Polymer for Android

Submitted in partial fulfillment of the requirements for

the degree of

Master of Science

in

Information Security

Honghanh Bui-Nguyen

B.S., Electrical Engineering and Computer Science, University of California, Berkeley

Carnegie Mellon University
Pittsburgh, PA

May, 2016

Acknowledgements

This work was supported in part by the Army Research Laboratory under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

I would first like to thank my thesis advisor, Dr. Lujo Bauer of the College of Engineering at Carnergie Mellon University, for his guidance during the course of this research and feedback that taught me how to write better papers.

I would also like to thank my reader, Dr. Limin Jia of the College of Engineering at Carnegie Mellon University, for her frank critique at my defense. It has greatly improved this manuscript.

I thank Xiaoxiao Tang, Ph.D. candidate at Singapore Management University, for her collaboration and constant support. Without her, this research would not accomplish as much as it does today.

I give special thanks to Yannis Mallios, Ph.D. candidate at Carnegie Mellon University. He was a good sounding board whenever Xiaoxiao and I got lost in ideas.

Last but not least, I thank my husband. Without his enduring love and care, I would not be able to get through the tough days of graduate school.

Abstract

Building on the Polymer system designed by Bauer, Ligatti and Walker, which allowed enforcing user-defined security policies on single-threaded Java applications, this research extends Polymer to enforce policies on multiple applications, possibly distributed across several hosts. Using Android as a case study, we adapted Polymer to equip each app with a monitor, and we added communication capability and central storage so that monitors can regulate interactions between apps and make decisions based on their shared state. Our central storage design also includes load-linked and store-conditional operations to support synchronization of parallel updates, and each communication module is accompanied by a non-circumvention policy designed to protect the integrity, authenticity and confidentiality properties of the channel. The non-circumvention policy can be composed with user-defined policies that involve two or more apps. To demonstrate the efficacy of the system, we implemented and tested three policies: the first prevents apps from making background calls caused by confused deputy attacks or collusion attacks; the second disallows sending background SMS messages exceeding a specified quota, and the third enforces a specified device location sampling rate among all apps on the device.

Table of Contents

List of Figures

1

Introduction

Among mobile platforms, Android is the most popular globally, and since Android apps require access to a wealth of personal data to provide numerous services, they can also use that data for purposes that are not approved by its owners such as advertising, user profiling, and sending SMSes to premium numbers once the access is allowed. To detect and stop this abuse of personal data, one major direction is to use run-time monitoring. TaintDroid [6] is a run-time monitoring system for Android devices that detects apps who transmit privacy-sensitive data such as phone numbers, IMEI, IMSI, and geo-coordinates to external servers without informing their users. Because apps can also provide useful services while leaking privacy-sensitive data, AppFence [7] is another run-time monitoring system that can fake the transmission of privacy-sensitive data to keep those apps running. TaintDroid and AppFence are special-purpose run-time monitoring systems whose sole focus is protecting sensitive user information. On the other hand, Aurasium [9] is a general-purpose run-time monitoring system that can enforce a variety of run-time policies: stopping sensitive data from being transmitted to external servers, stopping apps from sending SMSes to premium numbers or executing as root user, and more. Furthermore, it allows users

to decide at run-time whether an action is allowed or not. However, Aurasium has two major drawbacks. First, though Aurasium's researchers claimed that Aurasium could "enforce any defined policy", they have not published a language specification to write policies for their system. Second, Aurasium cannot enforce policies that involve multiple apps.

A general-purpose system gives Android users custom access control over what Android's permission system already provides because users can define conditions where access to a resource should be granted or not. In addition, it also gives users more control than TaintDroid and AppFence do because it can enforce user-defined policies regarding privacy-sensitive data such as allowing apps to send geo-coordinates to external servers a limited number times instead of stopping all transmission. We aim to design a general-purpose run-time monitoring system that can enforce a wide variety of policies like Aurasium can but without its drawbacks. Our system would have a clearly defined policy language, and it would be able to enforce both policies involving individual apps and policies involving groups of apps. In order to enable our system to enforce the latter type of policies, our monitors need to communicate and exchange information about their targets, and this research will explore different methods of communication between monitors. We choose our communication methods based on a few example policies that we want our system to enforce. We now present those policies.

An example policy that regulates individual apps is one that allows each app to send only a limited number of SMSes without user intervention. If apps can send SMSes without user intervention, they can send many SMSes without users' knowledge, but the users will have to pay for those SMSes. Thus, sending SMSes without user intervention is an action that needs to be monitored. A user may want to grant an app this ability if it provides a useful service in return while limiting financial damages should the app abuses its power. To do so, users can define the

maximum number of SMSes that apps can send without their knowledge in the above policy and use our run-time monitoring system to enforce it.

The second example policy belongs to a class of policies to stop one of Android's infamous privilege escalation attacks, in which an app that does not have access to a resource uses another app that has access to retrieve that resource. This policy demonstrates why monitor communication is essential. When monitors can communicate, the monitor of an app that starts another app can tell the monitor of the latter which resources its target can access, so the latter's monitor can decide whether the caller app is allowed to access the requested resource. If the caller app does not have the required access, then the callee app's monitor will stop it from returning the requested data or performing the requested action. This policy allows apps with permission to make background calls to make background calls themselves or by using other apps to do so while prohibiting apps without this permission from calling other apps to make background calls.

Another example policy where monitor communication is necessary is a policy that limits the number of geo-coordinates given to apps in a period of time. Although geo-coordinates can be used for extremely privacy-invasive purposes such as surveillance, they can also be used to provide great benefits such as navigation or getting a list of nearby amenities. Thus, to reduce the amount of data that can be used for privacy-invasive purposes, users may choose to enforce the described policy. If there is no monitor communication, the run-time monitoring system won't know if apps have gathered more geo-coordinates than the limit as a group while not exceeding that limit individually. Then, they can pool their knowledge, which would result in each app having more geo-coordinates than the users want. To effectively prevent apps from knowing more about users' movement, we need the monitor of each app on all location-enabled Android devices that users may carry with them at the same time (e.g. smart phones, smart watches and tablets) to tell each other how many

geo-coordinates their targets have gathered, so they can stop apps from exceeding the collective limit.

Our system can also provide on-demand authorization, where users can allow or prohibit an action at run-time. This is similar to a feature of Aurasium, where a dialog box appears whenever a privacy- or security-relevant action is intercepted, and Aurasium's users can choose "Yes" to permit the action, "No" to suppress it, and "Kill App" to stop the offending app. However, unlike Aurasium, our system would allow a user of one device to authorize an action on another device. With this capability, we can enforce a variety of parental control policies where monitors on a child's device can request permissions to execute from his parents, and they can grant or deny them using their own devices.

In addition to having communication capability, there are other requirements to make a mobile device's monitoring system usable. First, it should not modify the devices' platform because many users don't know how or don't like to root their devices. Second, it should have a policy language that is easy to understand and in which it is easy to express different policies. Third, it should have a user-friendly control interface to give feedback about policy enforcements to help users refine their policies. Useful features of this interface are: a list of monitored apps, a history of enforced policy decisions, and a place for users to specify policy parameters if they are needed.

There are two reasons why we chose to extend Polymer [4] into a general-purpose run-time monitoring system: first, it has a formal policy language that allows policy composition, which makes policy writing easier; second, it works for Java applications (albeit only single-threaded ones), and Android apps are typically written in Java. We also chose to modify only apps' deliverables, which is anything in an APK file. At a starting point, we limit ourselves to two high-level tasks: one, adapting Polymer to run on Android; two, adding communication capability as we discussed previously.

The layout of this paper is as follows: Section 2 will give the background on run-time monitoring, Polymer, and Android; Section 3 will discuss related work on Android and Java run-time monitoring; Section 4 will give an overview of our extensions to Polymer; Section 5 will talk about the implementation details; Section 6 will evaluate our resulting system; Section 7 contains our conclusions, and Section 8 discusses future work.

2

Background

2.1 Run-time monitoring

Before we dive into Polymer's system overview, this section will give a short discussion of run-time monitoring's concepts.

Conceptually, a run-time monitor runs in parallel with the program that it is monitoring. A program's execution is expressed as a sequence of actions; the monitor intercepts each action and decides whether to allow or disallow the action or execute another action altogether. A run-time policy defines the monitor's decisions. It can be thought of as a transformation function for action sequences, but with the requirement that it must create the output action sequence by transforming one action at a time because monitors make decisions as the program "runs". By extension, this also means that a run-time policy cannot undo its transformation of a past action; nor can it decide to transform a current action based on a future action that it hasn't intercepted. For action sequences in which no action depends on the output of a previous action, a run-time monitor can make its decision after observing the whole sequence by suppressing every action in the sequence until the last one, at

which time the monitor makes a decision and outputs the transformed sequence. An example sequence that has this property is `{ print("Hello"); print("World"); }`, and an example sequence that doesn't have this property is `{ x = input(); y = x * x;}`.

2.2 Polymer

Polymer is a run-time monitoring system with a combination of features that will later prove indispensible for writing policies that use monitor communication.

First, it has a policy language for defining how a program is to be altered. Among related research, only Java-MAC (section 3.2) and MOP (section 3.3) have formal language specifications. AppFence (section 3.1) has no language; the system defines a fixed transformation for each action. Aurasium (section 3.4) uses a configuration file, but its format was not published, so it is unclear what can or cannot be expressed by Aurasium configuration specifications. Then, out of the two systems with formal language specifications, Java-MAC's language allows only one modification to a program's behavior, which is raising an alarm when there is a policy violation. This is a system-defined action, and users cannot change it. However, Java-MAC language specifies actions in term of changes to variables and method calls while Polymer's language can only specify actions in term of the latter. MOP has several languages, all of which describe actions in term of method calls, and their various handlers are like Polymer's replace suggestions since they both allow users to define arbitrary code to be run when a condition is met. However, unlike Polymer, they also have expressions to describe action sequences, such as {`Iterator.hasNext()`; `Iterator.next()`;} and state transitions triggered by actions. In this regard, Polymer's users have to create their own methods to express state changes and action sequence matches.

