Decentralized Dynamic Security Enforcement for Mobile Applications with CliSeAuDroid

Tobias Hamann(✉) and Heiko Mantel

Department of Computer Science, TU Darmstadt, Darmstadt, Germany
{hamann,mantel}@cs.tu-darmstadt.de

Abstract. To date, Android is by far the most prevalent operating system for mobile devices. With Android devices taking a vital role in the everyday life of users, applications on these devices are handling vast amounts of private and potentially sensitive information, as well as sensitive sensor data like the device location. The built-in security mechanisms of the Android platform offer only limited protection for this data and device resources, and are not sufficient to enforce fine-grained policies on how data is used by applications. We present CliSeAuDroid, a runtime enforcement mechanism for Android applications that can enforce fine-grained security policies, either locally within a single application, across multiple applications, or even across multiple devices. We show that CliSeAuDroid can effectively ensure user-defined security requirements that protect sensitive data and resources on Android devices and adds only little runtime overhead to protected applications.

1 Introduction

Over the last decade, the Android platform has gained increasing popularity. To date, it is the most prevalent mobile operating system, with a market share of 85.9% in the first quarter of 2018.[1] Users are entrusting more and more private data to mobile applications, including, e.g., financial data for online banking, calendar entries, or health data. This makes mobile applications also a prime target for attackers. In addition to sensitive user data, mobile devices also expose a variety of privacy-relevant sensors, like the device camera or audio recording.

At the core of the built-in security mechanisms of the Android platform is the Android permission system. Whenever an application tries to access sensitive data or device resources, this access is controlled by the middleware layers of the Android platform. Before such an access can happen for the first time, the device user has to explicitly grant the corresponding permission to the application. Once an application has been granted a permission, however, the further usage of assets protected by this permission is only little controlled. Another challenge for Android security efforts is the fragmentation of the Android landscape into the many active Android versions. The slow migration of existing

[1] https://www.gartner.com/newsroom/id/3876865.

N. Zincir-Heywood et al. (Eds.): FPS 2018, LNCS 11358, pp. 29–45, 2019.
https://doi.org/10.1007/978-3-030-18419-3_3

devices to current Android versions means that new security features cannot be incorporated by applications running on older devices. While the current major version of the Android platform, Android 8, has been available for roughly one year, only 14.6% of all active devices are running on Android 8 in August 2018 [1]. Android versions older than Android 6, which introduced major improvements on permission management, are still used by roughly 32% of all active devices.

The shortcomings of the security mechanisms on the Android platform have attracted growing attention by the scientific community over the last years, leading to many security concepts in the area of Android security. Most of these concepts complement the built-in security mechanisms rather than replacing them. These concepts range from static analyses assessing the security of applications before they are installed (e.g., [5,8,18]), over adaptations or modifications of the operating system kernel or middleware layers (e.g., [6,7,9]), to dynamic approaches hardening applications on the application layer (e.g., [15,16,19]).

In this paper, we present CliSeAuDroid, a novel, flexible and light-weight dynamic enforcement mechanism for fine-grained security policies for Android applications.[2] CliSeAuDroid resides completely on the application level, and can be instantiated to address various security considerations not addressed by built-in security mechanisms. Our approach can be used to enforce all of local, cross-application, and cross-device security policies. By local policies, we refer to policies that involve only a single application. By cross-application and cross-device policies, we refer to policies that involve multiple applications on a single device or across devices, respectively. Such policies require coordination between applications and devices for effectively enforcing given security requirements. The capability to enforce cross-device policies for multiple devices of a user is desirable, and is not addressed by previous work on Android application security.

In detail, the contributions of this article are the following:

- We present CliSeAuDroid, a dynamic enforcement mechanism for user-defined, fine-grained security policies for Android applications. CliSeAuDroid can enforce both local policies and distributed policies on the same device or across devices. CliSeAuDroid incorporates the crosslining [11] technique that separates the interaction with the target program from the decision making. Such a separation is crucial for ensuring the reachability of the decision maker in distributed settings, especially on Android, where only one application can actively run in the foreground at a time.
- We evaluate the effectiveness and efficiency of the enforcement capabilities of CliSeAuDroid in case studies involving real, open-source Android applications. We show that CliSeAuDroid can effectively enforce both local policies and distributed policies. Our results indicate that the runtime overhead added by the enforcement are small, leading to no perceivable delay in the

[2] The implementation of CliSeAuDroid, all case study policies, and our results are available online. See Sect. 3 for details.

application execution for local enforcement. Distributed enforcement involving network communication shows a mean overhead below one second for up to four devices involved in the enforcement.

2 Android Application Security in a Nutshell

The Android platform provides various security mechanisms that are built into the platform implementation at different levels of the platform architecture. At the core of the security mechanisms for applications are a strong sandboxing mechanism, and a permission system for restricting access of applications to potentially sensitive user data and device resources.

