Aurasium: Practical Policy Enforcement for Android Applications

Rubin Xu
Computer Laboratory
University of Cambridge
Cambridge, UK
Rubin.Xu@cl.cam.ac.uk

Hassen Saïdi
Computer Science Laboratory
SRI International
Menlo Park, USA
hassen.saidi@sri.com

Ross Anderson
Computer Laboratory
University of Cambridge
Cambridge, UK
Ross.Anderson@cl.cam.ac.uk

Abstract

The increasing popularity of Google's mobile platform Android makes it the prime target of the latest surge in mobile malware. Most research on enhancing the platform's security and privacy controls requires extensive modification to the operating system, which has significant usability issues and hinders efforts for widespread adoption. We develop a novel solution called Aurasium that bypasses the need to modify the Android OS while providing much of the security and privacy that users desire. We automatically repackage arbitrary applications to attach user-level sandboxing and policy enforcement code, which closely watches the application's behavior for security and privacy violations such as attempts to retrieve a user's sensitive information, send SMS covertly to premium numbers, or access malicious IP addresses. Aurasium can also detect and prevent cases of privilege escalation attacks. Experiments show that we can apply this solution to a large sample of benign and malicious applications with a near 100 percent success rate, without significant performance and space overhead. Aurasium has been tested on three versions of the Android OS, and is freely available.

1   Introduction

Google's Android OS is undoubtedly the fastest growing mobile operating system in the world. In July 2011, Nielsen placed the market share of Android in the U.S. at 38 percent of all active U.S. smartphones [9]. Weeks later, for the period ending in August, Nielsen found that Android has risen to 43 percent. More important, among those who bought their phones in June, July, or August, Google had a formidable 56 percent market share. This unprecedented growth in popularity, together with the openness of its application ecosystem, has attracted malicious entities to aggressively target Android. Attacks on Android by malware writers have jumped by 76 percent over the past three months according to a report by

MacAfee [29], making it the most assaulted mobile operating system during that period. While much of the initial wave of Android malware consisted of trojans that masquerade as legitimate applications and leak a user's personal information or send SMS messages to premium numbers, recent malware samples indicate an escalation in the capability and stealth of Android malware. In particular, attempts are made to gain root access on the device through escalation of privilege [37] to establish a stealthy permanent presence on the device or to bypass Android permission checks.

Fighting malware and securing Android-powered devices has focused on three major directions. The first one consists of statically [20] and dynamically [12, 36] analyzing application code to detect malicious activities before the application is loaded onto the user's device. The second consists of modifying the Android OS to insert monitoring modules at key interfaces to allow the interception of malicious activity as it occurs on the device [19, 27, 17, 33, 13]. The third approach consists of using virtualization to implement rigorous separation of domains ranging from lightweight isolation of applications on the device [35] to running multiple instances of Android on the same device through the use of a hypervisor [26, 30, 11].

Two fundamental and intertwined problems plague these approaches. The first is that the definition of malicious behavior in an Android application is hard to ascertain. Access to privacy- and security-relevant parts of Android's API is controlled by an install-time application permission system. Android users are informed about what data and resources an application will have access to, and user consent is required before the application can be installed. These explicit permissions are declared in the application package. Install-time permissions provide users with control over their privacy, but are often coarse-grained. A permission granted at install time is granted as long as the application is installed on the device. While an application might legitimately re-

quest access to the Internet, it is not clear what connections it may establish with remote servers that may be malicious. Similarly, an application might legitimately require sending SMS messages. Once the SMS permission is granted, there are no checks to prevent the application from sending SMS messages to premium numbers without user consent. In fact, the mere request for SMS permission by an application can be deemed malicious according to a recent Android applications analysis [24], where it is suggested that 82 percent of malicious applications require permissions to access SMS. A recent survey [18] exposes many of the problems [22, 14] associated with application components interactions, delegation of permission, and permission escalation attacks due to poor or missing security policy specifications by developers. This prompted early work [21] on security policy extension for Android.

The second problem is that any approach so far that attempts to enhance the platform's security and privacy controls based on policy extensions requires extensive modification to the operating system. This has significant usability issues and hinders any efforts for widespread adoption. There exists numerous tablet and phone models with different hardware configurations, each running a different Android OS version with its own customizations and device drivers. This phenomenon, also known as the infamous Android version *fragmentation problem* [16] demonstrates that it is difficult to provide a custombuilt Android for all possible devices in the wild. And it is even more difficult to ask a normal user to apply the source patch of some security framework and compile the Android source tree for that user's own device. These issues will prevent many OS-based Android security projects from being widely adopted by the normal users. Alternatively, it is equally difficult to bring together Google, the phone manufacturers, and the cellular providers to introduce security extensions at the level of the consumer market, due to misaligned incentives from different parties.

Our Approach We aim at addressing these challenges by providing a novel, simple, effective, robust, and deployable technology called Aurasium. Conceptually, we want Aurasium to be an application-hardening service: a user obtains arbitrary Android applications from potentially untrusted places, but instead of installing the application as is, pushes the application through the Aurasium black box and gets a hardened version. The user then installs this hardened version on the phone, assured by Aurasium that all of the application's interactions are closely monitored for malicious activities, and policies protecting the user's privacy and security are actively enforced.

Aurasium does not need to modify the Android OS

at all; instead, it enforces flexible security and privacy polices to arbitrary applications by repackaging to attach sandboxing code to the application itself, which performs monitoring and policy enforcement. The repackaged application package (APK) can be installed on a user's phone and will enforce at runtime any defined policy without altering the original APK's functionalities. Aurasium exploits Android's unique application architecture of mixed Java and native code execution to achieve robust sandboxing. In particular, Aurasium introduces libc interposition code to the target application, wrapping around the Dalvik virtual machine (VM) under which the application's Java code runs. The target application is also modified such that the interposition hooks get placed each time the application starts.

Aurasium is able to interpose almost all types of interactions between the application and the OS, enabling much more fine-grained policy enforcement than Android's built-in permission system. For instance, whenever an application attempts to access a remote site on the Internet, the IP of the remote server is checked against an IP blacklist. Whenever an application attempts to send an SMS message, Aurasium checks whether the number is a premium number. Whenever an application tries to access private information such as the International Mobile Equipment Identity (IMEI), the International Mobile Subscriber Identity (IMSI), stored SMS messages, contact information, or services such as camera, voice recorder, or location, a policy check is performed to allow or disallow the access. Aurasium also monitors I/O operations such as write and read. We evaluated Aurasium against a large number of real-world Android applications and achieved over 99 percent success rate. Repackaging an arbitrary application using Aurasium is fast, requiring an average of 10 seconds.

