AppGuard — Fine-grained Policy Enforcement for Untrusted Android Applications

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

Technischer Bericht Nr. A/02/2013

AppGuard – Fine-grained Policy Enforcement for Untrusted Android Applications

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

[1] Saarland University, Saarbrücken, Germany
[2] Max Planck Institute for Software Systems (MPI-SWS)

Abstract Android's success makes it a prominent target for malicious software. However, the user has very limited control over security-relevant operations. This work presents AppGuard, a powerful and flexible security system that overcomes these deficiencies. It enforces user-defined security policies on untrusted Android applications without requiring any changes to a smartphone's firmware, root access, or the like. Fine-grained and stateful security policies are expressed in a formal specification language, which also supports secrecy requirements. Our system offers complete mediation of security-relevant methods based on callee-site inline reference monitoring and supports widespread deployment. In the experimental analysis we demonstrate the removal of permissions for overly curious apps as well as how to defend against several recent real-world attacks on Android phones. Our technique exhibits very little space and runtime overhead. The utility of AppGuard has already been demonstrated by more than 1,000,000 downloads.

1 Introduction

The rapidly increasing number of mobile devices creates a vast potential for misuse. Mobile devices store a plethora of information about our personal lives, and their sensors, GPS, camera, or microphone – just to name a few – offer the ability to track us at all times. The always-online nature of mobile devices makes them a clear target for overly curious or maliciously spying apps and Trojan horses. Social network apps, for instance, were recently criticized for silently uploading the user's entire contacts onto external servers [18, 45]. While this behavior became publicly known, users are most often not even aware of what an app actually does with their data. Additionally, fixes for security vulnerabilities in the Android OS often take months until they are integrated into vendor-specific OSs. Between Google's fix with a public 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 [1]. Access to personal information has to be explicitly granted at install time: When installing an app a list of permissions is displayed, which have to be granted in order to install the app. Users can neither dynamically grant and revoke permissions at runtime, nor add restrictions according to their personal

needs. Further, users (and often even developers, cf. [24,28]) usually do not have enough information to judge whether a permission is indeed required to fulfill a certain task.

1.1 Contributions

To overcome the aforementioned limitations of Android's security system, we present a novel policy-based security framework for Android called AppGuard.

- AppGuard takes an untrusted app and user-defined security policies as input and embeds the security monitor into the untrusted app, thereby delivering a secured self-monitoring app.
- Security policies are formalized in an automata-based language and displayed to the user in a convenient graphical interface. Security policies may specify restrictions on method invocations as well as secrecy requirements.
- AppGuard is built upon a novel approach for callee-site inline reference monitoring (IRM). The fundamental idea is to redirect method calls to the embedded security monitor and check whether executing the call is allowed by the security policy. Technically, this is achieved by altering method references in the Dalvik VM. This approach does not require root access or changes to the underlying Android architecture and, therefore, supports widespread deployment as a stand-alone app. Furthermore, it can handle even Java reflection (cf. section 4.6) and dynamically loaded code.
- Secrecy requirements are enforced by storing the secret within the security monitor. Apps are just provided with a handle to that secret. This mechanism is general enough to enforce the confidentiality of data persistently stored on the device (e.g., address book entries or geolocation) as well as of dynamically received data (e.g., user-provided passwords or session tokens received in a single sign-on protocol). The monitor itself is protected against manipulation of its internal state and forceful extraction of stored secrets.
- We support fully-automatic on-the-phone instrumentation (no root required) of third-party apps and automatic updates of rewritten apps such that no app data is lost. Our system has been downloaded by about 1,000,000 users so far and will be soon released to the Samsung Apps market after an explicit invitation from Samsung.
- 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 revoked permissions of excessively curious apps, demonstrate complex policies that do not necessarily involve system calls, and prevent several recent real-world attacks on Android phones, both due to in-app and OS vulnerabilities. We finally show that for the vast majority of 25,000 real-world apps, our instrumentation does not break functionality, thus demonstrating the robustness of our approach.

1.2 Key Design Decisions & Closely Related Work

Researchers have proposed several approaches to overcome the limitations of Android's security system, most of which require modifications to the Android platform. While there is hope that Google will eventually introduce a more fine-grained security system, we decided to directly integrate the security monitor within the apps, thereby requiring no change to the Android platform. The major drawback of modifying the firmware and platform code is that it requires rooting the device, which may void the user's warranty and affect the system stability. Besides, there is no general Android system but a plethora of vendor-specific variants that would need to be supported and maintained across OS updates. Finally, laymen users typically lack the expertise to conduct firmware modifications, and, therefore, abstain from installing modified Android versions.

Aurasium [49], a recently proposed tool for enforcing security policies in Android apps, rewrites low-level function pointers of the libc library in order to intercept interactions between the app and the OS. A lot of the functionality that is protected by Android's permission system depends on such system calls and thus can be intercepted at this level. A limitation of this approach is that the parameters of the original Java requests need to be recovered from the system calls' low-level byte arrays in order to differentiate malicious requests from benign ones, which "is generally difficult to write and test" [49] and may break in the next version of Android at Google's discretion. Similarly, mock return values are difficult to inject at this low level. In contrast, we designed our system to intercept high-level Java calls, which allows for more flexible policies. In particular we are able to inject arbitrary mock return values, e.g. a proxy object that only gives access to certain data, in case of policy violations. Additionally, we are able to intercept security-relevant methods that do not depend on the libc library. As an example consider the policy that systematically replaces MD5, which is nowadays widely considered an insecure hashing algorithm, by SHA-1. Since the implementation of MD5 does not use any security-relevant functionality of the libc library, this policy cannot be expressed in Aurasium. Finally, it is worth to mention that both Aurasium and AppGuard offer only limited guarantees for apps incorporating native code. Aurasium can detect an app that tries to perform security-relevant operations directly from native code, under the assumption, however, that the code does not re-implement the libc functionality. Our approach can monitor Java methods invoked from native code, although it cannot monitor system calls from native code.

