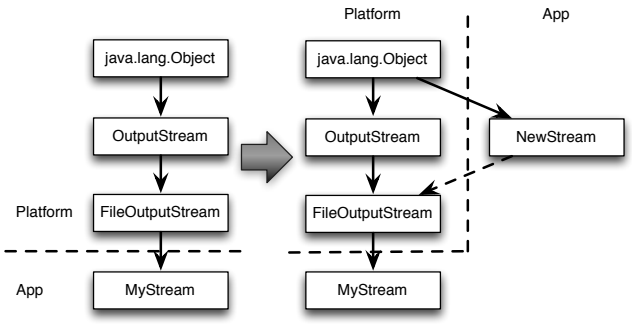

Figure 5: Class Hierarchy Comparison

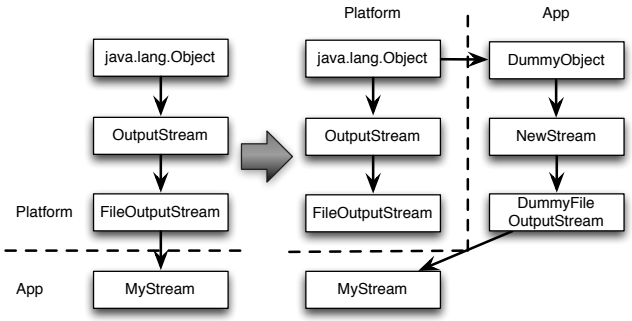

Figure 6: New Mirrored Class Hierarchy

```
MyStream ms = new MyStream();
FileOutputStream fos = (FileOutputStream) ms;
OutputStream os = (OutputStream) fos;
```

Simply replacing `OutputStream` with `NewStream` would produce:

```
MyStream ms = new MyStream();
FileOutputStream fos = (FileOutputStream) ms;
NewStream os = (NewStream) fos; // does not work
```

However, the type casting done in the last line does not work; unlike `OutputStream`, `NewStream` is not a parent class of `FileOutputStream`.

To make this code work, it may seem that we only need to make `FileOutputStream` a subclass of `NewStream`. However, we cannot do this since `FileOutputStream` is once again a platform class, which we cannot modify. Figure 5 gives a different view on this using two class hierarchies—the original class hierarchy on the left and the new hierarchy in question on the right. As shown by the dotted arrow, in order to make `NewStream` a drop-in replacement for `OutputStream` in the class hierarchy, we need to make `FileOutputStream` a subclass of `NewStream` instead of `OutputStream`, which we cannot.

Solution—Class Hierarchy Mirroring: To address the above problem, we use *class hierarchy mirroring* that exactly replicates the original *platform-side* class hierarchy in the *app side*. This is possible since we create replacement classes for all platform classes anyway, and we can mirror the entire platform class hierarchy by connecting the replacement classes the same way. Figure 6 illustrates this.

With this class hierarchy mirroring, we can correctly address the problem described above; in addition to `NewStream` replacing `OutputStream`, we have `DummyFileOutputStream` replacing `FileOutputStream`. These replacement classes mirror the original hierarchy in the app side, and correctly make the code work as follows:

```
MyStream ms = new MyStream();
DummyFileOutputStream fos =
    (DummyFileOutputStream) ms;
NewStream os = (NewStream) fos;
```

```
1  // This is the original.
2  public class MyActivity extends Activity {
3    protected void onCreate(Bundle savedState) {
4      super.onCreate(savedState);
5    }
6  }
7  // After using replacement classes
8  public class MyActivity extends ActivityRep {
9    protected void onCreate(DummyBundle savedState) {
10     super.onCreate(savedState);
11   }
12 }
```

Figure 7: Callback Method Example

3.4 Handling Callback Uses

Our third category is callbacks, one of the prominent ways of using platform classes on Android. Callbacks require app developers to create subclasses of callback-defining platform classes, and implement them. They are critical in Android apps since they are used in essential functions such as handling app lifecycle and UI events.

We present this category after discussing the previous two categories for a reason—we cannot consider callback uses of platform classes in isolation since they are tied together with basic and class hierarchy uses. Figure 7 illustrates this with two versions of code. The top one shows original app code, which is in fact a much-used code structure in Android; `MyActivity` (an app class) extends `Activity` (a platform class), and implements `onCreate()` (a callback), which expects `Bundle` (a platform class) as a parameter. This means that basic, class hierarchy, and callback uses of platform classes are all occurring together. This also demonstrates that we cannot consider individual categories separately.

Now, the bottom code is a version that uses our design described so far. It assumes that `ActivityRep` is a replacement class provided by a third party, and replaces `Activity` with `ActivityRep` and `Bundle` with `DummyBundle`.

However, this causes two problems for callbacks. First, we now break the relationship between `MyActivity` and `Activity`; `MyActivity` is now a subclass of `ActivityRep` instead of `Activity`. This is a problem as it violates the callback semantics of Android—`MyActivity` must be a subclass of `Activity`, either directly or indirectly. This is the only way for Android to correctly recognize it as a class that implements callbacks defined in `Activity`.

The second problem is that `onCreate()` now has a different method signature from the original one, since it accepts `DummyBundle` instead of `Bundle`. This is another violation of the callback semantics; Android passes a `Bundle` object, not a `DummyBundle` object when it calls `onCreate()`.

Solution—Compiler-Driven Selective Code Rewriting: To preserve Android's callback semantics, we *need* to retain the relationship between a callback-defining platform class and any app class that implements it. But as illustrated above, this conflicts with our current design.