Our main contributions are that

- We have built an automated system to repackage arbitrary APKs where arbitrary policies protecting privacy and ensuring security can be enforced.

- We have developed a set of policies that take advantage of advances in malware intelligence such as IP blacklisting.

- We provide a way of protecting users from malicious applications without making any changes to the underlying Android architecture. This makes Aurasium a technology that can be widely deployed.

- Aurasium is a robust technology that was tested on three versions of Android. It has low memory and runtime overhead and, unlike other approaches, is more portable across the different OS versions.

The paper is organized as follows: Section 2 provides the some background information on the architecture of Android and then goes through details about the architecture, enforceable policies and deployment methods of Aurasium. In Section 3 we evaluate Aurasium with respect to its robustness in repackaging applications, as well as the overhead introduced by the repackaging process. Section 4 describes threat models against Aurasium and mitigation techniques. Related work and conclusions are discussed in Section 5 and Section 6, respectively.

2 Aurasium

2.1 Android

Android, the open source mobile operating system developed by the Open Handset Alliance led by Google, is gaining increasing popularity and market share among smartphones. Built on top of a Linux 2.6 kernel, Android introduces a unique application architecture designed to ensure performance, security, and application portability. Rigorous compartmentalization of installed applications is enforced through traditional Linux permissions. Additional permission labels are assigned to applications during install time to control the application's access to security and privacy-sensitive functionalities of the OS, forming a mandatory access-control scheme.

Android employs an inter-process communication (IPC) mechanism called Binder [6] extensively for interactions between applications as well as for application-OS interfaces. Binder is established by a kernel driver and exposed as a special device node on which individual applications operate. Logically, the IPC works on the principle of thread migration. A thread invoking an IPC call with Binder appears as if it migrates into the target process and executes the code there, hopping back when the result is available. All the hard work such as taking care of argument marshalling, tracking object references across processes, and recursions of IPC calls is handled by Binder itself.

Android applications are mainly implemented in Java, with the compiled class files further converted into Dalvik bytecode, running on the proprietary register-based Dalvik VM. It is similar to the JVM, but designed for a resource-constrained environment with a higher code density and smaller footprint. Applications are tightly coupled with a large and function-rich Android framework library (c.f. J2SE). Also, applications are free to include compiled native code as standalone Linux shared object (`.so`) files. The interaction between an application's Java and native code is well defined by the Java Native Interface (JNI) specification and supported by Android's Native Development Kit (NDK). In reality, the complexity of using native code means that only a small number of applications employ native code for the most performance-critical tasks.

2.2 System Design

Aurasium is made up of two major components: the repackaging mechanism that inserts instrumentation code into arbitrary Android applications and the monitoring code that intercepts an application's interactions with the system and enforces various security policies. The repackaging process makes use of existing open source tools augmented with our own glue logic to re-engineer Android applications. The monitoring code employs user-level sandboxing and interposition to intercept the application's interaction with the OS. Aurasium is also able to reconstruct the high-level IPC communication from the low-level system call data, which allows it to monitor virtually all of Android's APIs.

2.2.1 Application-OS Interaction

Under the hood, some of Android's OS APIs are handled by the kernel directly, while others are implemented at user-mode system services and are callable via inter-process communication methods. However, in almost all scenarios the application does not need to distinguish between the two, as these APIs have already been fully encapsulated in the framework library and the applications just need to interact with the framework through well-documented interfaces. Figure 1 shows in detail the layers of the framework library in individual applications' address space.
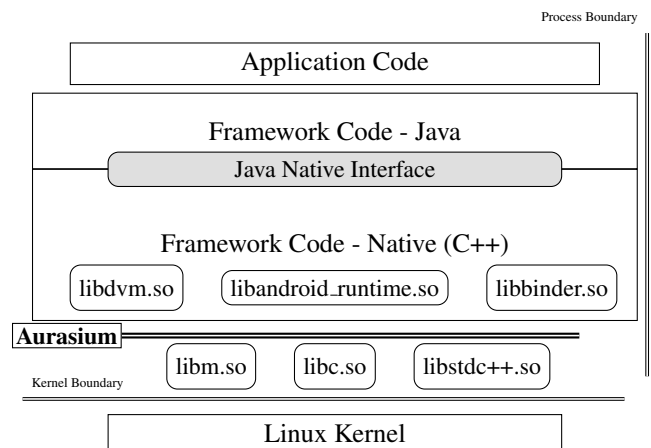

Figure 1: Android Application and Framework Structure

The top level of the framework is written in Java and is the well-documented part of the framework with which applications interact. This hides away the cumbersome

details from the application's point of view, but in order to realize the required operations it will hand over the request to the low-level part of the framework implemented in native code. The native layer of the framework consists of a few shared objects that do the real work, such as communicating with the Dalvik VM or establishing the mechanism for IPC communication. If we dive lower, we find that these shared objects are in fact also relying on shared libraries at even lower levels. There, we find Android's standard C libraries called *Bionic libc*. The Bionic libc will initiate appropriate system calls into the kernel that completes the required operation.

For example, if the application wants to download a file from the Internet, it has multiple ways to do so, ranging from fully managed `HttpURLConnection` class to low-level `Socket` access. No matter what framework APIs the application decides to use, they will all land on the `connect()` method in the `OSNetworkSystem` Java class in order to create the underlying TCP socket. This `connect()` method in turn transfers control to `libnativehelper.so`, one of the shared objects in the native layer of the framework, which again delegates the request to the `connect()` method in `libc.so`. The socket is finally created by libc issuing a system call into the Linux kernel.

No matter how complex the upper layer framework library may be, it will always have to go through appropriate functions in the Bionic libc library in order to interact with the OS itself. This gives a reliable choke point at which the application's interactions with the OS can be examined and modified. The next section explains how function calls from the framework into libc can be interposed neatly.

2.2.2 Efficient Interposition

Similar to the traditional Linux model, shared objects in Android are relocatable ELF files that are mapped into the process's address space when loaded. To save memory and avoid code duplication, all shared objects shipped with Android are dynamically linked against the Bionic libc library. Because a shared object like libc can be loaded into arbitrary memory address, dynamic linking is used to resolve the address of unknown symbols at load time. For an ELF file that is dynamically linked to some shared object, its call sites to the shared object functions are actually jump instructions to some stub function in the ELF's procedure linkage table (PLT). This stub function then performs a memory load on some entry in the ELF's global offset table (GOT) in order to retrieve the real target address of this function call to which it then branches. In other words, the ELF's global offset table contains an array of function pointers of all dynamically linked external functions referenced by its code.