Jeon et al. [38] advocate to place the reference monitor into a separate application. Their approach removes all permissions from the monitored app, as all calls to sensitive functionality are done in the monitoring app. This is fail-safe by default as it prevents both reflection and native code from executing such functionality. However, it has some drawbacks: If a security policy depends on the state of the monitored app, this approach incurs high complexity and overhead as all relevant data must be marshaled to the monitor. Besides, the monitor may not yet be initialized when the app attempts to perform security-relevant operations. Finally, this approach does not follow the principle of least

Table 1. Comparison of Android IRM approaches

Feature	No Firmware Mod.	On Phone Instr./Updates	Monitor	Native Methods	Reflection	Policy Lang.	Data Secrecy	Parametric Joinpoints	Runtime Overhead
Aurasium [49]	✓	–	I	◐	✓	–	–	–	14-35%
Dr. Android [38]	✓	–	E	◐	◐	–	–	–	10-50%
I-ARM-Droid [15]	✓	–	I	–	◐	–	–	✓	16%
AppGuard	✓	✓	I	◐	✓	✓	✓	✓	1-21%

privilege since the monitor must have the permissions of all monitored apps. We propose a different approach: Although the security policies are specified and stored within AppGuard, the policy enforcement mechanism is directly integrated and performed within the monitored apps. The policy configuration file is passed as input to the security monitor embedded in each app, thereby enabling dynamic policy configuration updates. This approach does not involve any inter-procedure calls and obeys the principle of least privilege, as AppGuard requires no special permissions.

Table 1 compares AppGuard with the most relevant related work that does not modify the firmware. Up to now, no other system can instrument an app and update apps directly on the phone. Dr. Android has an external monitor accessed via IPC; the other three approaches use internal monitors. Aurasium can monitor security-relevant native methods, Dr. Android only removes their permissions, which may lead to unexpected program termination, whereas our tool can prevent calls to sensitive Java APIs from native code. Both Aurasium and AppGuard handle reflection; Dr. Android does not handle it; I-ARM-Droid handles reflection and detects native calls. In case of native calls, however, the rewriting process is only aborted and the user notified. AppGuard is the only system that offers a high-level specification language for policies and supports hiding of secret data from, e.g., untrusted components in the monitored app. Both Aurasium and Dr. Android only support a fixed set of joinpoints where a security policy can be attached to. In contrast, I-ARM-Droid and AppGuard can instrument calls to any Java method. The last column displays the runtime

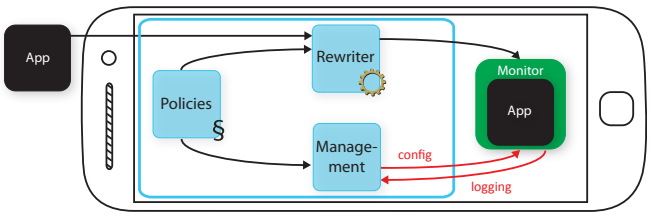

Figure 1. Schematics of AppGuard

overhead incurred in micro-benchmarks as reported by the respective authors. AppGuard is competitive in terms of runtime overhead with respect to concurrent efforts. For reasons of readability we postpone the discussion of further, less closely related work to section 6.

2 AppGuard

Runtime policy enforcement for third-party apps is challenging on unmodified Android systems. Android's security concept strictly isolates different apps installed on the same device. Communication between apps is only possible via Android's inter-process communication (IPC) mechanism. However, such communication requires both parties to cooperate, rendering this channel unsuitable for a generic runtime monitor. Furthermore, apps cannot gain elevated privileges that allow for observing the behavior of other apps.

AppGuard tackles this problem by following an approach pioneered by Erlingsson and Schneider [22] called *inline reference monitor* (IRM). The basic idea is to rewrite an untrusted app such that the code that monitors the app 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 app's 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 app. To actually enforce a *security policy*, the monitor controls the execution of the app by suppressing or altering calls to security-relevant methods, or even 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 [46]. Ligatti et al. differentiate security automata by their ability to enforce policies by manipulating the trace of the program [40]. Some IRM systems [16,22] implement truncation automata, which can only terminate the program if it deviates from the policy. However, this is often undesirable in practice. *Edit automata* [40] 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 mock value, thus 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.* On top of user-defined and app-specific policies (see section 3), AppGuard provides various generic security policies that govern access to platform API methods which 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 mapping from API methods to permissions from Song et al. [24].

2. *The program rewriter.* Android apps run within a custom register-based Java VM called *Dalvik*. Our rewriter manipulates Dalvik executable (`dex`) bytecode of untrusted Android apps and embeds the security monitor into the untrusted app. The references of the Dalvik VM are altered so as to redirect the method calls to the security monitor.
3. *A management component.* AppGuard offers a graphical user interface that allows the user to set individual policy configurations on a per-app basis. In particular, policies can be turned on or off and parameterized. In addition, the management component keeps a detailed log of all security-relevant events, enabling the user to monitor the behavior of an app.

3 Policies

Our policy language is a direct encoding of policy automata and allows us to express constraints on the execution of method calls as well as on the processing of confidential data. Intuitively, we represent security policies by edit automata [40], which we augment so as to specify whether or not the result of a method call is confidential. The method calls specified in the automaton are monitored and subject to the constraints imposed therein, while other method calls may be freely executed. Confidential data may only be processed as indicated by the edit automata, any other operation is forbidden: this allows us to uniformly reason about the confidentiality of persistent data (e.g., address book entries or geolocation) and dynamically received data (e.g., user-provided passwords or session tokens received in a single sign-on protocol). Even though there is no taint tracking outside the monitor, this mechanism allows information flow control policies. Data can be labeled as confidential and declassification policies can make the result of a function public even if its parameters contain confidential data (cf. Figure 4). As long as no equality tests are permitted, also implicit information flows are prevented. Otherwise, the result would be essentially declassified and thus may leak some information. A typical confidentiality policy, however, will be of the form "send this data only to this address" and, thus, not permit boolean tests.

Using our policy language, we can specify, for instance, an upper bound on the number of times the `android.telephony.SmsManager.sendTextMessage()` method may be called by an app or declare that the password provided by the user in a certain input box can only be sent to a specific IP address in encrypted form and cannot be processed by the app in any other way.

Our policy language is based on the SPoX policy language introduced by Hamlen et al. [34, 35], which was originally designed for specifying declarative aspect-oriented security policies. SPoX defines a security automaton where nodes correspond to security states and edges are labeled with conditions under which a certain method call is allowed. We extended the language to express the confidentiality of values and on-the-fly replacement of method calls (e.g., we can require that `http` connections are systematically replaced by `https` connections). We refer to the extended version as Security-Oriented SPoX (SOSPoX).

3.1 SOSPoX

The syntax of SOSPoX is reviewed in Table 2 (for the syntax of SPoX, we refer to [35]). We extended the language to support both labeling return values of function calls as secret, as well as to support rewriting existing function calls with new function calls.