2.1 The Android Security Architecture

The Android software stack is built on top of a Linux kernel. End-user applications run on top of multiple layers of middleware offered by the system architecture, consisting of the Android runtime, libraries, and the Java API framework. These middleware layers manage most of the interaction of applications with lower system levels, including access to protected resources on the device or to implementations of the IPC mechanisms.[3]

All applications on the Android platform are strictly sandboxed. Conceptually, this is achieved by providing distinct Linux UIDs to each application together with mandatory access control enforced by SELinux on the kernel level of the Android platform.[4] This sandboxing also applies for privileged system applications, as well as native code parts of applications. The strict sandboxing ensures that applications cannot access data that is local to other applications. Furthermore, the sandboxing mechanism ensures that access to protected system resources and data is managed by the Android platform implementation for each application, adhering to security policies established at the kernel level.

The core of the Android security architecture that is building on the strong sandboxing is its permission system that restricts the usage of certain designated operations and resources on a device.[5] These permissions are divided into three categories: normal permissions, signature permissions, and dangerous permissions. Data and resources protected by normal permissions are considered as low-risk operations and are granted to applications automatically when they request them, not requiring user confirmation. Signature permissions are granted to each application that requests them, provided that the application defining the permission has been signed with the same certificate as the requesting application. This can, for instance, be facilitated for custom-defined permissions that are shared between applications from the same developer. Dangerous permissions are considered to pose a significant risk for the user's privacy or the device's functionality. Examples for those dangerous permissions are sending

[3] https://developer.android.com/guide/platform/.

[4] https://developer.android.com/guide/components/fundamentals.

[5] https://developer.android.com/guide/topics/permissions/overview.

SMS messages (*SEND_SMS*) or accessing the current device location via GPS (*ACCESS_FINE_LOCATION*). Dangerous permissions are not granted to the application automatically, but have to be accepted by the user manually.

Prior to Android 6, permission requests were only posed at installation time of an application. Hence, users had only limited control over the behavior of applications: an installed application either got access to all requested permissions, or could not be installed on the device. From Android 6 onwards, these install-time requests were replaced with runtime requests. When installing an application, users are still presented with all permissions that the application requests. The actual granting of these permissions, however, is performed when the application tries to use them for the first time. Dangerous permissions can be revoked or regranted at any time using the system settings of the device.

Exemplary Shortcomings of Built-in Security Mechanisms. The built-in security mechanisms of the Android platform offer only a limited granularity for controlling sensitive data and resources on devices. For instance, the granting procedure for permissions at the first time of use is not sufficient to provide users with a fine-grained control over sensitive data and resources. Consider, for instance, a simple security requirement stating that an installed application may not send SMS messages to expensive premium SMS services (this requirement has been considered in other work, e.g., by DroidForce [19]). Using the built-in permission system, users have the choice to either grant the application with the permission to send SMS messages when it first asks for it, or can deny it. How the permission is actually used after granting the permission is not controlled by the Android system. In particular, an application requiring the permission to send SMS messages for benign purposes can also abuse this permission to send expensive messages without users noticing.

Naturally, on-device security mechanisms are limited to controlling applications on that very device. However, this can be insufficient to enforce user-specific security policies. Consider, for instance, a variant of the premium SMS requirement stating that multiple installed applications may send SMS messages also to expensive premium numbers, but within each 24-h interval only three such messages may be sent altogether from all devices of the user. In order to ensure this property, global knowledge of the execution history across devices is required. This cannot be achieved by the built-in Android security architecture.

2.2 State of the Art

To overcome the limitations of Android's built-in security mechanisms, a large variety of security solutions has been proposed in the literature, ranging from static analyses that assert application security before installation, to dynamic approaches that enforce security requirements at runtime.

Static Approaches. Static analysis techniques can be used for establishing trust in applications, enabling informed user decisions whether to install a given application or not. A large number of static analysis techniques for Android applications has been proposed. A recent, comprehensive literature survey of static analysis mechanisms for Android has analyzed over 120 research papers on static analysis approaches for Android applications [17]. These static approaches range from tools that build on analysis techniques that are proven to be sound (like, e.g., [8,18]) to tools that aim for a high recall in combination with high precision, but do not build on such formal foundations (like, e.g. [5]).

While static approaches can detect potential security violations in applications, they do not modify application behavior at runtime to make it compliant to a given security policy. In addition, static approaches can be overly restrictive, as they cannot take runtime information into consideration to determine whether sensitive information is actually leaked for a certain program run.

OS-Level and Middleware-Level Dynamic Enforcement. Application security can be ensured dynamically by enforcement mechanisms that are integrated into the operating system kernel or the middleware. Such approaches can offer a high level of protection by providing additional security features on devices. However, such approaches usually require substantial modifications of the platform running on the device, like rooting the device or flashing modified system images.

The Android Security Framework (ASF) [6] provides a module-based mechanism to extend the Android security mechanisms. The ASF resides on multiple layers of the Android platform, providing an API for security module developers. Security modules are provided in the form of code, and can be used to implement security enforcement for applications running on top of the ASF. The CRePE mechanism introduced context-aware enforcement of security policies for the Android platform [9]. CRePE resides in the Android middleware, and provides a hook-based system to detect access to permission-protected APIs of the Android system. It consists of a centralized policy provision and management system that is able to track the current device context, like, e.g., the current location of the device. Similar to CRePE, the Security Enhanced Android Framework (Seaf) provides a modified Android middleware layer that incorporates hooks to detect access to protected resources [7]. It provides both, a more fine-grained access control model for Android and behavior-based enforcement of security policies. This behavior-based enforcement can, e.g., be used to detect suspicious orderings of permission usages that indicate malicious behavior of a target application.

