A Programming Model and Framework for Comprehensive Dynamic Analysis on Android

Haiyang Sun†* Yudi Zheng* Lubomír Bulej*‡ Alex Villazón*§
Zhengwei Qi† Petr Tůma‡ Walter Binder*

* Università della Svizzera italiana (USI), Switzerland
† Shanghai Jiao Tong University, China
‡ Charles University, Czech Republic
§ Universidad Privada Boliviana, Bolivia

jysunhy@sjtu.edu.cn, yudi.zheng@usi.ch, lubomir.bulej@usi.ch, avillazon@upb.edu
qizhwei@sjtu.edu.cn, petr.tuma@d3s.mff.cuni.cz, walter.binder@usi.ch

Abstract

The multi-process architecture of Android applications combined with the lack of suitable APIs make dynamic program analysis (DPA) on Android challenging and unduly difficult. Existing analysis tools and frameworks are tailored mainly to the needs of security-related analyses and are not flexible enough to support the development of generic DPA tools. In this paper we present a framework that, besides providing the fundamental support for the development of DPA tools for Android, enables development of cross-platform analyses that can be applied to applications targeting the Android and Java platforms. The framework provides a convenient high-level programming model, flexible instrumentation support, and strong isolation of the base program from the analysis. To boost developer productivity, the framework retains Java as the main development language, while seamless integration with the platform overcomes the recurring obstacles hindering development of DPA tools for Android. We evaluate the framework on two diverse case studies, demonstrating key concepts, the flexibility of the framework, and analysis portability.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Frameworks

General Terms Languages, Measurement

Keywords Dynamic analysis, Dalvik Virtual Machine, Android, Java Virtual Machine, bytecode retargeting and instrumentation

1. Introduction

Android has become the dominant software platform for mobile devices and the most popular platform among mobile software developers. The growth in dominance and popularity is accompanied by a rising number and complexity of mobile applications, which in turn increase the demand for appropriate software development tools. This concerns especially program analysis tools that provide insight into the application behavior and help in tasks such as finding bugs, identifying design and performance problems, or determining important traits of third-party code.

When developing program analysis tools, particular challenges emerge with dynamic program analysis (DPA) [14]. To observe the dynamic application behavior, DPA tools need to integrate with the programming model and the execution environment of the target platform—developing DPA tools thus requires understanding the detailed operation of the execution platform, which likely occurs at an entirely different level of abstraction than application development. A good example that illustrates the challenges involved are tools that examine various security aspects of Android applications; such tools employ a variety of approaches to observe the program events of interest, ranging from modifications of the operating system kernel [6, 7, 9] and customization of the virtual machine [6, 10, 13], through various instrumentation techniques [5, 8, 12, 22], to running the virtual machine in a CPU emulator [18, 26].

As the range of approaches used by security analysis tools amply illustrates, Android lacks the means to develop DPA tools using a high-level programming model, without resorting to complex platform-specific implementation. To address this drawback, we present a new DPA development framework that remedies the lack of tooling support in the Dalvik Virtual Machine (DVM) and provides a high-level, aspect-oriented programming model for developing DPA tools in the resource-constrained environment of Android. In summary, the paper presents the following contributions:

- We modify the DVM to enable development of DPA tools. In particular, our modifications provide support for (a) handling of essential events such as class loading and initialization, (b) tracking of virtual machine, thread, and object lifecycle, and (c) exportable object identities, application event notifications, and component communication tracking. These modifications constitute an indispensable DPA tooling support on the DVM.
- We introduce a high-level programming model for DPA tools on the DVM which features an aspect-oriented specification of the desired instrumentation. The model permits a compact implementation of DPA tools even for applications consisting of multiple communicating components in multiple processes, as it is common on Android.

MODULARITY '15, March 16–19, 2015, Fort Collins, CO, USA.

http://dx.doi.org/10.1145/2724525.2724566

- We combine the DVM tooling support with the DPA programming model in a framework that features load-time application instrumentation, high code coverage (including coverage of core class libraries and component communication) and strong analysis isolation.
- We evaluate our approach on two case studies, one implementing a dynamic analysis for collecting application code coverage data on Android and Java platforms, the other implementing a dynamic analysis for detecting runtime use of permissions in Android applications. These case studies demonstrate the basic concepts of the programming model, the flexibility of our framework, and analysis portability.

This paper is organized as follows. In Section 2, we provide background essentials necessary for establishing the context of our work. We then present a high-level view of the framework, outlining the major design decisions in Section 3, and reviewing the programming model in Section 4. We complement the overall perspective with a discussion of the fundamental design and technical issues and the responsibilities tied to the deployment and execution of dynamic analyses in Section 5. We demonstrate and evaluate the framework on case studies presented in Section 6, and review related work in Section 7. Finally, we discuss the strengths and limitations of our approach in Section 8 and conclude in Section 9.

2. Background

In this section we aim at establishing the context of our work. We first provide an overview of the Android platform essentials and then review our prior work on bytecode instrumentation, modularization and composition of dynamic program analyses, and techniques for achieving high analysis coverage and strong isolation of the observed program from the analysis. Either of the following subsections can be skipped if the reader is familiar with the topic.

2.1 The Android Platform

Android applications are written in the Java programming language. To facilitate reuse and separation of concerns, applications are built from mutually interconnected application components. Each component plays a specific role and serves as an entry point to an application. The Android application framework defines four types of application components. Each type serves a distinct purpose and has a corresponding lifecycle.

Components responsible for interacting with users are called *activities*, each representing a single screen with a user interface. Activities are not supposed to perform any long-running operations or work for remote processes—this is the responsibility of *service* components. Services run in the background so as not to block user interaction with an activity, and do not provide any user interface—they are only accessible through an API provided to other components. If an application wants to expose private persistent data to other applications, it has to define a *content provider* component, which provides read/write access to the data, subject to application-defined restrictions. The last component type is a *broadcast receiver*, which allows an application to receive notifications about system-wide events.

Application components can communicate with other components using several mechanisms. For high-level communication among loosely-coupled components, the Android core framework provides a message passing system based on *intents*, messaging objects that are primarily used to start activities, to start/stop and bind services, and to broadcast system-wide event notifications. An intent identifies an action to be performed and the data to operate on. An implicit intent does not specify the name of the target component and will be delivered by the Android core framework to any component advertising the ability to perform the requested action with the requested data. An explicit intent specifies the name of the target component directly.

When a more tightly-coupled interaction with a service is required, a component can use an intent to bind to a service, obtaining in return a service object that can be used to interact with the service directly in one of three ways. When connected to a service running in the same process, a component can directly invoke the service's API methods. If a service executes in a different process, a component has to use either asynchronous messaging or remote procedure calls (RPC) to interact with the service.

At the OS level, the high-level inter-process communication (IPC) mechanisms are implemented using the *binder*, a low-level IPC mechanism for indirectly-addressed point-to-point bulk data exchange between processes. The endpoints of the communication are identified by a *binder token*, a system-wide identifier that can be shared among processes. Sending data from one process to another is called a *transaction*, and except in case of special transactions, the communication is synchronous and follows the request-reply model. The client is therefore suspended until the server provides a response.

Android is a Linux-based multi-user operating system, where applications execute in a private sandbox; each installed application has a distinct user ID, executes in a separate process, and can only access its own files. Data and resources can be shared only through the IPC mechanisms provided by the platform. In this way, the Android system implements the *principle of least privilege*, contributing to platform security [23].

Android applications, while written in Java, execute in the Dalvik Virtual Machine (DVM), which is similar to a Java Virtual Machine (JVM) in that it provides a managed execution environment with garbage collection, but conceptually implements a *register-based* instruction set architecture (ISA) in contrast to the *stack-based* ISA implemented by the JVM. Consequently, application classes need to be converted from Java bytecode to Dalvik bytecode before deployment. By default, application components belonging to the same application execute in a single DVM instance. However, any application component can be configured to execute in a separate process, and thus a separate DVM.

Launching a new DVM instance entails considerable latency due to VM bootstrap and initialization of the core libraries. Because the system needs to start and terminate processes frequently, the overhead of DVM initialization would be detrimental to performance. Android therefore starts a special process called *Zygote* early during system boot, which only bootstraps the DVM and initializes the core classes. The *Zygote* thus becomes a live snapshot of a freshly initialized DVM. Thanks to the copy-on-write implementation of the `fork()` system call, this snapshot can be efficiently duplicated when requested; that is, whenever a new DVM instance is needed, the *Zygote* process is simply forked, producing a child DVM which is almost immediately ready to execute application code.

2.2 Prior Work

This work is part of our ongoing effort aimed at simplifying the development of dynamic program analyses for modern managed platforms. Observing the runtime behavior of a program, which lies at the heart of dynamic program analysis, is generally plagued by a multitude of issues [14]. To simplify DPA tool development, we aim at providing a comprehensive framework that addresses those issues while balancing several antagonistic requirements: *simplicity*, meaning that it should take away most of the complexity not inherent to a particular analysis; *isolation*, meaning roughly that observing a program does not cause it to deviate from the path it would ordinarily take; *coverage*, meaning the ability to observe all relevant events during execution, including both user code and system code; and

performance, meaning the minimization of slowdown caused by the analysis.