The new **nodes** statement in Table 2 is used to either label a return value as secret, or to remove such a secret label. The return value is referenced via the object identifier `oid`, which, in order to change the secrecy label, is either added or removed to the list `SI` that keeps track of all object identifiers that are labeled as secret.

The pointcut syntax of SOSPoX is shown in Table 3. The **call**-statement has been extended by two additional identifiers: `oid` is an object identifier that refers to the object returned by a method call (`oid` is an equivalence class for all objects returned by a particular type of method call), whereas `cid` is an identifier for a particular method call.

In order to react gracefully to a policy violation (e.g., to prevent an app from crashing), most cases require a change in the control flow of a program. This is reflected by security automata that provide the transformation capabilities of an edit automaton. To that end, we introduce the **rewrite**-statement that allows to specify how a method call is replaced by new method calls: the newly introduced methods define the alternative control flow of the program that is to be executed in case of a policy violation. Notice that predicates that are applied in conjunction with new methods (second argument of the **rewrite**-statement) are to be considered as conditions that need to be fulfilled by the concrete implementation of the policy. We modified the **argval**- and the **argtype**-statement to include also the call-identifiers such that it is possible to express conditions related to the arguments of arbitrary method calls inside rewrite statements. The `order` pointcut allows to specify the order in which a series of new method calls introduced by the rewrite statement are supposed to be executed.

3.2 Policy Examples

In the following we present some policy examples, first by their automaton and afterwards how they can be expressed in SOSPoX. The automaton in Fig. 2 controls the usage of contacts: the app is authorized to access the contact list so as to publish a hash of each contact's phone number, but the contacts themselves are kept secret. This functionality is crucial to implement messaging apps (e.g., WhatsApp), which have to check for the presence of a certain entry in the contact list, in a privacy-preserving manner. The first edge declares that the cursor `id1` returned by the method call `CR.query(Contacts,PhoneNumber)` is to be regarded as a secret. A node is defined as a set of bindings between variables and values. In particular, each node maps a special variable `SI` to the list of secret identifiers. For instance, the second node maps `SI` to [`id1`].

The looping edge in the second node allows for processing the `id1` object via the `id1.moveToNext()` method call, which moves the cursor forward by one

Table 2. SOSPoX policy syntax. Additions to SPoX are marked by †.

	$n \in \mathbb{Z}$	**integers**
	$c \in C$	**class names**
	$sv \in SV$	**state variables**
†	$oid \in ID,\ O_{ID} \subseteq SV$	**object identifier**
	$x ::= c \mid oid$	**callee identifier**
†	$cid \in C_{ID}$	**call identifier**
	$iv \in IV$	**iteration vars**
	$en \in EN$	**edge names**
	$pn \in PCN$	**pointcut names**
	$pol ::= np^* sd^* e^*$	**policies**
	$np ::=$ (pointcut name = "pn" pcd)	**named pointcuts**
	$sd ::=$ (state name = "sv")	**state declarations**
	$e ::=$	**edges**
	(edge name = "en" [after] pcd ep^*)	edgesets
	| (forall "iv" from a_1 to a_2 e^*)	iteration
	$ep ::=$	**edge endpoints**
	| (nodes "sv" a_1, a_2)	state transitions
	| (nodes "sv" $a_1, \#$)	policy violations
†	| (nodes oid [+,-])	setting secrecy-level of object identifiers
	$a ::= a_1 + a_2 \mid a_1 - a_2 \mid b$	**arithmethic**
	$b ::= n \mid iv \mid b_1 * b_2 \mid b_1 / b_2 \mid (a)$	

position. Notice that the security state does not change, i.e., the cursor is still secret. Furthermore, the boolean value returned by the method call is not marked as secret and, thus, is freely accessible by the app. The outgoing edge in the second node permits reading the phone number at the cursor into `id3`, which is also marked as secret. In the third node, the cursor can either be moved forward via the left looping edge, the phone number at the current position of the cursor can be requested via the right looping edge, or the phone number can be encoded into a secret byte array `id4` via the method call `id4=call byte[] id3.getBytes()`. Notice that for the looping edges the security state does not change. Finally, the loop at the fourth node enables the hashing of `id4`. As the resulting hash `id5` is not marked as a secret, it can be freely processed and, e.g., sent over the Internet.

As previously mentioned, the policy language further supports on-the-fly replacement of method calls. The automaton in Fig. 3, for instance, declares that secret data can only be sent using `https` and not `http`. Whenever the method `Net.Connect` is called with secret data (`issec(Data)`) and the `http` parameter, this method call is replaced by the corresponding call using `https`.

We exemplify the syntax of SOSPoX in Fig. 4 and Fig. 5 by providing the actual policies that created the security automata presented in Fig. 2 and Fig. 3, respectively. Notice that the edge labels in the earlier automaton were simplified for demonstration. The full version of edge labels is provided in the policies in Fig. 4 and Fig. 5, respectively.

Table 3. SOSPoX pointcut syntax. Additions and modifications to SPoX are marked by † and ‡, respectively.

	$re \in RE$	**regular expressions**
	$md \in MD$	**method names**
	$fd \in FD$	**field names**
	$pcd ::=$	**pointcuts**
‡	$\mid (cid : oid = \texttt{call}\ mo^*\ rt\ x.md)$	method calls
‡	$\mid (\texttt{argval}\ cid\ n\ vp)$	stack args (values)
‡	$\mid (\texttt{argtyp}\ cid\ n\ c)$	stack args (types)
	$\mid (\texttt{and}\ pcd^*)$	conjunction
	$\mid (\texttt{or}\ pcd^*)$	disjunction
	$\mid (\texttt{not}\ pcd)$	negation
†	$\mid (\texttt{rewrite}\ pcd\ pcd)$	rewriting
†	$\mid (\texttt{order}\ cid^*)$	call order
	$mo ::= \texttt{public} \mid \texttt{private} \mid \ldots$	**modifiers**
	$rt ::= c \mid \texttt{void} \mid \ldots$	**return types**
	$vp ::= (\texttt{true})$	**value predicates**
†	$\mid (\texttt{secret})$	secrecy predicate
	$\mid (\texttt{isnull})$	object predicates
	$\mid (\texttt{inteq}\ n) \mid ((\texttt{intne}\ n)$	integer predicates
	$\mid (\texttt{intle}\ n) \mid ((\texttt{intge}\ n)$	
	$\mid (\texttt{intlt}\ n) \mid ((\texttt{intgt}\ n)$	
	$\mid (\texttt{streq}\ re)$	string equality
†	$\mid (\texttt{argeq}\ cid\ n)$	argument equality

4 Architecture

AppGuard[3] is a stand-alone Android app written in Java and C that comprises about 9000 lines of code. It builds upon the *dexlib* library, which is part of the *smali* disassembler for Android by Ben Gruver [33], for manipulating `dex` files. The size of the app package is roughly 2 Mb.