Application-Level Dynamic Enforcement. As an orthogonal approach to modifications of the operating system kernel or the Android middleware, enforcement mechanisms can reside completely on the application level. While such mechanisms are not closely integrated into the low-level parts of the system, they come with the advantage that usually no modification of the platform is required for enforcing security policies. Since this approach does not offer monitoring capabilities on the lower levels of the system (like, e.g., hooks), application-level enforcement mechanisms usually involve application instrumentation.

DroidForce [19] is a tool for enforcing complex, data-centric, system-wide policies for Android applications. Conceptually, DroidForce consists of a policy enforcement point that is inlined into target applications, and a central enforcement decision point on the device. Hence, enforcement decision in

DroidForce are always made centrally, and can consider the state of multiple applications on the same device. For detecting security-relevant program points, DroidForce incorporates a machine-learning approach. The framework presented in [16] provides capabilities to protect data usage on Android devices. The focus of this framework is the enforcement of attribute-based usage control depending on local or remote attributes that may change over time. Data providers embed usage control policies in data, which are then enforced by a central data protection system application on the device. Approaches that build on dynamic taint-tracking systems like, for instance, TaintDroid [10] follow an approach that is focused on data-flow tracking. Such approaches can analyze the flow of sensitive data inside or across applications, e.g., in order to prevent processing of sensitive data by third-party libraries. Many variants of dynamic or hybrid taint-tracking have been proposed, ranging from basic data-flow tracking to more sophisticated analyses that, e.g., consider native code inside apps [15].

3 CliSeAuDroid

With CliSeAuDroid, we present a novel runtime enforcement mechanism that enables the enforcement of fine-grained, user-defined security policies for Android applications both locally (i.e., within a single application), and in a distributed fashion (i.e., across different applications on the same device, or across different devices). CliSeAuDroid operates completely on the application layer, and can be applied to applications running on unmodified and unrooted Android devices. Policies for CliSeAuDroid are provided as Java source code, and are compiled for execution on devices. This approach provides support for using method and field invocations at the target program in the decision-making process, in particular involving the Android application lifecycle. In terms of the Android permission system, CliSeAuDroid offers a more fine-grained possibility to specify permission access, as well as a more sophisticated way to specify how data and device resources may be used after the corresponding permissions have been granted to an application. The implementation of CliSeAuDroid is provided as open-source software and available online. We provide the source code of CliSeAuDroid, sample security policies, and our evaluation results at www.mais.informatik.tu-darmstadt.de/assets/tools/cliseaudroid.zip.

3.1 Architecture

Figure 1 shows the architecture of CliSeAuDroid. It consists of four components: the interceptor, the coordinator, the local policy, and the enforcer. The interceptor component is responsible for monitoring the target application (Arrow 1), and communicating intercepted program events to the coordinator (Arrow 2). The coordinator component, in turn, checks with the local policy component whether the observed program behavior complies with the enforced security requirement (Arrow 3a). In case the observed behavior would violate the security requirement, a suitable countermeasure is determined and communicated to the enforcer component (Arrow 4). Finally, the enforcer component implements this countermeasure at the interaction point with the target application (Arrow 5). CliSeAuDroid supports the enforcement of distributed, system-wide security policies by communicating with other encapsulated applications. By system-wide policies, we refer to policies that consider all nodes in a distributed system, and that require global knowledge of the system state. This communication might involve other applications on the same device, or applications on a different device. In such settings, the coordinator can delegate the decision-making to other encapsulated applications instead of deciding locally (Arrow 3b).