During dynamic linking this table is filled with appropriate function pointers; this is controlled by the metadata stored in the ELF file, such as which GOT entry maps to which function in which shared object.

This level of indirection introduced by dynamic linking can be exploited to implement the required interposition mechanism neatly: it is sufficient to go through every loaded ELF file and overwrite its GOT entries with pointers to our monitoring functions. This is equivalent to doing the dynamic linking again but substituting function pointers of interposition routines[1].

Because Java code is incapable of directly modifying process memory, we implemented our interposition routines in C++ and compiled them to native code. All the detour functions are also implemented in C++ and they will preprocess the relevant function call arguments before feeding them to Aurasium's policy logic. We try to minimize the amount of native code because it is generally difficult to write and test. As a result most of the policy logic is implemented in Java, which also means it can take advantage of many helper functions in the standard Android framework. However, in the preprocessing step of the IPC calls we make an effort to reconstruct the inter-process communication parameters as well as high-level Java objects out of marshalled byte streams in our native code. It turns out that despite the system changes between Android 2.2, 2.3 and 3.*x*, the IPC protocol remains largely unaffected[2] and hence our interposition code is able to run on all major Android versions reliably.

With all these facilities in place, Aurasium is capable of intercepting virtually all framework APIs and enforcing many classes of security and privacy policies on them. It remains to be discussed what policies we currently implement (Section 2.3) and how reliable Aurasium's sandboxing mechanism is (Section 4). But before that, let us explain how we repackage an Android application such that Aurasium's sandboxing code is inserted.

2.2.3 APK Repackaging

Android applications are distributed as a single file called an Android Application Package (APK) (Figure 2). An APK file is merely a Java JAR archive containing the compiled manifest file `AndroidManifest.xml`, the application's code in the form of dex bytecode, compiled XML resources such as window layout and string constant tables, and other resources like images, sound and native libraries. It also includes its own signature in a form identical to the standard Java JAR file signatures.

---

[1] We did not consider other advanced dynamic linking techniques such as lazy linking here because they are not adopted in the current Android OS. They can be dealt with similarly.

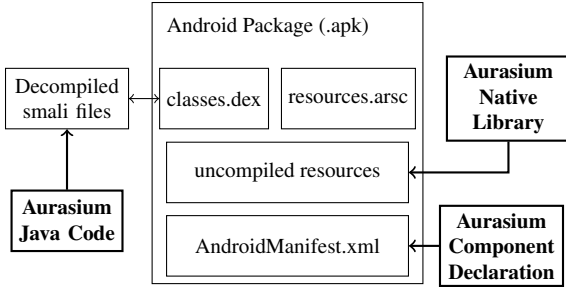[2] An exception is the introduction of `Strict Mode` from version

Figure 2: Android Application Package

Because the Aurasium code contains both a native library for low-level interposition and high-level Java code that executes the policy logic, we need a way of inserting both into the target APK. Adding a native library is trivial as native libraries are standalone Linux shared object (`.so`) files and are stored as is. Adding Java code is slightly tricky because Android requires all the application's compiled bytecode to reside in a single file called `classes.dex`. To insert Aurasium's Java code into an existing application, we have to take the original `classes.dex`, disassemble it back to a collection of individual classes, add Aurasium's classes, and then re-assemble everything back to create the new `classes.dex`.

There exist open source projects that can perform such task. For example, `smali` [7], an assembler/disassembler for dex files, and `android-apktool` [1], which is an integrated solution that can process not only code but also compiled resources in APK files.[3] In Aurasium we adopt `apktool` in our repackaging process. In the decode phase, `apktool` takes in an APK file, disassembles its dex file, and produces a directory such that each bytecode file maps to a single Java class, and its path corresponds to the package hierarchy, together with all other resources in the original APK file. Aurasium's Java code is then merged into the directory and `apktool` is engaged again to assemble the bytecode back into a new `classes.dex` file. Together with other resources, a new APK file is finally produced.

In reality, before producing the final APK file there is one more thing to do: merely merging Aurasium code into the target application does not automatically imply that it will run. We need to make sure that Aurasium code is invoked somehow, preferably before any of the original application code, so that the application does not execute any of its code before Aurasium's sandboxing is established. One option would be to modify the application's entry point so that it points to Aurasium. This turns

out to be not as easy as one might expect. Android applications often possess many possible entry points, in the sense that every public application component including activity, service, broadcast receiver, and content provider can be invoked directly and hence they all act as entry points.

In Aurasium we take a different approach: The Android SDK allows an application to specify an `Application` class in its manifest file which will be instantiated by the runtime whenever the application is about to start. By declaring Aurasium as this `Application` class, Aurasium runs automatically before any other parts of the application. There is a small caveat that the original application may have already defined such `Application` class. In this case, we trace the inheritance of this class until we find the root base class. This class will have to be inherited directly from `Application`, and we modify its definition (which is in the decompiled bytecode form) such that it inherits from Aurasium's `Application` class instead. This allows Aurasium to be instantiated as before, and being the root class ensures that Aurasium gets run before the application's `Application` class is instantiated.

Figure 2 illustrates the composition of an APK and the various Aurasium modules added at repackaging time.

2.2.4 Application Signing

The last thing to worry about is that when an application is modified and repackaged, its signature is inevitably destroyed and there is no way to re-sign the application under its original public key. We believe this is a problem, but manageable. Every Android application is indeed required to have a valid signature, but signatures in Android work more like a proof of authorship, in the sense that applications signed by the same certificate are believed to come from the same author, hence they are trusted by each other and enjoy certain flexibilities within Android's security architecture, e.g., signature permission. Application updates are also required to be signed with the same certificate as the original application. Other than that, signatures impose few other restrictions, and developers often use self-signed certificates for their applications.

This observation means that Aurasium can just re-sign the repackaged application using a new self-signed certificate. To preserve the authorship relation, Aurasium performs the re-signing step using a parallel set of randomly generated Aurasium certificates, maintaining a one-to-one mapping between this set to arbitrary developer certificates. In other words, whenever Aurasium is about to re-sign an application, it first verifies the validity of the old signature. If it passes, then Aurasium will proceed to sign the application with its own

---

[3]`apktool` is actually built on top of a fork of smali.

certificate that corresponds to the application's original certificate, or a newly generated one if this application has not been encountered earlier. In this way, the equivalence classes of authorship among applications are still maintained by Aurasium's re-signing procedure. Problems can still arise if Aurasium re-signs only a partial set of applications in the cases of application updates or applications intending to cooperate with their siblings. We consider these cases non-severe, with one reason being that Aurasium is more likely to be applied to a standalone application from a non-trusted source where application updates and application cooperation are not common.

