AppGuard — Real-time policy enforcement for third-party applications

*Michael Backes, Sebastian Gerling,
Christian Hammer, Matteo Maffei,
and Philipp von Styp-Rekowsky*

Technischer Bericht Nr. A/02/2012

AppGuard - Real-time policy enforcement for third-party applications

Michael Backes[1,2] Sebastian Gerling[1] Christian Hammer[1] Matteo Maffei[1]
Philipp von Styp-Rekowsky[1]

[1] *Saarland University*
[2] *Max Planck Institute for Software Systems*

Abstract

Android has become the most popular operating system for mobile devices, which makes it a prominent target for malicious software. The security concept of Android is based on app isolation and access control for critical system resources. However, users can only review and accept permission requests at install time, or else they cannot install an app at all. Android neither supports permission revocation after the installation of an app, nor dynamic permission assignment. Additionally, the current permission system is too coarse for many tasks and cannot easily be refined. We present an inline reference monitor system that overcomes these deficiencies. It extends Android's permission system to impede overly curious behaviors; it supports complex policies, and mitigates vulnerabilities of third-party apps and the OS. It is the first solution that provides a practical extension of the current Android permission system as it can be deployed to all Android devices without modification of the firmware or *root* access to the smartphone. Our experimental analysis shows that we can remove permissions for overly curious apps as well as defend against several recent real-world attacks on Android phones with very little space and runtime overhead. AppGuard is available from the Google Play market[1].

1 Introduction

Smartphones and tablet computers have become our every day companions, providing assistance and comfort both in business and in our private lives. In 2008 the Open Handset Alliance, a group of companies led by Google [1], joined the smartphone market with the open source software stack Android. By now, it has become the most popular operating system for these devices [27]. More than 450,000 apps in the official market Google Play and more than 850,000 activations of new devices per day demonstrate the success of Android [42].

However, the rapidly increasing numbers of mobile devices also creates a vast potential for misuse as the development of new security concepts did not keep pace with the development of new and fancy features. Mobile devices store a plethora of information about our personal lives, and their sensors, GPS, camera, or microphone – just to name a few – potentially track us at all times. Further, the always-online nature of mobile devices makes them both interesting and exposed targets for attackers and overly curious or maliciously spying apps and trojan horses that hide their true nature in an unsuspicious app. For instance, social network apps were recently criticized for silently downloading and storing the user's entire contacts to the network's servers [17, 43]. While this behavior became publicly known, users are most often not even aware of what an app actually does with their data. Additional complications arise because even fixes for recent security vulnerabilities in the Android operating system often take months until they are integrated into the vendor-specific OS by all vendors (a process called late updates). During the time between Google's fix (and the corresponding, publicly available vulnerability description) and the vendor's update, an unpatched system becomes the obvious target for exploits.

Android's security concept is based on isolation of third-party apps and access control [2]. Access to personal information has to be explicitly granted at install time: During the installation of an app the user is provided with a list of permissions the app requests. The user can either accept all of these permissions, or else the app will not be in-

[1] https://play.google.com/store/apps/details?id=com.srt.appguard.mobile

stalled. Users can neither dynamically grant and revoke permissions at runtime, nor add restrictions according to their personal needs. Furthermore, users are often not aware of a permission's impact. They usually do not have enough information to judge whether a permission is indeed required to fulfill a certain task. Research has shown that this is the case even for Android app developers [24, 28].

In order to overcome the current limitations with Android's permission system, researchers have proposed several approaches. Some work [40, 39, 38, 12] focuses on extending the current permission system, e.g. to prevent critical permission combinations [20] (e.g. camera and Internet access could be used to implement a bug), but most work [9, 19, 28, 18, 29, 41] targets the detection of privacy leaks and malicious third-party apps. While most of these approaches solve one particular problem in theory, the vast majority of these systems rely on modifications of the underlying software stack, which prevents deployment to off-the-shelf Android phones. So far, no solution simultaneously removes permissions for overly curious behavior, supports fine-grained security policies, prevents malicious applications from exploiting security vulnerabilities, and is easily deployed on all existing Android devices.

1.1 Contributions

In this paper we present a novel policy-based security framework for Android that overcomes the aforementioned limitations of Android's security system. Our approach proposes solutions for a variety of situations, in particular for:

1. *Revoking Android permissions dynamically.* Permissions can be granted and revoked at any time after installation of an app. We support graceful revocation by selectively suppressing undesired operations without terminating the program.
2. *Enforcing complex stateful and fine-grained security policies.* Policies can transform the sequence of program actions in case the program deviates from the security policy. We support all predicates over that sequence as security policy, which enables enforcement of any security property [37].
3. *Policy-based quick-fixes for vulnerabilities in third-party applications.* When an app uses the API in an insecure way, we can transform the execution to leverage alternative, secure functionality. While it would be cleaner to fix the app directly, a rapid work-around provides a temporary solution until the vendor provides an update.
4. *Policy-based mitigation for vulnerabilities in the operating system itself [10].* Known OS vulnerabilities pose a major threat to system security. Our framework defends against malicious apps that try to exploit a vulnerability, which is paramount to relieve the late update problem, where vendors integrate the fix at a later time than Google, or do not provide security patches at all.

We have built a prototypical implementation called AppGuard that supports all features listed. Our system does not require any modification to the core software stack of the Android device and thus supports widespread deployment as a stand-alone app. AppGuard is based on inline reference monitoring [21]. It takes user-defined policies as input and delivers a secured self-monitoring app. Our evaluation on typical Android apps has shown very little overhead in terms of space and runtime. The case studies demonstrate the effectiveness of our approach: we successfully limit the excessive curiosity of apps, demonstrate complex policies and prevent several recent real-world attacks on Android phones. To the best of our knowledge, this is the first defense against these attacks on phones with standard firmware.

2 The Android Monitor

Android's current permission system fails to address some challenges of mobile applications: apps basically dictate the permissions deemed necessary and users have to grant them as requested short of not installing the application. Therefore, the individual security requirements of users are ignored as they are not really given a choice. In particular, they are unable to grant each permission individually and to grant and revoke certain permissions at a later time. This deficiency becomes particularly severe as not only users have problems with Android's permission system. Song et al. [24] have shown that even application developers have problems requesting the "correct" permissions for the functionality of their applications, since the Android documentation lacks completeness. If in doubt, they will therefore request too many permissions in order to make their application work.