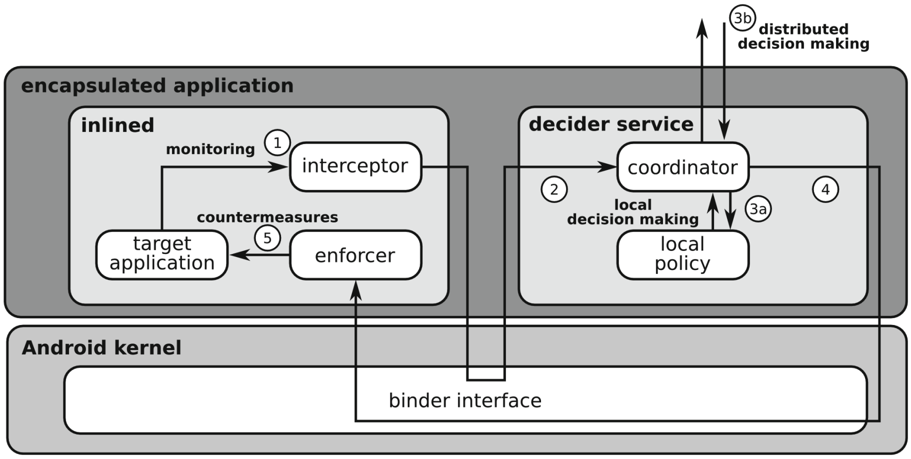


Fig. 1. Architecture of CliSeAuDroid

All components of the enforcement mechanism are located at the application level, depicted by the upper, dark-gray box in Fig. 1. This is achieved by instrumenting the target application APK file, placing the interceptor component and the enforcer component inlined into the target application code. The decision-making part of the mechanism, i.e., the coordinator component and the local policy component are located in a separate *decider service*. This technique of splitting the enforcement into an inlined entity and an entity outside the original application code, known as crosslining [11], guarantees that the decision making components are reachable for other applications, even if the target application is not actively running. The instrumentation results in an encapsulated application, in which all components of CliSeAuDroid are included into the target application. The inlined part of the security mechanism and the decider service communicate via the binder interface provided by the Android kernel, depicted by the lower, light-gray box of Fig. 1.

The component design of CliSeAuDroid implements the concept of service automata, a parametric framework for enforcing security requirements in local and distributed systems at runtime [14]. The service automata framework is parametric in the enforced security requirement, the possible countermeasures on the target, and the monitored program events. For all communication within the enforcement and for the decision-making, we abstract from target application events by generating internal events based on the observed behavior. Implementing the concept of Service Automata, CliSeAuDroid enables the decentralized coordination between different target applications running on a single device or across devices, and thus the enforcement of system-wide policies. A decentralized coordination that does not require a central decision-making entity can be beneficial depending on the application scenario, especially in distributed settings involving mobile devices that might not be reachable continuously.

3.2 Implementation Details

The implementation of CliSeAuDroid builds on CliSeAu [11], an implementation of the service automata framework for Java programs. CliSeAu is designed in a modular fashion that already enabled its extension for Ruby target programs [12]. The modular design of CliSeAu enabled us to reuse large parts of the existing codebase, since most components are working with an internal event abstraction that is target-language independent.

In addition to the existing codebase of CliSeAu, we developed software components that are specifically tailored to the peculiarities of the Android platform. The decision-making process is implemented as a background service, that is kept alive even when the target application is not actively running. The communication of the interceptor and the enforcer components with the decision-making service facilitate the Android inter-process communication mechanisms, in particular the Android binder interface using Intents. Different target applications instrumented with CliSeAuDroid communicate over plain Java sockets. Currently, this requires a-priori knowledge of the IP addresses of all targeted devices. However, CliSeAuDroid is designed in a modular fashion that enables its adaptation to different communication strategies. Hence, the socket-based communication can be adapted to a setting that is agnostic of the actual IP addresses of target devices, e.g., using cloud-based communication mechanisms like Firebase Cloud Messaging [3]. Regardless of the actual communication strategy, applications instrumented by CliSeAuDroid require the permission for internet access for distributed enforcement scenarios. We also enhanced the instrumentation infrastructure of CliSeAu to enable the instrumentation of Android applications using the AspectBench Compiler (abc) [4].

3.3 Application Instrumentation

The instrumentation process of CliSeAuDroid operates on the application package file (APK) of the target application. APKs are container files, including the binary files of the application as well as necessary dependencies and application resources. Figure 2 visualizes the instrumentation process of CliSeAuDroid. The instrumentation operates on three input artifacts: the APK file of the target application, an instantiation of CliSeAuDroid for the target application, and a pointcut specification of the security-relevant program points.

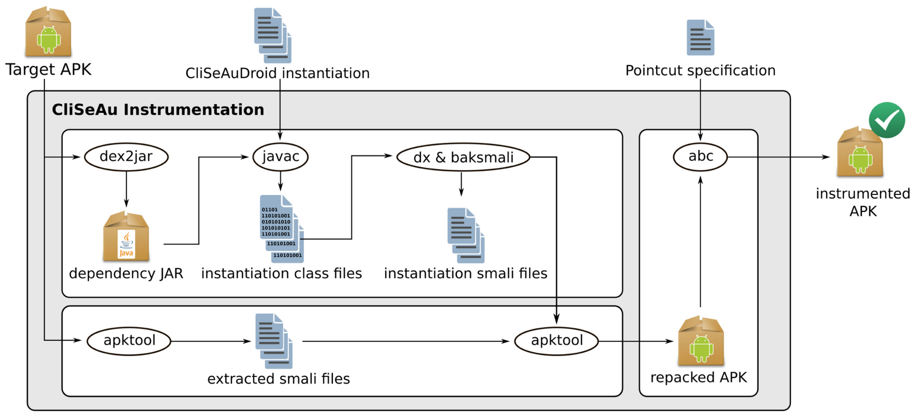


Fig. 2. Application instrumentation

The instantiation of CliSeAuDroid for the target program and policy consists of an implementation of the local policy component, the enforcer component, a factory class for abstracting from intercepted program behavior, and a factory class for creating enforcer objects from decisions. These implementations can be reused between different application scenarios, e.g., when the same policy shall be enforced but different program points are relevant for the monitoring process. The security-relevant program points are specified as AspectJ pointcuts. Program points matching these pointcuts are intercepted during the enforcement and translated to the internal event abstraction by a factory class.