In general, the developer of a DPA tool has to deal with two principal concerns at different levels of abstraction. One concern is the design and implementation of the analysis itself, which comprises the analysis algorithms, data structures for maintaining analysis state, and reaction to base program events that drive the analysis. The other concern is ensuring that the events of interest occurring in the base program will be reported to the analysis; this often requires modification of the base program code or special hooks in the execution platform.

In our previous work, we tackled various aspects of DPA tool development. In DiSL [17], we provide a domain-specific aspect language that enables rapid development of efficient instrumentations. The code to be inserted into the base program is expressed using *snippets* of Java code, with the desired location of the code specified declaratively in snippet annotations. In FRANC [1], we provide support for modularization of DPA tools, enabling composition of analysis tools from modules capturing recurring instrumentation and analysis tasks. Finally, in ShadowVM [16] we provide a system for dynamic analysis of programs running within the JVM, which achieves strong isolation and high coverage. The ShadowVM approach is based on executing the analysis code in a separate JVM, asynchronously with respect to the observed program. The observed program is instrumented using DiSL to emit the events of interests, which are then forwarded to the analysis executing in the separate JVM.

We necessarily build on our previous efforts—we rely on DiSL for instrumentation, and on the ShadowVM concept of separate analysis server for isolating analysis execution from the base program. The novelty of this work lies in enabling development of DPA tools on Android, which goes far beyond mere reuse of existing components, both in scope and effort. In targeting the Android platform, we enable development of cross-platform analyses using a common programming model, which allows using a single analysis for both Android and Java platforms. To achieve that, we address many specific design and technical issues, such as support for applications executing in multiple virtual machines, inter-component and inter-process communication, and many others, which we present in the following sections.

3. DPA Framework Architecture

The design of the DPA framework seeks to accommodate the major dynamic analysis requirements—high coverage, strong isolation, programming simplicity and acceptable performance—in the context of applications that follow the Android component model. The most visible trait of the design is the separation of analysis from the executing application, which is not common in conventional DPA frameworks, where it may appear reasonable to colocate the analysis with the executing application to achieve low overhead in accessing the application state. With applications consisting of multiple components possibly executing in separate DVM instances, colocating the analysis with the application would result in distributing the analysis state with the application components. This might not matter with analyses that can partition their state to mirror the partitioning of the application, but in general, partitioning analysis state would needlessly complicate the analysis code.

The component model also brings the existence of multiple entry points; instead of being launched through a single entry point, as is common with desktop and server applications, each application component can be activated by the system independently. The components execute in response to asynchronous messages (intents) sent by other components. In fact, an application can also integrate components from other applications by using implicit intents that do not specify a specific receiver. The potentially complex control flow

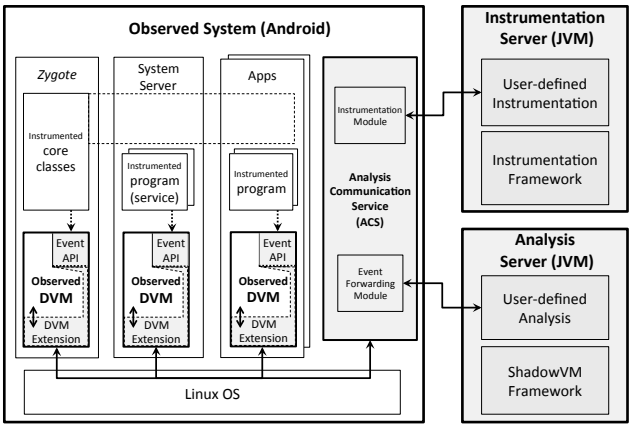

Figure 1: The ShadowVM architecture for Android.

is another factor for separating the analysis from the application. It also necessitates that the DPA framework tracks the component communication and informs the analysis about the implied control flow.

Finally, Android applications execute in what is (at least from the perspective of contemporary desktop and server platforms) a resource-constrained environment. While it is possible to execute the analysis server in the Android environment, doing so would typically make it compete for resources with the analyzed application. We therefore execute the analysis on a remote system, which is fed with the observed events. The remote execution also contributes to comfortable analysis development. The analysis code is not constrained to those Java features available on Android. It can also interface directly with the common development environments, for example for reporting results. On the DPA framework side, the remote analysis execution requires adjusting for the differences between the DVM and the JVM platforms, in particular the difference in the bytecode formats used by the two environments.

Motivated by the factors outlined above, the overall DPA framework architecture is depicted in Figure 1. At its core is the *observed system*, where the analyzed application executes in potentially multiple DVM instances. The application is instrumented to notify the analysis about events of interest through the new *event API*, added to the DVM as a part of our *DVM extension*. The communication takes place through the *Analysis Communication Service* (ACS), which forwards the notifications to a separate *analysis server*. The instrumentation itself is done by the *instrumentation server*, which takes care of injecting the code that generates event notifications into the application as directed by the aspect-oriented analysis model.

Although the entire design is geared to deliver strong isolation between the application and the analysis, the analysis developer is shielded from many of the separation issues by the DPA programming model. The model allows the developer to retain Java as the instrumentation language, masking the fact that the application DVM and the analysis JVM use different bytecode. The event notifications carry the necessary contextual information that allows the analysis to identify the application components where the events originate, to track the communication between components, and to handle special situations such as DVM forking. We describe the DPA programming model next, and then return to the more advanced aspects of the DPA framework implementation.

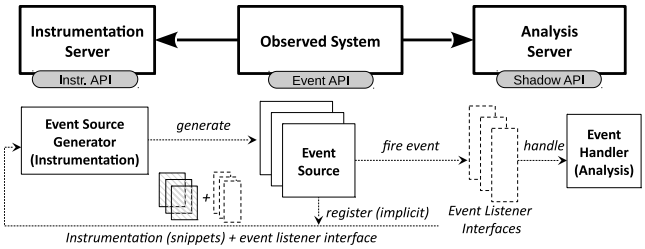

Figure 2: Overview of the programming model as an event-based system.

4. DPA Programming Model

The programming model follows from the requirement for strong isolation and is essentially a distributed event-processing system, as illustrated in Figure 2. An instrumented base program executes on the observed system, producing event notifications that drive the analysis. Both the instrumentation and the analysis execution are isolated from the observed system.

An analysis developer is responsible for identifying the events of interest and for generating the corresponding event notifications during program execution. The code of the observed program thus needs to be instrumented to include hooks that trigger invocation of analysis stubs, which use the framework-provided *Event API* to generate event notifications. The hooks are represented as DiSL snippets (c.f. Section 2.2) that are woven into the code of the observed program; this is performed in a dedicated VM, which allows retaining Java as the instrumentation language. When intercepting common events, such as method entry/exit, basic-block entry/exit, monitor entry/exit, object allocation, or field read/write, the developer can reuse existing snippets from a library of instrumentations. For certain events that are impossible to capture via instrumentation, such as the VM lifecycle or an object being freed by the garbage collector, the notifications are generated automatically by the framework.

The framework buffers the notifications and delivers them to the analysis, while respecting the ordering configuration required by the analysis. The available ordering configurations include, among others, global and per-thread ordering. Global ordering is provided for analyses that typically observe global application behavior and need to correlate events from all application threads. Per-thread ordering can be used for analyses with relaxed ordering requirements, to avoid completely serializing the base program, and to execute analysis code in a matching number of threads.

The notifications are delivered in form of method invocations on an analysis class deployed in a separate analysis VM. For user-defined analysis events, the interface of the analysis class is determined by the developer. To receive the framework-generated events, the analysis class needs to implement event-specific interfaces to subscribe to those events.

An analysis method corresponding to an event receives as its arguments the data from notifications generated by the hooks in the observed program. The data can be either primitive values or object references, with the latter reified as ShadowObject instances when delivered to the analysis. The mapping between references in the observed program and the shadow objects is a partial[1] function, i.e., a reference from the observed VM will always map to the same shadow object. A ShadowObject is the basis of the *Shadow API* provided by the framework. Besides representing object identity, it provides access to class information and allows an analysis to associate state with any ShadowObject instance. Instances of the String, Thread, and Class classes in the observed program are reified as specialized ShadowObject instances—a ShadowString, providing the string value, a ShadowThread, providing basic thread attributes, and a ShadowClass, providing reflective information (any class loaded by the observed VM will have a corresponding ShadowClass instance on the analysis VM). Any other state-related data from the observed program need to be explicitly extracted by the hooks that reify analysis events and marshalled into event notifications.