Our AppGuard puts the user back in charge of application permissions. With our system users are given the opportunity to decide both at installation time of an app, as well as at any later point in time what an app is allowed to access. AppGuard provides more fine-grained policies than Android does itself, and, in contrast to other proposed solutions [38, 12], it is oblivious of the installed Android firmware. It can be installed on all existing Android phones without modifying the core system, which enables wide-spread deployment.

Runtime policy enforcement for third-party applications is not an easy feat on unmodified Android systems. Android's security concept strictly isolates different applications installed on the same device, preventing them from interfering with each other at runtime. Furthermore, applications cannot gain elevated privileges that would enable them to observe the behavior of other applications. Communication between apps is only possible via Android's inter-process communication (IPC) mechanism. So far, such communication requires both parties to cooperate, rendering this channel unsuitable for a generic runtime monitor.

AppGuard tackles this open problem by following an approach pioneered by Erlingsson and Schneier [22] called *inline reference monitor* (IRM). The basic idea is to rewrite an untrusted application such that the code that monitors the application is directly embedded into its code. To this end, IRM systems incorporate a *rewriter* or *inliner* component, that injects additional security checks at critical points into the application bytecode. This enables the monitor to observe a trace of *security-relevant events*, which typically correspond to invocations of trusted system library methods from the untrusted application. To actually enforce a *security policy*, the monitor controls the execution of the application by suppressing or altering calls to security-relevant methods, or even by terminating the program if necessary.

In the IRM context, a policy is typically specified by means of a security automaton that defines which sequences of security-relevant events are acceptable. Such policies have been shown to express exactly the policies enforceable by runtime monitoring [45]. Ligatti et al. differentiate security automata by their ability to enforce policies by manipulating the trace of the program [37]. Some IRM systems [22, 15] implement simple truncation automata, which can only terminate the program if it deviates from the policy. However, this is often undesirable in practice. In their paper [37], Ligatti et al. formulate the notion of *edit automata*, which can transform the program trace by inserting or suppressing events. Monitors based on edit automata are able to react gracefully to policy violations, e.g. by suppressing an undesired method call and returning a dummy value, but allowing the program to continue.

AppGuard is an IRM system for Android with the transformation capabilities of an edit automaton. Figure 1 provides a high-level overview of our system. We distinguish three main components:

1. A set of security policies. AppGuard provides various Android-specific security policies that govern access to platform API methods that are protected by coarse-grained Android permissions. These methods comprise e.g. methods for reading personal data, creating network sockets, or accessing device hardware like the GPS or the camera. As a starting point for the security policies, we used the Android permission map by Song et al. [24].
2. The program rewriter. Android applications run within a custom register-based Java VM called *Dalvik*. Our rewriter manipulates Dalvik executable (`dex`) bytecode of untrusted Android applications and generates monitoring code according to the policies to harness the untrusted app.
3. A management component. It offers a graphical user interface that allows the user to set individual policy configurations on a per-application basis. In particular, policies can be turned on or off and be parameterized. In addition, the management component keeps a detailed log of all security-relevant events, enabling the user to monitor the behavior of an application.

3 Implementation

AppGuard is a stand-alone Android application written in pure Java and comprises about 6500 lines of code. It builds upon the *dexlib* library, which is part of the *smali* disassembler for Android by Ben Gruver [32], for manipulating `dex` files. The size of the application package is roughly 750 Kb.

3.1 Policies

In our system, a policy is defined by a set of security-relevant method signatures and corresponding callback methods. In our system poli-

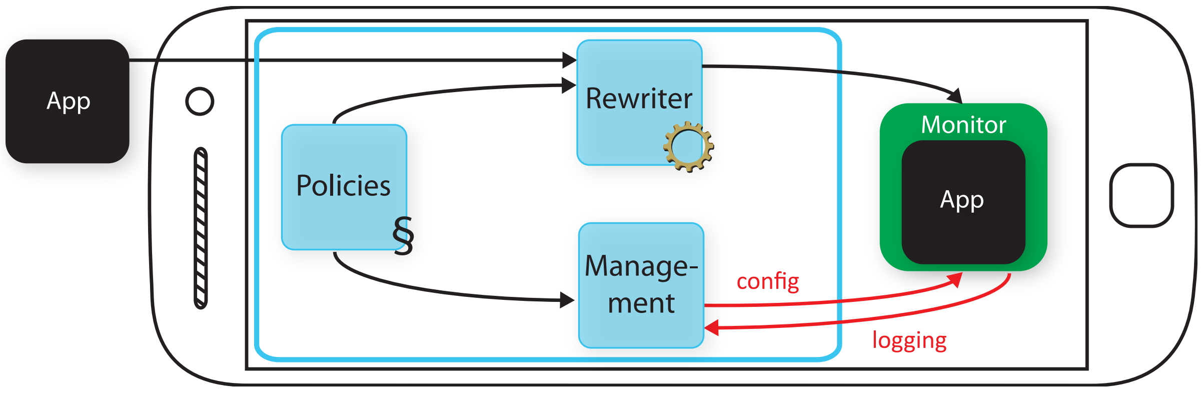

Figure 1: Schematics of the AppGuard

cies are implemented as Java classes. The callback methods are mapped to a set of method signatures using a custom method annotation `MapSignatures`. Consider Fig. 2 as a basic example. This policy guards access to the `openConnection()` method in the `java.net.URL` class and only allows connections to the host "wetter.com".

A policy callback method gains access to the arguments of the original method call by declaring an equivalent list of parameters. In our example, the `checkConnection(URL)` callback method uses the `URL` parameter to decide whether a connection should be allowed. If allowed, the callback method will simply return such that the original method call can proceed. If the connection is not allowed, an exception will be thrown that is either caught by the surrounding application code or by a synthetic handler introduced by the inliner. In both cases the original method call will be prevented by this scheme. Details will be discussed in the next subsection.

Furthermore, policies can be stateful and store security state information in member variables of the policy class. The values of these variables are preserved across callback method invocations. Member variables could also be used to store the complete history of intercepted methods.