To resolve this conflict, we devise a strategy where we retain all callback relationships, while still incorporating the uses of replacement classes. Figure 8 demonstrates how we retain callback

```
1  // After handling callbacks
2  public class MyActivity extends SuperRep {
3    protected void onCreate(Bundle savedState) {
4      DummyBundle dBd = new DummyBundle(savedState):
5      this.onCreateWrapped(dBd);
6    }
7    protected void onCreateWrapped(DummyBundle dBd) {
8      super.onCreate(dBd);
9    }
10 }
```

Figure 8: Wrapped Callback Example

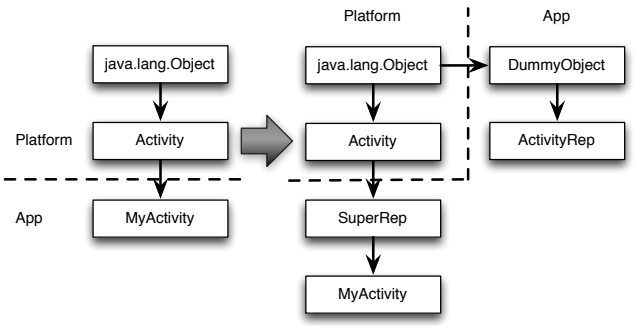

Figure 9: New Class Hierarchy for Callbacks

relationships by extending the previous example in Figure 7. Line 3 shows that `MyActivity` still implements `onCreate()` defined in `Activity`, without modifying the signature. In addition, we make `MyActivity` an indirect subclass of `Activity` as shown in line 2 (`MyActivity` extends `SuperRep`, which extends `Activity`; we discuss `SuperRep` later). In other words, it functions as a proper callback-implementing class.

We then use three techniques that *selectively rewrite* app code to incorporate the uses of replacement classes. The first technique incorporates the basic uses of platform classes, i.e., variable, parameter, and return type uses. The second technique incorporates two of the class hierarchy uses, inheritance and `super` call uses. The third technique incorporates the last of the class hierarchy uses, type casting uses. In implementing these techniques, we leverage a compiler (Soot [37]) to identify places where rewriting needs to occur. This is not difficult as it only needs to detect callback class uses in app code.

Our first technique is *callback wrapping*, as demonstrated in lines 4-5 and lines 7-9 in Figure 8. `onCreate()` calls a new method `onCreateWrapped()`, which wraps the original `onCreate()` code with replacement classes. This callback wrapping allows us to keep original callback signatures so that Android can correctly invoke callbacks, while using replacement classes in actual callback handling code.

Our second technique is to introduce a new type of replacement class called *super replacement class*. We show an example (`SuperRep`) in Figure 9. We insert this super replacement class between `MyActivity` and `Activity` to correctly handle two of the three features of the Java class hierarchy—`super` calls and inheritance. Since `SuperRep` is now a parent class of `MyActivity`, it intercepts all invocations of `super` calls and inherited method calls. This means that we can put third-party code that replaces `Activity` *inside* of `SuperRep`, so that all inherited calls and `super` calls in `MyActivity` use the third-party code correctly.

However, there are two subtleties involved further. First, there are now two replacement classes for a single third-party reimplementation for a platform class, e.g., `SuperRep` and `ActivityRep` for `Activity`. In order for both replacement classes to use the same third-party code, we ask developers to write static methods, as described in Section 3.7. Both replacement classes call these static methods to use the same code. We note that a super replacement class exists only for a callback-defining platform class, and as part of the class hierarchy.

Another subtle issue is that we cannot override `final` methods in a super replacement class to replace them. For example, in `SuperRep` we cannot override `final` methods defined in `Activity`, which prevents us from replacing them. In order to overcome this issue, we add a common suffix to all `final` methods when we put them in a super replacement class. We then rewrite all call sites that make inherited or super calls to `final` methods.

Our final technique for incorporating replacement classes is *explicit casting* between an app class and a platform class. This enables the remaining feature of the Java class hierarchy (type casting) with callbacks. To illustrate this strategy, consider once again the class hierarchy example in Figure 9. If an app originally casted `MyActivity` to `Activity` in its code, our design described so far would try casting `MyActivity` to `ActivityRep` since we replace `Activity` with `ActivityRep`. However, this casting does not work since there is no subclass relationship between the two classes.

Thus, we *explicitly* create an instance of the corresponding replacement class and rewrite the casting statement as follows (we assume `myAtv` is of type `MyActivity`):

```
// Original
Activity atv = (Activity) myAtv;

// After API virtualization
ActivityRep atv = new ActivityRep(myAtv);
```

Conversely, when it gets cast back to the original type, we use the `getRealObj()` method in the replacement class to get the original object as follows:

```
// Original
MyActivity myAtv = (MyActivity) atv;

// After API virtualization
MyActivity myAtv = (MyActivity) atv.getRealObj();
```

Technically, explicit casting could be used in general cases, i.e., we could use explicit casting instead of mirroring the platform-side class hierarchy. However, this incurs unnecessary overhead of creating a new replacement class instance whenever there is casting. Thus, we still mirror the class hierarchy and only use explicit casting in a limited fashion.

3.5 Handling Interface Return Type Uses

Our fourth category is related to interfaces, another feature of Java used for polymorphism that only contain class and method definitions. An interface must be implemented by a concrete class, and one cannot instantiate an interface. This poses a challenge for class replacement, since the return type of a method in a platform class can be an interface.

For example, a platform class `Context` has `getSharedPreferences()` that returns a platform interface `SharedPreferences`. `SharedPreferences` is a popular data storage option provided by Android. If we create a replacement class for `Context`, it will be:

```
public class DummyContext {
  DummySharedPreferences getSharedPreferences(...){
```

```
    SharedPreferences sp =
        realObj.getSharedPreferences(...);
    return new DummySharedPreferences(sp);
  }
}
```

However, this code does not work as is; `SharedPreferences` is an interface and we need to make its corresponding `DummySharedPreferences` also an interface to mirror the class hierarchy. This means that we cannot instantiate `DummySharedPreferences` as it is done above.

Solution—Using Concrete Dummy Classes: In order to solve the problem, whenever we mirror an interface with a replacement interface, we also create a concrete class that implements it. For example, for `DummySharedPreferences`, we create `ConcreteDummySharedPreferences` class that implements it. The only case where a concrete class is used is when a platform method returns an interface. For example, the replacement class for `Context` becomes:

```
public class DummyContext {
  DummySharedPreferences getSharedPreferences(...){
    SharedPreferences sp =
        realObj.getSharedPreferences(...);
    return new ConcreteDummySharedPreferences(sp);
  }
}
```

This way, we can correctly use a replacement class even when a platform class method returns an interface. We note that `abstract` platform classes have a similar problem, but the solution is much simpler—for each replacement class of an `abstract` platform class, we simply make it a concrete class so we can instantiate it.