[1] The mapping is only performed for objects that are exposed to analysis.

```
1 public final class Context {
2   public int userId();
3   public int processId();
4   public String processName();
5   public Collection <ShadowObject> shadowObjects();
6   public static Collection <Context> contexts();
7 }
```

Figure 3: The Context representing a VM.

4.1 Virtual Machine Context

Because the components of an Android application can execute in multiple DVM processes, an analysis may receive events originating from different processes. To determine event origin or to distinguish between events from different virtual machines, the framework can (optionally) associate context information with each event notification delivered to an analysis.

To receive the context information, the last parameter in the signature of a user-defined analysis method must be of type Context, shown in Figure 3. When invoking such a method in response to a particular event notification, the framework will pass a Context instance as an argument to the method. The Context instance represents the VM in which a particular event originated and can be only created by the framework. The information identifying a virtual machine is immutable and represents the Context identity. A Context instance can be therefore used as a key in associative data structures. In addition, a Context provides access to all shadow objects associated with the VM it represents.

To receive framework-generated notifications for special events, an analysis class needs to implement the corresponding listener interfaces. Figure 4 shows interfaces for various lifecycle event notifications. The signatures of the receiver methods are prescribed by the framework, therefore the Context parameter is always included where appropriate.

The onObjectFree() notification signals that a particular object has been released by the GC, and is the last notification referring to that object. The onThreadStart() and onThreadExit() notifications delimit the start and the exit of a thread in a VM and are the first and the last notifications for that thread. The onVmStart() notification captures the creation of a new VM on the observed system and is the very first event for that context. The parent context will be null for a standalone/bootstrap VM—the *Zygote* in the Android case. The DVM instances forked off the *Zygote* to run an application or a service will receive the *Zygote* context as their parent. The onVmExit() notification signals the exit of a VM and is the very last notification for that context. We note that due to the distributed nature of Android applications, this notification alone is not sufficient to detect the end of application execution—user-defined hooks need to be employed to detect when different application components are being shut down.

4.2 Inter-process Communication Events

In addition to the resource lifecycle events, the framework also generates notifications for inter-process communication events, with the corresponding listener interfaces shown in Figure 5. We discuss these separately, because they are instrumental in addressing the

```
1 interface ObjectFreeListener {
2   void onObjectFree (ShadowObject object, Context ctx);
3 }
4 interface ThreadStartListener {
5   void onThreadStart (ShadowThread parent,
6     ShadowThread thread, Context ctx);
7 }
8 interface ThreadExitListener {
9   void onThreadExit (ShadowThread thread, Context ctx);
10 }
11 interface VmStartListener {
12   void onVmStart (Context parent, Context ctx);
13 }
14 interface VmExitListener {
15   void onVmExit (Context ctx);
16 }
```

Figure 4: Resource lifecycle events.

```
1 interface RequestSentListener {
2   void onRequestSent(Endpoint target,
3     NativeThread client, Context ctx);
4 }
5 interface RequestReceivedListener {
6   void onRequestReceived(Endpoint target,
7     NativeThread client, NativeThread server, Context ctx);
8 }
9 interface ResponseSentListener {
10   void onResponseSent(Endpoint target,
11     NativeThread server, Context ctx);
12 }
13 interface ResponseReceivedListener {
14   void onResponseReceived(Endpoint target,
15     NativeThread server, NativeThread client, Context ctx);
16 }
```

Figure 5: Inter-process communication events.

design challenge related to the multi-process architecture of Android applications.

The Android application framework provides the IBinder interface, which represents the base interface of a remotable object and serves as an abstraction of the low-level *binder* mechanism (c.f. Section 2.1). Transactions performed using the IBinder interface are by default synchronous, unless a special flag requesting one-way transaction (to an out-of-process target) is used. The high-level IPC mechanisms, such as intents, messaging, and RPC, are implemented using the IBinder abstraction.

An analysis designed to observe high-level Android communication can observe most of the high-level behavior using user-defined instrumentation. But when the control reaches native code that uses the low-level *binder* API to perform a transaction, the last step of an IPC operation becomes opaque. To make this last step observable, our DPA framework generates event notifications corresponding to a transaction client sending a request and receiving a response, and to a transaction server receiving a request and sending a response. Each notification includes a Context representing the VM in which the event occurred, along with an Endpoint representing the transaction's target binder token, and NativeThread instances representing threads (not necessarily corresponding to application threads) involved in the communication. With this information, an analysis can identify and distinguish among individual transactions.

We emphasize that these notifications only provide the necessary means to enable development of analyses that observe communication between application components and correlate IPC events from multiple virtual machines. Additional user-defined hooks in the Android core libraries are required to capture and track communication from the high-level abstractions (intents, messaging, RPC) used by the application to the lower-level layers using the *binder* kernel API.

4.3 Analysis State Replication

Analyses that need to shadow base-program objects typically associate their state with the shadow objects using associative data structures. To relieve analysis developers from repeatedly implementing mapping between shadow objects and analysis state, the ShadowObject class provides a convenience API, shown in Figure 6 (lines 3–6), which allows attaching arbitrary analysis-specific data to a ShadowObject instance. However, this API is not sufficient for platforms such as Android, where multiple virtual machines can be executing components of a single application.

```
1 public class ShadowObject {
2   ...
3   public Object getState();
4   public <T> T getState (Class <T> type);
5   public void setState (Object state);
6   public Object setStateIfAbsent (Object state);
7 }
8
9 interface Replicable {
10   Replicable replicate ();
11 }
```

Figure 6: API for management of analysis state.

The general contract for a ShadowObject instance is that it represents the identity of a single object. However, when a new VM is started by forking a parent VM, the objects allocated on the parent VM prior to the fork are duplicated in the child VM. Consequently, any shadow objects associated with the parent VM should be cloned as well, so as to maintain the ShadowObject contract of representing a single object in a single virtual machine. The framework naturally honors the contract by cloning the ShadowObject instances before issuing the onVmStart() notification to the analysis, but it does not know how to clone analysis-specific data.

This is not a concern if the analysis only targets application classes, because their instances will never be created in the *Zygote*, and thus will never be cloned when creating a new DVM for the application components. However, when an analysis also targets core libraries (many of which are initialized in the *Zygote*), the developer has to provide a replication strategy for the shadow state.

Any state data attached to a ShadowObject that is being cloned is expected to implement the Replicable[2] interface, shown in Figure 6 (lines 9–11), so that it can be replicated for the new ShadowObject clone. If the analysis never attaches state to shadow objects created during *Zygote* initialization, the framework will have no need for state replication, and the state class will not need to be replicable.

5. Advanced Technical Challenges

The programming comfort provided by our DPA programming model—compared to the DPA approaches listed in the introduction—requires extensive design effort on the part of the DPA framework, which abstracts away from numerous details of the underlying platform. We describe the more elaborate issues step by step, starting with the mechanisms related to code instrumentation and following with the event notification generation and delivery.

Bytecode. As mentioned earlier, Android application classes must be translated from the *stack-based* Java bytecode to the *register-based* Dalvik bytecode for execution. Manipulating the Dalvik bytecode represents an added burden on DPA analysis development without any technical merit—the Dalvik bytecode was developed primarily to circumvent licensing issues. With a plethora of mature tools for manipulating Java bytecode, there is little interest in duplicating the effort for directly manipulating the Dalvik bytecode, which is (officially) only ever produced by conversion from Java bytecode. We therefore use bytecode retargeting tools to convert between the

[2] The Cloneable interface in Java is generally considered broken, because a successful cast of an Object to a Cloneable does not allow calling the clone() method on the result.

two representations as necessary, and adjust the instrumentation process accordingly.

When instrumenting Android application classes, the instrumentation server extracts the classes to be instrumented from the corresponding .dex file, converts them from Dalvik to Java bytecode using the bytecode retargeting tool, and instruments the resulting Java classes using DiSL. The hooks employed by the analysis are expressed as Java code snippets and are compiled into Java bytecode, which is used by DiSL directly, without any transformation. After instrumentation, the server converts the instrumented classes back to Dalvik bytecode, repackages them into a .dex file, and sends it back to the DVM via the ACS.

Batch class loading. The Java language specification requires that classes are initialized lazily, in general immediately before first access but not sooner. The JVM also loads the classes lazily—just before initialization. It is therefore customary to instrument classes at load time (otherwise it is difficult to enumerate the exact set of classes belonging to the application). In contrast, the DVM loads (maps) classes into memory from per-application .dex files, in which all classes are laid out to facilitate memory mapping and constant deduplication [23]. Before loaded into memory, these .dex files still need to be optimized specifically to the runtime into .odex formatted files. These .odex files are cached to boost loading time for other DVM instances. The DVM thus reads the .odex files, maps multiple classes at the same time, and then initializes them lazily when they are needed. As a consequence, the DVM does not support class redefinition, and does not allow instrumenting individual classes upon loading or before initialization—once mapped into memory, the class code cannot be changed.