4.1 Instrumentation

A key aspect of any IRM system is to instrument the target app such that the control flow of the program is diverted to the security monitor whenever a security-relevant method is about to be invoked. There are two strategies for passing control to the monitor: Either at the call-site in the app 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 easily identified and less in number [5]. Furthermore, callee-site rewriting can handle obfuscated apps as it does not require to "understand" the untrusted code. Unfortunately, in our setting, standard callee-site rewriting is not feasible for almost all security-relevant methods, as they are defined in Android system libraries, which cannot be modified.

[3] http://www.srt-appguard.com/en/

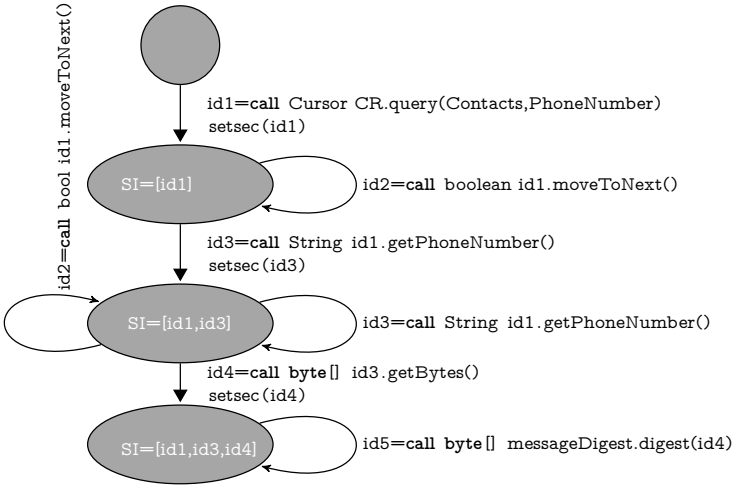

Figure 2. Security automaton for reading contacts

```
rewrite
(call Net.Connect(Scheme,URL,Data) && Scheme==HTTP && issec(Data))
(call Net.Connect(Scheme,URL,Data) && Scheme==HTTPS)
```

Figure 3. Security automaton for replacing `http` connections with `https` connections (`Net.Connect` is a shortcut for `Ljava/net/URL;->openConnection()`).

In order to achieve the same effect as callee-site rewriting, AppGuard employs a novel dynamic call-interposition approach [47]. This approach diverts calls to security-relevant methods to functions in the monitor (called *guards*) that perform a security check. In order to divert the control flow we replace the reference to a method's bytecode in the VM's internal representation (e.g., a virtual method table) with the reference to our security guard. The security guards reside in an external library that is dynamically loaded on app startup. Therefore, we do not need to reinstrument the app when a security policy is modified. Additionally, we store the original reference in order to access the original function later on, e.g., in case the security check grants the permission to execute the security-critical method. This procedure also reduces the risk of accidentally introducing infinite loops by a policy, since we usually call the original method.

With this approach, invocations of security-relevant methods do *not* need to be rewritten statically. Instead, we use Java Native Interface (JNI) calls at runtime to replace the references to each of the monitored functions. More precisely, we call the JNI method `GetMethodID()` which takes a method's signature, and

```
(edge name="SecretCursor"
   (and (cid1:id1=call "Cursor ContentResolver.query(URI,Content)")
        (argval cid1 1 (streq "Contacts"))
        (argval cid1 2 (streq "PhoneNumber")))
        (nodes id1 +))
(edge name="MoveToNode"
   (cid2:id2=call "boolean id1.moveToNext()))
(edge name="SecretPhoneNumber"
   (cid3:id3=call "String id1.getPhoneNumber()")
   (nodes id3 +))
(edge name="MoveToNode2"
   (cid4:id2=call "boolean id1.moveToNext()))
(edge name="SecretPhoneNumber2"
   (cid5:id3=call "String id1.getPhoneNumber()"))
(edge name="SecretByte"
   (cid6:id4=call "byte[] id3.getBytes()")
   (nodes id4 +))
(edge name="Declassification"
   (cid7:id5=call "byte[] MessageDigest.digest(id4)"))
```

Figure 4. Policy for enforcing secrecy of Contacts

```
(edge name="EnforceHTTPS"
   (rewrite (and (cid1:id1=call "Net.Connect(Scheme,URL,Data)")
                 (argval cid1 1 (streq http))
                   (argval cid2 3 (secret )))
                (and (cid2:id2 = call "Net.Connect(Scheme,URL,Data)")
                     (argval cid2 1 (streq https )))))
```

Figure 5. Policy for enforcing `https` for secret data

returns a pointer to the internal data structure describing that method. This data structure contains a reference to the bytecode instructions associated with the method, as well as metadata such as the method's argument types or the number of registers. In order to redirect the control flow to our guard method, we overwrite the reference to the instructions such that it points to the instructions of the security guard's method instead. Additionally, we adjust the intercepted method's metadata (e.g., number of registers) to be compatible with the guard method's code. This approach works both for pure Java methods and methods with a native implementation.

Figure 6 illustrates how to redirect a method call using the functionality provided by our instrumentation library. Calling `Instrumentation.replaceMethod()` replaces the instruction reference of method `foo()` of class `com.test.A` with the reference to the instructions of method `bar()` of class `com.test.B`. It returns the original reference, which we store in a variable `A_foo`. Calling `A.foo()` will now invoke `B.bar()` instead. The original method can still be invoked by `Instrumentation.callOriginalMethod(A_foo)`. Note that the handle `A_foo` will be a secret of the

```
public class Main {
    public static void main(String[] args) {
        A.foo(); // calls A.foo()
        MethodHandle A_foo = Instrumentation.replaceMethod(
            "Lcom/test/A;->foo()", "Lcom/test/B;->bar()");
        A.foo(); // calls B.bar()
        Instrumentation.callOriginalMethod(A_foo); // calls A.foo()
}}
```

Figure 6. Example illustrating the functionality of the instrumentation library

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

security monitor in practice. Therefore the original method can no longer be invoked directly by the instrumented app.

4.2 Policies

We transform the high-level policy descriptions in SOSPoX into their concrete Java counterparts. For each policy, we generate a Java class which declares security-relevant method signatures and corresponding guard methods. We use a custom method annotation `MapSignatures` to map guard methods to a set of method signatures. Security state variables are stored in instance fields of the respective policy class, such that the values of these variables are preserved across guard method invocations.