3.6 Runtime, Exception, and JNI Uses

The remaining categories we handle are runtime, exception, and JNI uses of platform classes. Among these, the runtime use category includes reflection, `synchronized`, reference access, and `instanceof`. For reflection, we leverage the fact that a reflection call uses a plain string to identify which class to reflect on. We add extra code that, at run time replaces a class string with the string for the corresponding replacement class. The following (simplified) code demonstrates this:

```
// Original (a call to java.lang.System.exit())
String str0 = "java.";
String str1 = "lang.System";
Class c = Class.forName(str0+str1);
Object t = c.newInstance();
Method m = c.getDeclaredMethod("exit", ...);
Object o = m.invoke(t, ...);

// After API virtualization
// Assume that str0 and str1 are replaced already
DummyClass c = DummyClass.forName(str0+str1);
DummyObject t = c.newInstance();
DummyMethod m = c.getDeclaredMethod("exit", ...);
DummyObject o = m.invoke(t, ...);
```

In our `DummyClass.forName()`, we implement logic that replaces a passed-down string (e.g., `str0+str1`) with its replacement class string. Since there are only a few prefixes for platform API classes (e.g., `android.`, `java.`, etc.), we can easily identify at run time if a string is for a platform class. In our implementation, all replacement classes use the same prefix (`reptor.`), and we simply prepend our prefix. We emphasize that this is different from statically replacing a string with another string. There are many ways to compose a string dynamically (e.g., as shown in the example above), and it is generally difficult to infer a string statically. We are able to handle reflection gracefully, since we replace and instrument reflection classes.

For all other runtime uses, i.e., `synchronized`, reference access (e.g., using $==$ or $!=$ to compare object references), and `instanceof`, we call `getRealObj()` to use the actual instance and the original type. This way, we preserve the semantics of app code.

We also wrap all exceptions, but use original exceptions whenever necessary to preserve Java exception semantics. This means that when app code has a try-catch block, we do not change the catch statement, but add extra code to wrap the caught exception before it is used in the rest of the code. On the other hand, when app code throws an exception, we use `getRealObj()` to throw the actual exception. The following code demonstrates this:

```
// Original
try {
  ...
} catch (IOException e) {
  e.printStackTrace();
  throw e;
}

// After API virtualization
try {
  ...
} catch (IOException e) {
  DummyIOException de = new DummyIOException(e);
  de.printStackTrace();
  throw de.getRealObj();
}
```

For JNI calls, we still wrap them from the Java side but do not touch native code. We do this since Android is primarily a Java platform, and our focus is virtualizing the Java APIs of Android. Other techniques exist that give the ability to intercept native API calls [38, 17, 39], and we could potentially leverage those as well.

3.7 Implementation

Our prototype called *Reptor* implements our design. It uses Soot [37] for injecting replacement classes and rewriting bytecode. There are two noteworthy features of our prototype. First, it asks a third-party developer to provide a reimplementation for an API method as a static method, for which we provide a template. For example, a third-party developer replaces `Activity.onCreate()` by filling the following template:

```
public class ActivityTemplate {
  static void onCreate(Activity is, DummyBundle b){
    // Third-party code
  }
}
```

This static method is called within both a regular replacement class and a super replacement class as discussed in Section 3.4. The first parameter is always a reference to a real object of the original type (e.g., `Activity`), so that a third-party developer can call the original method if necessary. The rest of the parameters use replacement classes, and third-party developers can use both replacement and original types in their code as they see fit. One limitation is that since we inject a third-party reimplementation into an app, it cannot perform any privileged operation; however, this is inevitable for an app-only approach.

The second noteworthy feature of our prototype is that it sometimes generates multiple DEX files. DEX is a bytecode format that Android uses, and has a limitation—it only allows 64K methods to be included in a single DEX file. Since we create replacement classes, sometimes we go over this limit. If that happens, we generate multiple DEX files [3].

4. USE CASES

We demonstrate the extensibility Reptor affords by implementing three use cases, and discussing how we can improve the correctness of previous systems. We acknowledge that these use cases can be developed independently; thus, our primary purpose of the use cases is showcasing that Reptor is a general tool that makes it easy, correct, and complete for a third party to add new platform-level functionality.

4.1 Vendor-Tied Library Switching

There is a growing concern that Google has started to offer critical functionality *not* as part of stock Android but as part of a proprietary, closed-source library called "Google Play Services," in order to maintain their competitive edge [16]. It provides important features such as location services (maps), search, in-app purchasing, etc. Although it is installed as a separate system app, it works much like built-in platform APIs—a large number of popular Android apps do not work if Google Play Services is not installed. Even worse, it does not work on non-Google-approved devices such as Amazon Fire. If an app is developed with it, the app does not run on Amazon devices. To alleviate this, a few vendors such as To alleviate this, a few vendors such as Amazon and Samsung offer compatible, alternate libraries. However, a developer now has the burden of implementing the use of multiple libraries.

Given this status quo, our first use case is to automatically transform an app developed with Google's library to use Amazon's library. It even allows a *user* with an Amazon device, without any help from original developers, to download an app that uses Google's library, transform it to make the app use Amazon's library instead, and run it on the user's Amazon device. To demonstrate this, we have implemented a mechanism that replaces Google Maps APIs with Amazon Maps APIs. Although Amazon Maps APIs provide matching classes for Google Maps classes, switching from Google's to Amazon's is not a simple matter of replacing Google API calls with Amazon API calls; in order to use Google Maps, an app must not only make API calls, but also provide callbacks to receive location updates—this is done by extending `abstract` classes or implementing interfaces defined in the Google APIs. Thus, proper replacement of all classes is necessary.

Our test app for this use case is Airbnb and we have posted a video [1] that demonstrates this use case. The original Airbnb app was not developed to use Amazon Maps, but our instrumented Airbnb app uses Amazon Maps on an Amazon Fire HD tablet.

4.2 Runtime Permission

We have implemented a runtime permission mechanism that asks a user to grant a permission to an app at run time, mirroring the iOS permission model. Recent versions of Android have added a similar runtime permission mechanism, introduced from version 6.0 [11]. Using Reptor we can enable such a mechanism for apps running on older Android versions. Google reports that the market share of Android 6.0 is 24% as of November 2016 [4], which means that the majority of devices do not benefit from runtime permissions.

With Reptor, we can replace sensitive API calls, wrap them with a permission-asking dialog, and ask a user for a permission. We have posted a video [14] that demonstrates this use case using Twitter app running on Android 4.4. Our instrumented Twitter app asks a user to grant a permission to access the Internet at login, unlike the original version that does not ask for any runtime permission.