To preserve the concept of instrumentation at load time—essential to relieve the developers from instrumenting the application manually before installation—we modify the DVM class loading process to instrument classes in batches before they are mapped to memory. To enable different instrumentation strategies, we also make sure the .odex files won't be cached by removing the caches at boot time. Before mapping a .dex file, the modified DVM sends it via the ACS to the instrumentation server, which instruments the classes and sends back another .dex file which is then optimized and mapped to memory. This ensures that both the application classes and the associated libraries are instrumented.

Core libraries. When the Android system starts, the first DVM process—the *Zygote*—loads and initializes the core classes. All subsequent DVM instances (obtained by forking the *Zygote*) therefore share the code of the core libraries. This concerns full coverage analyses, which require instrumentation of the core libraries, because it is technically impossible to only instrument the core libraries for the application that is being analyzed—the instrumentation is present at all times.

To prevent flooding the analysis with events from the core libraries generated due to activities of other applications, the instrumentation is equipped with a bypass functionality [19]. The instrumentation is only enabled in the analyzed application, and bypassed with minimum overhead in the other applications.

Reflective information. Connected to class loading is the issue of providing reflective information, available to the analysis through ShadowClass instances. The reflective metadata is extracted by the analysis server from the bytecode of loaded classes. To achieve that, the modified DVM sends an internal notification to the analysis server prior to class initialization, and the analysis server in turn requests the class bytecode from the instrumentation server and creates the corresponding ShadowClass instance containing the reflective information. The analysis notifications that depend on that particular ShadowClass to be available are delayed until the ShadowClass is created.

Generating analysis event notifications. Following the loading and instrumentation phases, the major responsibility of the DPA framework is to generate notifications for the analysis events as directed by the aspect-oriented snippet code. The code inserted by instrumentation into the application typically assumes the form of simple hooks, which capture the application state that is relevant to the event being reified and use the *Event API* to generate the event notifications. The hooks themselves are provided by the analysis developer and woven into application code.

The code that actually generates the event notifications resides in a separate (stub) class and invokes the native helper methods of the *Event API*—provided by the modified DVM—to create a notification, marshal the captured state data into it, and submit it for delivery to the analysis. The code is always of the same restricted form and can be therefore generated. Each notification is automatically augmented with an identifier of the DVM process, so that the analysis server can locate the *Context* representing the DVM process from which the notification originated.

Object identity. The DPA programming model provides the analysis with a convenient abstraction for the identities of objects that have been exposed to the analysis. On the JVM, support for exportable object identities—that is, identities that can be used outside the analyzed application—can be implemented through a specialized tool building API, the JVMTI [24]. The DVM, however, does not provide such an API.

To maintain correspondence between the objects in the observed DVM and their ShadowObject counterparts on the analysis server, we have extended the DVM to support object tagging, or, the ability to associate an arbitrary long integer value with any object. All objects passed into an event notification are tagged with a unique (within the DVM) identifier. Whenever an object identifier is received in the event notification, the analysis server uses the DVM process identifier from the notification to locate the Context in which to look for or create the corresponding shadow object.

Generating special event notifications. Besides generating notifications for the analysis events from the aspect-oriented snippet code, the DPA framework must also generate notifications for certain essential events that are impossible or unduly difficult to capture by instrumentation. This includes events related to lifecycle of various DVM resources, as well as events related to low-level inter-process communication.

Among the lifecycle events, the onObjectFree() notification is the most common. Without this notification, an analysis would not be able to determine when to free the shadow objects, and would therefore be bound to run out of memory on a long enough execution. Again, the notification can be implemented on the JVM through a specialized tool building API, but no such API exists for the DVM—we have therefore modified the DVM to emit the notification when releasing a tagged object, that is, an object that has been exposed to the analysis.

To produce the onVmStart() notifications, the *Zygote* was modified to emit an internal notification to the analysis server after initializing the bootstrap DVM, and after creating a new DVM by forking itself. The internal notification triggers the creation of a new Context for the new DVM, as well as cloning of all shadow object instances (and possibly replication of the attached state data) associated with the parent DVM. The onThreadStart() notification is generated before the DVM creates a thread, while the onThreadExit() notification is generated when the thread's resources are released—just after the thread had been detached from the DVM.

The notifications for inter-process communication events are produced from hooks in the *binder* communication API (native implementation of the transact() and onTransact() methods). To identify the communicating threads, the transaction data is extended to include the thread identity of the sender thread. This is similar to IPC taint tracking [10].

Together, our modifications of the DVM can form a new tool interface for use by other tools independent of our DPA framework. In addition to these modification, we introduce two details related to the notification transport layer—notification buffering and notification delivery.

Notification buffering. The framework extension of the DVM employs carefully designed buffering and threading strategies to enable asynchronous analyses while respecting ordering constraints. Events produced by the application threads are stored and marshalled into buffers in the context of the *Event API* invocations. Completed buffers are then sent to the ACS. This is handled by a dedicated thread in all DVM instances except the *Zygote*, where this is handled on the main thread. When the *Zygote* is about to fork off a new DVM, all notification buffers are flushed to ensure that the child DVM does not send out events originating in the parent. After forking the *Zygote*, a dedicated sending thread is created in the child DVM.

Notification delivery. For each event notification received from the ACS, the analysis server unmarshals the data, determines the corresponding VM Context, maps object identifiers to shadow objects, and invokes the appropriate analysis method. User-defined analysis event notifications are dispatched according to the selected notification ordering model, which allows processing notifications from multiple base program threads in matching number of analysis threads. The framework-generated special event notifications are dispatched in a dedicated analysis server thread. The internal threading model in the analysis server guarantees proper ordering of the special event notifications with respect to the analysis event notifications originating in the base program.

6. Case Studies

In this section we present two case studies to illustrate the use of our framework for developing dynamic analyses for Android. The first case study is a conventional dynamic analysis for determining application code coverage. The second case study is a custom, Android-specific analysis that allows tracing the usage of permissions in Android applications.

The case studies were evaluated with an implementation of our framework, built on Android 4.1 Jelly Bean (API level 16). The bytecode retargeting was performed using the dex2jar[3] tool. We used the standard Android ARM emulator to run our framework, with the instrumentation and analysis servers running in separate JVMs on an Ubuntu Desktop Linux host with 8 GB of RAM.

We note that the purpose of these case studies is solely to showcase some of the concepts provided by the framework, as well as substantiate our claims with a working implementation. At this point, we do not make any performance claims, because using a full system emulator does not allow us to undertake an extensive performance evaluation.

6.1 Code Coverage Analysis

Code coverage is generally used to judge the effectiveness of a test suite and the quality of individual tests [20]. Code coverage metrics span from high-level program elements, such as classes and methods, to lower-level code elements, such as basic block, lines of code, or individual instructions. Branch coverage belongs among the lower-level metrics and represents the percentage of branches executed at least once in a program in response to a particular input.

The Android SDK supports collecting coverage data for test suites. This was originally based on EMMA[4], which provided mainly high-level coverage metrics, and is now based on JaCoCo[5], which also provides many low-level coverage metrics.

To collect test coverage data for an application requires instrumenting the application code prior to deployment. This process also produces metadata that serves to identify various program and code elements. The tools are integrated into the Android SDK build system to simplify their usage for the developer. When the instrumented application is deployed, the tests are run using a custom implementation of the Instrumentation class from the android.app package, which also retrieves the collected coverage data when the tests finish executing. Because of tight integration with the SDK, replacing or extending the test coverage tool supported by Android SDK is cumbersome—a developer would need to either modify the existing test runner, or create a custom implementation of the Instrumentation class.

To demonstrate the strengths of our framework, we implemented a dynamic analysis to collect branch, basic block, method, and class coverage metrics for an application. The tested application can be deployed unmodified, because the framework will instrument the application when it starts. There is no need to plug our analysis into the SDK or the testing framework—all the developer needs is to deploy the application and run the tests.

We now overview the two aspects of a DPA tool implementation—the instrumentation, responsible for capturing the events of interest in the base program, and the analysis, responsible for producing results based on the observed events. We then provide a short evaluation of the resulting analysis.

Instrumentation. To capture the events required for the analysis, we recasted JaCoCo's probe insertion strategy for branch and basic-block coverage[6] in DiSL. For branch coverage, the instrumentation code identifies conditionals and assigns every branch in every method a local identifier (from 0 to $local - 1$, where $local$ is the number of branches in a method. A sum of the number of branches in all methods provides the $total$ number of branches in a class. This is performed for each instrumented class, and is similar to the metadata collection step when using other coverage tools—except when using DiSL, the information is inlined into the code inserted into the base program during instrumentation and becomes the payload of an analysis event notification. Other hooks capture branch execution, producing analysis event notifications containing an identifier of the branch in the currently executing method. For basic-block coverage, we track entry to and exit from each basic block, and maintain a bitmap representing the executed basic blocks for each method. We do not show the DiSL instrumentation here, because it is outside the scope of this paper. We instead focus on demonstrating the use of the DPA framework programming model in the analysis implementation.