In general, policy callbacks can perform arbitrary operations. As an example, consider a policy that intercepts `http` connections and relays them to encrypted `https`, if available (cf. Fig. 3.) After calling the original method with the new arguments, it throws an exception containing the returned value, which will be substituted for the return value of the original method as described in the next section.

3.2 Inliner

The task of the inliner component is to divert the control flow of the target application to the monitoring code at invocation instructions to security-relevant methods. There are two strategies for passing control to the monitor: Either at the call-site in the application code, right before the invocation of the security-relevant method, or at the callee-site, i.e. at the beginning of the security-relevant method. The latter strategy is simpler and more efficient, because callee sites are easier to identify and less in number [5]. Unfortunately, callee-site rewriting is not applicable in our scenario, as security-relevant methods are declared in the trusted Android platform libraries, which are part of the non-modifiable firmware image. Thus, our inliner implements a call-site control flow diversion strategy. The inlining process consists of three steps:

1. Merging the policy classes into the target application's `classes.dex` file, which contains all Dalvik bytecode for the app.
2. Generating the `MonitorInterface`, which serves as a bridge between application code and policies.
3. Injecting monitoring code around invocations of security-relevant methods.

The policy classes are stored precompiled in a separate `dex` file. In the first step, the inliner copies all class declarations from this file to the `classes.dex` file of the untrusted application.

In order to connect invocations of security-relevant methods to their policy callbacks, the inliner generates a utility class called `MonitorInterface` as the second step of the process. For each security-relevant method specified by the policies, the inliner generates a static guard method in the `MonitorInterface` class. The purpose of this guard

```
class InternetPolicy extends Policy {
  @MapSignatures({"Ljava/net/URL;->openConnection()"})
  public void checkConnection(URL url) throws Exception {
    if (!"wetter.com".equals(url.getHost()))
      throw new IOException();
}}
```

Figure 2: Example policy protecting calls to `java.net.URL.openConnection()`. The callback method `checkConnection(URL)` allows connections to one host only.

```
class HttpsRedirectPolicy extends Policy {
  @MapSignatures({"Ljava/net/URL;->openConnection()"})
  public void checkConnection(URL url) throws Exception {
    if (redirectToHttps(url)) {
      URL httpsUrl = new URL("https", url.getHost(), url.getFile());
      URLConnection returnValue = httpsUrl.openConnection();
      throw new MonitorException(returnValue);
}}}
```

Figure 3: Example policy that redirects `http` connections to `https` if available.

method is two-fold: First, callback methods of different policies may be defined for a single security-relevant method. Thus, the guard method invokes all policy callbacks defined for this method signature. Second, the guard method shares the signature of the security-relevant method, including the receiver object for virtual calls, which is passed as the first method argument, if available. Thus, calls to the `MonitorInterface` require only minimal modifications to the application code.

In the final step, the inliner identifes all call sites of security-relevant methods. If a matching instruction is found, the inliner adds monitoring code around the method call as depicted in Fig. 4 in the appendix. First, a method call to the corresponding guard method in the `MonitorInterface` is inserted right before the invocation of the security-relevant method. Second, the inliner adds a new **try/catch**-block around the inserted guard and the original method call. This block enables policy callbacks to pass a return value to the application code if the original call is suppressed, which also allows to protect security-relevant constructor methods. To this end, policy callbacks can throw a special `MonitorException` that carries the return value to the application code. In the inserted catch block this value is assigned to the intended variable (possibly after type conversion).

As pointed out earlier, another option for the policy is to throw an exception that will be caught by the original application code. Our example policy in Fig. 2 makes use of this technique: If a connection is not allowed, the policy callback throws an `IOException`, which resembles the behavior of the original `URL->openConnection()` method if a connection error occurs.

3.3 Management

The management component of AppGuard allows for monitoring the behavior of inlined apps and for configuring policies at runtime. The policy configuration is provided to the inlined app as a world-readable file. Its location is hardcoded into the monitor code during the inlining process. This is motivated by the fact that invocations of security-relevant methods can occur before the inlined application is fully initialized and able to perform Android IPC.

The management component provides a log of all security-relevant events, which enables the user to make informed decisions about the current policy configuration. The log is maintained based on the security-relevant method invocations encountered in the self-monitoring application, which sends its events to the management application. For this direction of the communication we are leveraging a standard Android Service component. The asynchronous nature of Android IPC is not an issue, since security-relevant method invocations that occur before the service connection is established are buffered locally.

3.4 Challenges

In our implementation we faced two main difficulties: The handling of reflection and the handling of virtual methods. In the following we will discuss both in detail.

3.4.1 Reflection

The Java Reflection API enables the inspection and manipulation of classes, fields, methods, and constructors at runtime without knowing their names at compile time. It can also be used to instantiate new objects and to invoke methods. The latter two features are relevant to our IRM system as they provide alternative ways to invoke security-relevant methods without their signatures appearing in the application bytecode.

We deal with reflection by monitoring critical methods of the Reflection API. To this end, we implement a `ReflectionPolicy` that guards invocations of `java.lang.reflect.Method->invoke()`, `Constructor->newInstance()`, and `java.lang.Class->newInstance()`. Whenever one of these methods is invoked, the policy identifies the target method of the reflective call by inspecting the method parameters. If the target matches a security-relevant method, the policy reflectively invokes the corresponding guard method in the MonitorInterface with the arguments of the reflective call. This scheme ensures that the Reflection API cannot be used to circumvent the inlined monitor.

3.4.2 Virtual methods

Virtual methods are a core concept of object-oriented programming. The target of a virtual method invocation is not determined statically, but dynamically based on the runtime type of the receiving object. More specifically, an invoke-virtual instruction with static target `A->m` can be resolved to any method `B->m` at runtime, where `B` is a subclass of `A` or `A` itself. Thus, virtual method invocations require special treatment by our inliner.

We analyze the class hierarchy of application code before the rewriting step and identify classes that inherit security-relevant methods. If a class does *not* override the security-relevant method, we add the signature of the inherited method to the set of security-relevant methods. The new signature is associated with the same guard method as the security-relevant method in the base class. If the class *does* override a security-relevant method, we can safely ignore invocations referencing the overridden method because any calls to the base class within the overridden method will be already protected by the inliner.