Despite this shortcoming, Polymer's policy language has an important feature

that no other language has, which is the support for policy composition because it separates the making of a policy decision from the enforcement of that decision. When two policies are composed, both of them maybe asked to make a decision, but only one decision may be enforced. Since none of MOP's languages has this distinction, it is not possible to compose policies in a methodical way using them. While a MOP policy writer can still write one policy that is the combination of two sub-policies, this process is not simplified by any mechanism provided by MOP's languages. Java-MAC has the same drawback as MOP, and since Aurasium does not publish its configuration format, it is unknown whether they have that support. Policy composition is important because it allows a complex policy to be built from simpler ones. This not only makes policy writing easier, but it also makes proving the correctness of a policy easier because proving the correctness of a policy means showing that the written policy both captures all of functions relevant to the user-intended policy and makes the appropriate decisions about them, and reasoning about a policy that captures only five functions is easier than reasoning about one that captures twenty.

The remainder of this section will be a detailed discussion of Polymer's system architecture, its many components, how policies are compiled, how function calls are captured and how the monitoring process works.

2.2.1 Polymer's system architecture

The overall design of Polymer is illustrated in Figure 2.1. The system requires three inputs: the policy definition written in Polymer language (this file has a .poly extension), the target program as a collection of .class files, and a file that lists all security-relevant functions. The third file is used to reduce the number of functions that need to be monitored and thus reduces the performance overhead that a monitor imposes on its target. The diagram only shows one policy file for clarity, but users can

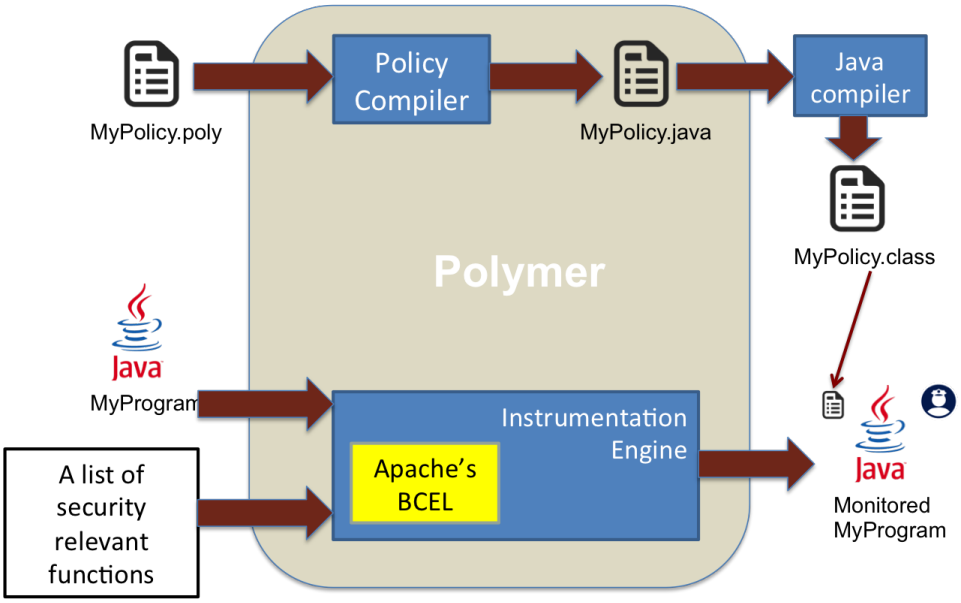

Figure 2.1: Polymer system overview

input multiple policy definitions along with Java source code for any support classes. After the inputs are processed, the system produces the instrumented program as a collection of .class files, along with the policies' .class files.

There are two components that make up Polymer: a policy compiler and an instrumentation engine. The policy compiler translates `.poly` files to `.java` files. Then, any Java compiler can be used to compile those files to `.class` files. The instrumentation engine is responsible for adding the monitor's software to the target program. This software is a library that includes two important functions: the first defines the policy to be enforced, and the second tells the monitor whenever a security-relevant action is about to be executed and which one it is. This is how the monitor knows when it needs to make decisions according to which policy. Since these APIs are to be called from inside the target program, the instrumentation engine modifies the main function to define a policy before starting the program; it also modifies all security-relevant functions to call the monitor before executing.

```
aswitch(a) {
  case <void android.app.Activity.startActivity (Intent i)>:
  // Compute monitor decision
  case <* android.telephony.SmsManager.sendTextMessage(..)>:
  // Compute monitor decision
}
```

Figure 2.2: aswitch example

In order to do this, it uses Apache's BCEL [1] library to manipulate classes and methods that are compiled in bytecode. The modified target program is then packed with compiled policies from before.

2.2.2 Policy language

The policy language is an extension of Java. One extension is a new `aswitch` statement that accepts Polymer's `Action` objects as input. `aswitch`'s `case` selector uses action pattern expressions. Figure 2.2 shows two types of matching: the first is exact matching where the method's fully-qualified name, return type and all inputs are specified; the second is partial matching where a wildcard character `"*"` replaces the return type and a wildcard string `".."` replaces the arguments of the function. In the first case, only calls to the function `android.app.Activity.startActivity` that take a single input of type `Intent` and the return type `void` would match, whereas in the second case, any call to any function with the name `android.telephony.SmsManager.sendTextMessage` would match. The character `"*"` would match any return type, and the string `".."` would match any type and any number of arguments.

Figure 2.3 shows a simplified `Policy` class. In order to define a policy, policy authors must sub-class `Policy` class and define the `query()` method. Optionally, they can also overload the `accept()` and `handleResult()` methods if their policies

```
public abstract class Policy {
  abstract public Suggestion query(Action a);

  public void accept(Suggestion s) { }

  public void handleResult(Suggestion s, Object result, boolean
      wasExnThn) { };
}
```

Figure 2.3: Policy class

are not stateless. Every time the monitor intercepts a security-relevant function, it calls the `query()` method. This method returns its transformation in the form of a `Suggestion` object. Then, before the monitor enforces the transformation represented by `Suggestion`, it calls `accept()`. After enforcement, if the executed action returns a value, the monitor sends that value to policy by calling `handleResult()`. If the executed action results in an exception, the monitor also calls `handleResult()` with `wasExnThn=true`.

Figure 2.4 shows that there are six types of `Suggestion`, and each type is a sub-class of the `Suggestion` class.

- Irrelevant: an identity transformation. Policies return this suggestion when an Action is not relevant to them.
- OK: also an identity transformation. Policies return this when an Action is relevant and is allowed to run.
- Replace: transforms an Action to another Action.
- Insert: transforms action A to the sequence (B, A) where B is defined by the policy.
- Suppress: transforms A to nil, meaning the Action is not executed.

```
public abstract class Suggestion {
  public abstract boolean isIrrelevant();
  public abstract boolean isOK();
  public abstract boolean isReplace();
  public abstract boolean isSuppress();
  public abstract boolean isHalt();
  public abstract boolean isInsertion();
  public abstract Action getTrigger();
  public abstract Policy getSuggestingPolicy();
}
```

Figure 2.4: Suggestion class

```
import ../polyandroid/comm/policy/DataBankComm.ifc;

<* android.app.Activity.startActivity(..)>
<* android.location.LocationManager.requestSingleUpdate(..)>
<* *.onCreate(..)>
<* android.telephony.SmsManager.sendTextMessage(..)>
```

Figure 2.5: List of security-relevant functions

- Halt: not only transform A to nil, but also stop the application.

2.2.3 List of security-relevant functions

This list describes all security-relevant functions that the target program may have. Since there can be many security-relevant functions especially due to method overloading, they are expressed as action patterns like the one used in `aswitch`, and each pattern may match more than one function. The list is written in one file or more; there is one pattern per line, and the `import` statement allows one file to include other files. Figure 2.5 is an example of this file.

```
public class EditAutoMonitor {
  public static void setPolicyFile(String filename) {}
  public static void run_$$POLY_METHOD$$(Action a) {}
  public static void after_$$POLY_METHOD$$(Object result,
      boolean wasExnThrown) {}
  public static void haltTarget(Policy p) {}
}
```

Figure 2.6: EditAutoMonitor

2.2.4 Polymer monitors

`EditAutoMonitor` class is the implementation of a Polymer monitor. Since there is only one monitor per application, all of `EditAutoMonitor` functions are static. Figure 2.6 shows the relevant functions of `EditAutoMonitor`. The function `setPolicyFile()` is called in the program's `main()` function to define the policy class. `run_$$POLY_METHOD$$()` is called whenever a function in the security-relevant list is about to be invoked. In turn, it calls the policy module's `query()` and `accept()`. If the policy returns HaltSuggestion, it calls `haltTarget()`. If the policy returns ReplaceSuggestion, it executes the replacement action and calls `after_$$POLY_METHOD$$()`, which calls the policy's `handleResult()`. `after_$$POLY_METHOD$$()` is not called when SuppressSuggestion is returned because the action won't be executed, so there is no result. For the remaining Suggestions, OK and Irrelevant, `after_$$POLY_METHOD$$()` is called after the original function runs.

2.2.5 Instrumentation

The previous section discussed when various functions of `EditAutoMonitor` are called to perform their roles in monitoring a target program; this section will discuss how the target program is modified, so those functions are called at the right time.

The instrumentation engine uses BCEL library to parse .class files, add new func-

```
public static void main(String[] args) {
  EditAutoMonitor.setPolicyFile("policy.MyPolicy");
  // original code of main()
}
```

Figure 2.7: Policy installation

```
public class MyClass {
  private static Scanner openFile (String name) {
    return new Scanner(new File(name));
  }

  public static void main (String[] args) {
    Scanner s = openFile( "hello.txt");
    // more code
  }
}
```

Figure 2.8: An example of uninstrumented program

tions, and change existing ones. Figure 2.7 shows the addition of `setPolicyFile()` at the beginning of the `main()` function. That is an example of a simple modification; the modification needed to call `run_$$POLY_METHOD$$()` and `after_$$POLY_METHOD$$()` for each security-relevant function is more extensive. There are two variants: call definition instrumentation modifies the body of the intercepted functions and their classes while call site instrumentation modifies the callers of those functions and the callers' classes. To demonstrate how each variant intercepts a method, we created a simple example.