Because all private keys of the generated certificates need to be stored[4] for future queries, the re-signing process contains highly confidential information and, hence, requires careful protection. It should be (physically) separated from Aurasium's other services and perceived as an oracle with minimal interfaces to allow re-signing an already-signed application. For higher assurance, hardware security modules could be used.

2.2.5 Aurasium's Security Manager

Aurasium-wrapped applications are self-contained in the sense that the policy logic and the relevant user interface are included in the repackaged application bundle, and so are remembered user decisions stored locally in the application's data directory. Alternatively, Aurasium Security Manager (ASM) can also be installed, enabling central handling of policy decisions of all repackaged application on the device. Depending on the enforced polices at repackaging time, an application queries the ASM for a policy decision via IPC mechanisms with intents describing the sensitive operation it is about to perform, and the ASM either prompts the user for consent, uses a remembered user decision recorded earlier, or automatically makes a decision without user interaction by enforcing a predefined policy embedded at repackaging time. The policy logic in individual applications prefers delegating policy decisions to the ASM, and will fall back to local decisions only if a genuine ASM instance is not detected on the device.

Using ASM for central policy decision management has one major advantage: policy logic can be controlled globally, and it can also be improved by updating the ASM instance on the device. For example, IP address blacklisting and whitelisting can be managed and kept up to date by ASM. Repackaged applications are able to take advantage of better policy logics once ASM is updated, even after they have been repackaged and deployed to users' devices. There is a tradeoff between the flexibility of ASM and the efficiency of repackaged application, though. In extreme cases, a repackaged application can proxy every IPC call to ASM, but this would be vastly inefficient. In our implementation ASM is consulted only with high-level summaries of potential sensitive operations, the set of which is fixed at repackaging time.

2.3 Policies

Now that we have demonstrated the ability to repackage arbitrary applications with Aurasium to insert monitoring code, we discuss various security policies that leverage this technique. It is important to point out that these are just some examples that we implemented as a proof of concept so far. Aurasium itself provides a flexible framework under which many more potent policies are possible.

We are interested primarily in enforcing some security policy that protects the device from untrusted applications. This includes not only attempts by the application to access sensitive information, leaking to the outside world or modifying it, but also attempts by the application to escalate privilege and to gain root access on the device by running suspicious system calls and loading native libraries. Aurasium's architecture and design allow us to implement many of the already-proposed policies such as dynamically constraining permissions [33], or setting up default dummy IMEI and IMSI numbers as well as phone numbers, as in [27].

The following subsections describe a set of policies that are easily checkable by Aurasium. The enforcement of these policies is supported by Aurasium intercepting the following functions:

- `ioctl()`

  This is the main API through which all IPCs are sent. By interposing this and reconstructing the high-level IPC communication, Aurasium is able to monitor most Android system APIs and enforce the privacy and SMS policies, and modifying the IPC arguments and results on the fly. In certain cases such as content providers, Aurasium replaces the returned `Cursor` object with a wrapper class to allow finer control over the returned data.

- `getaddrinfo()` and `connect()`

  These functions are responsible for DNS resolving and socket connection. Intercepting them allows Aurasium to control the application's Internet access.

- `dlopen()`, `fork()` and `execvp()`

  The loading of native code from Java and execution of external programs are provided by these functions, which Aurasium monitors.

---

[4]Alternatively, these new certificates can be generated from the original certificate under a master key.

- `read()`, `write()`

  These functions reflect access to the file system. Intercepting them allows Aurasium to control private and shared files accesses.

- `open()` and reflection APIs in `libdvm.so`[5]

  These functions are intercepted to prevent malicious applications from circumventing Aurasium's sandboxing. Because Aurasium may stores policy decisions in the application's local directory, it must prevent the application from tampering with the decision file. `open()` is hooked such that whenever it is invoked on the decision file it will check the JNI call stack and allow only Aurasium code to successfully open the file. The various reflection APIs are also guarded to prevent malicious applications from modifying Aurasium's Java-based policy logic by reflection.

2.3.1 Privacy Policy

The most obvious set of policies that can be defined relates to users' privacy. These policies protect the private data of the user such as the IMEI, IMSI, phone number, location, stored SMS messages, phone conversations, and contact list. These policies can be checked by monitoring access to the system services provided by the Framework APIs. While many APIs are available to access system services, they all translate to a single call to the `ioctl()` system call. By monitoring calls to `ioctl()`, and parsing the data that is transmitted in the call, we are able to determine which service is being accessed and alert the user.
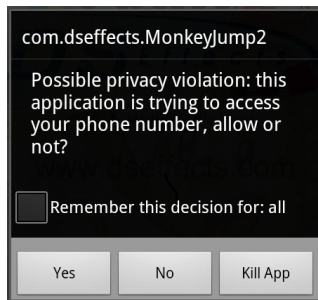
Figure 3: Enforcement of Privacy Policies: Access to Phone Number

Figure 3 illustrates how Aurasium intercepts a request made by an application to access the user's phone number. Aurasium displays a warning message and prompts

---

[5] `Dalvik_java_lang_reflect_Method_invokeNative()`, `Dalvik_java_lang_reflect_Field_setField()` and `Dalvik_java_lang_reflect_Field_setPrimitiveField()`

the user to either accept the requested access or deny it. The user can also make Aurasium store that user's answer to the request so that the same request never prompts the user for approval again and the cached answer is used instead. Finally, the user has the option to terminate the application.

Aurasium is capable of intercepting requests for the IMEI (Figure 4) and IMSI identifiers. Both the IMEI and IMSI numbers are often abused by applications to track users and devices for analytics and advertisement purposes, but are also used by malware to identify victims.

Similar policies are also implemented for accessing device location and contact list. In all of the above cases, if the user denies an request for the private information, Aurasium will provide shadow data to the application instead, similar to the approach in [27];
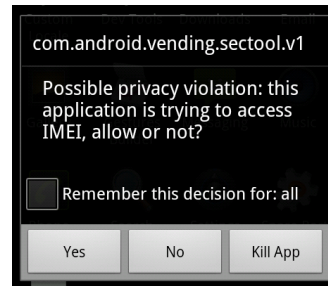
Figure 4: Enforcement of Privacy Policies: Access to IMEI from repackaged Android Market Security Tool malware.