To enable this, we have mainly replaced `URLConnection.connect()`, even though typical apps (including Twitter) often use `HttpURLConnection.connect()` to access the Internet. We do this because, (1) `HttpURLConnection` inherits `connect()` from `URLConnection` and (2) other classes such as `HttpsURLConnection` (used for HTTPS) also inherit `connect()` from `URLConnection`. Since Reptor correctly handles inheritance, we can uniformly enforce the runtime permission for both HTTP and HTTPS by replacing `URLConnection.connect()`.

4.3 HTTP-to-HTTPS Translator

Most web browsers such as Firefox, Chrome, and Opera provide an extension that automatically switches many major websites from HTTP to HTTPS for more secure browsing. However, Android platform does not provide such functionality for Android apps. Thus, we have implemented an HTTP-to-HTTPS translator, which automatically makes apps use HTTPS instead of HTTP if HTTPS is available on a requested URL. To implement this, we replace `URL` class so that it returns an HTTPS object instead of an HTTP object. Although previous work [21] introduces this use case, it in fact requires correct API replacement. We have posted a video for this as well [6].

4.4 Improving Previous Systems

As mentioned in Section 2, previous systems [33, 32, 36] use call replacement to augment certain API calls. Using Reptor, these systems can automatically get correct augmentation. In addition, previous instrumentation tools for Android [24, 21] discuss use cases where in fact correct API replacement is necessary. These include an API call timing profiler, an ad blocker, etc. Reptor can provide correct API replacement for these use cases as well. Lastly, BlueMountain [20] discusses a vision of replacing file system API calls to automatically add advanced storage functionality. An example can be automatic cloud integration as mentioned earlier, where local storage read/write calls are augmented to perform not only local operations but also cloud operations. Reptor can replace storage APIs for BlueMountain.

5. EVALUATION

This section characterizes the overhead of Reptor with microbenchmark apps of our own as well as 1,200 real apps downloaded from Google Play. Since our approach creates replacement classes for all platform classes, we have an extra layer that wraps the Android APIs and creates extra objects of replacement class types. Thus, we primarily measure the raw overhead of this extra layering, i.e., the overhead of passing all API calls through our API virtualization layer that does not implement any extra functionality.

We use Samsung Galaxy Nexus (1GB RAM, 64MB per-app heap size unless otherwise mentioned) running Android 4.4 to measure our run-time overhead. This is an older version of Android, and we detail our reasons for using this in Section 6. For our app instrumentation, we run Reptor on a desktop PC with a 3.10 GHz Intel Core i5-2400 CPU, 16GB of RAM, and a single 7200 RPM hard disk. We have measured all our instrumentation-time overhead in this setting.

Reptor has processed roughly 234.8M lines of code to produce the results presented in this section. We have verified that the runtime behavior is correct for every app Reptor transforms for our

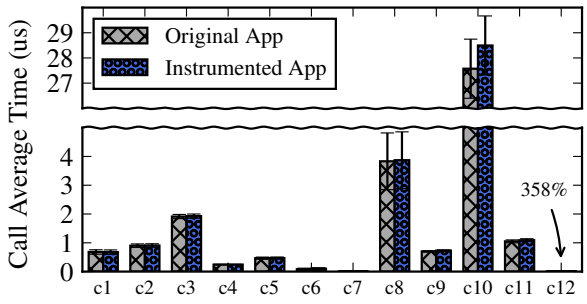

Figure 10: Call Latency for Regular Platform Calls

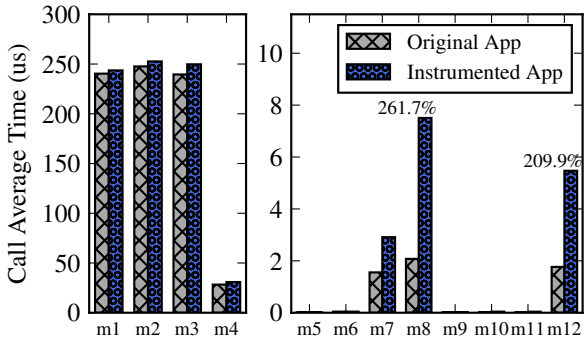

Figure 11: Call Latency for Memory Operations

evaluation, using our automated UI testing system described in Section 5.4 as well as deep manual code introspection.

5.1 Latency Overhead Characterization

We first characterize our latency overhead with a suite of micro-benchmark apps. There are two types of calls we measure—regular platform API calls that just go through our replacement classes, and API calls that involve memory operations.

Latency for Regular Platform Calls: For measuring the overhead of regular API calls, we have written an app that calls eleven platform methods from five categories: device information, network, sensing (GPS), storage, and reflection. Figure 10 shows the result. The methods in the device information category are `getLine1Number` (c1) and `getDeviceID` (c2). The methods in the network category are `getCellLocation` (c3) and `isWifiConnected` (c4). The method in the sensing category is `getLastKnownLocation` (c5). The methods in the storage category are `putValueInPreference` (c6), `getValueFromPreference` (c7), `putValueInFile` (c8), `getValueFromFile` (c9), `putValueInSQLite` (c10), and `getValueFromSQLite` (c11). The method in the reflection category is `forName` (c12). Since the time consumed by an individual call is too small to measure, we make 1K invocations of each method, and consider this an atomic unit of measurement. We perform 1K such measurements for a total of 1M invocations of each method. Most calls do not show any noticeable overhead except SQLite write (c10; 3.3%) and reflection class lookup (c12; 358%). The reflection class lookup (`forName`) has the overhead of string comparison and translation as mentioned in Section 3.6. Regardless, since all the calls add less than 1 μs per call, we believe that our approach would still be feasible in practice.

Call Latency for Memory Operations: The next micro-benchmark app measures memory-related calls, including creating, initializing, and retrieving data from two data structures—an array and a `HashMap`. The app uses an integer array, a string array, and an `Object` array as well as a `HashMap`. There are twelve methods we use: `createIntArray` (m1), `createStringArray` (m2), `createObjectArray` (m3), `createHashMap` (m4), `initializeIntArray` (m5), `initializeStringArray` (m6), `initializeObjectArray` (m7), `putValueInHashMap` (m8), `retrieveIntArray` (m9), `retrieveStringArray` (m10), `retrieveObjectArray` (m11), and `getValueFromHashMap` (m12).