- First, Figure 2.8 shows an example of a program's original code that calls a security-relevant function, `openFile()`. The definition of `openFile()` is also shown.

- Next, Figure 2.9 shows the transformation to the program when a function's

```
private static Scanner openFile (String name) {
  Object[] arrayOfObject = { name };
  Action monaction = new Action(null, "MyClass", "openFile",
    "(Ljava/lang/String;)Ljava/util/Scanner",
    arrayOfObject, "openFile");
  try {
    EditAutoMonitor.run_$$POLY_METHOD$$(monaction);
  } catch (ReplaceException replaceException) {
    Scanner result = (Scanner) replaceException.getValue();
    return result;
  } catch (SuppressException suppressException) {
    return null;
  }

  try {
    Scanner result = MyClass.openFile_WRAP(name);
    EditAutoMonitor.after_$$POLY_METHOD$$(result, false);
    return result;
  } catch (RuntimeException e) {
    EditAutoMonitor.after_$$POLY_METHOD$$(e, true);
    throw e;
  }
}

private static Scanner openFile_WRAP(String name){
  return new Scanner(new File(name));
}

public static void main (String[] args) {
  // openFile is security-relevant function
  Scanner s = openFile("hello.txt");
  // more code
}
```

Figure 2.9: Example call definition instrumentation

call definition is instrumented. The code body of `openFile()` is moved to a new function, `openFile_WRAP()`, which has the same parameter, return type, and modifiers. `openFile()` has a new body in which `run_$$POLY_METHOD$$()` and `after_$$POLY_METHOD$$()` are called. Because the body of the original function is changed, whenever it is called, the monitor is called. The call site in `main()` is unchanged.

```
private static Scanner openFile (String name) {
  return new Scanner(new File(name));
}

public static void main (String[] args) {
  Scanner s = openFile_WRAP(null, "hello.txt");
  // more code
}

private static Scanner open_WRAP(String param1) {
  Object[] arrayOfObject = { param1 };
  Action monaction = new Action(null, "MyClass", "openFile",
    "(Ljava/lang/String;)Ljava/util/Scanner", arrayOfObject, "openFile"
      );
  try {
    EditAutoMonitor.run_$$POLY_METHOD$$(monaction);
  } catch (ReplaceException replaceException) {
    Scanner result = (Scanner) replaceException.getValue();
    return result;
  } catch (SuppressException suppressException) {
    return null;
  }

  try {
    Scanner result = MyClass.openFile(param1);
    EditAutoMonitor.after_$$POLY_METHOD$$(result, false);
    return result;
  } catch (RuntimeException localRuntimeException) {
    EditAutoMonitor.after_$$POLY_METHOD$$(localRuntimeException, true);
    throw localRuntimeException;
  }
}
```

Figure 2.10: Example call site instrumentation

- Alternatively, Figure 2.10 shows the transformation to the program when call site instrumentation is used. The original function is unchanged while the call site in `main()` is modified to call a new function, `openFile_WRAP()`. This function invokes the monitor. In this example, `openFile()` has only one call site in `MyClass.main()`, so there is only one `openFile_WRAP()` created in `MyClass`. If `MyClass` contains more calls to `openFile()`, then all of them would be changed to call `MyClass.openFile_WRAP()`. If there are classes `Foo` and `Bar` that have

calls to `MyClass.openFile()`, then there will be a `Foo.openFile_WRAP()` and a `Bar.openFile_WRAP()`; the call sites in `Foo` will be changed to the former, and the call sites in `Bar` will be changed to the latter. This is different from call definition instrumentation where all changes are in `MyClass`. This also means that an implementation of call site instrumentation may not intercept all invocations of a function: for example, if a function is invoked by native code, and native code is not instrumented, then that invocation won't trigger the monitor.

While there are minor differences between the two implementations, such as how the original code is called once it is allowed to run, they are largely the same. When a function is invoked, `run_$$POLY_METHOD$$()` is called. `after_$$POLY_METHOD$$()` is called after the original function body is executed, but in the case where the original function body won't be executed (e.g. when Replace, Suppress, or Halt Suggestion is returned by a policy), `run_$$POLY_METHOD$$()` will invoke `after_$$POLY_METHOD$$()`. Replace or SuppressException are used to deliver a return value when a Replace or SuppressSuggestion is enforced.

2.2.6 Conjunction combinator

Combinators are used to compose policies. While there are other combinators, this section will only discuss the `Conjunction` combinator because it is used to compose user-defined policies with non-circumvention policies.

A conjunction combinator is composed of two sub-policies; it is also a Policy. When it is queried, it queries the two sub-policies and returns the most restrictive Suggestion of the two. However, if either sub-policy returns InsertSuggestion or ReplaceSuggestion, a HaltSuggestion is returned by the combinator because an Action is altered in Insert or ReplaceSuggestion, and it is not possible to quantify the restrictiveness of each.

2.3 Android Application

Since Polymer is designed to monitor Java applications running inside JVMs, transforming it into a monitoring system for Android apps requires changes necessitated by the design of the Android system and its apps. Therefore, this section will give a brief overview of Android and how apps are packed and distributed.

Android system executes each app in the app's own process with its own username. The process runs a Dalvik VM, which is similar to JVM but is designed for efficiency. Android apps are mainly written in Java, but some apps also contain native code. Typically, if apps are developed on Android Studio, Android Studio will compile Java source code to Dalvik bytecode, but if another development method is used, Java source code can be compiled to JVM bytecode, then be converted to Dalvik bytecode with a number of open-source tools; dex2jar [2] is one tool.

Each Android app can have multiple entries: each Activity, Service, ContentProvider, and BroadcastReceiver is one. Each of these entries must be declared in AndroidManifest.xml. Android apps are typically multi-threaded: if an app has both a UI component and a Service, then each runs in their own thread. Furthermore, apps can spawn background threads to handle long-running tasks so that they won't freeze device screens.

To access a resource, apps need to request permission from users. There are over 100 defined resources with associated permissions in Android. Prior to Android 6.0, apps declare the permissions they need in AndroidManifest.xml, a file that accompanies each app. The Android system uses this file to ask users to grant permissions when apps are being installed. As of Android 6.0, in addition to declaring permissions in AndroidManifest.xml, apps also need to call `requestPermissions()` as well because Android only asks users to grant permissions whenever `requestPermissions()` is called.

Apps' deliverables are packed into .apk files. Users can install these files onto devices using "adb" program provided by Android SDK. `apktool` [3] is an open-source tool that we use to unpack .apk files in our research.

3

Related Work

This section discusses several run-time monitoring systems where users can alter target programs' control flow in some ways. We also describe their drawbacks.

3.1 AppFence

AppFence [7] is designed to prevent privacy-sensitive data from being leaked by apps while keeping the apps running by fooling them with shadow data. An example of shadow data is an empty contact list. It can also stop exfiltration attempts, which is when privacy-tainted data is sent to external servers, by intercepting OS-provided network functions and dropping buffers containing private data. Then, it will fool the offending apps into thinking that the data has been sent or that there is no network connection by manipulating the return value of network functions. The first drawback of this system is that it enforces only the policy discussed here because it is a special-purpose system. The second drawback is that it modifies many components of Android.

3.2 Java-MAC

Java-MAC [8] is a run-time verification system that can verify user-defined properties. It has a language to define events and conditions to be tracked. According to the taxonomy of this system, events occur when the program is running, and conditions are states that hold for a period of time. Its language allows users to define which events and conditions to monitor and when to raise alarms. However, it does not allow policy writers to modify the behavior of the monitored programs beyond raising alarms, which is a system-defined action. Its language does not have expressions to distinguish the monitor of one program from another's because it was designed to monitor individual applications only.

3.3 MOP

MOP [5] is one of the most prominent run-time monitoring system for Java applications. The MOP framework has several languages for specifying run-time properties and handlers for when violations are detected; each of the languages is called a logic module in MOP's taxonomy. Handlers are a way to divert the control flow of a program when a condition is met. Therefore, while there is no native expression for communication between monitors in MOP, policy authors can create their own by defining a handler. However, MOP can only enforce policies on an individual process, and there is no expression in its language to compose policies, so adding the same communication mechanism to different policies requires each policy to be updated differently to prevent the communication from being corrupted. Additionally, the resulting policies will need to be reviewed entirely for correctness.

3.4 Aurasium

Aurasium [9] is a run-time monitoring system for Android that can enforce a wide range of policies without modifying Android. It claims to be able to intercept virtually any calls to Android functions from apps by adding a few native code modules to APK files. These modules would trap all calls from an app to the Linux kernel to call its monitor first. Aurasium can intercept calls regardless whether they are from Java code or native code. However, the monitoring system is not yet accompanied by any policy language, and without a clear policy language, users can't be sure that they have expressed a policy that makes the monitors work the way they want it to. Another drawback is that Aurasium only monitors individual apps so far.

4

Overview of Polyandroid

This section will provide an overview of the additions to Polymer. Each sub-section will having a matching sub-section in "Implementation of Polyandroid" (Section 5). Apps and policies that are mentioned in this section and the next are also summarized in "Evaluation" (Section 6) along with our evaluation of these additions.

From this point on, to distinguish this extended version of Polymer that monitors Android apps from the original Polymer that monitors regular Java applications, the former will be referred to as Polyandroid.

4.1 Design constraints

This section will list the constraints that influence Polyandroid's designs. One of them is self-imposed, and the other are to simplify its requirements.

The first constraint, which is self-imposed, is that Polyandroid cannot modify any component of the Android system, or any Google apps because they are signed by Google's key, and therefore they have more privileges than the other apps. We set this constraint so that the users of Polyandroid won't have to root their devices.