Analysis. The responsibility of the analysis is to gather coverage data based on the received event notifications. The skeleton of the analysis is shown in Figure 7. We only show code for recording branch coverage, because the code for recording basic-block coverage is similar. For brevity, we omit the public modifiers and some of the implementation details in the listing.

The CodeCoverageAnalysis class (as shown) defines handlers for three analysis event notifications (lines 21–41), represented

[3] https://code.google.com/p/dex2jar

[4] http://emma.sourceforge.net

[5] http://www.eclemma.org/jacoco

[6] http://www.eclemma.org/jacoco/trunk/doc/flow.html

```
1  class CodeCoverageAnalysis VmExitListener {
2   // Per-method analysis state
3   static class CoverageState implements Replicable {
4    String className; // method owner
5    boolean[] branchExecuted; // branch coverage bitmap
6    int branchesTotal; // number of branches per class
7
8    private CoverageState(String, boolean[], int) { ... }
9
10   CoverageState(String className, int local, int total) {
11    this(className, new boolean[local], total);
12   }
13
14   @Override Replicable replicate() {
15    return new CoverageState(className,
16      new boolean [branchExecuted.length], branchesTotal);
17   }
18  }
19
20  // Analysis methods corresponding to event notifications
21  void registerMethod(ShadowString className,
22   ShadowString methodSig, int total, int local) {
23   // Associate per-method state with the signature
24   if (methodSig.getState() == null) {
25    methodSig.setStateIfAbsent(new CoverageState(
26      className.getValue(), local, total));
27   }
28  }
29
30  void commitBranch(ShadowString methodSig, int idx) {
31   CoverageState s = methodSig.getState(CoverageState.class);
32   s.branchExecuted[idx] = true;
33  }
34
35  void endAnalysis() {
36   // Calculate branch coverage for all processes
37   for (Context ctx : Context.contexts()) {
38    for (ShadowObject o : ctx.shadowObjects()) {
39     ... // get state to compute coverage
40   }}
41  }
42
43  @Override void onVmExit(Context ctx) {
44    ... // Calculate brach coverage per process
45  }
46 }
```

Figure 7: Skeleton of the code coverage analysis related to collecting branch coverage data.

by the registerMethod(), commitBranch(), and endAnalysis() methods.

The metadata and branch coverage information for a method is held in instances of the CoverageState class (lines 3–18), which also implements the Replicable interface to support shadow state replication (c.f. Section 4.3). A replica contains a copy of the metadata, but the coverage data is reset (lines 14–17).

The registerMethod() notification is emitted when a method is executed, and in response the analysis associates a lazily-instantiated CoverageState with the ShadowString instance representing the method signature (lines 21–28).

The commitBranch() notification is emitted when a branch is executed, and in response the analysis updates the state associated with the indicated method signature by setting a bit corresponding to the executed branch to true (lines 30–33). Because registerMethod() notifications are always delivered before commitBranch() notifications, we can assume the shadow state to be always present in the latter.

Because DVM instances usually do not terminate (unless killed by the system), the reporting of the coverage data must be triggered explicitly using the endAnalysis() notification. The handler for that notification simply collects and prints the data from ShadowObject instances in each VM Context (lines 35–41). To avoid losing coverage data when a DVM does terminate, the analysis class implements the VmExitListener interface, thus subscribing to receiving the onVmExit() notifications, to which it reacts by outputting the coverage data for the terminated process (not shown).

Evaluation. The goal of this evaluation is to demonstrate basic concepts of the programming model provided by our framework, and to show that: (a) our analysis tool produces valid results, (b) the tool is cross-platform and that bytecode retargeting does not invalidate its results, and (c) the tool can be used with actual Android applications.

Table 1: Coverage results for GrinderBench, collected using JaCoCo on the JVM, and using our code coverage analysis (CCA) on the JVM and DVM. Results for CCA also include coverage data for the system libraries. [T] stands for the *total* number, while [E] stands for the number of *executed* branches, basic blocks, method, and classes.

	BC	Branches		Basic Blocks		Methods		Classes	
App/package	**%**	**[T]**	**[E]**	**[T]**	**[E]**	**[T]**	**[E]**	**[T]**	**[E]**
(JaCoCo/JVM)									
GrinderBench	**55.3**	1673	925	—	—	505	380	102	85
(CCA/JVM)									
GrinderBench	**56.5**	1636	925	2229	1494	478	380	93	85
(CCA/DVM)									
GrinderBench	**55.3**	1672	925	2278	1467	505	380	102	85
(CCA/JVM)									
java.*	**9.7**	12024	1170	16663	2200	4156	749	265	153
sun.*	**11.6**	2994	346	4019	644	908	204	95	55
(CCA/DVM)									
java.*	**2.6**	28819	753	40730	1393	11681	553	1140	118
dalvik.*	**2.8**	432	12	772	27	284	13	50	7
libcore.*	**3.5**	2557	89	3669	163	1112	63	123	22

To validate the implementation of our analysis, we compare the results produced by our analysis tool to results produced by JaCoCo when run on the JVM, with the GrinderBench[7] suite as the base program. The suite contains 5 reproducible benchmarks targeting embedded Java devices, and provides a workload sufficient for the validation. The first block of results in Table 1 shows the coverage data for the GrinderBench suite, collected using JaCoCo and our code coverage analysis tool (CCA) on the JVM.

We observe that the numbers of actually executed (the "[E]" columns) branches, methods, and classes produced by our tool are identical to those produced by JaCoCo. The total numbers (the "[T]" columns) of branches, methods, and classes differ, because our tool collects summary metadata during instrumentation, which is performed at load time, whereas JaCoCo collects metadata for all classes in a given .jar file.

To confirm that the analysis can be used even after bytecode retargeting, we used it to collect coverage data for the GrinderBench suite on the DVM. Because GrinderBench is not an Android application, we could not use the normal process for launching Android applications. Instead, we used an isolated DVM launched directly, without the *Zygote*. The second block of results in Table 1 shows the coverage data obtained using our tool. Note that the results do not show coverage data collected using JaCoCo, because the integration of JaCoCo with the new Gradle-based build system in the Android SDK is still unstable, and we did not manage to get it running.

We again observe that the numbers of executed branches, methods, and classes are identical to the results obtained with both tools on the JVM. The total numbers of methods and classes match the results produced by JaCoCo on the JVM, but differ from those produced by our tool on the JVM. This is because when running on Android, our framework has to instrument the classes in bulk (and not one-by-one as loaded by a JVM), thus obtaining the same summary data as JaCoCo. The total number of branches is off-by-one, which was caused by an optimization during bytecode retargeting. A conditional statement using an empty then branch and non-empty else branch (2 branches in total) was transformed into a statement with an inverted condition and only the then branch, reducing the

[7] http://www.grinderbench.org/

Table 2: Coverage results for the ImplicitFlow4 application from DroidBench, with 4 different human actions.

	BC	Branches		BBC	BasicBlocks	
Scenario/Method	**%**	**[T]**	**[E]**	**%**	**[T]**	**[E]**
(1–bad user)						
checkUsernamePassword	**0**	2	0	**60**	5	3
lookup	**17**	6	1	33	6	2
(2–good user, bad password)						
checkUsernamePassword	**50**	2	1	**60**	5	3
lookup	**50**	6	3	**67**	6	4
(3–good user and password)						
checkUsernamePassword	**50**	2	1	**60**	5	3
lookup	**50**	6	3	**67**	6	4
(4–sequence of scenarios 1–3)						
checkUsernamePassword	**100**	2	2	**100**	5	5
lookup	**83**	6	5	**100**	6	6

total number of branches by one. Finally, we observe a slight mismatch both in the total number and the number of executed basic blocks compared to results produced by our tool on the JVM. This was again caused by an optimization, this time eliminating empty branches targeted by tableswitch bytecode instructions, thus reducing the amount of basic blocks, resulting in a slight bias in the coverage results.

Despite the slight bias in the basic block coverage, we note that our framework allowed us to use a single implementation of the code coverage analysis to collect coverage data on both the JVM and the DVM. In addition, thanks to strong isolation of the analysis from the base program, our analysis can easily collect coverage data even for classes in the core libraries. The coverage of system classes when running GrinderBench is shown in the third block of results in Table 1. As expected, GrinderBench mostly uses the Java API (java.*). On the Android platform, several Android specific classes are used due to class loading and I/O (dalvik.* and libcore.*).

To show that the tool can be used with Android applications written using the Android application framework and component model, we applied the analysis tool to DroidBench[8], which is a set of real-life Android applications for validating static and dynamic security tools. Using our tool, we collected coverage data for the ImplicitFlow4 application from the DroidBench suite, shown in Table 2. While the application itself is very simple—it presents two input fields for user name and password, and a button for submitting the credentials—we test it in four different scenarios, yielding different coverage. Scenario 1 corresponds to a user submitting the credentials with incorrect user name, Scenario 2 corresponds to submitting correct user name, but incorrect password, Scenario 3 corresponds to submitting correct user name and password, and Scenario 4 performs scenarios 1–3 in sequence. We verified that coverage results obtained using our tool for two key methods, checkUsernamePassword() and lookup() are correct.