2.3.2 Preventing SMS Abuse

Figure 5 illustrates how Aurasium intercepts SMS messages sent to a premium number, which is initiated by the malicious application AndroidOS.FakePlayer [2] found in the wild. Aurasium displays the destination number as well as the SMS's content, so users can make informed decision on whether to allow the operation or not. In this case, the malware is most likely to attempt to subscribe to some premium service covertly. We also observed malware NickySpy [3] leaking device IMEI via SMS in another test run. We believe automatic classification on SMS number and content is possible to further reduce user intervention.

2.3.3 Network Policy

Similarly to the privacy policies, we enforce a set of network policies that regulate how an application is allowed to interact with the network. Since the Android permission scheme allows unrestricted access to the Internet when an application is installed, we enforce finer-grained
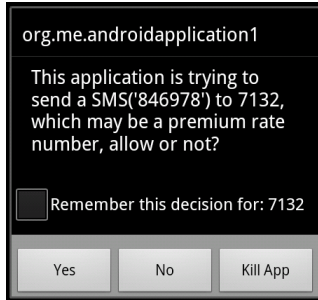
Figure 5: Enforcement of SMS Sending

policies that are expressed as a combination of the following:

- restrict the application to only a particular web domain or set of IP addresses

- restrict the application from connecting to a remote IP address known to be malicious

Figure 6: Enforcement of Network Policies: Access to an IP address with an unverified level of maliciousness

We use an IP blacklisting provided by the Bothunter network monitoring tool [4] to harvest information about malicious IP addresses. For each connection, the service retrieves information about the remote location, and the warning presented to the user indicates the level of maliciousness of the remote location (Figure 6). We also display the geo-location of the remote IP. It would be possible to include more threat intelligence from various diverse sources.

2.3.4 Privilege Escalation Policy

In addition to the privacy policy and the network policy, we implement a policy that warns the user when a suspicious `execvp` is invoked. Aurasium intervenes whenever the application tries to execute external ELF binaries.

Knowing the attack signatures based on suspicious executables can prevent certain types of escalation of privilege attacks. Figure 7 illustrates an interception of the `su` command. The Aurasium warning indicates that the application is trying to gain root access on a potentially rooted phone by executing the `su` command.

In another scenario, Aurasium warns the user when the application is about to load a native library. Malicious native code can interfere with Aurasium and potentially break out of its sandbox, which we discuss further in section 4.

Figure 7: Enforcement of Privilege Escalation Policy

2.3.5 Automatic Embedding of policies

Our implementation allows us to naturally compare the behavior of an application against a policy expressed not as a single event such as a single access to private data, to a system service or a single invocation of a system call, but as a sequence of such events. We plan on automatically embedding into an application code an arbitrary user-defined policy expressible in an automaton.

2.4 Deployment Models

Driven by the need for deployable mobile security solutions for Android and other platforms, we support multiple deployment models for Aurasium. The unrestricted and open nature of the Android Market allows us to provide Aurasium hardened and repackaged applications to users directly. Here, we discuss several deployment models for Aurasium that users can directly use without modifying the Android OS on their phones.

2.4.1 Web Interface

We have a web interface[6] that allows users to upload arbitrary applications and download the Aurasium repackaged and hardened version. Aurasium can be employed to repackage any APKs that the user possesses.

---

[6] www.aurasium.com

2.4.2 Cooperation with Application Markets

We are exploring collaborations with Android markets run by mobile service providers to deploy Aurasium. Subscribers to the mobile service who get their applications from the official Android market supported by the mobile provider will have all their applications packaged with Aurasium for protection.

2.4.3 Deployment in the Cloud

Another deployment model consists of writing a custom download application that runs on a user's phone so that whenever a user browses an Android market and wishes to download an application, the application is pulled and sent to the Aurasium cloud service where the application is repackaged and then downloaded to the user's phone. This may be more accessible as users no longer need to interact with Aurasium's web interface manually.

2.4.4 Phone Deployment

Similarly to the cloud service, we plan on porting the repackaging tool to the Android phone itself. That is, we will be able to repackage an application on the device itself.

2.4.5 Corporate Environment

Many corporations have security concerns about mobile devices in their infrastructure and Aurasium can help to establish the desired security and privacy polices on applications to be installed on these devices. These Android devices should be configured to allow installing only Aurasium-protected applications (by means of APK signatures for example), while the applications can be provided by some methods described above, such as an internal repackaging service or a transparent repackaging proxy between the application market and the device.

3 Evaluation

We have evaluated Aurasium on a collection of Android applications to ensure that the application repackaging succeeds and that our added code does not impede the original functionality of the application. We have conducted a broad evaluation that includes a large number of benign applications as well as malware collection. Our evaluation was conducted on a Samsung Nexus S phone running Android 2.3.6 "Gingerbread".

3.1 Setting Up An Evaluation Framework

Aurasium consists of scripts that implement the repackaging process described in Figure 2. It transforms each APK file in the corpus to the corresponding hardened repackaged application. We scripted to load the application onto the Nexus S phone, start the application automatically, and capture the logs generated by Aurasium. Android Monkey [8] is used to randomly exercise the user interface (UI) of the application.

Monkey is a program running on Android that feeds the application with pseudo-random streams of user events such as clicks and touches, as well as a number of system-level events. We use Monkey to stress-test the repackaged applications in a random yet repeatable manner. The captured logs allow us to determine whether the application has started and is being executed normally or whether it crashes due to our repackaging process. As a random fuzzer, Monkey is fundamentally unable to exercise all execution paths of an application. But in our setup, running random testing over a large number of independent applications proves useful, covering most of Aurasium's policy logic and revealing several bugs.

3.2 Repackaging Evaluation

We first performed an evaluation to determine how many APK files can successfully be repackaged by Aurasium. Table 1 shows a breakdown of the Android APK files corpus on which we ran our evaluation. We applied Aurasium to 3491 applications crawled from a third-party application store[7] and 1260 known malicious applications [39]. Table 1 shows the success rate of repackaging for each category of applications.

Type of App	#of Apps	Repackaging Success Rate
App store corpus	3491	99.6%(3476)
Malware corpus	1260	99.8%(1258)

Table 1: Repackaging Evaluation Results

We have a near 100% success rate in repackaging arbitrary applications. Our failures to repackage an application are due to bugs in apktool in disassembling and reassembling the hardened APK file. We are working on improving apktool to achieve a 100% success rate.