Since these method calls also take very little time, we combine 100 method calls as a unit of measurement, and repeat each measurement 10K times for a total of 1M calls. While the overhead for creating the data structures is negligible (the left plot in Figure 11), there is higher overhead involved in creating object arrays, and to put/get values in/from `HashMap` (the right plot in Figure 11)). Our manual analysis has revealed that the overhead for `HashMap` is because Reptor also virtualizes the key and value classes used by `HashMap`, resulting in nested levels of indirection. This is inevitable as it is necessary for the correct functioning of the data structure. While the percentage overhead seems high, we note that the overhead in absolute time is very little (4-6 μs per call on average) leading us to believe that this would also be feasible in practice. Common compiler optimization techniques, e.g., method inlining, will likely help as well.

5.2 Memory Overhead Characterization

Since our approach creates extra objects for replacement classes, it adds to the memory requirements of an app. Thus, we study the increase in memory requirements for three types of workload—compute-intensive, basic memory, and memory-intensive.

Memory Performance on the Compute-Intensive App: For the compute-intensive workload, we have written an app that calculates π to n decimal place as a result of the infinite series:
$\pi = 4*(1-1/3+1/5-1/7+1/9...+1/(2n-1)-1/(2n+1))$
For our tests, we set $n = 100000000$.

We then conduct heap size variation tests. This is performed by changing the heap size given to the app during execution. As shown in Figure 12 (a), we vary the heap size from 1MB to 64MB for both the original and the instrumented versions. The purpose is to measure how execution time varies depending on available memory. Unlike the memory-related apps discussed below, our compute-intensive app does not require a large amount of memory, hence does not exhibit much difference in performance across different heap sizes.

Memory Performance for the Basic Memory App: For the basic memory workload, we use the same micro-benchmark app from the second part of Section 5.1, which creates, initializes, and retrieves data from four data structures—an integer array, a string array, an `Object` array, and a `HashMap`.

Figure 12 (b) shows the performance of the instrumented and non-instrumented versions with varying the heap size from 8MB to 64MB. Under the heap size of 8MB, our app throws an out-of-memory exception, and we could not collect any data point. As shown, smaller heap sizes require much longer execution times to finish. This is due to the activities of the garbage collector; it gets frequently invoked to reclaim the heap. However, both versions exhibit similar overall behavior in terms of memory usage.

Memory Performance for the Memory-Intensive App: For the memory-intensive workload, we have written an app that creates a `HashMap` array with 10,000 elements, copies even elements to odd elements, deletes all even elements, creates 5,000 elements again, and assigns them to even elements. The app repeats this process 100 times. We emphasize that this has been done to create a rather *extreme case* where we comprehensively stress-test all memory-

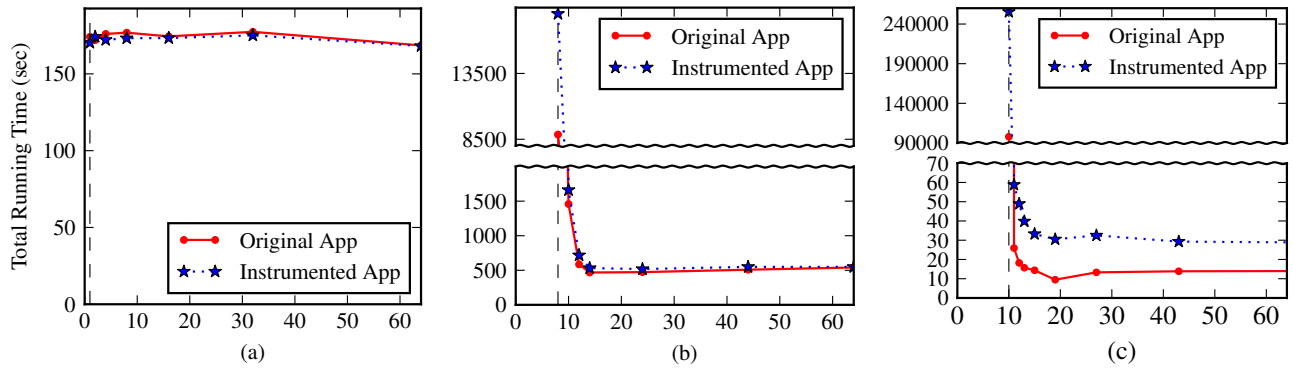

Figure 12: Heap Size Variation (X-axis: size in MB)

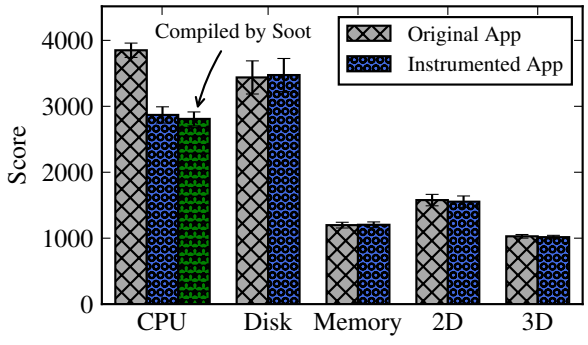

Figure 13: PassMark Benchmark App Results

related operations such as allocation, reading, writing, and deallocation. This kind of memory usage is unrealistic, but effectively fragments the memory and creates pressure on the garbage collector—it triggers the garbage collector to frequently scan the heap to keep track of live objects with active references.

Figure 12 (c) shows the results. We vary the heap size from 10MB to 64MB. Our app throws an out-of-memory exception below 10MB, which means that 10MB is roughly the minimum heap size required to run the app. Besides this, we make two observations. First, the original version takes ∼98,000 seconds at 10MB; we can extrapolate that the instrumented version roughly requires 10.5 MB of heap to finish within the same amount of time. This means that there is a slight increase in the minimum heap size requirement to maintain the same performance. Second, both the original and the instrumented versions start to stabilize at 13 MB, and the instrumented version takes approximately three times longer to finish at steady state. This is not too surprising as we have created an extreme case that intentionally stresses the memory in various ways.

5.3 Benchmark App Performance

In addition to our own micro-benchmark apps, we have downloaded and instrumented a benchmark app called PassMark [10] from Google Play. We have chosen this app since it uses Java for benchmarking code. PassMark has 19 benchmarks in 5 categories—CPU, Disk, Memory, 2D, and 3D. We show average scores based on 10 runs.