The second constraint is that for each communication method used by a monitor,

its non-circumvention policy must be enforced on every app that can access the medium used by the method. Each communication method is accompanied by a non-circumvention policy as will be discussed in Section 4.3. Non-circumvention policies detect unauthorized accesses to monitors' communication media by non-monitors, and they are only effective when every app that can access those media is monitored. For example, if inter-process communication (IPC) is used by two monitors, then every app on the same device will need to be monitored for unauthorized accesses to the same IPC channel; if two monitors on different devices communicate by leaving messages on a web server, then all apps on both devices need to be monitored for making unauthorized HTTP requests to that server, and the server must also block all connections that are not from monitored devices. Still, users can enforce different policies on each app; they would only need to combine their policies with non-circumvention policies. There is one consequence due to the first constraint: Because we cannot modify Google apps, we cannot enforce policies on them. This means that Google apps are trusted components in our system.

The third constraint is to monitor only single-threaded apps because Polymer only works on single-threaded Java applications, and monitoring multi-threaded applications requires thoughtful design changes that are outside the scope of this research. This means that we can monitor apps that only have Activities and create no background threads because Android multiplexes the execution of all activities of an app onto a single UI thread. A monitored single-threaded app can still interact with unmonitored multi-threaded apps without affecting the efficacy of its monitor, but as a consequence of the second constraint, since multi-threaded apps cannot be monitored, existing monitors cannot use any communication medium that an unmonitored multi-threaded app can access.

4.2 Communicators

Like Polymer, Polyandroid is a Java library that is added to apps by the instrumentation engine. Communicators are classes in this library; each communicator class implements communication using a different medium. Since each app is equipped with a monitor, policy writers can choose a suitable communicator class for their policies to exchange messages between monitors of different apps. We created two communicator classes, `IntentExtraComm` and `DataBankComm`; they are the primary additions to Polymer. This section will describe general APIs and requirements of a communicator class, and Section 5.2 will discuss them in more detail.

Monitors send each other messages to help make policy decisions, so each message is a parameter to a decision function. Therefore, each message is designed as a key-value pair to identify the parameter and its value. A communicator class will have methods for sending and receiving messages, and a method to specify its non-circumvention policy.

Because efficient power consumption is very important to mobile devices, communicators cannot be constantly running in the background when there may be no communication. Because monitors are active only when the monitored apps are running, not all monitors are active at the same time; therefore, it is very likely that a message's intended recipient may not receive a message until long after it is sent. Thus, most communicators' design should include a way to store messages until they are retrieved. This also means that sending operations are asynchronous. Receiving operations may or may not be synchronous. Our current communicators only have synchronous receive functions because neither of them takes more than a few seconds to receive a message. However, we expect inter-device communicators to have asynchronous receive functions because an operation may take more than a few seconds due to congested network traffic, busy devices or weak signal strength. The lack of

synchronous send functions implies that if policy decisions depend on one monitor's answer to another monitor's question, the policy must be written to put the monitored apps on hold in the background or to stop it temporarily until an answer is received.

There are two kinds of communication: one-to-one is a communication between only two monitors, and one-to-many is a communication between a group of monitors. Currently, we only have a global group, which includes all monitors that communicate via the same communicator class. In order to have different groups defined by their members, we will need membership registration and group IDs; this is a future task. `IntentExtraComm` is an example of one-to-one communicators and `DataBankComm` is an example of one-to-many communicators. However, it is possible to extend `DataBankComm` to support one-to-one communication as well.

4.3 Preventing circumvention of communicators

From the perspective of the Android operating system, monitors are part of the monitored apps; therefore, they run with the same privileges. Since communicators store messages that directly affect policy decisions, the messages are attractive targets for corruption, meaning unauthorized modification, removal, or insertion of messages done by non-monitors. This section gives an overview of how to craft non-circumvention policies for communicators.

There are two places where messages can be corrupted. The first is during transmisison, and the second is during storage. Since communicators uses Android APIs to transmit and store messages, they rely on the Android operating system to protect messages in transmission and storage. This means that they rely on the guarantees that messages won't be read or changed except through well-defined APIs provided by Android. For example, a communicator stores its messages in a private file of `MonitorDataBank` app, which has a ContentProvider. The ContentProvider's

`insert()` is the only function that reads or writes to this file. That communicator relies on two guarantees given by the Android system. First, no app other than `MonitorDataBank` can read or write to the file. Second, messages won't be modified during IPC, which facilitates exchanges with ContentProviders.

Consequently, preventing message corruption is reduced to defining a non-circumvention policy to intercept all functions that would modify the messages in the target program and halt the program when such a modification is attempted. This means that the more functions there are, the larger the policy is, and the more overhead it adds to the target program. Therefore, a good communicator's design should restrict the number of APIs that apps can use to insert, remove or modify messages.

4.4 Declaring Android permissions

Sometime a policy module requires monitors to have certain Android permissions in order to enforce its decisions. This section will discuss possible reasons for requiring permissions, and the mechanism by which permissions are obtained.

One reason to require permissions is so that monitors can communicate. For communication between apps on different devices, the `INTERNET` permission is typically needed, but the monitored apps may not need Internet and so may not declare that permission.

Even without communication, a policy may also require some permissions. For example, a policy that logs the invocations of security-relevant functions may need `WRITE_EXTERNAL_STORAGE` permission if the logs are on an external storage, and this permission needs to be declared if it isn't already declared by the target apps.

The instrumentation engine is responsible for modifying the AndroidManifest.xml file of each app to add any permission that is required by a policy but is not already declared by the app itself. In order for the instrumentation engine to know which

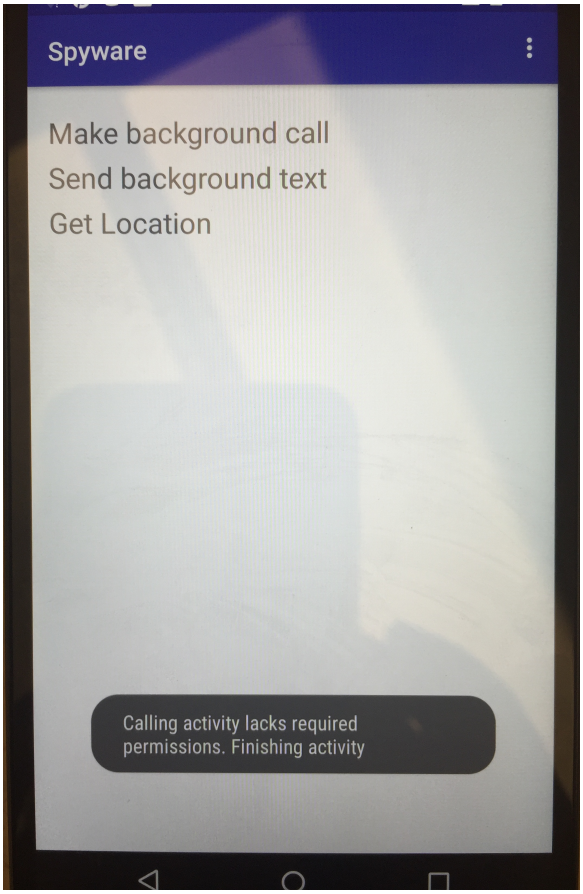

Figure 4.1: Example of a monitor notifying users about an enforcement

permissions to add, a new API is added to `Policy` class, and this API returns a list of required permissions.

4.5 Informing users of enforcement

Because the goal of Polyandroid is to give users more controls over the apps they use, the execution of the monitors themselves should not be a blackbox. Otherwise, users will not know whether their policies are enforced as they intended. Figure 4.1 shows an example of a monitor informing a device's user that the Spyware app cannot make background calls because it lacks that permission. The Spyware app in this

example is attempting to use the Utilities app to make background calls, which is a privilege escalation attack that the example's policy disallows. This is a notification feature that monitors use to inform users of important actions. While users may not be notified of every monitor action because there are likely to be too many to constantly disrupt users' focus, this is a first step in letting them know when their policies are enforced.

5

Implementation of Polyandroid

Our first task is porting Polymer to Android. Sub-section 5.1 will talk about changes that we made to Polymer to accomplish this task and a challenge that we encountered while doing so. Our second task is creating APIs for different communication methods that policy writers can choose for their monitors. We encapsulated the implementation of each communication method in its own Java class. Each class has its own public functions that can be used to facilitate communication. However, we also created a Java interface, called `Communicator`, to impose a set of common functions that all communication classes should provide, and each class implements this interface. Sub-section 5.2 describes the `Communicator` interface and two classes that implement two communication methods, `IntentExtraComm` and `DataBankComm`, along with the use case for each communication method which affects their designs. Because Section 4.3 already mentioned why non-circumvention policies were needed and what they contained at a high-level, our third task is to show readers how to construct non-circumvention policies via an example. Sub-section 5.3 will show the non-circumvention policy of one of our communication classes, `DataBankComm`, and explain how it prevents messages from being corrupted. Our fourth task is to explain

how the instrumentation engine adds permissions that a policy needs to AndroidManifest.xml and to discuss the implications of adding new permissions to apps. This is carried out in Sub-section 5.4. Sub-section 5.5 will carry out our final task, which is detailing how the user-notification feature that was mentioned in Section 4.5 is implemented.

5.1 Porting Polymer to Android

Before we can implement communicators, we must adapt Polymer to work on Android first, so this section will discuss the necessary changes to do so.

5.1.1 Policy installation

The first change is where policies are installed on the apps. Originally, the policies are installed in `main()` functions (see Figure 2.7). However, Android apps do not have `main()` functions, so policies will have to be installed somewhere else.