In summary, using our framework we developed a cross-platform code coverage analysis tool, in modest amount of code and without having to worry about low-level platform-specific details. With respect to the goals of this evaluation, we have shown that the analysis (a) produces valid results on the JVM, (b) is cross-platform and robust with respect to bytecode retargeting, and (c) can be used with real-world Android applications. In addition, we did not need to plug the analysis into the Android testing framework, instrument applications prior to installation, or modify their manifest files. We were also able to collect coverage data system-wide, for all processes—including shared code in the core library.

[8] http://sseblog.ec-spride.de/tools/droidbench/

6.2 Permission Usage Tracking

The Linux kernel underneath the Android system provides basic security mechanisms and isolation to Android applications. However, it does not exert any control over what applications can do and how they communicate with each other. The Android platform provides applications with numerous services that are sensitive in nature, e.g., a service to initiate phone calls. To prevent malicious software from abusing the services, access to them is restricted and requires user's consent. The Android platform therefore employs a concept of permissions that an application needs to use the restricted APIs. Each application declares the set of required permissions in its manifest file, and the user is required to grant the requested set of permissions to the application prior to installation. Failure to obtain any of the requested permission aborts the application installation. During execution, whenever an application calls a restricted API through its respective proxy, a call on behalf of the application will be made to the *System Server* process (which provides all system services) to check whether the calling process is permitted to use the API.

An important motivation for this case study is the fact that the user is made aware of the permissions required by the application only at installation time, and is only allowed to either grant all the permissions, or none. When using the application, the user is not aware when and in what context the application uses these permissions. This allows a seemingly benign application to acquire permissions that are later used for malign purposes, or trick vulnerable overprivileged applications to perform undesired actions on behalf of another application. While static analyses can detect usage of sensitive APIs and map them to actual permissions using Stowaway [11], the static analysis may fail in cases where dynamic loading and reflection are used.

We therefore developed a simple dynamic analysis to detect runtime usage of API permissions and to pinpoint their usage to application methods by providing the user with a calling context leading to the permission check. Because all API permission checks are performed by invoking a remote service on the system server, the analysis has to cope with distributed nature of these invocations, and the calling context therefore spans multiple processes. Consequently, the analysis is also able to detect permission usage transitively, i.e., in situations where one application requests a service from another application, which leads to a permission check. This is made possible by utilizing the support for IPC events provided by our framework, which allows us to connect the permission check in the *System Server* with the calling application thread.

We now again review the instrumentation used to obtain the events of interest, demonstrate the use of IPC events in the analysis implementation, and conclude with a short evaluation.

Instrumentation. To detect permission usage, we instrument methods and classes of the Android framework that are related to permission checks. This involves primarily the checkPermission() method in the ActivityManagerService class. From these invocations, we extract the name of the permission that is being checked. The events corresponding to invocation of these methods merely indicate permission usage. To pinpoint them to specific locations in applications on whose behalf the permission checks were made, we also need events that allow the analysis to track calling context across processes. We therefore instrument entry to and exit from all methods, which allows us to identify the application method causing a permission check. We also instrument all method invocations in a method body, which allows us to identify a particular invocation that caused the check.

Analysis. The responsibility of the analysis is to identify the method invocation which ultimately resulted in a permission check, across multiple process boundaries. The skeleton of the analysis is shown

in Figure 8. For brevity, we omit the public modifiers and some of the implementation details (specifically, the ThreadState and Logger classes) in the listing.

The PermissionUsageAnalysis class (as shown) defined handlers for three analysis event notifications (lines 2–23), represented by the boundaryStart(), boundaryEnd(), and permissionUsed() methods.

The boundaryStart() and boundaryEnd() notifications enable the analysis to maintain a call-stack for each thread. In response to these notifications, the analysis obtains a ThreadState instance for the corresponding thread and pushes or pops an invocation boundary to or from the call stack associated with the per-thread state (lines 2–7 and 9–14). The permissionUsed() notification indicates that a permission check occurred for the given permission. In response to this notification, the analysis finds all threads in the call-chain spanning multiple processes and associates with each the permission (lines 16–23).

The analysis then defines four handlers for the framework-generated IPC event notifications (lines 25–66), represented by the onRequestSent(), onRequestReceived(), onResponseSent(), and onResponseReceived() methods. These notifications enable the analysis to reconstruct the call-chain across threads in multiple processes, and trigger the reporting of used permissions along with the location in the application which triggered a check for those permissions.

The onRequestSent() notification indicates the start of a *binder transaction* in a client thread. In response to this notification, the analysis just records the event in the history of events received for the client thread (lines 25–30. The onRequestReceived() notification indicates that the request has been received by a server thread. Because this notification originates in a different process and may be received out-of-order by the analysis server, the analysis first ensures that it has received the onRequestSent() notification corresponding to the target from the client. The event is then recorded in the history of events received for the server thread (lines 32–41). The onResponseSent() notification is handled similarly to the onRequestSent() notification, the analysis just updating the server thread event history (43–48). Finally, the onResponseReceived() notification indicates the end of the *binder transaction*, and triggers the reporting of used permissions. In response to this notification, the analysis first ensures that it has received the onResponseSent() notification corresponding to the target and the client from the server. If the application used any permissions, they are reported along with the call stack in the client thread leading to their usage. In the last step, the analysis clears the event history related to the transaction (lines 50–66).

```
1  class PermissionUsageAnalysis implements ... {
2   void boundaryStart(ShadowString signature,
3     ShadowThread thr, Context ctx) {
4     // Update call-stack on method entry/before invocation
5     ThreadState state = ThreadState.get(ctx, thr);
6     state.pushBoundary (signature.getValue());
7   }
8
9   void boundaryEnd(ShadowString signature,
10    ShadowThread thr, Context ctx) {
11    // Update call-stack on method exit/after invocation
12    ThreadState state = ThreadState.get(ctx, thr);
13    state.popBoundary (signature.getValue());
14  }
15
16  void permissionUsed(ShadowString permission,
17    ShadowThread thr, Context ctx) {
18    // Associate permission with threads in the call-chain
19    ThreadState state = ThreadState.get(ctx, thr);
20    for (ThreadState caller : state.getCallers()) {
21      caller.addPermission(permission);
22    }
23  }
24
25  @Override void onRequestSent(Endpoint target,
26    NativeThread client, Context ctx) {
27    // Add this event to client event history
28    ThreadState clientState = ThreadState.get(client);
29    clientState.recordRequestSent(target, ctx);
30  }
31
32  @Override void onRequestReceived(Endpoint target,
33    NativeThread client, NativeThread server, Context ctx) {
34    // Ensure corresponding onRequestSent() was received
35    ThreadState clientState = ThreadState.get(client);
36    clientState.waitForRequestSent(target);
37
38    // Add this event to server event history
39    ThreadState serverState = ThreadState.get(server);
40    serverState.recordRequestReceived(target, client, ctx);
41  }
42
43  @Override void onResponseSent(Endpoint target,
44    NativeThread server, Context ctx) {
45    // Add this event to server event history
46    ThreadState serverState = ThreadState.get(server);
47    serverState.recordResponseSent(target, ctx);
48  }
49
50  @Override void onResponseReceived(Endpoint target,
51    NativeThread server, NativeThread client,Context ctx){
52    // Ensure corresponding onResponseSent() was received
53    ThreadState serverState = ThreadState.get(server);
54    serverState.waitForResponseSent(target, client);
55
56    // Print used permissions and thread call-stack
57    ThreadState clientState = ThreadState.get (client);
58    if (clientState.permissionCount() > 0) {
59      Logger.reportPermissionUsage(clientState);
60      clientState.clearPermissions();
61    }
62
63    // Discard notification related to this transaction
64    clientState.discardEventHistory(target, server);
65    serverState.discardEventHistory(target, client);
66  }
67 }
```

Figure 8: Skeleton of permission usage analysis.

Evaluation. The goal of this case study is primarily to demonstrate the use of the special IPC event notifications generated by our framework in an analysis implementation. Consequently, in this evaluation we aim at showing that our analysis tool can be run with actual Android applications and provide results that may escape detection by static analysis.