3.3 Runtime Robustness

As we pointed out earlier, Aurasium is able to run on all major Android versions (2.2, 2.3, 3.x) without any problem. We performed the robustness evaluation on a Samsung Nexus S phone running Android 2.3.6 (which is among the most widely used Android distributions

---

[7] http://lisvid.com

2.3.3 − 2.3.7 [10]). For each hardened application we use `Monkey` to exercise the application's functionalities by injecting 500 random UI events. These hardened applications are built with a debug version of Aurasium that will output a log message when Aurasium successfully intercepts an API invocation. Out of 3476 successfully repackaged application, we performed tests on 3189 standalone runnable applications[8] on the device. We were able to start all of the applications in the sense that Aurasium successfully reported the interception of the first API invocation for all of them.

3.4 Performance Evaluation

We take two Android benchmark applications from the official market and apply Aurasium to them in order to check if Aurasium introduces significant performance overhead to a real-world application. In both cases, the benchmark scores turn out to be largely unaffected by Aurasium (Table 2).

Benchmark App	without Aurasium	with Aurasium
AnTuTu Benchmark	2900 *Pts*	2892 *Pts*
BenchmarkPi	1280 *ms*	1293 *ms*

Table 2: Performance on Benchmark Applications

Aurasium introduces the most overhead when the application performs API invocations, which is not the most important test factor of these benchmarks. So we synthesized an artificial application that performs a large number of API invocations, in order to find Aurasium's performance overhead in the worst cases. Because these APIs all involve IPC with remote system services, they are expected to induce the most overhead as Aurasium needs to fully parse the Binder communication. Results in Table 3 show that Aurasium introduces an overhead of 14% to 35% in three cases, which we believe is acceptable as IPC-based APIs are not frequently used by normal applications to become the performance bottleneck. In objective testing we did not feel any lagging when playing with an Aurasium-hardened application.

3.5 Size Overhead

We evaluated application size after being repackaged with Aurasium code, as shown in Figure 8. On average, Aurasium increases the application size by only 52

---

[8]The rest are applications that do not have a main launchable Activity, and applications that fail to install due to clashes with pre-installed version.

200 API Invocations	Without Aurasium	With Aurasium	Overhead
Get Device Info	106 *ms*	143 *ms*	35%
Get Last Location	41 *ms*	55 *ms*	34%
Query Contact List	1270 *ms*	1340 *ms*	14%

Table 3: Performance on Synthesized Application

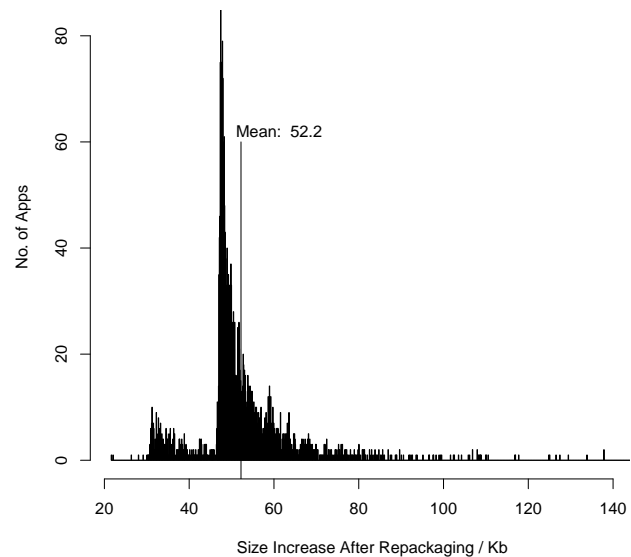Kb, which is a very small overhead for the majority of applications.

Figure 8: Application Size Increase After Repackaging.

3.6 Policies Enforcement

We observe the various behaviors intercepted from the 3031 runnable applications that were previously repackaged and run on the Nexus S device under `Monkey`. Table 4 shows a breakdown of the application corpus into permission requested in the manifest file of the applications. It also shows which applications actually make use of the permission to access the requested service.

Permission	Requested	Accessed
Internet Permission	2686	1305
GPS Permission	846	132
Phone State Permission	1243	378

Table 4: Permission Requested and Permissions Used

Due to the random fuzzing nature of our evaluation,

the accessed permission is most likely to be an underestimate. We also observed that 226 applications included native code libraries in their application bundle.

4 Attack Surfaces

Because fundamentally Aurasium code runs in the same process context as the application's code, there is no strong barrier between the application and Aurasium. Hence, it is non-trivial to argue that Aurasium can reliably sandbox arbitrary Android applications. We describe possible ways that a malicious application can break out of Aurasium's policy enforcement mechanism and discuss possible mitigation against them.

4.1 Native Code

Aurasium relies on being able to intercept calls to Bionic libc functions by means of rewriting function pointers in a module's global offset table. This is robust against arbitrary Java code, but a malicious application can employ native code to bypass Aurasium completely either by restoring the global offset table entries, by making relevant system calls using its own libc implementation rather than going through the monitored libc, or by tampering with the code or private data of Aurasium. However, because Android runtime requires applications to bootstrap as Java classes, the first load of native code in even malicious applications has to go through a well-defined and fixed pathway as defined by JNI. This gives us an upper hand in dealing with potential untrusted native code: because of the way our repackaging process works, Aurasium is guaranteed to start before the application's code and hence be able to intercept the application's first attempt to load alien native code (invocation of `dlopen()` function in libc). As a result, Aurasium is guaranteed to detect any potential circumvention attempts by a malicious application.

What can Aurasium do with such an attempt? Silently denying the load of all native code is not satisfactory because it will guarantee an application crash and some legitimate applications use native code. Even though Aurasium has the power to switch off the unknown native code, the collateral damage caused by false positives would be too severe.

If Aurasium is to give binary decisions on whether or not to load some unknown native code, then it reduces to the arms race between malware and antivirus software that we have seen for years. Aurasium tries to classify native code in Android applications, while malware authors craft and obfuscate it to avoid being detected. It is better not to go down the same road; and a much neater approach would be letting the native code run, but not with unlimited power.

Previous work [28, 40, 34] on securely executing untrusted native code provides useful directions for example, by using dynamic binary translation. In our scenario we are required to restrict the application's native code from writing to guarded memory locations (to prevent tampering with Aurasium and the libc interposition mechanism), using special machine instructions (to initiate system calls without going through libc), and performing arbitrary control flow transfer into libc. Due to time constraints we have not implemented such facilities in Aurasium. Currently, Aurasium prompts the user for a decision, and informs the user that if the load is allowed then Aurasium can be rendered ineffective from this point onwards. We consider this problem a high priority for future work.