At the current state of the implementation, policy callbacks have to examine the runtime type of the receiver if they want to parameterize according to the target object. In our experiments, we have not encountered any case where this was required.

3.5 Deployment

In comparison to existing approaches AppGuard's novel security framework can be used on existing Android phones without requiring any changes to the operating system. In particular, it does not require *root* access to the smartphone at any time.

Third party applications installed on Android are usually assigned distinct user ids. By default, application A can neither access nor modify application B.[2] Therefore, our rewriter cannot simply modify already installed applications. Instead, we leverage the fact that installation packages of third party applications can be read from the file system by any application. We read and unpack the application packages of the application to be secured, inline the security monitor, and finally repackage and reinstall the application. In order to start this installation process, the user is asked in a preparatory step to uninstall the existing version of the application. Afterwards, it is possible to install the repackaged application without further problems.

As mentioned, all Android applications need to be signed with a developer key. Since our rewriting process breaks the original signature, we sign the modified app with a new key. However, apps signed with the same key can access each other's data if they declare so in their manifests. Thus, rewritten apps are signed with keys based on their original signatures in order to preserve the intended behavior. In particular, two apps that were originally signed with the same key, are signed with the same new key after the rewriting process.

Moreover, we ask the user to enable the OS-option to allow installation of apps that have not been signed by the Google Android market. Due to these two user interactions, no additional root privileges are required for AppGuard.

[2] Applications signed with the same developer key and those that have the "shared user id"-flag set constitute special cases.

4 Experimental Evaluation

In this section, we present the results of our experimental evaluation. It focuses on the performance of our framework and the evaluation of its effectiveness in different case studies. As testbed we used the Google Galaxy Nexus smartphone with Android 4.0.2. It has a dual-core 1.2 GHz ARM CPU from Texas Instruments (OMAP 4460) and features 1GB RAM. For our off-the-phone evaluation we use a notebook with an Intel Core i5-2520M CPU (2.5 Ghz, two cores, hyper-threading) and 8GB RAM.

4.1 Performance Evaluation

AppGuard modifies apps installed on an Android device by adding code at the bytecode level. We analyze the time it takes to inline an app and its impact on both size and execution time of the modified app.

Table 1 provides an overview of our performance evaluation for the inlining process. We have tested AppGuard with 13 apps and inlined each of the apps with 9 policies (cf. section 4.2 for details on the policies). In particular, we list the following results for each of the apps: size of the original application package (Apk), size of the `classes.dex` file before and after the inlining process (Dex and Inl, respectively) and the resulting file size difference (Diff), total number of instructions in the application code (Total), number of instructions that have been instrumented by the inliner (Chg), and, finally, the duration of the whole inlining process, both on the laptop and smartphone (PC and Phone, respectively).

The size of the `classes.dex` file increases on average by approximately 45 Kb. The majority of this increase results from merging the monitoring framework and policy class definitions into the application code, while the inserted security checks only have a minor influence on the file size. The applications in our benchmark exhibit significant differences in the total number of instructions as well as in the size of the application package. These differences are reflected in the execution times of the inliner. In most cases, the total instruction count has the largest impact on the runtime, as all instructions in the application code need to be scanned in order to identify invocations of security-relevant methods. For a few apps (e.g. Angry Birds), however, the runtime is dominated by rebuilding and compressing the application package file (which is essentially a zip archive). The evaluation also clearly reveals the difference in computing power between the laptop and the phone. While the inlining process takes considerably more time on the phone than on the laptop, we argue that this should not be a major concern as the inliner is only run once per application.

The runtime overhead introduced by the inline reference monitor is measured through micro-benchmarks (cf. Table 2.) We compare the execution time of single function calls in three different settings: the original code with no inlining as well as the inlined code with disabled and enabled policies (i.e. policy enforcement turned on or off) . Additionally, we present the overhead incurred for the case where policies are disabled. We list the average execution time for each function call. For the case where we enforce policies we prevent the execution of the respective function.

For all function calls the instrumentation adds a small runtime overhead due to additional code. However, when enabled policies prevent the particular function call, the control flow change leads to a smaller overall execution time. Therefore, it is incomparable to the other execution times, so that we compute the overhead only for the disabled policies. In either case, the incurred runtime overhead is negligible and does not adversely affect the application's performance.

4.2 Case Study Evaluation

The conceptual design of AppGuard focuses on flexibility and introduces a variety of possibilities to enhance Android's security features. In this section, we evaluate our framework in several case studies by applying different policies to real world apps from Google's application market Google Play [30]. As a disclaimer, we would like to point out that we use apps from the market for exemplary purposes only, without implications regarding their security unless we state this explicitly.

For our evaluation, we implemented 9 different policies. Five of them are designed to revoke critical Android platform permissions, in particular the Internet permission (`InternetPolicy`), access to camera and audio hardware (`CameraPolicy`, `AudioPolicy`), and permissions to read contacts and calendar entries (`ContactsPolicy`, `CalendarPolicy`). Furthermore, we introduce a complex policy that tracks possible fees incurred by untrusted applications (`CostPolicy`). The `HttpsRedirectPolicy`

Table 1: Inliner Evaluation: sizes of apk file, classes.dex, inlined classes.dex, diff. of dex file, # of total and changed instructions, inlining time on PC and phone.

App (Version)	**Size** [Kb]				**Instructions**		**Time** [sec]	
	Apk	**Dex**	**Inl**	**Diff**	**Total**	**Chg**	**PC**	**Phone**
Angry Birds (2.0.2)	15018	994	1038	+44	79311	100	6.5	43.4
Barcode Scanner (4.0)	508	352	397	+45	46337	31	1.8	4.1
Chess Free (1.55)	2240	517	561	+45	52615	71	3.2	7.9
Dropbox (2.1.1)	3252	869	913	+44	90334	86	1.9	14.1
Endomondo (7.0.2)	3263	1635	1680	+45	134452	88	2.6	23.0
Facebook (1.8.3)	4013	2695	2744	+48	224285	218	3.2	47.3
Instagram (1.0.3)	12901	3292	3337	+46	254032	137	4.2	66,4
Post mobil (1.3.1)	858	1015	1056	+41	84407	58	1.7	11.6
Shazam (3.9.0)	3904	2642	2690	+48	259644	221	2.8	47.5
Tiny Flashlight (4.7)	1287	485	531	+46	46878	109	1.8	7.3
Twitter (3.0.1)	2218	764	813	+48	105594	107	3.6	16.7
Wetter.com (1.3.1)	4296	958	1000	+43	89655	36	2.2	15.7
WhatsApp (2.7.3581)	5155	3182	3230	+48	437874	235	3.0	57.5

Table 2: Runtime comparison with micro-benchmarks for function calls in unmodified apps and inlined apps with policies disabled and enabled. The runtime overhead is presented for the inlined app with disabled policies.