To make the results verifiable, we apply the tool to an application from the DroidBench suite, specifically the Reflection_Reflection4X[9] application, which was designed to confuse analysis tools by its use of reflection to invoke privileged APIs. The application invokes the getDeviceId() method in the TelephonyManager class, and the sendTextMessage() method in the SmsManager class. These APIs perform a permission check on behalf of the application to determine whether the application is authorized to invoke the API. The permission check passes through another process before reaching the ActivityManagerService class in the *System Server*. The actual interaction for the getDeviceId() method, as recovered by our tool, is illustrated in Figure 9 as a combination of sequence diagrams spanning multiple processes. The MainActivity component of the application triggers the invocation of the getDeviceId() method on the TelephonyManager, which uses a proxy to invoke the method remotely on the *Phone* service running in a different process. The communication across process boundaries is implemented as a *binder transactions*. Before performing the invocation on the Phone class, the PhoneSubInfoProxy uses the ActivityManagerProxy to trigger a permission check on behalf of the activity. The ActivityManagerProxy again uses the *binder* to perform a remote invocation of the checkPermission() on the ActivityManagerService hosted by the *System Server*, where use of the READ_PHONE_STATE permission is detected by the analysis. When control returns to the application component, the use of the permission is reported along with the corresponding location in application code. The situation is similar when invoking the sendTextMessage() method on the SmsManager class, we therefore do not illustrate it here.

[9] https://github.com/secure-software-engineering/DroidBench/tree/master/eclipse-project/Reflection_Reflection4

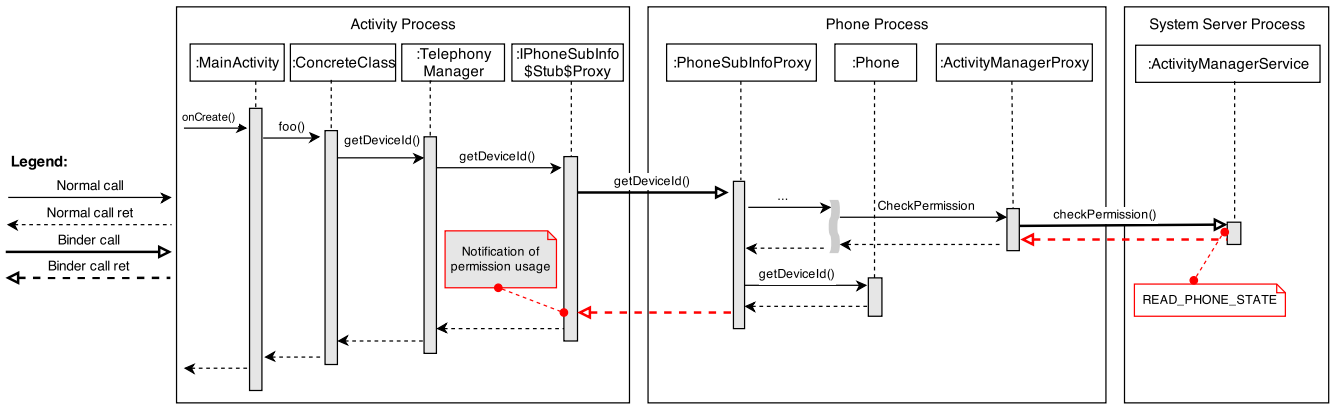

Figure 9: Multi-process interaction resulting from an activity invoking a restricted API method on a *Phone* service, triggering a permission check in the *System Server*.

```
Warning in de.ecspride(pid-1092):
Detected use of permission(s) #READ_PHONE_STATE
Calling Stack:
#0 android.telephony.TelephonyManager.getDeviceId
#1 de.ecspride.ConcreteClass.foo
#2 de.ecspride.MainActivity.onCreate

Warning in de.ecspride(pid-1092):
Detected use of permission(s) #SEND_SMS and #WAKE_LOCK
Calling Stack:
#0 android.telephony.SmsManager.sendTextMessage
#1 de.ecspride.ConcreteClass.bar
#2 de.ecspride.MainActivity.onCreate
```

Figure 10: Console output for the sample in analysis server

The output of the analysis tool for the Reflection_Reflection4X application is shown in Figure 10. Whenever a permission usage is detected, the tool outputs the name of the permission along with the call-stack in the application component triggering a permission check. We verified the results by checking the source code of the application and the Android framework.

In summary, using our framework, we managed to develop a simple dynamic analysis that can obtain information about application behavior which is often hidden from static analysis tools. We developed this analysis entirely in Java, without having to modify the application in any way, and without having to deal with low-level deployment issues. The use of special IPC events generated by our framework was essential for observing the inter-component communication and for reconstructing the call-chain across process boundaries.

7. Related Work

Existing dynamic analyses for Android are implemented using different techniques and at different levels of the Android platform, ranging from the application/framework level (Java) [5, 8, 12, 22], through native library level [6, 10, 13], OS kernel/driver level [6, 7, 9], to emulator level [18, 26]. Most of the work on dynamic analysis has been done in the context of Android security, and only few of the existing tools provide a framework for developing new analysis tools.

DroidScope [26] is an emulator-level analysis platform for Android that rebuilds two levels of semantic information: OS and Java. It provides an instrumentation interface for writing analysis plugins, such as API tracing (capturing invocation an execution of methods), native instruction tracing (gathering each instruction with operand and values), Dalvik instruction tracing (decode instructions with values and symbolic information) and taint tracking (keep track of data propagation). Similar to our approach, DroidScope provides an event based interface for dynamic analysis development. However, the available events are low-level and analyses need to be developed using native code. User-defined events are not supported, and the analyses cannot be deployed on real devices, because DroidScope is bound to the Android emulator. The support for dynamic analysis development using DroidScope is thus more similar to a low-level binary instrumentation tool. In contrast, our approach allows developing both analysis and instrumentation in Java, and provides the developer with a high-level programming model and convenient abstractions.

The In-Vivo [5] approach for Android is based on inserting hooks for monitoring and dynamic analysis into application bytecode. The analysis code is mixed with the application code and the instrumentation is performed directly on the device. This has several drawbacks. First, running the base program code together with the analysis code breaks isolation and is bound to introduce perturbations [14]. Second, the transformation of DVM to JVM bytecode together with the instrumentation are executed as application code, making the instrumentation of the core library code impossible. Finally, instrumentation can be resource demanding and therefore not suitable for devices with limited processing and storage capacities. Our approach addresses all these issues.

TaintDroid [10] is a dynamic analysis tool for Android that allows detecting unauthorized leakage of sensitive data. It employs dynamic taint analysis to label data propagating through the system. TaintDroid applies taints at different levels, facilitated by modifications to the DVM and the OS (to enable file-level, method-level, variable-level tracking) and by modifying Android's IPC core libraries (to enable message-level tracking). The latter tracks messages instead of data within messages to minimize IPC overhead. Even though our framework does not primarily target taint analysis, it can be used for basic message-level tracking thanks to IPC event notifications and full coverage support.

Finally, retargeting Dalvik bytecode to Java bytecode is a general technique to leverage existing static and dynamic analysis frameworks for Java on the Android platform. For example, the Dexpler [4] plugin allows the Soot [15] framework to operate on Dalvik bytecode. This in turn allows using AspectJ and Tracematches for the development of high-level monitoring tools for Android [2], or

using Soot as a basis for FlowDroid [3], a highly precise static taint analysis for Android applications. Similarly, our framework uses bytecode retargeting to leverage the ability of DiSL [17] to reconcile developer productivity with high instrumentation flexibility and performance. Thanks to load-time instrumentation, the bytecode retargeting is completely transparent for the user.

8. Discussion

We now discuss additional benefits and overreaching implications of our work. We also discuss a few principal limitations, either due to the platform itself, or due to our choice of DiSL as the instrumentation language.

Diverse deployment scenarios. The decoupling of the base program from the analysis advocated by our approach provides DPA tool developers with a wide range of deployment scenarios. Our modifications to the Android platform can be deployed on a system emulator (for example, a QEMU-based ARM emulator running on an Intel system), a virtual machine in which code executes natively, or an actual physical device. Our modified DVM uses a local socket to communicate with the ACS, therefore only the ACS needs a network connection to the instrumentation and analysis servers. This connection is readily available when running in a system emulator or a virtual machine. The situation is more difficult when running on a physical device. Because the wireless network interface only becomes available after the system finishes booting, we have to send data over USB, using the Android debugging interface. Our framework allows developing instrumentation-heavy analyses (such as the object lifetime profiler) that generate an enormous amount (gigabytes) of notification data, making the network connection a potential bottleneck. We consider employing fast on-the-fly compression to reduce the amount of data in exchange for slight computational overhead—preliminary testing reveals that the stream of notification data contains significant redundancy which can be exploited even by compression techniques focusing on speed, such as LZ4 and LZO. For light-weight analyses, we can consider running the analysis server directly in the Android environment, but this arrangement only makes sense for deployment on a real device.

Cross-platform analysis. One of the most important implications of our work is that it enables development of analyses that can be used with applications running on different platforms, without having to implement the analysis separately for each platform. Using our framework, both analysis and instrumentation code can be implemented in Java, with the analysis code deployed on the analysis server, reacting to event notifications from a platform-specific front-end. Many DPA tools targeting Java are written in C or C++, using the JVMTI interface. Such tools cannot be used with Android application without spending considerable effort at modifying the DVM.