In Android, an application has multiple entries: each Activity, ContentProvider, Service, and BroadcastReceiver is an entry point. Whenever one of these components is used, Android automatically instantiates it if there is no existing instance of that component. Then, right after a component is instantiated, its `onCreate()` method is called. BroadcastReceiver is the exception since that component does not have an `onCreate()` method. We can install the policy module by calling `EditAutoMonitor.setPolicyFile()` in `onCreate()` for Activities, ContentProviders, and Services, while for BroadcastReceivers, we can put this call in their constructors. However, since we limit our scope to monitor only Activities, we did not instrument the other three components yet.

```
public abstract class Policy {
  private Activity activity;

  // existing APIs

  public void setActivity (Activity a) { this.activity = a; }
  public Activity getActivity () { return this.activity; }
}
```

Figure 5.1: Policy tracking of current Activity

5.1.2 Tracking Android Activities in policies

In Android, Activities are UI components, and Android multiplexes their execution onto a single UI thread for each app. However, sometime it is useful for a policy to stop an Activity instead of halting the app because halting is a severe punishment. An app is halted by killing its process, which means killing all of its running components in addition to quitting without closing any open resources like files or sockets, leaving those tasks to the operating system. In order to allow policy writers to tell monitors to quit an Activity, we added two APIs to Policy class: `getActivity()` and `setActivity()` (see Figure 5.1). Whenever a policy wants to finish the current activity, it creates a ReplaceSuggestion whose value is the action `getActivity().finish()`.

Even though all Activities are multiplexed on the same thread, separating the execution trace of each activity may be useful. An Activity can be thought of as an encapsulated module that interacts only through well-defined APIs which are `startActivity()`, `startActivityForResult()` and `onActivityResult()`. These methods allow one Activity to invoke another, specify parameters and receive results. While it is possible for two Activities of the same app to interact by calling each other's methods, or passing data through app-global variables, this is not a typical design. By separating the execution trace of one Activity from another's, one can eas-

```
public static void setMain(Activity m);
public static void resetMain (Activity m);
```

Figure 5.2: setMain() and resetMain()

ily write a policy that analyzes the behavior of each Activity and guesses its purpose. Thus, we create a different Policy instance for each running Activity. This is achieved by instantiating the policy class in `EditAutoMonitor.setPolicyFile()`, which is called in `Activity.onCreate()`. Then, to associate a policy instance with an Activity, we added two functions to EditAutoMonitor, `setMain()` and `resetMain()` (see Figure 5.2). `setMain()` is called after `setPolicyFile()` in `Activity.onCreate()`, and it calls `Policy.setActivity()` of the policy instance that was just created. The function `resetMain()` is needed because Android Activities can be paused and unpaused as they come out and come back in focus on a device's screen. When an Activity is paused, another Activity may be unpaused. EditAutoMonitor must replace the policy instance of the old Activity with one of the new Activity, and this is accomplished by calling `resetMain()`. This function looks up a mapping from Activities to Policies to find the existing Policy object associated with the input Activity. The call to `resetMain()` is added to `Activity.onResume()` method.

5.1.3 Making BCEL work with dex2jar

To our surprise, we discovered that the dex2jar tool sometimes won't convert the bytecode that we modified with BCEL. To make sure that we did not have a bug in our implementation, we wrote a simple test in which:

1. We parse the original code with BCEL, and create a `MethodGen` object, which can be used to modify Java methods, for each method in every class.

2. Then we call `MethodGen.toMethod()` to write out the original method without

```
private void foo (Object arg) {
    // code
    arg = new String[2];
    // code
}
```

Figure 5.3: Decompilation of some methods from Dalvik bytecode

making any changes.

Even when we didn't make any changes, the code that is generated by BCEL is different from the original code, causing dex2jar to fail. Upon investigation, we found that for some methods in some .class files that are written by BCEL have `local_variables_table` attributes that didn't exist before. The existence of this attribute in combination with the quirk of Dalvik bytecode shown Figure 5.3 causes dex2jar to fail when it detects an object of one type is assigned to a variable of a different type.

We bypassed this problem by calling `MethodGen.removeLocalVariables()` before calling `MethodGen.toMethod()` to remove the `local_variables_table` attribute from all methods' bytecode.

5.2 Communicators

Our design of `IntentExtraComm` is to enable enforcement the anti-privilege escalation policy that we mentioned in "Introduction" (Section 1), where an app with the permission to make background calls is permitted to make them, but an app without that permission is not allowed to use another app to make background calls. Our design of `DataBankComm` is to help enforce a simpler version of another policy that we also mentioned in "Introduction", a policy that limits the number of geo-coordinates that all apps can collect as a group. However, instead of monitoring all apps on all

```
public interface Communicator {
        public void send (String receiver, String key, String value);
        public void send (String receiver, String key, byte[] value);
        public Message recv (String key);
        public Message[] recvAllMessages ();
        public static Class<? extends Policy> getNonCircumventionPolicy
            () {
                return null;
        }
}
```

Figure 5.4: Communicator interface

devices that are at the same location, we limit ourselves to monitor only apps on a single device. Each of the communicators implements our `Communicator` interface in Figure 5.4. In the process of perfecting these communicators' designs, we discover some important requirements that a communicator should have. This section will discuss those requirements and the implementation of each communicators in detail.

5.2.1 Communicator interface

Figure 5.4 shows the basic APIs of a communicator: some for sending messages, some for receving messages, and one for returning a non-circumvention policy to be composed with the policies that use the communicator. When sending, a monitor can optionally specify a receiver ID. If no ID is specified, the message can be read by any monitor that has access to the communication medium. A message's value can be a `String` or an array of bytes. The `Message` class is a way for `recv()` and `recvAllMessages()` functions to return tuples of `(senderID, key, value)`. The communicator automatically attaches the sender's ID to each message that it sends. How monitor IDs are given to communicators depends on the design of each communicator.

5.2.2 IntentExtraComm

`IntentExtraComm` is a one-to-one communicator, created to enable enforcement of anti-privilege escalation policies like our example. Because this type of privilege escalation always involves one app starting another app, and an Android's Intent object is delivered by the Android system from the caller app to the callee app in the process, `IntentExtraComm` uses this object to deliver messages from the caller's monitor to the callee's monitor. Each message is added to the Intent object as an extra parameter.

To demonstrate how `IntentExtraComm` is used, we will describe its use case in detail, starting with how privilege escalation works followed by how a policy can detect and stop offending apps.

Making background calls with privilege escalation

1. Suppose Utilities is an app with `CALL_PHONE` permission, which is the permission for making background calls. To make calls, it will create an Intent with `ACTION_CALL`, and use that to start an activity. The Android system will process this Activity and dial the number specified in the Intent object. This feature is implemented in Utilities' CallActivity class.

2. Utilities' CallActivity is declared as a public activity, and any app can start it by calling `startActivity()`.

3. Suppose Spyware is an app that does not have `CALL_PHONE` permission. To make calls, it will start Utilities' CallActivity.

Policy definition

`BackgroundCallPolicy` prevents Spyware from making background calls while permitting Utilities to do so. Even though Utilities doesn't use a third app to make

background calls in our example, this policy would also allow Utilities to use one because it has `CALL_PHONE` permission.

The actual policy definition has about 100 lines, but Figure 5.5 shows the pseudocode version. Below is an explanation of how this policy works:

- The policy prevents privilege escalation attacks by intercepting calls to `Activity.startActivity()` function.
- `startActivity()` is chosen for interception because by intercepting its calls we can capture two pertinent events: the first is when an Activity attempts to make background calls directly; the second is when an Activity starts another Activity to make calls.
- In the caller Activities, the policy uses this logic: For those making background calls directly by calling `startActivity()` with `Intent.ACTION_CALL`, the policy let Android decide whether those Activities can make background calls or not; for those that start another Activity, the policy uses `IntentExtraComm` to send the list of permissions their apps have to the Activity that is about to be started.
- In the callee Activities, the policy uses this logic: If the Activity is not started by another, then it's started by the user, and since there is no privilege escalation, it is allowed to continue; if the Activity is started by another Activity, the policy will use `IntentExtraComm` to retrieve the list of permissions of the caller Activity and will call `Activity.finish()` if the caller Activity does not have `CALL_PHONE` permission.

```
public Suggestion query (Action a) {
  aswitch (a) {
    case <* Activity.startActivity(Intent i)>:
      String action = i.getAction();
      if (action != null &&
        !action.equals(Intent.ACTION_CALL))
        // Not making background calls
        return OKSuggestion(this, a);

      Activity myActivity = getActivity();
      if (action == null) {
        // starting an Activity in some app
        IntentExtraComm comm = new IntentExtraComm(i, myActivity.
            getPackageName());
        String permissions = getAllPermissions(myActivity);
        comm.send(null, PERMS_KEY, permissions);
        return OKSuggestion(this, a);
      }

      // action.equals(Intent.ACTION_CALL), which means
      // that this is the receiving Activity
      Intent startingIntent = myActivity.getIntent();
      if (startingIntent == null)
        // Activity is started by users
        return OKSuggestion(this, a);

      IntentExtraComm comm = new IntentExtraComm(startingIntent, null);
      String permissions = comm.recv(PERMS_KEY);
      if (!hasCallPerms(permissions))
        return ReplaceSuggestion(
        /* replaced by myActivity.finish() with reason
        "Don't have permissions"*/);

      return OKSuggestion(this, a);

    default: return IrrSuggestion(this, a);
  }
}
```

Figure 5.5: BackgroundCallPolicy

Implementation of IntentExtraComm

Each instance of `IntentExtraComm` uses an Intent object to send messages, by calling `Intent.putExtra()`. The receiving functions are implemented by calling `Intent.getStringExtra()` or similar functions for different types of data. The Intent object is delivered by Android to the started Activity, so the use of Intents as a delivery mechanism has several benefits:

1. The Android system guarantees that the message is sent to the intended recipient.

2. The ID of the recipient is not needed in `send()` because it is specified as a component's name in the Intent object.

3. The monitor is the last module to modify the Intent object, so the starting app cannot overwrite any of its monitor's messages.

4. This communication method has very low overhead: the only overhead is the time it takes to read messages from and write messages to Intent objects.

5.2.3 DataBankComm

Because `IntentExtraComm` only delivers messages between two monitors, it is difficult to use when a policy requires communication between a group of monitors (the monitor would need to make M exchanges, where M is the number of monitors in the group). Thus, `DataBankComm` is created. `DataBankComm` is a one-to-many communicator for all monitors on the same device. It uses a ContentProvider to store and retrieve messages. Monitors send messages by sending them to the ContentProvider, and they receive messages by querying it. This ContentProvider is not a part of any monitored app; instead, it belongs to a Polyandroid special app, called `MonitorDataBank`.