Function Call	**Original App**	**Inlined App**		**Overhead**
		Pol. disabled	**Pol. enabled**	
`Socket-><init>()`	0.2879 ms	0.3022 ms	0.0248 ms	5.0%
`ContentResolver->query()`	10.484 ms	11.138 ms	0.1 ms	6.2%
`Camera->open()`	150.8 ms	152.36 ms	0.6 ms	1.0%

and `MediaStorePolicy` address security issues in third-party apps and the OS. Finally, the `ReflectionPolicy` described in section 3.4.1 monitors invocations of Java's Reflection API. In the following case studies, we highlight 6 of these policies and evaluate them in detail on real-world apps.

Our case studies focus on (a) the possibility to revoke standard Android permissions - which is arguably the feature Android users desire most. Additionally, it is possible to (b) enforce fine-grained permissions that are not supported by Android's existing permission system, and, (c) to enforce complex and stateful policies based on the current execution trace. Finally, our framework provides quick-fixes and mitigation for vulnerabilities both in (d) third-party apps and (e) the operating system.

(a) Revoking Android permissions
Many Android applications request more permissions than necessary for achieving the intended functionality. A prominent example is the Internet permission *android.permission.INTERNET*, which allows sending and receiving arbitrary data to and from the Internet. Although the majority of applications requests this permission, it is not required for the core functionality of an app in many cases. It is often used just for providing in-app advertisements. However, overly curious apps that, e.g., upload the user's entire contact list to their servers, and even trojan horses are recently reported on a regular basis. Unfortunately, users cannot simply add, revoke, or configure permissions dynamically at a fine-grained level. Instead, users have to decide at installation time whether they accept the installation of the app with the listed permissions or they reject them with the consequence that the app cannot be installed at all.

AppGuard overcomes this unsatisfactory all-or-nothing situation by giving users the chance to safely revoke permissions at any time at a fine-grained level. We aim at a "safe" revocation of permissions, so that applications with revoked permissions will not be terminated by a runtime exception. To this end, we carefully provide proper dummy return values instead of just blocking unsafe function calls [35]. We tested the revocation of permissions on several apps, of which we highlight two in the following.

Case study: Twitter

As an example for the revocation of permissions, we chose the official app of the popular micro-blogging service Twitter. It recently attracted attention in the media [43] for secretly uploading phone numbers and email addresses stored in the user's address book to the Twitter servers. While the app "officially" requests the permissions to access both Internet and the user's contact data, it did not indicate that this data would be copied off the phone. As a result of the public disclosure, the current version of the app now explicitly informs the user before uploading any personal information.

We can stop the Twitter app from leaking any private information by completely blocking access to the user's contact list. The contact data is used as part of Twitter's "Find friends" feature that makes friend suggestions to new users based on information from their address book. Since friends can also be added manually, AppGuard leverages the `ContactsPolicy` to protect the user's privacy at the cost of losing only minor convenience functionality. Actual policy enforcement is done by monitoring queries to the `ContentResolver`, which serves as a centralized access point to Android's various databases. Data is identified by a URI, which we examine to selectively block queries to the contact list by returning a mock result object. Our tests were carried out on an older version of the Twitter app, which was released prior to their fix.

Case study: Tiny Flashlight

The core functionality of the Tiny Flashlight app is to provide a flashlight, either using the camera's LED flash, or by turning the whole screen white. At installation time, the app requests the permissions to access the Internet and the camera. Manual analysis indicates that the Internet permission is only required to display online advertisements. However, in combination with the camera permission this could in principle be abused for spying purposes, which would be hard to detect without further detailed code or traffic analysis. AppGuard can block the Internet access of the app with the `InternetPolicy` (cf. section 3.1 and Fig. 2), which, in this particular case, has the effect of an ad-blocker. We monitor constructor calls of the various `Socket` classes, the `java.net.url.openConnection()` method as well as several other network I/O functions, and throw an `IOException` if access to the Internet is forbidden.

Apart from the Internet permission, users might not easily see why the camera permission is required for this app. Here, our analysis indicates that – depending on the actual smartphone hardware – the flashlight can in some cases be accessed directly, while in others only via the camera interface. Although requesting this permission seems to be benign for this app, our approach offers the possibility to revoke camera access. We enforce the `CameraPolicy` by monitoring the `android.hardware.Camera.open()` method. The policy simulates hardware without a camera by returning a null value. The Tiny Flashlight app gracefully handles the revocation of the camera permission by falling back to the screen-based flashlight solution.

(b) Enforcing fine-grained permissions

Besides the revocation of existing permissions, it is also possible to design fine-grained permissions that restrict the access of third-party apps. These permissions can add new restrictions to a functionality that is not yet limited by the current permission system and to a functionality that is already protected, but not in the desired way. Here, again, the Internet permission is a good example. From the user's point of view, most apps should only communicate with a limited set of servers.

The wetter.com app provides weather information and should only communicate with its servers to query weather information. The `InternetPolicy` of AppGuard provides fine grained Internet access enabling a consequent white-listing of web servers on a per-app basis. For this particular app we restrict the Internet access as illustrated in the first case study and extend it with regular-expression-based white-listing: `^(.+\.)?wetter\.com$`. Similar to the Tiny Flashlight app, no more advertisements are shown while the application's core functionality is preserved. The refined Internet policy can also be applied in a general setting as the white-listing can be configured in the management interface by choosing from a list of hosts the app has tried to connect to in the past.