Figure 13 shows that most of the benchmarks report similar numbers except CPU. The reason is that there are six benchmarks in CPU, and three of them (random string sort, encryption, and compression) have shown a performance decrease by 40%-70%. To determine the root cause of the overhead, we have manually examined the code and determined that it is due to a difference between Android's compiler and Soot, the baseline compiler we use to implement Reptor—they produce distinct bytecode that gives different performance results. To prove this point, we have used Soot to just decompile and recompile PassMark without any of our Reptor implementation, and tested the performance. The extra third bar in Figure 13 shows the result. We can see that the CPU performance numbers are similar between the second and the third bars.

5.4 Stock App Performance

Although micro-benchmarks provide a way to gauge different types of overhead under very specific conditions, they do not necessarily mirror real-world considerations. To demonstrate the feasibility of Reptor in the real world, we have instrumented apps downloaded from Google Play, verified the correctness of our instrumentation, and characterized the instrumentation performance as well as the run-time performance. We have chosen 1,200 popular and representative apps from Google Play. Below, we first discuss our strategy for verifying instrumentation correctness. We then discuss our overhead results.

Instrumentation Correctness: With 1,200 real apps, we have verified the instrumentation correctness of Reptor. Our verification consists of two steps—instrumentation-time validation and run-time testing. First, after we instrument an app through Reptor, we pass the instrumented app through a *validator* that we have developed. Our validator checks static properties of app code, such as type matching, local variable declaration correctness, method declaration correctness, etc. In the end, this static validator makes sure that the Java bytecode produced by our instrumentation for an app is not malformed (to the best it can).

Second, we verify the run-time correctness of an instrumented app by using an *automated UI testing system* that we have also developed. Our UI testing system leverages Android's UI Automator [2] which can inspect all UI elements and their hierarchy for an app running on a device. Using this information, it generates random UI events, such as button clicks and text input, and tests the run-time behavior of an app. It also captures all logs that an app generates during testing using `adb` (Android Debugging Bridge).

With this UI testing system, we run an app both with and without Reptor's instrumentation. After that, we compare the logs produced by the instrumented version of the app to the logs produced by the original version. If we find that an instrumented app has finished its execution without crashing and has not produced any extra ex-

Category	Examples	Inst. Time Avg. (Min./Max.)	APK Size Avg. (Min./Max.)	APK Size Increase Avg. (Min./Max.)	LoC Avg. (Min./Max.)	Loc Increase Avg. (Min./Max.)
Game	Farm Heroes Saga, Glow Hockey, Turbo Driving Racing 3D	85.7s (22.5s/248.1s)	24.5M (6.2M/99.7M)	160.7K (31.0K/1.3M)	220.4K (2.0K/681.0K)	330.3K (77.0K/888.0K)
Entertainment	Xbox 360 SmartGlass, Roku, TWC TV	79.9s (22.5s/308.5s)	9.0M (655.4K/54.2M)	410.0K (5.0K/2.3M)	210.7K (4.0K/802.0K)	346.3K (102.0K/839.0K)
Media	SiriusXM, Marvel Unlimited, Merriam-Webster	60.6s (12.0s/212.9s)	6.7M (38.3K/49.8M)	366.0K (74.0K/2.0M)	149.2K (1.0K/595.0K)	266.0K (67.0K/745.0K)
Education	NeuroNation, NASA, Mobile Learn (Blackboard)	63.7s (13.1s/192.3s)	8.4M (1014.7K/45.0M)	307.2K (475.0K/1.1M)	153.2K (1.0K/591.0K)	279.4K (72.0K/803.0K)
Personalization	Backgrounds HD, Twemoji, XFINITY home	53.3s (12.4s/303.2s)	5.9M (128.8K/54.7M)	311.6K (89.0K/1.0M)	120.5K (1.0K/683.0K)	234.1K (68.0K/821.0K)
Productivity	Wunderlist, Google Now Launcher, Microsoft Office Mobile	60.9s (13.4s/228.6s)	3.4M (336.7K/27.3M)	409.0K (390.0K/1.8M)	143.2K (1.0K/804.0K)	263.2K (82.0K/772.0K)
Business	Mint, Chase Mobile, Microsoft Remote Desktop	68.4s (13.1s/247.8s)	6.5M (397.9K/46.2M)	386.3K (19.0K/1.8M)	174.9K (2.0K/735.0K)	315.6K (76.0K/902.0K)
Social	WordPress, GasBuddy, Pixlr	87.7s (18.9s/312.4s)	10.3M (892.3K/48.4M)	340.4K (64.0K/1.5M)	242.0K (2.0K/714.0K)	372.4K (96.0K/826.0K)
Total	N/A	70.0s (12.0s/312.4s)	9.4M (38.3K/99.7M)	336.4K (5.0K/2.3M)	176.8K (1.2K/804.2K)	300.9K (67.0K/902.0K)

Table 2: Instrumentation Results for 1200 Popular Apps (150 apps in each category)

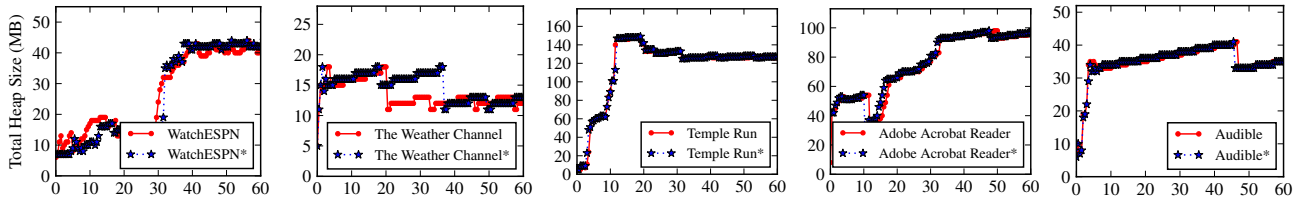

Figure 14: Heap Usage (*Instrumented apps): The x-axis shows elapsed time in seconds.

ception compared to the original, we mark that the verification has passed successfully for the app.

Using these verification steps, we have first verified that all 1,200 apps have passed our validator after instrumentation. In addition, we have verified that no instrumented app crashed during our UI testing and there is no extra exception thrown by an instrumented app compared to its original version. We have run each of 1,200 apps 10 times for 30 seconds, both with and without Reptor instrumentation. The total testing time is 100 hours.