Unlike the filtering-based hybrid sandboxes that are prone to the 'time of check/time of use' race conditions [25, 38], Aurasium's sandboxing mechanism is delegation based and hence much easier to defend against this class of attack.

4.2 Java Code

A possible attack on Aurasium would be using Java's reflection mechanism to interfere with the operation of Aurasium. Because currently Aurasium's policy enforcement logic is implemented in Java, a malicious application can use reflection to modify Aurasium's internal data structures and hence affect its correct behavior. We prevent such attacks by hooking into the reflection APIs in `libdvm.so` and preventing reflection access to Aurasium's internal classes.

Note that dynamically loaded Java code (via DexClassLoader) poses no threat to Aurasium, as the code is still executed by the same Dalvik VM instance and hence cannot escape Aurasium's sandbox. Native Java methods map to a dynamically loaded binary shared object library and are subject to the constraints discussed in the previous section, which basically means that attempts of using them will always be properly flagged by Aurasium.

4.3 Red Pill

Currently Aurasium is not designed to be stealthy. The existence of obvious traces such as changed application signature, the existence of Aurasium native library and Java classes allow applications to find out easily whether it is running under Aurasium or not. A malicious application can then refuse to run under Aurasium, forcing the user to use the more dangerous vanilla version. A legitimate application may also verify its own integrity (via application signature) to prevent malicious repackaging by malware writers. Due to Aurasium's control over the application's execution, it is possible to clean

up these traces for example by spoofing signature access to `PackageManager`, but fundamentally this is an arms race and a determined adversary will win.

5 Related Work

With the growing popularity of Android and the growing malware threat it is facing, many approaches to securing Android have been proposed recently. Many of the traditional security approaches adopted in desktops have been migrated to mobile phones in general and Android in particular. Probably the most standard approach is to use signature-based malware detection, which is in its infancy when it comes to mobile platforms. This approach is ineffective against zero-day attacks, and there is little reason to believe that it will be more successful in the mobile setting. Program analysis and behavioral analysis have been more successfully applied in the context of Android.

Static Analysis Static analysis of Android application package files is relatively more straightforward than static analysis of malware prevalent on desktops in general. Obfuscation techniques [41] used in today's malware are primarily aimed at impeding static analysis. Without effective ways to deobfuscate native binaries, static analysis will always suffer major drawbacks. Because of the prevalence of malware on x86 Windows machines, little effort has been focusing on reverse engineering ARM binaries. Static analysis of Java code is much more attainable through decompilation of the Dalvik bytecode. The DED [20] and dex2jar [5] are two decompilers that aim at achieving translation from Dalvik bytecode to Java bytecode.

Dynamic Analysis Despite its limitations, dynamic analysis remains the preferred approach among researchers and antivirus companies to profile malware and extract its distinctive features. The lack of automated ways to explore all the state space is often a hindering factor. Techniques such as multipath exploration [31] can be useful. However, the ability of mobile malware to load arbitrary libraries might limit the effectiveness of such techniques. The honeynet project offers a virtual machine for profiling Android Applications [36] similar to profiling desktop malware. Stowaway [23] is a tool that detects overprivilege in compiled Android applications. Testing is used on the Android API in order to build the permission map that is necessary for detecting overprivilege, and static analysis is used to determine which calls an application invokes.

Monitoring The bulk of research related to securing Android has been focused on security policy extension and enforcement for Android starting with [21]. TaintDroid [19] taints private data to detect leakage of users' private information modifying both Binder and the Dalvik VM, but extends only partially to native code. Quire [17] uses provenance to track permissions across application boundaries through the IPC call chain to prevent permission escalation of privilege attacks. Crepe [15] allows access to system services requested through install-time permission only in a certain context at runtime. Similarly, Apex [33] uses user-defined runtime constraints to regulate applications' access to system services. AppFence [27] blocks application access to data from imperious applications that demand information that is unnecessary to perform their advertised functionality, and covertly substitute shadow data in place. Airmid [32] uses cooperation between in-network sensors and smart devices to identify the provenance of malicious traffic.

Virtualization Recent approaches to Android security have focused on bringing virtualization technology to Android devices. The ability to run multiple version of the Android OS on the same physical device allows for strong separation and isolation but comes at a higher performance cost. L4Android [30] is an open source project derived from the L4Linux project. L4Android combines both the L4Linux and Google modifications of the Linux kernel and thus enables running Android on top of a microkernel. To address the performance issues when using virtualization, Cells in [11], is a lightweight virtualization architecture where multiple phones run on the same device. It is possible to run multiple versions of Android on a bare metal hypervisor and ensure strong isolation where shared security-critical device drivers run in individual virtual machines, which is demonstrated by [26]. Finally, logical domain separation, where two single domains are considered and isolation is enforced as a dataflow property between the logical domains without running each domain as a separate virtual machine, can also be employed [35].

6 Conclusion and Future Work

We have presented Aurasium, a robust and effective technology that protects users of the widely used Android OS from malicious and untrusted applications. Unlike many of the security solutions proposed so far, Aurasium does not require rooting and device reflashing.

Aurasium allows us to take full control of the execution of an application. This allows us to enforce arbitrary policies at runtime. By using the Aurasium security

manager (ASM), we are able to not only apply policies at the individual application level but across multiple applications simultaneously. This allows us to effectively orchestrate the execution of various applications on the device and mediate their access to critical resources and user's private data. This allows us to also detect attempts by multiple applications to collaborate and implement a malicious logic. With its overall low overhead and high repackaging success rate, it is possible to imagine Aurasium implementing an effective isolation and separation at the application layer without the need of complex virtualization technology.

Even though Aurasium currently only treats applications as black boxes and focuses on its external behavior, the idea of enforcing policy at per-application level by repackaging applications to attach side-by-side monitoring code is very powerful. By carefully instrumenting the application's Dalvik VM instance on the fly, it is even possible to apply more advanced dynamic analysis such as information flow and taint analysis, and we leave this as a possible direction of future work. We also plan on expanding our investigation of the potential threat models against Aurasium and provide practical ways to mitigate them, especially in the case of executing untrusted native code.

7   Acknowledgments

References

[1] Android apktool: A tool for reengineering Android apk files. code.google.com/p/android-apktool/.

[2] Android.OS/Fakeplayer. www.f-secure.com/v-descs/trojan_androidos_fakeplayer_a.shtml.

[3] Android.OS/NickiSpy. www.maikmorgenstern.de/wordpress/?tag=androidnickispy.

[4] Bothunter community threat intelligence feed. http://www.bothunter.net.