(c) Enforcing complex and stateful policies

Stateless permissions, as discussed in the previous case studies, cannot be used to enforce policies that depend on the trace of the current execution. Using AppGuard it is also possible to implement complex stateful policies, e.g. to limit the number of text messages or phone calls to costly numbers, or to block the Internet access after sensitive information like contacts or calendar entries has been accessed.

The Post mobil app provided by the German postal service Deutsche Post offers, besides informative services, the possibility to buy stamps online via premium service calls or text messages. To limit the cost incurred by this application, it is necessary to track the number of previous calls. AppGuard tracks these numbers and provides the `CostPolicy` that limits the number of possible charges. We monitor the relevant function calls for sending text messages and for making phone calls, e.g. `android.telephony.SmsManager.sendTextMessage()`. In order to monitor the start of phone calls, it is necessary to track so-called *Intents*, Android's message format for inter- and intra-app communication. Intents contain two parts, an *action* to be performed and parameter data encoded as URI. For example, intents that start phone calls have the action `ACTION_CALL`. We track intents by monitoring intent dispatch methods like `android.app.Activity.startActivity(Intent)`.

(d) Quick-fixes for vulnerabilities in third-party apps
Our system can also fix vulnerabilities in third-party applications. As an example, some applications still transmit sensitive information over the Internet via the `http` protocol. Although most apps use encrypted `https` for the login procedures to web servers, there are still some applications that return to unencrypted `http` after successful login, thereby transmitting their authentication tokens in plain text over the Internet. Attackers could eavesdrop on the connection to impersonate the current user [36].

The Endomondo Sports Tracker uses the `https` protocol for the login procedure only, and returns to the `http` protocol afterwards, which transmits the unencrypted authentication token. As the Web server supports `https` for the whole session, the `HttpsRedirectPolicy` of AppGuard enforces the usage of `https` connections throughout the session (cf. Fig. 3), which protects the user's account and data from identity theft. Instead of opening an `http` connection, we open an `https` connection (cf. the monitored method invocations in the first case study). Depending on the monitored function, we either return the redirected `https` connection, or the content from the redirected connection.

(e) Mitigation for operating system vulnerabilities
We also found our tool useful to mitigate operating system vulnerabilities. As we cannot change the operating system itself, we instrument all applications with a global security policy to prevent exploits.

Case study: Access to photos without a permission
A recent example for an operating system vulnerability is the lack of protection of the user's photos on Android phones. Any application can access these photos on the phone without any permission check [10]. Together with the Internet permission, an app could copy all photos to arbitrary servers on the Internet. This was demonstrated by a proof-of-concept exploit that – disguised as an inconspicuous timer app – uploads the user's personal photos to a public photo sharing site.

Android stores photos in a central media store, that can be accessed via the `ContentResolver` object, similar to contact data in the first case study. Leveraging the `MediaStorePolicy`, we block access to the stored photos, successfully preventing the exploit.

Case study: Local cross-site scripting attack
Similar to the mitigation of the photo access bug, it is also possible to fix security vulnerabilities in core applications that cannot be inlined directly. The Android browser that comes with all devices is vulnerable to a local cross-site scripting attack [3] up to Android version 2.3.4.

If the Android browser receives `VIEW` intents from another app with an `http/https` URI, it opens a new browser window and loads the requested web site. Similarly, it also handles `VIEW` intents with a `javascript:` URI, however, up to Android version 2.3.4, the browser reuses the currently active window. Consequently, the JavaScript code given in the intent will be executed in the context of the current web site, which leads to a local cross-site scripting vulnerability.

This attack can be mitigated by disallowing this combination of intents. The `InternetPolicy` monitors `startActivity(Intent)` calls and throws an exception if the particular intent is not allowed. The same approach can be leveraged to preclude third-party apps with no Internet permission from using intents with an `http/https` URI to send data to arbitrary servers on the Internet.

4.3 Discussion

The presented framework solves a pressing security problem of the Android platform. Coarse-grained and static policies like the access control mechanism of Android open the door for silent privacy violations and trojan horses, as the user never sees what an application actually does with the requested permissions. Our fine-grained dynamic policies can, e.g., be used to distrust the app and only grant a permission once the user finds that the app does not perform as expected. The logging-based approach in our tool allows a user to see which API calls were denied, possibly with the value of significant parameters. Granting access to those calls that are deemed necessary with restrictions on parameters (like accessible host names) will eventually lead to a minimal set of permissions that fulfills the privacy and security needs of a user.

We demonstrated that our solution is practical, as the runtime overhead and the increase in package sizes are negligible. The actual runtime overhead obviously depends on the complexity of the policy. However, when a policy denies access, the program will in general take a different execution path that usually leads to shorter times. The user experience does not suffer from rewriting the application. In particular, we did not notice any delays using the rewritten app. The rewriting process proceeds fast even on the limited hardware of a mobile phone. The inlining time is already reasonable, but we still see a large potential for reducing this time with some optimizations.

We outline some challenges and future work in the following: We currently do not monitor any code outside of the `classes.dex` file, in particular we might miss code in native libraries accessed via Java's Native Interface (JNI), dynamically loaded classes (from external sources), and external programs accessed via inter-procedure calls.

Android programs are multi-threaded by default. Issues of thread safety could therefore arise in the monitor. While we do not yet offer policies that take the relative timing of method calls in different threads into account, we plan to extend our system to support *race-free policies* [13] in the future.

5 Related Work

Since the release of Android in 2008, researchers have worked on various security aspects of this operating system and proposed many security enhancements.

One line of work analyzed Android's permission based access control system. Barrera et al. [4] have conducted an empirical analysis of Android's permission system on 1,100 Android applications and suggested improvements to its granularity. In 2011, Felt et al. [25] analyzed the effectiveness of application permissions using case studies on Google Chrome extensions and Android apps.