Instrumentation Overhead: Table 2 shows our instrumentation overhead results. We have re-categorized 1,200 apps into eight categories based on Google's categorization to concisely present our results. Each category contains 150 apps. We show average, maximum, and minimum values for instrumentation times, original APK sizes, APK size increases after instrumentation, original lines of code (LoC), and LoC increases after instrumentation. We observe that the instrumentation time for each app is modest and finishes in 6 minutes or less. However, since our design generates replacement classes, there is an inevitable cost of static code size increase. On average, the APK size increases by 21.8% and the LoC by 538.9%. The large percentage increase in average LoC is partly because we have many small apps that have a far fewer number of app classes compared to the platform classes they use. In general, we expect that other standard compiler optimization techniques will help reduce our code size.

Heap Usage and Energy Consumption Overhead: For more realistic evaluation, we have manually used five apps and measured two aspects—heap usage and energy consumption. The apps are WatchESPN, The Weather Channel (TWC), Temple Run, Adobe Acrobat Reader, and Audible. We have played video clips on WatchESPN, looked up weather on TWC, played a game on Temple Run, viewed PDF files on Adobe Acrobat Reader, and played audio books on Audible.

App Name	Average (J)	Std Dev (J)
WatchESPN	781.5	20.0
WatchESPN*	790.3	17.4
The Weather Channel	170.2	13.6
The Weather Channel*	181.4	8.9
Temple Run	991.5	29.5
Temple Run*	993.0	22.3
Adobe Acrobat Reader	838.3	8.9
Adobe Acrobat Reader*	850.0	10.1
Audible	787.5	18.5
Audible*	778.0	8.9

Table 3: Energy Consumption (*Instrumented apps)

Shown in Figure 14 is the heap usage over time. We use Android tools to record heap allocation sizes every 0.3s while we run each app for 60s. While the heap usage is slightly higher for the instrumented apps, both versions show similar behavior across the five apps. At steady state (typically around 40s $\sim$ 50s), the worst-case heap usage increase we observe is 4.2% for WatchESPN, 9.5% for TWC, 0.6% for Temple Run, 8.4% for Adobe Acrobat Reader, and 0.6% for Audible.

For energy measurement, we have used a Monsoon Power Monitor and run the five apps for ten minutes five times. Table 3 shows the results. All the averages are within statistical deviations of each other indicating that there is no significant energy overhead added by our instrumentation.

We have also created a video [13] of two users playing Temple Run side-by-side. The game was instrumented to display notifications whenever a touch event occurred. We have observed that there is no noticeable delays in game play, even though we use a Galaxy Nexus released five years ago.

6. DISCUSSION

Security: A potential concern for any instrumentation tool including Reptor is that it can be abused to inject malicious code. While we acknowledge that we do not improve the status quo, we argue that malicious actors do not need Reptor to inject malicious code. There are many tools available already for creating malicious apps as evidenced by the massive quantity of malicious repackaged apps [40]. Using these tools, malicious actors are already downloading paid apps, injecting malicious code to the apps, and publishing them on online stores as free versions.

Ethics: Another common concern that instrumentation tools raise is whether or not it is ethical (or even legal) to modify apps that other developers have published. For this, we envision a new ecosystem where there are two different types of developers that *cooperate* to provide better experiences for end users—app developers and extension developers. App developers write and distribute apps, and extension developers write and distribute new platform API extensions. In this ecosystem, Reptor becomes a tool that integrates extensions and apps.

Experience with Real Apps: In the early stage of our development, we hand-crafted app code that implements common uses of platform classes, and made Reptor work correctly with it. Our hope was that by doing so, we would eventually be able to transform at least a small set of real apps. However, it turns out that real apps, even when their code sizes are small, are often non-trivial and use platform classes in a wide variety of ways. Thus, it was *required* for us to thoroughly examine the Android documentation and the Java specification to determine what we need to address—it was impossible to transform real apps until we did it.

Formal Correctness: A formal proof is the most rigorous way to show Reptor's correctness and completeness. However, it is an extremely difficult task since we essentially need a formal proof of our implementation against two highly complex systems, Java and Android. Thus, we take a more pragmatic approach and show the correctness and completeness of Reptor empirically by instrumenting over a thousand apps and two hundred million lines of code. The apps we instrument are non-trivial, real-world apps that contain complex code. By instrumenting large bodies of code we have a good estimation of completeness across Java features and Android constructs.

API 19 (Android 4.4): We started our development with API 19 roughly two and a half years ago when it was the latest release, and have not changed the version since. This is because API 19 is still a popular version and we want to keep our development environment stable. The biggest difference in recent versions is the new VM (ART), but our basic idea of using replacement classes do not have any dependency on specific VMs or versions. This is because we generate a list of all platform classes automatically by leveraging a compiler (Soot), and from the list, we create corresponding replacement classes. This can be done on any Android API version. It is our future work to transition to the latest version.

7. RELATED WORK

Traditional techniques that allow extensibility in desktop environments do not provide good deployability on mobile platforms, since they require low-level system access. These include library overloading (using `LD_PRELOAD` or Java class loaders), and syscall library or Java VM hooks. Other techniques, such as Java bytecode rewriting [37, 35, 19], I-ARM-Droid [22], SIF [24], RetroSkeleton [21], `mach_inject` [34], Detours [26], and Windows Hooks [30], enable general instrumentation specific to a language, a platform, or a combination of them. Reptor addresses a different set of challenges previously not addressed by these techniques, i.e., how to replace a platform class with an app-level replacement class on Android. In addition, Reptor provides a more general form of proxy objects (e.g. [28]) and is able to handle the situation where certain classes (e.g., platform classes in Android) are not allowed to be proxied. Aspects (e.g., AspectJ [29]) provides an ability to instrument, but requires access to all target classes to instrument them.

Aurasium [38] and Boxify [17] have proposed app-level techniques for access control and sandboxing. Aurasium intercepts the Global Offset Table (GOT) of a process, and redirects `libc` calls; Boxify monitors and intercepts inter-process communication channels and system calls. These approaches are complementary to Reptor as they provide different capabilities for intercepting native API calls.