Consider Fig. 7 as a basic example. This policy implementation controls access to the `openConnection()` method in the `java.net.URL` class and only allows connections to the host "wetter.com". The guard method has access to the arguments of the original method call by declaring a compatible list of parameters. In the example, the guard method uses the `url` parameter to decide whether a connection should be allowed. If allowed, the guard method will simply return, indicating that the original method call should proceed. If the connection is not allowed, an exception is thrown. The guard method throws an `IOException`, which bubbles up to the surrounding app code, imitating the behavior of the original `URL->openConnection()` method in case a connection error occurs.

As a second example, consider the policy presented in Fig. 3 that intercepts `http` connections and relays them to encrypted `https`. Fig. 8 presents an excerpt from the corresponding implementation. After calling the original method with the new arguments, the guard method needs to return an alternative return

value to the app code. To this end, it throws a special `MonitorException` that is caught by a handler in the monitor and returned to the app.

The guard methods declared in the policies can not be used directly as call-diversion targets by our instrumentation library. In the following we will explain why we generate a utility class called `MonitorInterface`, which serves as a bridge between app code and security guards. For each security-relevant method specified in the policies, we generate a static trampoline method in the `MonitorInterface` class (cf. Fig. 9). The purpose of this trampoline method is fourfold: First, the trampoline method is always signature-compatible[4] to the security-relevant method (including the instance object for virtual calls, which is passed as the first method argument, if available.) A compatible method signature is required to successfully divert the control flow using our dynamic instrumentation approach. Second, guard methods of different policies may be defined for a single security-relevant method. Thus, the trampoline method invokes all guards associated with this method signature. Third, the trampoline method contains a try/catch block around the calls to the security guards. This block enables guard methods to pass an alternative return value back to the app via a `MonitorException`[5] (cf. Fig. 8, a viable alternative would be to simply return a value instead of using Exceptions.) Fourth, if none of the guard methods throws an exception, the trampoline method invokes the original function and returns its return value, if available.

The generated policy classes, the `MonitorInterface` class, and the instrumentation library are stored into a monitor package, which is dynamically loaded into the target app at runtime.

4.3 Rewriter

The task of the rewriter component is to insert code into the target app, which dynamically loads the monitor package into the app's virtual machine. To ensure instrumentation of security-sensitive methods before their execution, we

[4] For technical reasons we use static methods instead of instance methods.

[5] Policies can only throw checked exceptions that are handled by the surrounding code or a MonitorException, which is handled by the monitor. The goal is to keep the program running even in case of a policy violation.

```
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 8. Example policy that redirects `http` connections to `https`.