The inflexible and coarse-grained permission system of Android inspired many researchers to propose extensions. In 2009, Ongtang et al. [40] extended the current permission system for inter-app communication in a system called *Saint*. Enck et al. [20] introduce a policy based system called *Kirin* to detect malware candidates at install time based on an app's permissions. In 2010, Ongtang et al. introduce a policy-based system called *Porscha* [39] for digital rights management on smartphones. Nauman et al. [38] present a modification of the Android software stack called *Apex* that enables dynamic permission revocations. Conti et al. [12] go one step further and integrate with *CRePE* a context-related policy enforcement mechanism to the Android software stack. Grace et al. [31] detect capability leaks on Android by analyzing phone images of different vendors. In contrast to our approach, none of these approaches combines a fine-grained stateful policy enforcement mechanism with the ability to deploy the system to unmodified stock Android phones.

Another open problem of the Android system is the lack of completeness of its documentation. Using automated testing techniques Felt et al. show that the mapping of permissions to API-calls is only insufficiently documented [24]. Even for honest developers it is quite difficult to implement apps according to the principle of *least privilege*. Their analysis showed that roughly one-third of the tested applications were over-privileged. Vidas et al. [46] assist Eclipse developers to follow the principle of least privilege when programming Android apps.

Further, some papers focus on problems arising from inter-app communication. Davi et al. [14] demonstrate privilege escalation attacks on Android. These attacks are possible when an app exposes permission protected functionality via an interface (intentionally or unintentionally) to an app without the permission. Felt et al. modify the Android framework to inspect inter-process communication and to mitigate this problem [26]. Dietz et al. [16] and Bugiel et al. [7] address privilege escalation attacks as well. In the latter paper, the authors show how their approach even detects Soundcomber, a Android trojan based on covert channels [44]. Bugiel et al. have recently extended their framework to additionally detect colluding apps that aim at an intentional privilege escalation [8].

The concept of inlined reference monitors has received considerable attention in the literature. It was first formalized by Erlingsson and Schneider in the development of the SASI/PoET/PSLang systems [23, 22], which implement IRM's for x86 assembly code and Java bytecode. Several other IRM implementations for Java followed. Polymer [6] is a IRM system based on edit automata, which supports composition of complex security policies from simple building blocks. The Java-MOP [11] system offers policy-writers a rich set of formal policy specification languages. SPoX [33] enforces declarative aspect-oriented security policies by rewriting Java bytecode. IRM systems have also been developed for other platforms. Mobile [34] is an extension to Microsoft's .NET Common Intermediate Language (CIL) that supports certified inlined reference monitoring. Finally, the S3MS.NET Run Time Monitor [15] enforces security policies expressed in a variety of policy languages for .NET desktop and mobile applications on Windows phones.

6 Conclusions

We have presented a practical approach to overcome Android's limitations regarding secure, user-driven permission management. The system is based on an inline reference monitor and can be deployed to all Android devices as it does not rely on modifying the firmware. Most prominently, the system can curb the pervasive overly curious behavior of Android apps. Apart from that, we are able to enforce complex stateful security policies and mitigate vulnerabilities of both third-party apps as well as the OS. Our experimental analysis demonstrates that the overhead of both space and runtime are negligible. Further, the case studies illustrate the prevention of several real-world attacks on Android vulnerabilities.

7 Acknowledgements

This work was supported by the German Ministry for Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) and both the initiative for excellence and the Emmy Noether program of the German federal government. Further, we would like to thank Bastian Könings for pointing us to interesting Android apps.

References

[1] Android.com: Philosophy and Goals | Android Open Source (2010), http://source.android.com/about/philosophy.html

[2] Android.com: Security and Permissions (2012), http://developer.android.com/guide/topics/security/security.html

[3] Backes, M., Gerling, S., von Styp-Rekowsky, P.: A Local Cross-Site Scripting Attack against Android Phones (2011), http://www.infsec.cs.uni-saarland.de/projects/android-vuln/android_xss.pdf

[4] Barrera, D., Kayacık, H.G., van Oorschot, P.C., Somayaji, A.: A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android. In: Proc. 17th ACM Conference on Computer and Communication Security (CCS 2010). pp. 73–84 (2010)

[5] Bauer, L., Ligatti, J., Walker, D.: A Language and System for Composing Security Policies. Tech. Rep. TR-699-04, Princeton University (January 2004)

[6] Bauer, L., Ligatti, J., Walker, D.: Composing security policies with polymer. In: Proc. ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 2005). pp. 305–314 (2005)

[7] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R.: XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks. Tech. Rep. TR-2011-04, Technische Universität Darmstadt - Cased (2011)

[8] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R., Shastry, B.: Towards Taming Privilege-Escalation Attacks on Android. In: Proc. 19th Annual Network and Distributed System Security Symposium (NDSS 2011) (2012)

[9] Chaudhuri, A., Fuchs, A., Foster, J.: SCanDroid: Automated Security Certification of Android Applications. Tech. Rep.

CS-TR-4991, University of Maryland, College Park (2009), http://www.cs.umd.edu/~avik/papers/scandroidascaa.pdf

[10] Chen, B.X., Bilton, N.: Et Tu, Google? Android Apps Can Also Secretly Copy Photos (2012), http://bits.blogs.nytimes.com/2012/03/01/android-photos/

[11] Chen, F., Roşu, G.: Java-MOP: A Monitoring Oriented Programming Environment for Java. In: Proc. 11th International Conference on Tools and Algorithms for the construction and analysis of systems (TACAS 2005). vol. 3440, pp. 546–550. Springer-Verlag (2005)

[12] Conti, M., Nguyen, V.T.N., Crispo, B.: CRePE: Context-Related Policy Enforcement for Android. In: Proc. 13th International Conference on Information Security (ISC 2010). pp. 331–345 (2010)

[13] Dam, M., Jacobs, B., Lundblad, A., Piessens, F.: Security Monitor Inlining and Certification for Multithreaded Java. Mathematical Structures in Computer Science (2011)

[14] Davi, L., Dmitrienko, A., Sadeghi, A.R., Winandy, M.: Privilege Escalation Attacks on Android. In: Proc. 13th International Conference on Information Security (ISC 2010). pp. 346–360 (2010)

[15] Desmet, L., Joosen, W., Massacci, F., Naliuka, K., Philippaerts, P., Piessens, F., Vanoverberghe, D.: The S3MS.NET Run Time Monitor. Electron. Notes Theor. Comput. Sci. 253(5), 153–159 (Dec 2009)

[16] Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., Wallach, D.S.: QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In: Proc. 20th Usenix Security Symposium (2011)