To demonstrate how `DataBankComm` is used, we will again describe the use case in detail followed by an explanation of the policy's definition.

How apps can collude to exfiltrate more location readings per 15 minutes

To demonstrate the enhanced enforcement when monitors can communicate, we will assume that the apps in the below example are already monitored individually to prevent each from getting more than one location reading per 15 minutes.

1. Utilities is an app with `FINE_LOCATION` permission, which is used for getting precise location reading.

2. Spyware is another app that also has `FINE_LOCATION` permission.

3. The developers of Utilities and Spyware agree to share the location data that their respective app gathers.

4. Utilities will start listening for a new location whenever the current time's minute value is divisible by 10. It will send every reading it gets to an external server.

5. Spyware will do the same but it will start whenenver the current time's minute value is one of 5, 15, 25, 35, 45, or 55. It will also send every reading it gets to the same external server.

6. When an app reads its first location value, its monitor starts a clock which resets every 15 minutes. Before the clock resets, any calls to read new location will return the previous value.

When the two apps pool their knowledge, the users' location will leak twice every 15 minutes instead of once because Utilities and Spyware stagger the start time of their monitors.

Policy definition

This policy allows at most one geo-coordinate to be read by any app on a device every 15 minutes. Repeated attempts to read more geo-coordinates by the same app or a different one will receive the old value until the 15-minute period expires. We called this policy `LocationPolicy`.

Figure 5.6 shows the `query()` method of LocationPolicyHelper, which is the user-defined portion of LocationPolicy. Basically, the policy intercepts calls to `LocationManager.requestSingleUpdate()` to swap an app's LocationListener with the monitor's own listener. Note that `LocationManager.requestSingleUpdate()` is only one of several methods that install a LocationListener. For this policy to be complete, all of them should be intercepted, but as this is a proof of concept, we intercept only `requestSingleUpdate()`. In order to explain the logic of `MonitorLocationListener` clearly, I will explain its simplest version first before explaining more complex versions.

Figure 5.7 shows the pseudocode of MonitorLocationListener, which uses DataBankComm to read and write the value of the last published location. Read is done by calling `recv()`, and write is done by calling `send()`. If the last published location is recorded not more than 15 minutes before the current time, then `MonitorLocationListener` will call the app's listener with the old value; otherwise, it will call the app's listener with the new value.

Conceptually, this algorithm is simple. Nonetheless, its implementation can be fraught with synchronization problems as shown in Figure 5.7. Adding synchronization APIs is discussed in the following sub-section.

One-to-many communicators' synchronization needs

The synchronization issue demonstrated in Figure 5.7 arises because `DataBankComm`'s `send()` and `recv()` are used in a similar manner to write and read operations to

```
public Suggestion query (Action a) {
  aswitch(a) {
  case <public void android.location.LocationManager.
      requestSingleUpdate
    (String provider, LocationListener listener, Looper looper)> :
    MonitorLocationListener l = new MonitorLocationListener(listener,
    getActivity());
    return new ReplaceSuggestion(this, l, a);
  }
  return new IrrSuggestion(this, a);
}
```

Figure 5.6: LocationPolicyHelper's query() method

```
// LAST_LOCATION is a keyword

MonitorLocationListener.onLocationChanged (newLocation) {
  DataBankComm comm = new DataBankComm(this.activity);

  l = comm.recv(LAST_LOCATION);      // 1

  if ((newLocation.time - l.time) < 15 minutes) {
    app_listener.onLocationChanged(l);
    return;
  }

  comm.send(null, LAST_LOCATION, newLocation);      // 2
  app_listener.onLocationChanged(newLocation);
}

// what if another app modified LAST_LOCATION between
// 1 and 2?
```

Figure 5.7: MonitorLocationListener without synchronization

and from shared memory locations. The reason that `send()` and `recv()` are used in this manner is because all monitors may not be active at the time messages are sent, and it is unlikely they ever will be. Thus, receiving a message means reading from shared storage, and sending a message means writing to shared storage.

To solve a synchronization problem, one's first thought is to use `lock()` and `unlock()`. However, they are not the best solution in the case of LocationPolicy. Figure 5.8 shows how `lock()` and `unlock()` operations can be used to provide synchronization. These functions would be additional APIs of the communicator. Since the communicator can be used to transmit different shared values, `lock()` and `unlock()` APIs take a conversation ID to know which conversation to lock or unlock. `lock()` and `unlock()` functions are fairly simple to use, but it requires three operations even if there is no update to the shared value and four if there is. `DataBankComm`'s communicaton is between processes, and is much slower comparing to `IntentExtraComm`'s; therefore, policy authors should not use more operations if they can do with less.

We found that using load-linked (or `ll()`) and store-conditional (or `sc()`) would provide synchronization in a fewer number of operations. Figure 5.9 shows how these operations are used.

`ll()` is the same as `recv()`, but it is designed to work in concert with `sc()`. `sc()` is similar to `send()`, but it only succeeds when no other monitors called `sc()` since the last time this monitor called `ll()`.

In Figure 5.9, MonitorLocationListener uses `ll()` and `sc()` to detect if another app has updated the shared LAST_LOCATION after this app makes a decision based on the value that it just read. If no app has updated LAST_LOCATION, then this app updates it and returns the new location. If another app updated LAST_LOCATION, then this app reads the new value and re-tests whether the new location should be returned. Because MonitorLocationListener only updates

```
MonitorLocationListener.onLocationChanged(newLocation) {
  DataBankComm comm = new DataBankComm(this.activity);

  comm.lock(LAST_LOCATION);          // 1
  l = comm.recv(LAST_LOCATION);      // 2

  if ((newLocation.time - l.time) < 15 minutes) {
    comm.unlock(LAST_LOCATION);      // 3
    app_listener.onLocationChanged(l);
    return;
  }

  comm.send(null, LAST_LOCATION, newLocation);     // 4
  comm.unlock(LAST_LOCATION);        // 5
  app_listener.onLocationChanged(newLocation);
}
// When there is an update to LAST_LOCATION, there are
// 4 operations: 1, 2, 4, 5.
// When there isn't an update to LAST_LOCATION, there are
// 3 operations: 1, 2, 3.
```

Figure 5.8: MonitorLocationListener with lock()/unlocked()

LAST_LOCATION with a more recent one, eventually the loop will terminate.

In our algorithm, typically, the `sc()` calls should succeed as not a lot of time has passed between the `ll()` call at 1 and the `sc()` call at 2, so it should typically take only two operations (1, 2). In the unlikely event that there is contention, it takes three operations (1, 2, 3).

It is possible that one app can take more iterations in the do-while loop if it is constantly interrupted by the Android system after it executes 1 but before it executes 2, and other apps constant update LAST_LOCATION at the same time, but this is unlikely.

In general, it is more efficient to use `ll()` and `sc()` than using `lock()` and `unlock()` when the shared values are not frequently updated.

```
public class MonitorLocationListener {
  private DataBankComm comm;
  private Location l;

  MonitorLocationListener(LocationListener originalListener, Activity
      activity) {
    comm = new DataBankComm(activity);
    l = comm.read(LAST_LOCATION);
  }

  public void onLocationChanged(newLocation) {
    do {
      if ((newLocation.time - l.time) < 15 minutes) {
        app_listener.onLocationChanged(l);
        return;
      }

      l_temp = ll(LAST_LOCATION); // 1
      if (l_temp == l) {
        if (sc(LAST_LOCATION, newLocation) == SUCCESS) { // 2
          l = newLocation;
          app_listener.onLocationChanged(l);
          return;
        }
        // sc() failed when L is updated after ll() call
        l_temp = ll(L); // 3
      }
      // l_temp != l when l is too outdated

      l = l_temp;
    } while (true);
    // loop will terminate because l increase with every iteration
  }
// Typically require 2 operations to update, 3 when
// there is contention, plus 1 when Activity is
// created.
}
```

Figure 5.9: MonitorLocationListener with ll()/sc()

Implementation of DataBankComm

`DataBankComm` uses a ContentProvider from the MonitorDataBank app to send, receive and store messages. This section will explain the implementation of `send()`, `recv()`, `ll()`, and `sc()` in detail.

The `send()` operation is implemented by calling the `update()` method of MonitorDataBank's ContentProvider (see Figure 5.10). Similarly, the `recv()` operation is implemented by calling the `query()` function.

Figure 5.11 shows how `ll()` and `sc()` methods are implemented. Just like `send()` and `recv()`, `ll()` and `sc()` also call `query()` and `update()` methods. However, they use an additional argument, `selection`, to distinguish themselves from the regular `send()` and `recv()` by setting `selection` to "LL" or "SC". The following explains how `sc()` can detect when the value it's about to change is already updated by another monitor after `ll()` is called:

1. When `query()` is called with "LL", an additional value is returned along with the message. Let's call this the `lockValue`. This value is saved by the communicator to use with the next `sc()` call.

2. When `update()` is called with "SC", `selectionArgs` is used to transmit the previously recorded `lockValue`. The ContentProvider will compare this `lockValue` with the one that is currently associated with the message. If the two match, then it will write the message; otherwise, it won't.

3. Once `lockValues` matches, not only is the message written by `sc()`, but the associated `lockValue` is also updated. The new value is provided by the monitor that calls `sc()`, and this value must be unique among all potentially parallel sequences of `ll()` and `sc()`.

4. To create unique `lockValues`, whenever a `DataBankComm` object is instantiated,

```
/**
 * @param receiver is ignored because this is a
 * one-to-many communicator
 * @param convoID
 * @param message
 */
public void send(String receiver, String convoID, byte[] message) {
  // error checking

  // ContentProvider.update()
  ContentResolver mContentResolver = activity.getContentResolver();
  ContentValues values = new ContentValues();
  values.put(convoID, message);

  String uri = STORE_URI + "/" + convoID;
  mContentResolver.update(Uri.parse(uri), values, null, null);
  return;
}
```

Figure 5.10: DataBankComm's send()

it generates a `channelID` based on an app-global counter and the app's name. Thus, this ID is unique among all `DataBankComm` instances among all apps. In an `sc()` operation, the communicator passes this value to the ContentProvider by putting it in the second column of `values`.