```
public class MonitorInterface {
  public static URLConnection java_net_URL___openConnection(URL _this)
      throws Exception {
    try {
      InternetPolicy.instance.checkConnection(_this);
      HttpsRedirectPolicy.instance.checkConnection(_this);
    } catch (MonitorException e) {
      return (URLConnection) e.getValue();
    }
    return (URLConnection) Instrumentation.callOriginalMethod(
      java_net_URL___openConnection, _this);
}}
```

Figure 9. Example of a trampoline method in the MonitorInterface

create an application class that becomes the superclass of the existing application class[6]. Our new class contains a static initializer, which becomes the very first code executed upon app startup. The initializer uses a custom class loader to load our monitor package. Afterwards, it calls an initializer method in the monitor that uses the instrumentation library to rewrite the method references.

4.4 Separation of Secrets

Policies in our system can specify that the return values of certain functions are to be kept secret. In order to prevent an app from leaking secret values, we control access to these secrets. To this end, the monitor intercepts all calls to methods that the policy annotates as "secret-carrying", i.e. methods that can produce secret output or receive secret input. Whenever the invocation of such a method produces a new secret output, the monitor returns a dummy value, which serves as a reference to the secret for further processing. If such a secret reference is passed to a method that supports secret parameters, the trampoline method invokes the original method with the corresponding secret instead and returns either the actual result or a new secret reference, in case the return value was marked as secret in the policy. The dummy reference values do not contain any information about the secret itself and are thus innocuous if processed by any method that is not annotated in the policy.

4.5 Management

The management component of AppGuard monitors the behavior of instrumented apps and allows to configure policies at runtime. The policy configuration is provided to the instrumented app as a world-readable file. Its location is hardcoded into the monitor code during the rewriting process. This is motivated by the fact that invocations of security-relevant methods can occur before

[6] In case no application class exists, we register our class as the application class.

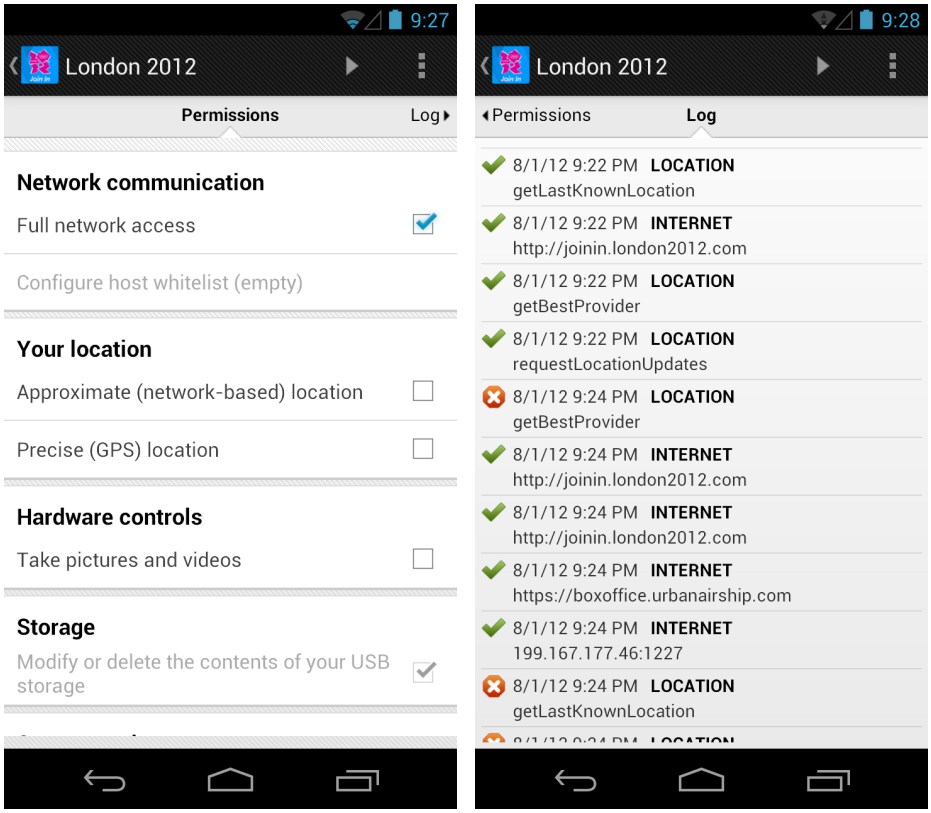

Figure 10. Screenshot of our management app. The left shows permission revocation policies, the right an extract of the event log.

the management app is fully initialized and able to react on Android IPC. The management component provides a log of all security-relevant method invocations for each app, which enables the user to make informed decisions about the current policy configuration. We report these invocations to the management app using 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. A screenshot of a policy configuration and a log are shown in Fig. 10.

4.6 Monitor Protection

In our system, the inlined monitor is part of the monitored app. A malicious app might try to circumvent the monitor by tampering with its internal state. Furthermore, an app could try to subvert secrecy policies by directly extracting stored secrets from the monitor. Since the monitor package containing secret data and pointers to the original methods is unknown at compile time

and due to strong typing, a malicious app would need to rely on reflection to access the monitor. To thwart such attacks, we implement a `ReflectionPolicy` that intercepts function calls to the Reflection API. In particular, we monitor operations that access Java classes and fields like `java.lang.Class->forName()` or `java.lang.Class->getField()` and prevent thereby effectively the access to the monitor package.

4.7 Deployment

On unmodified Android systems, app sandboxing prevents direct modifications of the code of other apps installed on the device. AppGuard leverages the fact that the app packages of installed third-party apps are stored at a world-readable location in the filesystem. This allows to inline the monitor into any app installed on the device by processing the corresponding `apk` file. In the end, AppGuard produces a self-monitoring app package that replaces the original version. Since stock Android does not allow automatic (un)installation of other apps, the user is prompted to confirm both the removal of the original app as well as the installation of the instrumented app. Moreover, we ask the user to enable the OS-option "Unknown sources: Allow installation of apps from sources other than the Play Store". Due to these two user interactions, no root privileges are required for AppGuard.

All Android apps 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.

Finally, due to the different signature, instrumented apps would no longer receive automatic updates, which may negatively impact device security. Therefore, AppGuard assumes the role of the Play Store app and checks for updates of instrumented apps. If a new version is found, AppGuard prompts to download the app package, instrument it, and replace the existing version of the app.

5 Experimental Evaluation

In this section we present the results of our experimental evaluation. It reports robustness and performance results of our instrumentation and evaluates the effectiveness of AppGuard in different case studies. For the evaluation we used a Google Galaxy Nexus smartphone with Android 4.1.2, a dual-core 1.2 GHz ARM CPU from Texas Instruments (OMAP 4460), and 1GB RAM. The off-the-phone evaluation was conducted on a notebook with an Intel Core i5-2520M CPU (2.5 GHz, two cores, hyper-threading) and 8GB RAM.

Table 4. Robustness of rewriting and monitoring

App Market	Apps	Stable	Dex verified	Stable Instr.
Google Play	9508	8783	9508 (100%)	8744 (99.6%)
SlideMe	15974	14590	15974 (100%)	14469 (99.1%)
Total	25482	23373	25482 (100%)	23213 (99.3%)

5.1 Robustness

To evaluate the robustness of our approach, we tested AppGuard on more than 25,000 apps from two different app markets and report the results in Table 4. The stability of the original apps is tested using the UI/Application Exerciser Monkey provided by the Android framework with a random seed and 1000 injected events (third column). To evaluate the robustness of the rewriting process we check the validity of the generated `dex` file (fourth column) and test the stability of the instrumented app using the UI Monkey with the random seed (fifth column). Note that we only consider the stability of instrumented apps where the original version did not crash.

The reported numbers indicate a very high reliability of the instrumentation process: we found no illegal `dex` file and over 99% of the stable apps were also stable after the instrumentation. The majority of the remaining 1% does not handle checked exceptions gracefully (e.g. IOException), which may be thrown by AppGuard when suppressing a function call. This bad coding style is not found in popular apps. Other apps terminate when they detect a different app signature. In rare cases, the mock values returned by suppressed function calls violate an invariant of the program. Note, however, that our test with the UI Monkey does not check for semantic equivalence.

5.2 Performance

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

Table 5 provides an overview of our performance evaluation for the rewriting process. We tested AppGuard with 15 apps and list the following results for each of the apps: size of the original app package (Apk), size of the `classes.dex` file, and the duration of the rewriting process both on the laptop and smartphone (PC and Phone, respectively).

The size of the `classes.dex` file increases on average by approximately 3.7 Kb. This increase results from merging code that loads the monitor package into the app. Since we perform callee-site rewriting and load the our external policies dynamically, we only have this static and no proportional increase of the original dex file.

For a few apps (e.g. Angry Birds) the instrumentation time is dominated by re-building and compressing the app package file (which is essentially a zip

Table 5. Sizes of apk and dex files with rewriting time on PC and phone.

App (Version)	Size [Kb]		Time [sec]	
	Apk	Dex	PC	Phone
Angry Birds (2.0.2)	15018	994	5.8	39.3
APG (1.0.8)	1064	1718	0.7	10.1
Barcode Scanner (4.0)	508	352	0.1	2.6
Chess Free (1.55)	2240	517	0.3	4.2
Dropbox (2.1.1)	3252	869	0.5	10.2
Endomondo (7.0.2)	3263	1635	0.7	16.6
Facebook (1.8.3)	4013	2695	1.2	26.4
Instagram (1.0.3)	12901	3292	3.0	44.3
Post mobil (1.3.1)	858	1015	0.2	5.8
Shazam (3.9.0)	3904	2642	1.2	26.1
Tiny Flashlight (4.7)	1287	485	0.1	2.9
Twitter (3.0.1)	2218	764	0.3	8.9
Wetter.com (1.3.1)	4296	958	0.4	10.7
WhatsApp (2.7.3581)	5155	3182	0.8	27.7
Yuilop (1.4.2)	4879	1615	0.8	19.7