Reference Hijacking [39] has recently proposed an app-level technique that allows an app to load custom system libraries. It intercepts the launch sequence of an Android app, loads custom system libraries, and restarts the app with the newly-loaded libraries. However, this approach has a few limitations compared to Reptor. First, system libraries are customized for and specific to vendors (e.g., Google, Amazon, or Samsung), Android versions, target devices, or some combinations of them. It is difficult, if not impossible, for a third party to keep track of the dependencies and generate their own custom system libraries for each and every dependency. Second, their technique can only be applied to system libraries, thus cannot handle Google Play Services which is installed as a system app, not a library. Third, there is also memory and storage overhead since a modified system library needs to be distributed in its entirety. Fourth, the technique relies on the particular app launch sequence of Android, and is difficult to generalize even conceptually.

Many previous systems leverage app instrumentation as a tool to achieve different goals, such as UI automation [25], performance monitoring [33], mobile network control [31], and privacy [27, 18]. These systems further prove the applicability of app instrumentation, and Reptor provides a general way of replacing API classes.

8. CONCLUSIONS

In this paper, we have proposed a new system called Reptor that enables API virtualization on Android for practical openness. Reptor allows third-party developers to modify, instrument, or extend Android platform API call behavior as needed and deploy their modifications seamlessly. The practicality of Reptor comes from the fact that it takes an app-layer approach; it does not require any platform-level changes. Reptor combines many techniques to address the challenges specific to Android, including the use of replacement classes, super replacement classes, concrete classes for interfaces, class hierarchy mirroring, as well as selective code rewriting. Using various micro-benchmarks and real apps from Google Play, our evaluation proves that Reptor incurs minimal overhead in performance and energy consumption, and only modest overhead in terms of memory.

9. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and our shepherd, Aruna Balasubramanian, for their valuable feedback. We are also grateful to Feng Shen, Kyungho Jeon, Farshad Ghanei, and Sharath Chandrashekhara for their insightful comments. This work was supported in part by the generous funding from the National Science Foundation, CNS-1350883 (CAREER) and CNS-1618531.

10. REFERENCES

[1] Airbnb Demo. https://goo.gl/tyVoQX.

[2] Android UI Automator. https://developer.android.com/topic/libraries/testing-support-library/index.html#UIAutomator.

[3] Building Apps with Over 64K Methods. http://developer.android.com/tools/building/multidex.html.

[4] Dashboards. http://developer.android.com/about/dashboards/index.html.

[5] FUSE. http://fuse.sourceforge.net/.

[6] HTTP-to-HTTPS Translator Demo. https://goo.gl/GpW21u.

[7] Java Language and Virtual Machine Specifications. https://docs.oracle.com/javase/specs/.

[8] Kernel Probes (KProbes). https://www.kernel.org/doc/Documentation/kprobes.txt.

[9] NetFilter. http://www.netfilter.org/.

[10] PassMark Performance Test. https://play.google.com/store/apps/details?id=com.passmark.pt_mobile&hl=en.

[11] Requesting Permissions at Run Time. http://developer.android.com/training/permissions/requesting.html.

[12] SSHFS. https://github.com/libfuse/sshfs.

[13] TempleRun Demo. https://goo.gl/2kYMgh.

[14] Twitter Demo. https://goo.gl/05IEIC.

[15] G. Android. Android API Guides. http://developer.android.com/guide/index.html.

[16] ArsTechnica. Google's Iron Grip on Android: Controlling Open Source by Any Means Necessary. http://goo.gl/c7VhN5.

[17] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky. Boxify: Full-fledged App Sandboxing for Stock Android. In *Proceedings of the 24th USENIX Security Symposium*, USENIX Security '15, 2015.

[18] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, 2011.

[19] A. Chander, J. Mitchell, and I. Shin. Mobile Code Security by Java Bytecode Instrumentation. In *DARPA Information Survivability Conference and Exposition II, 2001. DISCEX '01. Proceedings*, volume 2, pages 27–40, 2001.

[20] S. Chandrashekhara, K. Marcus, R. G. M. Subramanya, H. S. Karve, K. Dantu, and S. Y. Ko. Enabling Automated, Rich, and Versatile Data Management for Android Apps with BlueMountain. In *Proceedings of the 7th USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage '15, 2015.

[21] B. Davis and H. Chen. RetroSkeleton: Retrofitting Android Apps. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, 2013.

[22] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen. I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications. In *Proceedings of the IEEE Mobile Security Technologies*, MoST '12, 2012.

[23] N. Feamster, J. Rexford, and E. Zegura. The Road to SDN: An Intellectual History of Programmable Networks. *SIGCOMM Comput. Commun. Rev.*, 44(2), Apr. 2014.

[24] S. Hao, D. Li, W. G. Halfond, and R. Govindan. SIF: A Selective Instrumentation Framework for Mobile Applications. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, 2013.

[25] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. PUMA: Programmable UI-Automation for Large-scale Dynamic Analysis of Mobile Apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, 2014.

[26] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd Conference on USENIX Windows NT Symposium - Volume 3*, WINSYM '99, 1999.

[27] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, 2012.

[28] M. Keil, S. N. Guria, A. Schlegel, M. Geffken, and P. Thiemann. Transparent Object Proxies in JavaScript. In *29th European Conference on Object-Oriented Programming*, ECOOP '15, 2015.

[29] R. Laddad. *AspectJ in Action*. Manning Publications, 2nd edition, 2009.

[30] Microsoft. Windows Hooks. http://goo.gl/r32d7B.

[31] L. Ravindranath, S. Agarwal, J. Padhye, and C. Riederer. Procrastinator: Pacing Mobile Apps' Usage of the Network. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, 2014.

[32] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and Scalable Fault Detection for Mobile Applications. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, 2014.

[33] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, 2012.

[34] J. Rentzsch. mach_inject. http://goo.gl/SnOZQW.

[35] A. Rudys and D. S. Wallach. Enforcing Java Run-time Properties Using Bytecode Rewriting. In *Proceedings of the 2002 Mext-NSF-JSPS International Conference on Software Security: Theories and Systems*, ISSS '02, 2003.

[36] H. Shen, A. Balasubramanian, A. LaMarca, and D. Wetherall. Enhancing Mobile Apps to Use Sensor Hubs Without Programmer Effort. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, 2015.

[37] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, 1999.

[38] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security '12, 2012.

[39] W. You, B. Liang, W. Shi, S. Zhu, P. Wang, S. Xie, and X. Zhang. Reference Hijacking: Patching, Protecting and Analyzing on Unmodified and Non-rooted Android Devices.

In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, 2016.

[40] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.