The CliSeAuDroid instantiation for the target application is provided as source code and compiled against a JAR file (generated with dex2jar) containing class information of the target application. In particular, this JAR includes dependencies inside the target application that are required for compiling the policy instantiations. These dependencies include, for instance, information about the fields of classes in the target application, or method signatures of these classes. After compiling the CliSeAuDroid instantiation to class files, they are converted to smali files using the dx and baksmali tools, because regular Java class files cannot be directly used inside APK containers. In order to include the generated smali files into the target APK, we first extract the existing smali files from the APK using the apktool and subsequently repackage all files into a single APK file using the apktool. Finally, the AspectBench Compiler (abc) [4] combines the repackaged target APK that now contains all components required for an enforcement with CliSeAuDroid and the pointcut specification that provides the program points of the target application relevant for the enforcement. The result is an instrumented APK that is monitored by CliSeAuDroid. In case policy violations are detected, suitable countermeasures are taken. Note that the instrumented APK file needs to be signed after the instrumentation, as the existing signature of the APK is not preserved during the instrumentation.

4 Security Evaluation

CliSeAuDroid can enforce user-defined usage control policies for granted permissions and device resources locally, across applications and across devices. We empirically evaluate the capabilities of CliSeAuDroid for enforcing such fine-grained and system-wide security policies using exemplary case study policies highlighting security aspects that cannot be enforced by the Android platform security architecture. The flexible and modular design of CliSeAuDroid enables the enforcement of user-defined policies for a variety of requirements. The policies can reuse existing code from other policy instantiations, or can be developed from scratch to match given scenarios. In this paper, we focus on example policies that we also provide as part of our implementation. For evaluating the effectiveness of enforcing these policies, we target open-source applications publicly available on the F-Droid store [2] that make use of specific permissions that we target in the case study policies.

4.1 Case Study Policies

We present three classes of security policies that we investigate in our case studies. For each of these classes, we present an exemplary instantiation for a specific permission-protected part of the Android API.

Explicit Permission Usage Control. Permissions on the Android platform are granted per application based on user confirmation. Once the permission has been granted, users are not asked for confirmation again when the application accesses an API protected by that permission. While there might be benign reasons for an application to access a permission-protected API at the time a permission is granted, the application can also misuse the permission later.

CliSeAuDroid can be instantiated to ask users for explicit confirmation whenever the target application tries to access an API protected by a permission. We evaluate an instantiation of such a policy for the *SEND_SMS* permission. The SmsUserConfirmation policy provides users with a fine-grained control over permission usage for sending SMS messages. Instead of granting the *SEND_SMS* permission forever, users can choose to be asked every time the application tries to send a SMS message. When the application first tries to access the permission-protected API method for sending SMS messages, the user is presented with a popup window. In this popup window, the user can decide to grant or deny access for the permission to the application. In addition, the user can choose to remember this decision for the application. If the user does not choose to remember this decision, he will be presented with a popup message asking for permission each time the application tries to send a SMS message. Enforcing this policy for applications running on older devices also provides a backport of the permission model of Android starting from Android 5 in case the user chooses to remember the decision.

Rate-Limiting Policies. Once an application has been granted with a specific permission, there is no limit on the frequency the application can use API methods protected by this permission. While users might be willing to grant applications with a specific permission, they might want to ensure that the permission is not used extensively.

CliSeAuDroid enables the enforcement of user-defined rate limits for accessing permission-protected API methods. We evaluate an instantiation of such a policy for the *SEND_SMS* permission. The SmsRateLimiting policy limits the amount of SMS messages that can be sent by a specific application. Using the distributed enforcement capability of CliSeAuDroid, this policy can also be applied to enforce rate limits across multiple applications and devices of the user. This enables settings where a user wants to use applications that make use of SMS messaging on more than one device, e.g., on a smartphone and on a tablet device. Note that for both local and distributed rate limiting, we can reuse the same instantiation of CliSeAuDroid. In distributed settings each unit is instrumented separately and installed on the corresponding device. The coordinator components of the different units will handle the decision-making during runtime in a transparent fashion for the end user.

Provision of Fake Data. Denying applications access to permission-protected APIs can lead to application crashes, as the application might depend on the presence of data queried from the APIs. While users might want to deny certain applications access to specific permissions, they might still have an interest in using other functionalities of the application. Hence, avoiding application crashes in such cases is a desirable goal for enforcement mechanisms.

CliSeAuDroid can be instantiated to deny applications access to specific permissions, providing fake data for the application instead. We evaluate an instantiation of such a policy for the *ACCESS_FINE_LOCATION* permission. The FakeLocationProvision policy can prevent application crashes by enabling the provision of fake location data to applications. Whenever the target application tries to access the current location of the device, the return values of the API calls are intercepted and modified to show a different location (e.g., in Antarctica). This enables users to still run the application, without providing it with the actual location of the device. Note that the generic architecture of CliSeAuDroid also allows the provision of more sophisticated return data, like, e.g., plausible movement profiles for fake locations. For the evaluation in this article, we limit ourselves to a static return value that is used as fake data.