Table 6. Runtime comparison with micro-benchmarks for normal function calls and guarded function calls with policies disabled as well as the introduced runtime overhead.

Function Call	Original Call	Guarded Call	Overhead
`Socket-><init>()`	0.0186 ms	0.0212 ms	21.4%
`ContentResolver->query()`	19.5229 ms	19.4987 ms	0.8%
`Camera->open()`	74.498 ms	79.476 ms	6.4%

archive). The evaluation also clearly reveals the difference in computing power between the laptop and the phone. While the rewriting process takes considerably more time on the phone than on the laptop, we argue that this should not be a major concern as the rewriter is only run once per app.

The runtime overhead introduced by the inline reference monitor is measured through micro-benchmarks (cf. Table 6 .) We compare the execution time of single function calls in three different settings: the original code with no instrumentation, the instrumented code with disabled policies (i.e. policy enforcement turned off.), and the incurred overhead. We list the average execution time for each function call.

For all function calls the instrumentation adds a small runtime overhead due to additional code. If we enabled policies, the changed control flow usually leads to shorter execution times and renders them incomparable. Even with disabled policies the incurred runtime overhead is negligible and does not adversely affect the app's performance.

5.3 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 on several case studies by applying different policies to real world apps from Google's app 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` and `MediaStorePolicy` address security issues in third-party apps and the OS. Finally, the `ReflectionPolicy` described in section 4.6 monitors invocations of Java's Reflection API and an app-specific policy. In the following case studies, we highlight 7 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. 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. Our framework provides quick-fixes and mitigation for vulnerabilities both in (d) third-party apps and (e) the operating system[7]. Finally, we present a general security policy that is completely independent of Android's permission 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 apps request this permission, it is not required for the core functionality of an app in many cases. Instead, it is often just used for providing in-app advertisements. At the same time, 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

[7] By providing policy recommendations based on a crowdsourcing approach, even laymen users can enforce complex policies (e.g. to fix OS vulnerabilities)

permissions will not be terminated by a runtime exception. To this end, we carefully provide proper mock return values instead of just blocking unsafe function calls [37]. 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 attracted attention in the media [45] 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. The 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 4.2 and Fig. 7), 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 Internet access with regular-expression-based white-listing: `^(.+\.)?wetter\.com$`. Similar to the Tiny Flashlight app, no more advertisements are shown while the app's core functionality is preserved. White-listing can be configured in the management interface by selecting from a list of hosts the app has attempted to connect to in the past.

(c) Enforcing complex and stateful policies. Using AppGuard it is also possible to implement complex stateful policies, e.g. to limit the number of text messages or phone calls to premium numbers, or to block Internet access after sensitive information like contacts or calendar entries has been accessed.

The Post mobil app provided by the German postal service offers the possibility to buy stamps online via premium service calls or text messages. To limit cost incurred by this app, 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 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. 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 user [39].

The Endomondo Sports Tracker uses the `https` protocol for the login procedure only and returns to the `http` protocol afterwards, thereby leaking 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. 8), which protects the user's account and data from identity theft. 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 permission. A recent example of an operating system vulnerability is the lack of protection of user photos on Android phones. Any app 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 [31] – 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, thereby 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 instrumented 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 with a `javascript:` URI, they are loaded in 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.

(f) Enforcing secure passphrases. APG is a public key encryption tool for Android that is capable of encrypting/decrypting files and emails via OpenPGP. It also offers the option to generate the key pairs used for encryption. While the tool prevents the user from generating keys without a passphrase, it does allow any non-empty passphrase to be used. AppGuard is able to enforce a policy that puts security requirements on the passphrase, such as having a minimum length or being comprised of a certain set of characters. To this end, we intercept the method responsible for creating new key pairs, check whether the passphrase entered meets the minimum requirements specified in the policy, and show an error message if the requirements are not met. Thus, the user can only complete the key generation step if he provides a passphrase allowed by the policy. Note

Table 7. Ratio of apps using native code

App Market	Overall		Games		No games	
	Apps	Nat. code	Apps	Nat. code	Apps	Nat. code
Google Play	9508	2212 (23%)	2838	1110 (39%)	6670	1102 (16%)
SlideMe	15974	1693 (10%)	5920	1244 (21%)	10054	449 (4.5%)
Total	25482	3905 (15%)	8758	2354 (26%)	16724	1551 (9.2%)

that this is an app-specific policy that defines joinpoints in the app code and not in the Android libraries as did the policies discussed before. In a similar fashion, we could enforce other security properties for the key pairs generated by APG, e.g. a minimum key size or the use of a specific algorithm.

5.4 Threats to Validity

Like in any IRM system, AppGuard's monitor runs within the same process as the target app. This makes it vulnerable to attacks from malicious apps that try to bypass or disable the security monitor. Our instrumentation technique is robust against attacks from Java code, as this code is strongly typed. It can handle cases like reflection or dynamically loaded libraries. However, a malicious app could use native code to disable the security monitor by altering the references we modified or tamper with the AppGuard's bytecode instructions or data structures. To prevent this, we could block the execution of any untrusted native code by intercepting calls to `System.loadLibrary()`, which is, however, not a viable solution in practice. Currently, AppGuard warns the user if an app attempts to execute untrusted native code.

In order to assess the potential impact of native code on our approach, we wanted to confirm our assumption that only a small percentage of apps rely on it. Our evaluation on 25,000 apps revealed that about 15% include native libraries (cf. Table 7), which is high compared to the 5% of apps reported in [50]. We conjecture that this difference is due to the composition of our sample. It consists of 30% games, which on Android frequently build upon native code based game engines (e.g., libGDX or Unity) to improve performance. Ignoring games, we found only 9% of the apps to be using native code, which makes AppGuard a safe solution for over 90% of these apps.

AppGuard monitors the invocation of security-relevant methods, which are typically part of the Android framework API. By reimplementing parts of this API and directly calling into lower layers of the framework, a malicious app could circumvent the security monitor. This attack vector is always available to attackers in IRM systems that monitor method invocations. Furthermore, AppGuard is not designed to be stealthy: due to the resigning of apps, instrumentation transparency cannot be guaranteed. There are many apps that verify their own signature (e.g. from the Amazon AppStore). If they rely on Android API to retrieve their own signature, however, AppGuard can hook these functions to return the original signature, thus concealing its presence. An app could

also detect the presence of AppGuard by looking for the presence of AppGuard classes in the virtual machine. In the end, both of these attacks boil down to an arms race, that a determined attacker will win. Up to now, we did not detect any app that tried to explicitly circumvent AppGuard.