In this design, a message remains in `DataBankComm`'s storage until a monitor overwrites it with a new message. `DataBankComm` doesn't guarantee that all monitors get to read the message before it is overwritten because many policies are only interested in the most updated values. In addition, making sure that all monitors get to read a message before it is deleted from the communicator can easily cause resource leaks if the communicator cannot promptly detect when the monitor that it is waiting on is uninstalled.

```
public byte[] ll(String key) {
  // Error checking code here

  String uri = STORE_URI + "/" + key;
  ContentResolver mContentResolver = activity.getContentResolver();
  Cursor cursor = mContentResolver.query(Uri.parse(uri),
    null, "LL", null, null);

  if (!cursor.moveToFirst()) {
    this.lockVal = "␣";
    return null;
  }

  // Lockval would be return in the second column
  this.lockVal = new String(cursor.getBlob(1));
  return cursor.getBlob(0);
}

public boolean sc(String key, byte[] message) {
  // Error checking code here

  ContentResolver mContentResolver = activity.getContentResolver();
  ContentValues values = new ContentValues();
  values.put(key, message);

  String id = this.channelID + "/" + key;
  values.put(key + LOCK_EXTENSION, id.getBytes()); // 1

  String uri = STORE_URI + "/" + key;
  int numUpdated = mContentResolver.update(Uri.parse(uri),
    values, "SC", new String[] {lockVal});
  return numUpdated == 1;
}
```

Figure 5.11: DataBankComm's ll() and sc()

5.3 Preventing circumvention of communicators

As mentioned in Section 4.3, communicators transport messages that directly affect policy decisions, and if monitored apps can modify these messages, they may be able to circumvent their monitors. If Spyware can add `CALL_PHONE` to the list of permissions sent to Utilities even though it doesn't have that permission, Utilities would make background calls on its behalf, which would mean that Spyware evaded

its monitor. If Spyware is allowed to write messages to `MonitorDataBank`'s ContentProvider, it will overwrite the timestamp of the last published location to an earlier time. Consequently, its monitor will always think that the 15-minute period has expired and permit Spyware to get the latest geo-coordinates.

The `DataBankComm` is an example of a communicator that needs a non-circumvention policy: the target apps can easily start the MonitorDataBank's provider and call `query()` and `update()` just like DataBankComm does in `send()` and `recv()`. This section will show how its non-circumvention policy is defined.

An app can only modify stored messages by calling `MonitorDataBank`'s ContentProvider's `update()` method and can only read messages by calling its `query()` method. These methods are actually called by calling a ContentResolver's methods with the same name. Thus, the non-circumvention policy for `MonitorDataBank` is simple: it intercepts `ContentResolver.update()` and `ContentResolver.query()` calls from the target apps, and if their URI input points to `MonitorDataBank`, then the apps are halted. Figure 5.12 shows a `DataBankComm`'s non-circumvention policy that does exactly this.

Since LocationPolicy uses `DataBankComm`, the full policy is then a composition of the user-defined policy and DataBankCommPolicy. Figure 5.13 shows how these policies are composed using Conjunction combinator. Please refer to Section 2.2.6 for a brief explanation of the conjunction combinator.

5.4 Declaring Android permissions

This section will describe how new permissions, which are required by policies but are not already declared by monitored apps, are added to apps. Then, it will discuss the consequences of adding new permissions.

Section 4.4 already explained why new permissions may be needed, and that AndroidManifest.xml needs to be changed in order to declare them. It also mentioned

```
public class DataBankCommPolicy extends Policy {
  // MONITOR_DATA_BANK is the URI to MonitorDataBank
  public Suggestion query (Action a) {
    aswitch(a) {
    case <public Cursor android.content.ContentResolver.query(Uri uri,
      String[] projection, String selection, String[] selectionArgs,
      String sortOrder)>:
      if (uri.getAuthority().equals(MONITOR_DATA_BANK))
        return new HaltSuggestion(this, a,
        "App tried to circumvent monitor and got killed");
      return new OKSuggestion(this, a);

    case <public Uri android.content.ContentResolver.update(Uri uri,
      ContentValues values, String selection, String[] selectionArgs)>:
      if (uri.getAuthority().equals(MONITOR_DATA_BANK))
        return new HaltSuggestion(this, a,
        "App tried to circumvent monitor and got killed");
      return new OKSuggestion(this, a);
    }

    return new IrrSuggestion(this, a);
  }
}
```

Figure 5.12: Example non-circumvention policy

a new API in `Policy` class to tell the instrumentation engine which permissons to add. That API is `Policy.getAdditionalPerms()`. This function returns an array of permissions required by a policy as `String` objects. The instrumentation engine compares the permissions in this array against the permissions listed in each app's AndroidManifest.xml to find permissions that aren't declared by the app. Then, it will declare the missing permissions by adding `<uses-permission>` elements to the AndroidManifest.xml file. Therefore, to define required permissions, policy authors overload this function in their policy files.

Prior to Android 6.0, permissions declared in AndroidManifest.xml are requested at an app's installation time. Since Android 6.0, the permissions are not requested at installation time though they still need to be declared in AndroidManifest.xml.

```
public class LocationPolicy extends Conjunction {
  private static final Policy COMM_POLICY;

  static {
    try {
      COMM_POLICY = DataBankComm.getNonCircumventionPolicy().
          newInstance();
    } catch (InstantiationException e) {
      throw new PolymerException(e);
    } catch (IllegalAccessException e) {
      throw new PolymerException(e);
    }
  }

  public LocationPolicy() {
    super(COMM_POLICY, new LocationPolicyHelper());
  }
}
```

Figure 5.13: Composition with non-circumvention policy

Apps must request each permission when they need it by calling `ActivityCompat.requestPermissions()`. However, once granted, the app can continue accessing the permission's associated resource until it is revoked. The instrumentation engine does not add calls to `ActivityCompat.requestPermissions()` to apps, so policy authors must call this function inside their policy.

Adding new permissions to an app must be done with caution because it would give an app more capabilities than it needs to have. While one may assume that if the original app doesn't request a given permission (e.g. `INTERNET`) that it would never use the associated resource (e.g. connecting to external servers), that assumption could be disastrously incorrect. Suppose that an app does not declare `INTERNET` permission, but its monitor informs a device's user that it needs `INTERNET` permission to monitor the app. The user trusts the monitor and thereby grants that permission. A malicious app that knows its users use this run-time monitoring system can take advantage of this scenario by continually attempting to exfiltrate users' data out of

their devices until it succeeds without ever requesting `INTERNET` permission. This is an example of how a monitoring system can create new security vulnerabilities by requesting additional permissions.

A potential solution to this problem is to add a new prevention policy to prevent apps from using privileges granted to monitors. However, this policy is far more complicated than non-circumvention policies for two reasons:

1. Certain resources have many associated APIs. For example, `WRITE_EXTERNAL_STORAGE` allows an app to write to files on external storage, and there are many file I/O APIs, which means the size of the prevention policy for this permission alone can be huge.

2. Some apps that a user wants to use may already have the requested permissions while others may not. Thus, the prevention policy must be composed with the user-defined policy for some apps and not for others. If there are multiple permissions involved, then the number of compositional combinations increases quickly. Polyandroid currently does not support this kind of dynamic policy compositions.

In summary, policy authors must be cautious in requesting permissions, possibly limiting themselves to permissions that an app already has or those that can cause less damage, such as `WAKE_LOCK` or `VIBRATE`.

5.5 Informing users of enforcement

This section will discuss how the current user notification feature shown in Figure 4.1 and Section 4.5 is implemented and its limitation.

The aforementioned feature displays notifications using Android's `Toast`, and the message to be displayed is specified in Suggestions. Whenever a policy returns

```
public void accept(Suggestion s) {
  if (activity == null) return;

  String reason = s.getReason();

  if (reason == null || reason.length() == 0)
    return;

  Toast.makeText(activity.getApplicationContext(),
    reason, Toast.LENGTH_SHORT).show();
}
```

Figure 5.14: Implementing user-notification

a Suggestion, it can optionally add a reason for making that Suggestion. Whenever a Suggestion is accepted, if it is accompanied by a reason, the monitor will display that reason as a `Toast`. Figure 5.14 shows the modified `Policy.accept()` method where this feature is implemented.

The use of `Toast` to display user notifications has a flaw, however. Because the code to display `Toasts` runs in an app's process, the reason for HaltSuggestion cannot be displayed because the process is halted right after `Policy.accept()` is called. This is undesirable because halting an app is the most severe enforcement for which an explanation is needed. In the future, we hope to employ MonitorDataBank to display the reasons as Android `Notifications` instead.

6

Evaluation

In order to test Polyandroid, we created three apps and wrote three policies. This section will discuss how we used them to evaluate the efficacy of Polyandroid and what we found.

6.1 Test apps

While there have been many references to the apps listed below in sections 4 and 5, this section will act as a reference of their features and their purposes.

6.1.1 MonitorDataBank

This is an app that is used solely by the monitoring system, and the users can install it on any device where Polyandroid is installed. Currently, its only functional component is a GenericContentProvider, which provides the communication media for DataBankComm. Its development was motivated by the needs of DataBankComm, but it has many other potential uses: as an interface for users to specify policy parameters, as a viewer of enforcement history, and as a user-notification system.

6.1.2 Utilities

Utilities represents apps that provide useful services and are directly granted permissions to access sensitive resources by its users. It has the following Activites:

- BackgroundCallActivity: Make background calls
- BackgroundSmsActvity: Send background SMSes
- LocationActivity: Read fine location, and start Google Maps activity to display the location on a map
- ContactActivity: Read contact list and display it on the screen

6.1.3 Spyware

Spyware represents apps that provide some useful services while exploiting users' data for nefarious purposes. Spyware has the following Activites:

- MainActivity: Start Utilities' BackgroundCallActivity to make background calls or start Utilities' BackgroundSmsActivity to send background SMSes
- GetLocationActivity: This Activity is identical to Utilities' LocationActivity. Spyware requests its own permission to get users' fine location here instead of starting Utilities' LocationActivity
- It has an optional feature that we can add or remove depending on each test case: Just before it attempts to read a location value, it contacts MonitorDataBank and changes the timestamp of the last published location, so if LocationPolicy is used, the policy module will always think that the last published location is outdated and so will return the latest location to Spyware.