4.2 Case Study Instantiations

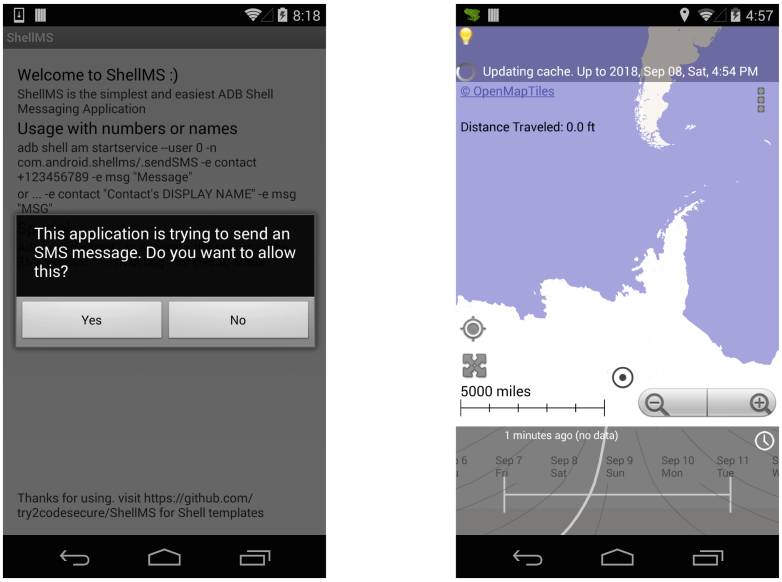
In our evaluations, we target two open-source applications that are publicly available on F-Droid: TinyTravelTracker[6] and ShellMS[7]. The TinyTravelTracker app collects GPS location data in the background in order to provide the user with movement profiles. Naturally, TinyTravelTracker requires access to the *ACCESS_FINE_LOCATION* permission for this purpose. The ShellMS app, in turn, provides a background service that can be used by other applications on the device or by users via the Android Debug Bridge (adb) for sending SMS messages. Naturally, ShellMS requires the *SEND_SMS* permission.

Evaluation Setup. We instantiate the FakeLocationProvision policy for TinyTravelTracker. We instantiate both the SmsUserConfirmation policy, and the SmsRateLimiting policies for ShellMS. For the SmsRateLimiting policy, we evaluate four variants: a purely local variant involving only one application instance and device, and distributed variants involving 2, 3, and 4 devices, correspondingly. We evaluated all policy instantiations on Google Nexus 5 devices running Android 4.4.3 in a local WiFi network.

Evaluation Results. The left-hand side of Fig. 3 shows a screenshot of an instrumented ShellMS instance using the SmsUserConfirmation policy. Our evaluation confirms that CliSeAuDroid can effectively enforce the policy, and will ask users for permission whenever the application tries to send SMS messages. No SMS message was sent without explicit user confirmation in our experiments. We further confirmed in our experiments that the SmsRateLimiting policy was correctly preventing ShellMS from sending more messages than the quota permits. We were able to confirm this both locally, and in our cross-device experiments. The quota limit was enforced across all involved devices, regardless of where the permitted messages originated before. The right-hand side of Fig. 3 shows a screenshot of an instrumented TinyTravelTracker instance for the FakeLocationProvision policy. As can be seen in the figure, the application is correctly prevented from access to the real device location and is provided with a fake location in Antarctica instead. Note that we do not evaluate in full rigor whether all invocations of security-relevant methods are intercepted and handled by CliSeAuDroid. We rely on the automatized instrumentation by abc for ensuring a sound instantiation of the monitoring. Defining suitable pointcuts for given security requirements is part of policy development and controlled by the user of CliSeAuDroid.

In summary, CliSeAuDroid succeeded in enforcing all of our security policy instantiations, both locally on a single device and in distributed settings involving up to four devices. In our experiments, we did not observe any disruption of regular application functionality that was not covered by the security policies.

[6] https://f-droid.org/en/packages/com.rareventure.gps2/.
[7] https://f-droid.org/en/packages/com.android.shellms/.



ShellMS

Welcome to ShellMS :)

ShellMS is the simplest and easiest ADB Shell Messaging Application

Usage with numbers or names

adb shell am startservice --user 0 -n com.android.shellms/.sendSMS -e contact +123456789 -e msg "Message"

or ... -e contact "Contact's DISPLAY NAME" -e msg "MSG"

This application is trying to send an SMS message. Do you want to allow this?

Yes No

Thanks for using. visit https://github.com/try2codesecure/ShellMS for Shell templates

Fig. 3. Screenshots of encapsulated target applications

5 Performance Evaluation

Dynamically enforcing security policies at runtime comes at the cost of a performance overhead. This overhead is caused by the additional time required for monitoring application behavior, the time required for the decision making, and the time required to impose countermeasures on the application. In addition to the runtime overhead, some preprocessing time is required when instrumenting the target application. In this section, we evaluate the performance overhead introduced by CliSeAuDroid in both of these dimensions for the case study policies presented in Sect. 4. We show that the runtime overhead of CliSeAuDroid achieves its goal of achieving a light-weight enforcement without adding extensive runtime overhead, and that the runtime overhead of CliSeAuDroid is not perceivable to end users for local enforcement.