[17] von Eitzen, C.: Apple: Future iOS release will require user permission for apps to access address book (February 2012), http://h-online.com/-1435404

[18] Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In: Proc. 9th Usenix Symposium on Operating Systems Design and Implementation (OSDI 2010). pp. 393–407 (2010)

[19] Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A Study of Android Application Security. In: Proc. 20th Usenix Security Symposium (2011)

[20] Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. In: Proc. 16th ACM Conference on Computer and Communication Security (CCS 2009). pp. 235–245 (2009)

[21] Erlingsson, Ú.: The Inlined Reference Monitor Approach to Security Policy Enforcement. Ph.D. thesis, Cornell University (January 2004)

[22] Erlingsson, Ú., Schneider, F.B.: IRM Enforcement of Java Stack Inspection. In: Proc. 2002 IEEE Symposium on Security and Privacy (Oakland 2002). pp. 246–255 (2000)

[23] Erlingsson, U., Schneider, F.B.: SASI enforcement of security policies: a retrospective. In: Proc. of the 1999 workshop on New security paradigms (NSPW 1999). pp. 87–95 (2000)

[24] Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android Permissions Demystified. In: Proc. 18th ACM Conference on Computer and Communication Security (CCS 2011) (2011)

[25] Felt, A.P., Greenwood, K., Wagner, D.: The Effectiveness of Application Permissions. In: Proc. 2nd Usenix Conference on Web Application Development (WebApps 2011) (2011)

[26] Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission Re-Delegation: Attacks and Defenses. In: Proc. 20th Usenix Security Symposium. pp. Want to prevent permission re–delegation attacks. (2011)

[27] Gartner: Gartner Says Worldwide Smartphone Sales Soared in Fourth Quarter of 2011 With 47 Percent Growth, http://www.gartner.com/it/page.jsp?id=1924314

[28] Gibler, C., Crussel, J., Erickson, J., Chen, H.: AndroidLeaks: Detecting Privacy Leaks in Android Applications. Tech. Rep. CSE-2011-10, University of California Davis (2011)

[29] Gilbert, P., Chun, B.G., Cox, L.P., Jung, J.: Vision: Automated Security Validation of Mobile Apps at App Markets. In: Proc. 2nd International Workshop on Mobile Cloud Computing and Services (MCS 2011 (2011)

[30] Google Play (2012), https://play.google.com/store

[31] Grace, M., Zhou, Y., Wang, Z., Jiang, X.: Systematic Detection of Capability Leaks in Stock Android Smartphones. In: Proc. 19th Annual Network and Distributed System Security Symposium (NDSS 2011) (2012)

[32] Gruver, B.: Smali: A assembler/disassembler for Android's dex format, http://code.google.com/p/smali/

[33] Hamlen, K.W., Jones, M.: Aspect-oriented in-lined reference monitors. In: Proc. 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS 2008). pp. 11–20 (2008)

[34] Hamlen, K.W., Morrisett, G., Schneider, F.B.: Certified In-lined Reference Monitoring on .NET. In: Proc. 1st ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS 2006). pp. 7–16 (2006)

[35] Hornyack, P., Han, S., Jung, J., Schechter, S., Wetherall, D.: "These Aren't the Droids You're Looking For": Retrofitting Android to Protect Data from Imperious Applications. In: Proc. 18th ACM Conference on Computer and Communication Security (CCS 2011) (2011)

[36] Könings, B., Nickels, J., Schaub, F.: Catching AuthTokens in the Wild - The Insecurity of Google's ClientLogin Protocol. Tech. rep., Ulm University (2011), http://www.uni-ulm.de/in/mi/mi-mitarbeiter/koenings/catching-authtokens.html

[37] Ligatti, J., Bauer, L., Walker, D.: Edit Automata: Enforcement Mechanisms for Runtime Security Policies. International Journal of Information Security 4(1–2), 2–16 (2005)

[38] Nauman, M., Khan, S., Zhang, X.: Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In: Proc. 5th ACM Symposium on Information, Computer and Communication Security (ASIACCS 2010). pp. 328–332 (2010)

[39] Ongtang, M., Butler, K.R.B., McDaniel, P.D.: Porscha: policy oriented secure content handling in Android. In: Proc. 26th Annual Computer Security Applications Conference (ACSAC 2010). pp. 221–230 (2010)

[40] Ongtang, M., McLaughlin, S.E., Enck, W., McDaniel, P.: Semantically Rich Application-Centric Security in Android. In: Proc. 25th Annual Computer Security Applications Conference (ACSAC 2009). pp. 340–349 (2009)

[41] Portokalidis, G., Homburg, P., Anagnostakis, K., Bos, H.: Paranoid Andoird: Versatile Protection For Smartphones. In: Proc. 26th Annual Computer Security Applications Conference (ACSAC 2010). pp. 347–356 (2010)

[42] Rubin, A.: Google+ post on the Android ecosystem (2012), https://plus.google.com/u/0/112599748506977857728/posts/Btey7rJBaLF

[43] Sarno, D.: Twitter stores full iPhone contact list for 18 months, after scan (February 2012), http://articles.latimes.com/2012/feb/14/business/la-fi-tn-twitter-contacts-20120214

[44] Schlegel, R., Zhang, K., Zhou, X., Intwala, M., Kapadia, A., Wang, X.: Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In: Proc. 18th Annual Network and Distributed System Security Symposium (NDSS 2011). pp. 17–33 (2011)

[45] Schneider, F.B.: Enforceable Security Policies. ACM Transactions on Information and System Security 3(1), 30–50 (2000)

[46] Vidas, T., Christin, N., Cranor, L.F.: Curbing Android Permission Creep. In: Proc. Workshop on Web 2.0 Security and Privacy 2011 (W2SP 2011) (2011)

Original snippet

```
URL url = new URL(loc);
URLConnnection conn =
  url.openConnection();
```

After inlining

```
URL url = new URL(loc);
URLConnection conn;
try {
  MonitorInterface.checkConnection(url);
  conn = url.openConnection();
} catch (MonitorException e) {
  conn = (URLConnection) e.value();
}
```

Figure 4: Illustration of the call-site monitoring code for the security-relevant method `java.net.URL.openConnection()`.