[5] dex2jar: A tool for converting Android's .dex format to Java's .class format. code.google.com/p/dex2jar/.

[6] OpenBinder. www.angryredplanet.com/~hackbod/openbinder/docs/html/.

[7] smali: An assembler/disassembler for Android's dex format. code.google.com/p/smali/.

[8] UI/Application exerciser Monkey. developer.android.com/guide/developing/tools/monkey.html.

[9] In U.S. market, new smartphone buyers increasingly embracing Android. blog.nielsen.com/nielsenwire/online_mobile/, sep 2011.

[10] ANDROID OPEN SOURCE PROJECT. Platform versions. developer.android.com/resources/dashboard/platform-versions.html.

[11] ANDRUS, J., DALL, C., HOF, A. V., LAADAN, O., AND NIEH, J. Cells: a virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 173–187.

[12] BLÄSING, T., SCHMIDT, A.-D., BATYUK, L., CAMTEPE, S. A., AND ALBAYRAK, S. An Android application sandbox system for suspicious software detection. In *5th International Conference on Malicious and Unwanted Software (MALWARE'2010)* (Nancy, France, France, 2010).

[13] BURGUERA, I., ZURUTUZA, U., AND NADJM-TEHRANI, S. Crowdroid: behavior-based malware detection system for Android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile devices* (New York, NY, USA, 2011), SPSM '11, ACM, pp. 15–26.

[14] CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2011), MobiSys '11, ACM, pp. 239–252.

[15] CONTI, M., NGUYEN, V. T. N., AND CRISPO, B. Crepe: context-related policy enforcement for Android. In *Proceedings of the 13th International Conference on Information Security* (Berlin, Heidelberg, 2011), ISC'10, Springer-Verlag, pp. 331–345.

[16] DEGUSTA, M. Android orphans: Visualizing a sad history of support. theunderstatement.com/post/11982112928/android-orphans-visualizing-a-sad-history-of-support.

[17] DIETZ, M., SHEKHAR, S., PISETSKY, Y., SHU, A., AND WALLACH, D. S. Quire: lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX Conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 23–23.

[18] ENCK, W. Defending users against smartphone apps: Techniques and future directions. In *Proceedings of the 7th International Conference on Information Systems Security* (Kolkata, India, dec 2011), ICISS.

[19] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–6.

[20] ENCK, W., OCTEAU, D., MCDANIEL, P., AND CHAUDHURI, S. A study of Android application security. In *Proceedings of the 20th USENIX conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 21–21.

[21] ENCK, W., ONGTANG, M., AND MCDANIEL, P. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 235–245.

[22] ENCK, W., ONGTANG, M., AND MCDANIEL, P. Understanding Android security. *IEEE Security and Privacy 7* (January 2009), 50–57.

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

[24] FELT, A. P., FINIFTER, M., CHIN, E., HANNA, S., AND WAGNER, D. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (Oct. 2011), SPSM '11, ACM, pp. 3–14.

[25] GARFINKEL, T., PFAFF, B., AND ROSENBLUM, M. Ostia: A delegating architecture for secure system call interposition. In *Proceedings of the Network and Distributed Systems Security Symposium* (February 2004).

[26] GUDETH, K., PIRRETTI, M., HOEPER, K., AND BUSKEY, R. Delivering secure applications on commercial mobile devices: the case for bare metal hypervisors. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile devices* (New York, NY, USA, 2011), SPSM '11, ACM, pp. 33–38.

[27] HORNYACK, P., HAN, S., JUNG, J., SCHECHTER, S., AND WETHERALL, D. 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* (New York, NY, USA, 2011), CCS '11, ACM, pp. 639–652.

[28] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. P. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium* (Berkeley, CA, USA, 2002), USENIX Association, pp. 191–206.

[29] LABS, M. McAfee threats report: Second quarter 2011. www.mcafee.com/us/resources/reports/rp-quarterly-threat-q2-2011.pdf, aug 2011.

[30] LANGE, M., LIEBERGELD, S., LACKORZYNSKI, A., WARG, A., AND PETER, M. L4Android: a generic operating system framework for secure smartphones. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile devices* (New York, NY, USA, 2011), SPSM '11, ACM, pp. 39–50.

[31] MOSER, A., KRUEGEL, C., AND KIRDA, E. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2007), SP '07, IEEE Computer Society, pp. 231–245.

[32] NADJI, Y., GIFFIN, J., AND TRAYNOR, P. Automated remote repair for mobile malware. In *Proceedings of the 2011 Annual Computer Security Applications Conference* (Washington, DC, USA, 2011), ACSAC '10, ACM.

[33] NAUMAN, M., KHAN, S., AND ZHANG, X. Apex: extending Android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2010), ASIACCS '10, ACM, pp. 328–332.

[34] SCOTT, K., KUMAR, N., VELUSAMY, S., CHILDERS, B., DAVIDSON, J. W., AND SOFFA, M. L. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the international symposium on Code Generation and Optimization: feedback-directed and runtime optimization* (Washington, DC, USA, 2003), CGO '03, IEEE Computer Society, pp. 36–47.

[35] SVEN, B., LUCAS, D., ALEXANDRA, D., STEPHAN, H., AHMAD-REZA, S., AND BHARGAVA, S. Practical and lightweight domain isolation on Android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile devices* (New York, NY, USA, 2011), SPSM '11, ACM, pp. 51–62.

[36] THE HONEYNET PROJECT. Android reverse engineering virtual machine. www.honeynet.org/node/783.

[37] VIDAS, T., VOTIPKA, D., AND CHRISTIN, N. All your droid are belong to us: a survey of current Android attacks. In *Proceedings of the 5th USENIX Workshop On Offensive Technologies* (Berkeley, CA, USA, 2011), WOOT'11, USENIX Association, pp. 10–10.

[38] WATSON, R. N. M. Exploiting concurrency vulnerabilities in system call wrappers. In *Proceedings of the first USENIX Workshop On Offensive Technologies* (Berkeley, CA, USA, 2007), USENIX Association, pp. 2:1–2:8.

[39] YAJIN, Z., AND XUXIAN, J. Dissecting android malware: Characterization and evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy* (may 2012).

[40] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: a sandbox for portable, untrusted x86 native code. *Commun. ACM 53* (January 2010), 91–99.

[41] YOU, I., AND YIM, K. Malware obfuscation techniques: A brief survey. In *Proceedings of the 2010 International Conference on Broadband, Wireless Computing, Communication and Applications* (Washington, DC, USA, 2010), BWCCA '10, IEEE Computer Society, pp. 297–300.