5.1 Instrumentation Overhead

Instrumenting a target application with CliSeAuDroid is a one-time operation. Once an instance of the target application has been instrumented for a specific device, the resulting APK file can be installed on the device like a regular application. The instrumentation process can thus be kept transparent to the end user, who is provided with the instrumented APK file. When instrumenting applications for a distributed setting, one instance of the target application is instrumented per unit of the distributed system.

Figure 4 summarizes the instrumentation times and size overheads for the different target applications and case study policies. Overall, our evaluation shows that the instrumentation time is below three minutes for all our case study policies and applications. Since the instrumentation is carried out only once before installing the target application on a device, we consider this an acceptable overhead. Regarding application size, the instrumentation added less than 1 MB to the original application size for each policy instantiation.

security policy	instrumentation time	size overhead
SmsUserConfirmation	35s	100.5kB (25.2%)
SmsRateLimiting	34s	96.7kB (24.3%)
FakeLocationProvision	150s	591.9kB (3.3%)

Fig. 4. Instrumentation time and application size overhead (per unit)

5.2 Runtime Overhead

We evaluate the runtime overhead introduced by CliSeAuDroid when running an encapsulated application by measuring the time spent within the enforcement components. For this, we measure the start time when we first intercept a program event that is relevant for the enforced security policy. We measure the end time at the point just before the enforcer applies the determined countermeasure to the target program. For our experiments, we carried out measurements for each policy instantiation, discarding outliers that lie more than three absolute standard deviations from the median. All experiments were carried out on Google Nexus 5 devices running Android 4.4.3. The start and end times, respectively, were logged to the device for evaluation. These log results were extracted from the devices using the Android debug bridge (adb).

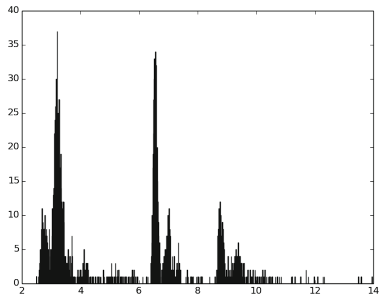
Figure 5 summarizes the mean overhead times (with 95% confidence intervals) and the standard deviation introduced by CliSeAuDroid for each policy instantiation. For the policies that do not involve direct user interaction (i.e., the SmsRateLimiting policy, and the FakeLocationProvision policy), we carried out 2,000 experiments. For the SmsUserConfirmation policy involving user interaction, we carried out 100 experiments. Our results show that for purely local enforcement within a single application, the mean overhead added by CliSeAuDroid is below 6 ms for each runtime check on the invocation of a security-relevant method. For distributed enforcement across devices, our results show a comparably high overhead that is increasing with the amount of network hops performed during the enforcement. In addition to the higher mean enforcement overhead, we can also observe that the standard deviation is significantly higher than for local enforcement. Figure 6 visualizes the distributions of overhead times for local enforcement of the FakeLocationProvision and SmsRateLimiting policies.

security policy	mean overhead	standard deviation
SmsUserConfirmation	1.6726 ± 0.0232 ms	0.2753 ms
SmsRateLimiting (local)	5.4051 ± 0.0443 ms	2.3608 ms
SmsRateLimiting (2 hops)	423.4710 ± 5.3791 ms	287.8840 ms
SmsRateLimiting (3 hops)	616.0131 ± 7.0449 ms	377.2245 ms
SmsRateLimiting (4 hops)	712.3791 ± 6.2084 ms	332.2642 ms
FakeLocationProvision	3.6122 ± 0.0227 ms	1.1995 ms

Fig. 5. Runtime overhead introduced by CliSeAuDroid enforcement

Our results indicate that the runtime overhead introduced by CliSeAuDroid is within limits that are not perceivable to the end user for local enforcement. Indeed, during our experiments we did not notice any disruption of application functionality. This observation lines up with the overhead added by CliSeAu for other target languages, and is competitive with other enforcement mechanisms for the Android platform, like, e.g., DroidForce [19].

For distributed policies, we can observe a much higher overhead above 400 ms. This magnitude of runtime overhead can be clearly perceivable to end users. However, depending on the application scenario, the security benefits can still outweigh the overhead. Our experiments show that even with up to four devices involved in the enforcement process, the overhead remains below 1 s. Interpreting the overhead in distributed settings, the biggest part of the overhead seems to stem from network communication overhead. We consider investigations of possibilities to decrease this overhead while still ensuring a sound enforcement as an interesting direction for future work. Decreasing this overhead might involve different communication technologies, or strategies to reduce the amount of communication required for decision making, e.g., by precomputing decisions. Previous evaluations of such precomputation strategies showed a significant potential for reducing overhead [13].