An example of such a tool is ElephantTracks [25] (ET), which allows collecting traces for estimating object lifetimes. The latest ET release is implemented using a mixture of C, C++ and Java, in roughly 14000 lines of code, with a split of 16:60:24 between the languages, and can be currently only used with the IBM J9 Java Virtual Machine. We have implemented an analysis tool similar to ET using our framework in less than 2000 lines of Java, and can use it for collecting object lifetime traces and comparing allocation behavior for applications running on both the DVM and the JVM.

Comprehensive client/server analysis. The support for developing cross-platform analyses can be taken even further. Many modern applications consist of a thin client executing in a DVM running on Android, and a server back-end executing in a JVM running on a dedicated machine. Our framework lays the ground work for developing analyses for such applications, with the analysis code executing on a single, dedicated analysis server, receiving event notifications from both the client and the server, each running on a different machine and a different platform. We consider this particular topic well suited for future work.

Coverage limitations. Achieving full coverage is complicated by the sharing of core libraries—any class can be instrumented, but there is a minor limitation involving 6 core classes[10] during the DVM bootstrap. To avoid executing instrumented code in processes that are not under analysis, we use a per-thread bypass mechanism to selectively execute either the original or the instrumented code [19]. This mechanism needs a reference to the current thread, which can be obtained by calling the `Thread.currentThread()` method. However, this method is implemented using JNI, which cannot be called during a short phase in the DVM initialization, because the JNI environment is not yet available—forcing us to ignore the bypass mechanism in the critical classes until JNI is ready. Therefore apart from a very short period during DVM bootstrap, the framework can provide full coverage for all classes in the core libraries.

Bytecode retargeting. Using DiSL for instrumentation brings about two additional considerations. DiSL was designed as a general purpose framework for JVM bytecode instrumentation—using it with Android applications requires converting application bytecode back and forth between two different representations. This is generally considered safe, because the bytecode is verifiable and the transformations are formally defined [21], but the developer needs to be aware that the conversion takes place. We rely on third party tools for the conversion, which may not be exempt of bugs—indeed, just transforming the Android core library back and forth (without instrumentation) using the `dex2jar` tool results in one class throwing a verification error on the DVM[11].

Bytecode-specific metrics. The second consideration concerns bytecode-specific metrics, such as the number of instructions in a method or a basic block, available from certain DiSL components in form of static information. These metrics are based on Java bytecode, and may be therefore biased in the context of Dalvik bytecode. Developers should be aware of this limitation, and adjust the design of their analyses accordingly—for certain kinds of analyses, this may require obtaining the static information directly from the Dalvik bytecode prior to instrumentation.

9. Conclusion

The Android platform was designed and optimized for widespread deployment on mobile devices, yet lacking fundamental support for observing applications executing on the Dalvik Virtual Machine. We overcome the fundamental shortcomings of the Android platform, and handle its specifics, such as multi-process applications and inter-process communication. We believe that our framework enables—and greatly simplifies—development of generic DPA tools for Android applications, and we demonstrate this on the two diverse case studies, each showcasing a different aspect of our framework.

Acknowledgments

The research presented here was conducted while Haiyang Sun, Lubomír Bulej, and Alex Villazón were with the Università della Svizzera italiana. It was supported by the European Commission (contract ACP2-GA-2013-605442), by the Swiss National

[10] The `Object`, `Class`, and `Thread` classes from `java.lang`, and the `Reference`, `ReferenceQueue`, and `FinalizerReference` classes from `java.lang.ref`.

[11] The `java.text.SimpleDateFormat` class fails transformation. https://code.google.com/p/dex2jar/issues/detail?id=206

Science Foundation (project CRSII2_136225), by a Sino-Swiss Science and Technology Cooperation (SSSTC) Institutional Partnership (project IP04–092010), by the European Commission (project ASCENS 257414), by the National Natural Science Foundation of China (project No.61272101), as well as by Shanghai Key Laboratory of Scalable Computing and Systems.

References

[1] D. Ansaloni, S. Kell, Y. Zheng, L. Bulej, W. Binder, and P. Tůma. Enabling modularity and re-use in dynamic program analysis tools for the Java virtual machine. In *Proc. 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 352–377. Springer, 2013.

[2] S. Arzt, S. Rasthofer, and E. Bodden. Instrumenting Android and Java applications as easy as abc. In *Proc. 4th Intl. Conf. on Runtime Verification*, volume 8174 of *LNCS*, pages 364–381. Springer, 2013.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proc. 35th ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI'14, pages 259–269. ACM, 2014.

[4] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with Soot. In *Proc. ACM/SIGPLAN Intl. W. on State of the Art in Java Program Analysis*, pages 27–38. ACM, 2012.

[5] A. Bartel, J. Klein, M. Monperrus, K. Allix, and Y. L. Traon. Improving privacy on Android smartphones through in-vivo bytecode instrumentation. *CoRR*, abs/1208.4536, 2012.

[6] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on Android. In *Proc. 19th Annual Network and Distributed System Security Symp.*, NDSS'12. ISOC, 2012.

[7] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. Practical and lightweight domain isolation on Android. In *Proc. 1st ACM W. on Security and Privacy in Smartphones and Mobile Devices*, SPSM'11, pages 51–62. ACM, 2011.

[8] B. Davis, B. S, A. Khodaverdian, and H. Chen. I-ARM-droid: A rewriting framework for in-app reference monitors for Android applications. In *Proc. Mobile Security Technologies*, MOST'12. IEEE, 2012.

[9] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *Proc. 20th USENIX Security Symposium*, pages 23–23. USENIX, 2011.

[10] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. 9th USENIX Conf. on Operating Systems Design and Implementation*, OSDI'10, pages 1–6. USENIX, 2010.

[11] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proc. 18th ACM Conf. on Computer and Communications Security*, CCS'11, pages 627–638. ACM, 2011.

[12] H. Hao, V. Singh, and W. Du. On the effectiveness of API-level access control using bytecode rewriting in Android. In *Proc. 8th ACM/SIGSAC Symp. on Information, Computer and Communications Security*, ASIA CCS'13, pages 25–36. ACM, 2013.

[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 *Proc. 18th ACM Conf. on Computer and Communications Security*, CCS'11, pages 639–652. ACM, 2011.

[14] S. Kell, D. Ansaloni, W. Binder, and L. Marek. The JVM is not observable enough (and what to do about it). In *Proc. 6th W. on Virtual Machines and Intermediate Languages*, VMIL'12, pages 33–38. ACM, 2012.

[15] P. Lam, E. Bodden, O. Lhotak, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*, CETUS'11, 2011.

[16] L. Marek, S. Kell, Y. Zheng, L. Bulej, W. Binder, P. Tůma, D. Ansaloni, A. Sarimbekov, and A. Sewe. ShadowVM: Robust and comprehensive dynamic program analysis for the Java platform. In *Proc. 12th Intl. Conf. on Generative Programming: Concepts and Experiences*, GPCE'13, pages 105–114. ACM, 2013.

[17] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi. DiSL: a domain-specific language for bytecode instrumentation. In *Proc. 11th Intl. Conf. on Aspect-oriented Software Development*, AOSD'12, pages 239–250. ACM.

[18] L. X. Min and Q. H. Cao. Runtime-based behavior dynamic analysis system for android malware detection. *Advanced Materials Research*, 756:2220–2225, 2013.

[19] P. Moret, W. Binder, and E. Tanter. Polymorphic bytecode instrumentation. In *Proc. 10th Intl. Conf. on Aspect-Oriented Software Development*, AOSD'11, pages 129–140. ACM, 2011.

[20] G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[21] D. Octeau, S. Jha, and P. McDaniel. Retargeting Android applications to Java bytecode. In *Proc. 20th Intl. Symp. on the Foundations of Software Engineering*, FSE'12, pages 6:1–6:11. ACM, 2012.

[22] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. Effective inter-component communication mapping in Android: An essential step towards holistic security analysis. In *Proc. 22nd USENIX Security Symposium*. USENIX, 2013.

[23] H.-S. Oh, B.-J. Kim, H.-K. Choi, and S.-M. Moon. Evaluation of Android Dalvik virtual machine. In *Proc. 10th Intl. W. on Java Technologies for Real-time and Embedded Systems*, JTRES'12, pages 115–124. ACM, 2012.

[24] Oracle Corporation. JVM Tool Interface (JVMTI) version 1.2. Web pages at http://download.oracle.com/javase/7/docs/platform/jvmti/jvmti.html, 2007.

[25] N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant Tracks: Portable production of complete and precise GC traces. In *Proc. Intl. Symp. on Memory Management*, ISMM'13, pages 109–118. ACM, 2013.

[26] L. K. Yan and H. Yin. DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *Proc. 21st USENIX Security Symposium*, Security'12, pages 29–29. USENIX, 2012.