Our instrumentation approach relies only on the layout of Dalvik's internal data structure for methods, which has not changed since the initial version of Android. However, our instrumentation system could easily be adapted if the layout were to change in future versions of Android.

5.5 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 app actually does with the requested permissions. Our fine-grained dynamic policies can be used to restrict the permissions of an app to those required to achieve the expected app behavior. As AppGuard denies access to certain functionality according to security policies, it may prevent proper functionality of an app. Policies use a best-effort approach to keep the program running even if security policies are violated by returning dummy values instead of e.g. null. We cannot guarantee, however, that security-breaking programs maintain their expected behavior.

AppGuard's log informs about all denied method calls and may contain the value of significant parameters. Granting access to those calls that are 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 while at the same time retaining the intended functionality. On top of generic policies for permission-revocation we offer app-specific policies that can even specify secret values to be kept inaccessible. We are currently implementing the automatic conversion of policies specified in SOSPoX into Java classes that monitor the app.

We demonstrate 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 app. 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 rewriting time is already reasonable, but we still see a large potential for reducing this time with some optimizations.

Android programs are multi-threaded by default. Issues of thread safety could therefore arise in the monitor when considering stateful policies that take the relative timing of events in different threads into account. While we did not yet experiment with such policies, we plan to extend our system to support *race-free policies* [13] in the future. In contrast, policies that atomically decide whether to permit a method call are also correct in the multithreaded setting.

6 Further 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 research [9, 19, 20, 29, 44] targets the detection of privacy leaks and malicious third-party apps. Another line of work analyzed Android's permission based access control system. Barrera et al. [4] conducted an empirical analysis of Android's permission system on 1,100 Android apps and suggested improvements to its granularity. Felt et al. [25] analyzed the effectiveness of app 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 [21, 32, 41–43]. Conti et al. [12] integrate a context-related policy enforcement mechanism to the Android software stack. Fragkaki et al. [27] recently presented an external reference monitor approach to enforce coarse grained secrecy and integrity policies called SORBET. In contrast, our intention was 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. [24] show that the mapping of permissions to API-calls is only insufficiently documented. 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 apps were over-privileged. Vidas et al. [48] 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 like privilege escalation attacks and the confused deputy problem [7, 8, 14, 17, 26].

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 [22, 23], 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 a rich set of formal policy specification languages. IRM systems have also been developed for other platforms. Mobile [36] is an extension to Microsoft's .NET Common Intermediate Language (CIL) that supports certified inlined reference monitoring. Finally, the S3MS.NET Run Time Monitor [16] enforces security policies expressed in a variety of policy languages for .NET desktop and mobile applications on Windows phones.

In our previous work [47] we presented the initial idea for diverting method calls in the Dalvik VM with a rudimentary implementation for micro-benchmarks only. It did *not* support a policy language, secrecy requirements, and on-the-phone instrumentation. Further, it did not include case studies. A recent tool paper [2] presented a previous version of AppGuard based on caller-site instrumentation.

7 Conclusions

We presented a practical approach to enforce high-level, fine-grained security policies on stock android phones. It is built upon a novel approach for callee-site inline reference monitoring and provides a powerful framework for enforcing arbitrary security and secrecy policies. Our system instruments directly on the phone and allows automatic updates without losing user data. Most prominently, the system curbs the pervasive overly curious behavior of Android apps. We enforce complex stateful security policies and mitigate vulnerabilities of both third-party apps and the OS. AppGuard goes even one step beyond and allows efficient protection of secret data from misuse in untrusted apps. Our experimental analysis demonstrates the robustness of the approach and shows that the overhead in terms of space and runtime are negligible. The case studies illustrate how AppGuard prevents several real-world attacks on Android. A recent release of AppGuard has already been downloaded by more than 1,000,000 users.

8 Acknowledgments

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: Security and Permissions (2012), http://developer.android.com/guide/topics/security/security.html
2. Backes, M., Gerling, S., Hammer, C., Maffei, M., von Styp-Rekowsky, P.: AppGuard - Enforcing User Requirements on Android Apps. In: Proc. 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013) (2013), to appear.
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 2012) (2012)
9. Chaudhuri, A., Fuchs, A., Foster, J.: SCanDroid: Automated Security Certification of Android Applications. Tech. Rep. CS-TR-4991, University of Maryland (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. Davis, B., Sanders, B., Khodaverdian, A., Chen, H.: I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications. In: Mobile Security Technologies 2012 (MoST 12) (2012)
16. 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)
17. 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)
18. von Eitzen, C.: Apple: Future iOS release will require user permission for apps to access address book (February 2012), http://h-online.com/-1435404
19. 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)
20. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A Study of Android Application Security. In: Proc. 20th Usenix Security Symposium (2011)
21. 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)
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. Fragkaki, E., Bauer, L., Jia, L., Swasey, D.: Modeling and Enhancing Android's Permission System. In: Proc. 17th European Symposium on Research in Computer Security (ESORICS 2012) (2012)
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. Gootee, R.: Evil Tea Timer (2012), https://github.com/ralphleon/EvilTeaTimer
32. 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 2012) (2012)
33. Gruver, B.: Smali: A assembler/disassembler for Android's dex format, http://code.google.com/p/smali/
34. 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)
35. Hamlen, K.W., Jones, M.M., Sridhar, M.: Chekov: Aspect-oriented Runtime Monitor Certification via Model-checking. Tech. Rep. UTDCS-16-11, University of Texas at Dallas (May 2011)
36. 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)
37. 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)
38. Jeon, J., Micinski, K.K., Vaughan, J.A., Reddy, N., Zhu, Y., Foster, J.S., Millstein, T.: Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android. Tech. Rep. CS-TR-5006, University of Maryland (December 2011)
39. 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
40. Ligatti, J., Bauer, L., Walker, D.: Edit Automata: Enforcement Mechanisms for Run-time Security Policies. International Journal of Information Security 4(1–2), 2–16 (2005)
41. 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)
42. 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)

43. 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)
44. 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)
45. 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
46. Schneider, F.B.: Enforceable Security Policies. ACM Transactions on Information and System Security 3(1), 30–50 (2000)
47. von Styp-Rekowsky, P., Gerling, S., Backes, M., Hammer, C.: Callee-site Rewriting of Sealed System Libraries. In: International Symposium on Engineering Secure Software and Systems (ESSoS'13). LNCS, Springer (2013), to appear.
48. 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)
49. Xu, R., Saïdi, H., Anderson, R.: Aurasium – Practical Policy Enforcement for Android Applications. In: Proc. 21st Usenix Security Symposium (2012)
50. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In: Proc. 19th Annual Network and Distributed System Security Symposium (NDSS 2012) (Feb 2012)