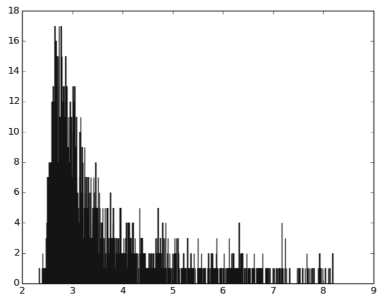


Fig. 6. Runtime overhead distribution introduced by local enforcement for the SmsRateLimiting policy (left) and FakeLocationProvision policy (right)

6 Conclusion

In this article, we presented CliSeAuDroid, a mechanism for dynamically enforcing both local and system-wide security policies for Android applications at runtime. CliSeAuDroid enables the enforcement of fine-grained security policies that cannot be enforced with built-in Android security mechanisms. Our mechanism is implemented completely at the application layer, and can be used on unmodified and unrooted Android devices.

We showed that CliSeAuDroid can effectively enforce realistic, user-defined security policies for applications running on a single device or across devices. Our experimental evaluation indicates that the performance overhead added by this enforcement is small, and within boundaries that are not recognizable by end users for local enforcement. When enforcing distributed policies, the performance overhead is significantly higher, but was still below 1 second in our experiments.

The capability to enforce cross-device security policies adds to our confidence that CliSeAuDroid is not just yet another tool for Android security, but provides a flexible and light-weight solution for security concerns in our increasingly connected world.

Acknowledgments. This work was supported by the DFG under the project RSCP (MA 3326/4-3) in the priority program RS^3 (SPP 1496).

References

1. Android Distribution Dashboard. https://developer.android.com/about/dashboards/. Accessed 3 Sept 2018
2. F-Droid. https://www.f-droid.org. Accessed 3 Sept 2018
3. Firebase Cloud Messaging (FCM). https://firebase.google.com/docs/cloud-messaging/. Accessed 3 Sept 2018
4. Arzt, S., Rasthofer, S., Bodden, E.: Instrumenting Android and Java applications as easy as abc. In: Legay, A., Bensalem, S. (eds.) RV 2013. LNCS, vol. 8174, pp. 364–381. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40787-1_26
5. Arzt, S., et al.: FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: PLDI 2014, pp. 259–269 (2014)
6. Backes, M., Bugiel, S., Gerling, S., von Styp-Rekowsky, P.: Android security framework: extensible multi-layered access control on Android. In: ACSAC 2014, pp. 46–55 (2014)
7. Banuri, H., et al.: An Android runtime security policy enforcement framework. Pers. Ubiquitous Comput. **16**(6), 631–641 (2012)
8. Chen, H., Tiu, A., Xu, Z., Liu, Y.: A permission-dependent type system for secure information flow analysis. In: CSF 2018, pp. 218–232 (2018)
9. Conti, M., Nguyen, V.T.N., Crispo, B.: CRePE: context-related policy enforcement for Android. In: Burmester, M., Tsudik, G., Magliveras, S., Ilić, I. (eds.) ISC 2010. LNCS, vol. 6531, pp. 331–345. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18178-8_29
10. Enck, W., et al.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. **32**(2), 5 (2014)

11. Gay, R., Hu, J., Mantel, H.: CliSeAu: securing distributed Java programs by cooperative dynamic enforcement. In: Prakash, A., Shyamasundar, R. (eds.) ICISS 2014. LNCS, vol. 8880, pp. 378–398. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13841-1_21
12. Gay, R., Hu, J., Mantel, H., Mazaheri, S.: Relationship-based access control for resharing in decentralized online social networks. In: Imine, A., Fernandez, J.M., Marion, J.-Y., Logrippo, L., Garcia-Alfaro, J. (eds.) FPS 2017. LNCS, vol. 10723, pp. 18–34. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75650-9_2
13. Gay, R., Hu, J., Mantel, H., Schickel, J.: Towards accelerated usage control based on access correlations. In: Lipmaa, H., Mitrokotsa, A., Matulevičius, R. (eds.) NordSec 2017. LNCS, vol. 10674, pp. 245–261. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70290-2_15
14. Gay, R., Mantel, H., Sprick, B.: Service automata. In: Barthe, G., Datta, A., Etalle, S. (eds.) FAST 2011. LNCS, vol. 7140, pp. 148–163. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29420-4_10
15. Graa, M., Cuppens-Boulahia, N., Cuppens, F., Lanet, J.-L.: Tracking explicit and control flows in Java and native Android apps code. In: ICISSP 2016, pp. 307–316 (2016)
16. Lazouski, A., Martinelli, F., Mori, P., Saracino, A.: Stateful data usage control for Android mobile devices. Int. J. Inf. Secur. **16**(4), 345–369 (2017)
17. Li, L., Bissyandé, T.F., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., Le Traon, Y.: Static analysis of Android apps: a systematic literature review. Inf. Softw. Technol. **88**, 67–95 (2017)
18. Lortz, S., Mantel, H., Starostin, A., Bähr, T., Schneider, D., Weber, A.: Cassandra: towards a certifying app store for Android. In: SPSM 2014, pp. 93–104 (2014)
19. Rasthofer, S., Arzt, S., Lovat, E., Bodden, E.: DroidForce: enforcing complex, data-centric, system-wide policies in Android. In: ARES 2014, pp. 40–49 (2014)