6.2 Implemented Policies

Like the apps, most of our policies are already discussed in earlier chapters, so this section will only summarize them.

6.2.1 BackgroundSmsPolicy

Unlike the other two, this policy does not use any communication module, so it hasn't been discussed until now. It is designed to prove that Polyandroid preserves Polymer's abilities to enforce individual apps. Following is its summary:

- The policy stops an app with permission to send background SMSes from sending more than 4 background SMSes.
- The policy enforces this by storing the count of background SMSes that an app has sent in a file.
- This policy is also an example of adding customized controls over Android permissions. In this case, the app has permission to send background SMSes, but it is subjected to a user-defined limit.
- A use case of this policy is for users to allow a small amount of a resource to be consumed in exchange for services provided by apps, but disallow apps from abusing this resource, especially without their notice.

6.2.2 BackgroundCallPolicy

This policy requires one-to-one communication. Following is its summary:

- The policy disallows an app that doesn't have `CALL_PHONE` from using another app to make background calls.

- The policy enforces this by using IntentExtraComm to send the list of permissions that the starting app has to the app that makes background calls, and the monitor in the latter will determine whether it should make the calls or not.

- When the starting app does not have `CALL_PHONE` permission, the started app finishes and displays a message to inform users that privilege esclation has happened.

- This is an example that Polyandroid can enforce specific cases of privilege escalation.

6.2.3 LocationPolicy

This policy requires one-to-many communication. Following is its summary:

- This policy returns the same value to every request to read fine location within 15-minute intervals, even if the requests are from different apps.

- This policy uses the DataBankComm to store the last location published, and whenever an app's `LocationListener.onLocationChanged()` is called, if the current time is less than 15 minutes from the timestamp of the last location, the last location is returned; otherwise, it is updated, and the new location is returned.

- This is an example of creating customized control over a granted resource.

- DataBankComm has an accompanying non-circumvention policy, so LocationPolicy is also an example of how to compose a user-defined policy with a non-circumvention policy.

6.3 Results

We tested our apps and policies on the Android 6.0 platform. We used both a Google Nexus 5 phone and Android emulators for testing.

After verifying that our system can successfully enforce the policies that we listed, we evaluated our system based on the ease of writing policies, the average amount of time needed to instrument an app, and the degradation of monitored apps' performance. We chose these criteria for the following reasons: First, if having monitors slow apps' performance greatly, then users will likely choose not to use our monitoring system; second, if it takes a lot of writing to express a policy that users desire, they would likely make mistakes or choose not to write that policy; however, because policies can be written by an expert and share with average users, having users writing long policy definitions is not necessarily detrimental to a run-time monitoring system; lastly, making apps' instrumentation take fewer steps and less time would be make the system more acceptable to users. However, our analyses are qualitative because our dataset is small.

6.3.1 Enforcement result

We found that Polyandroid can successfully enforce the above policies on the apps we created according to the following success criteria:

- When BackgroundSmsPolicy is in effect: Utilities can only send background SMSes 4 times. On the 5th time, instead of sending an SMS, an on-screen message appears telling the users that the app's limit for sending background SMSes has been reached.

- When BackgroundCallPolicy is in effect: Utilities can make background calls, but Spyware cannnot. Whenever Spyware failed to start, an on-screen message

appears and informs users that Spyware does not have permission to make background calls.

- When LocationPolicy is in effect (and calls to corrupt DataBankComm's messages are removed from Spyware): Utilities' LocationActivity is activated first, then Spyware's GetLocationActivity is activated 5 minutes later; both display the same location even though the tester has moved several blocks. The policy was tested several times, and the result showed that only when the interval between starting each Activity is 15 minutes or more will a new Location be returned to the later Activity.

- When LocationPolicy is in effect and Spyware invokes code to circumvent DataBankComm, Spyware cannot start.

6.3.2 Ease of writing policies

We measured the ease of writing policies by counting the number of lines of code to define a policy. `BackgroundSmsPolicy` comprises 120 lines. `BackgroundCallPolicy` comprises 104 lines. `LocationPolicy` includes 200 lines from the user-defined policy and 30 lines from `DataBankComm`'s non-circumvention policy.

6.3.3 Ease of compiling policies and instrumentating apps

Instrumenting Utilities or Spyware takes approximately the same amount of time: 65 seconds on average. Most of the time is used by `jar2dex` tool to convert JVM bytecode to Dalvik bytecode. Here is a list of factors that affect the run-time of instrumentation:

  - Size of apps: Utilities has approximately 600 lines of code, and Spyware has about 250 lines of code.

– Number of security-relevant functions to be intercepted: we intercept nine functions, including those that are checked by `DataBankComm`'s non-circumvention policy.

– The platform on which apps are instrumented: a personal laptop with quad-core processor, 2.4 GHz processor speed and 8 GB memory.

At this point, users have to download apps on their computers, where the apps are instrumented, and install them using the `adb` tool. Though this is a more cumbersome process than installing an uninstrumented app, privacy- and security-conscious users may be willing to accept.

6.3.4 Effects on performance

Using either `IntentExtraComm` or `DataBankComm`, the degradation of performance is imperceptible. Making background calls on Utilities doesn't take perceivably more time when there is a monitor comparing to when there is not. Comparing the time an app takes to receive geo-coordinates when there is a monitor to when there is not is more difficult because Android doesn't release these values at a constant rate, but there is no consistent delay in the former case. Still, both the size of apps and the number of functions to be intercepted that our tests used are small, and the effect of Polyandroid on the performance of typical apps in Googple Play Store may be more visible.

7

Conclusions

Our preliminary experiments show that it is possible to assert customized access control over Android apps. Using Polyandroid communicator modules, users can regulate the collective behavior of multiple apps such as stopping privilege escalation or limiting the release of privacy-sensitive data to a group of apps to a set amount. This access control is a customized version of Android-provided access control because it only grants apps limited access to a resource whereas Android's access control either grants unlimited access or none at all. Customized access controls are added by writing policies, and due to Polymer's ability to compose policies, users can easily combine their own policies with non-circumvention policies accompanying communicators of their choice, and be confident that the resulting policies do not diminish the non-circumvention policies due to incorrect composition.

During the course of our research, we also found a few requirements to designing monitor communication:

- Sending and receiving messages between monitors is likely to be asynchronous because senders and receivers won't be simultaneously available. This means

that there must be a way to store messages that are not yet retrieved by recipients.

- Because monitors are integrated in their targets, their communication can be corrupted by malicious targets. Thus, each communication module will have to be accompanied by a non-circumvention policy unless trusted components are used by the module to deliver and and store their messages.

- A communication module backed by a medium that has fewer APIs to access its messages is better than one with a medium that has more because the former would have a smaller non-circumvention policy.

- Synchronization is likely required for any group communication.

- The designer of a communication scheme must be careful while choosing designs that require Android permissions because any permission granted to a monitor is also granted to its target, and that would give an app more privileges when it is monitored than when it is not.

These requirements show that designing a communication scheme is a non-trivial task. Therefore, encapsulating communication into its own module and enabling reuse is important.

8

Future Work

This section lists several tasks remaining to be done before Polyandroid can be a useful tool for the average Android users.

8.1 Monitoring of multi-threaded apps

Android apps are likely to be multi-threaded because it is likely that they must retrieve data from external servers to enable many features, and it is also likely that data retrieval runs on background threads to avoid affecting the apps' interactivity. Examples of such features are retrieving user profiles or images from the cloud. Without a correct model for monitoring multiple threads, Polyandroid can only monitor an extremely limited set of real apps, maybe even none.

8.2 Tracking ContentProviders, Services, and BroadcastReceivers in Policy

Currently, policy writers can call `getActivity()` to finish an Activity if that Activity is about to violate a policy. This is a noninvasive way to stop a violation, but it only applies to Activity. We can also stop policy violation in ContentProviders by making

their methods such as `query()`, `update()`, `insert()`, etc., return empty results instead of to killing the app. Similarly, we stop a Service by calling `stopService()` or `stopSelf()`. As for BroadcastReceivers, when they violate a policy, we can call `abortBroadcast()`.

8.3 Instrument native methods

Some apps may have native code, which we haven't instrumented. In other to have more comprehensive monitoring, we should also instrument native code.

8.4 Adding features to MonitorDataBank

MonitorDataBank can be used as an interface between the monitoring system and users. We can add an Activity to let users view enforcement history, and another one to let users specify which policies to use, or parameters that can further personalized a policy.

Bibliography

[1] *Apache Commons BCEL*, 2014 (accessed April 25, 2016). [Online]. Available: https://commons.apache.org/proper/commons-bcel/.

[2] *dex2jar: Tools to work with android .dex and java .class files*, 2015 (accessed April 25, 2016). [Online]. Available: https://sourceforge.net/projects/dex2jar/.

[3] *Android apktool: A tool for reverse engineering Android apk files*, 2016 (accessed April 26, 2016). [Online]. Available: http://ibotpeaches.github.io/Apktool/.

[4] L. Bauer, J. Ligatti, and D. Walker, "Composing expressive runtime security policies," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 18, no. 3, p. 9, 2009.

[5] F. Chen and G. Roşu, "Mop: an efficient and generic runtime verification framework," in *ACM SIGPLAN Notices*, vol. 42, no. 10. ACM, 2007, pp. 569–588.

[6] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 5, 2014.

[7] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 639–652.

[8] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky, "Java-mac: A runtime assurance approach for java programs," *Formal methods in system design*, vol. 24, no. 2, pp. 129–155, 2004.

[9] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for android applications," in *Presented as part of the 21st USENIX Security*

Symposium (USENIX Security 12). Bellevue, WA: USENIX, 2012, pp. 539–552. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/xu_rubin.