URANOS: User-Guided Rewriting for Plugin-Enabled ANdroid ApplicatiOn Security

Daniel Schreckling, Stephan Huber, Focke Höhne, Joachim Posegga

URANOS: User-Guided Rewriting for Plugin-Enabled ANdroid ApplicatiOn Security

Daniel Schreckling, Stephan Huber, Focke Höhne, and Joachim Posegga

Institute of IT-Security and Security Law
University of Passau, Innstraße 43, Passau, Germany
{ds,sh,fh,jp}@sec.uni-passau.de

Abstract. URANOS is an Android application which uses syntactical static analysis to determine in which component of an Android application a permission is required. This work describes how the detection and analysis of widely distributed and security critical adware plugins is achieved. We show, how users can trigger bytecode rewriting to (de)activate selected or redundant permissions in Android applications without sacrificing functionality. The paper also discusses performance, security, and legal implications of the presented approach.

1 Introduction

Many Smartphone operating systems associate shared resources with permissions. API calls accessing such resources require permissions to gain the required privileges. Once an application obtains these privileges, it can generally access all the items stored in the respective resource. Additionally, such privileges are often valid until the deinstallation or an update of the application. These properties conflict with the emerging privacy needs of users. Increasing sensitivity encourages the protection of data which helps applications, vendors, or providers to generate individual user profiles. Unfortunately, current coarse grained permission systems only provide limited control or information about an application. Hence, informed consents to the use of permissions are far from being available.

In Android, numerous analyses of permissions requested by an application [3, 11, 14, 20, 21] substantiate this problem. Permissions increase the attack surface of an application [4, 2, 12] and the platform executing it. Thus, granting permissions in excessive manners induces new exploit techniques. Static analysis and runtime monitoring frameworks have been developed to detect permission-based platform and application vulnerabilities. There are also Android core extensions enabling the deactivation of selected permissions. However, such frameworks either interfere with the usability of the application and render it unusable or they only provide permission analysis on separate hosts.

Thus, there is a strong need for flexible security solutions which do not aim at generality and precision but couple lightweight analysis and permission modification mechanisms.

We define URANOS, an application rewriting framework for Android which enables the selective deactivation of permissions for specific application contexts,

e.g. plugins. The contributions of this paper include an on-device static analysis to detect permissions and their usage, selective on-device rewriting to guarantee user-specific permission settings, and a prototype implementing detection and rewriting in common Android applications.

Our contribution is structured as follows: Section 3 provides a knowledge base for this contribution, Section 2 gives a high-level overview of URANOS. Its components are explained in Section 4. Section 5 discusses performance, limitations, and legal implications. Finally, Section 6 lists related work before Section 7 summarizes our conclusions.

2 Approach Overview

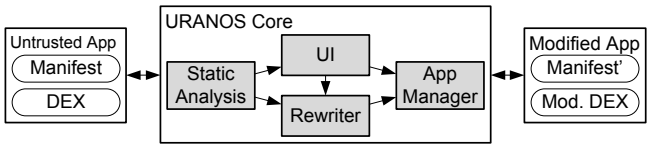

Fig. 1. High-level overview of URANOS

We strive for an efficient on-device framework (see Figure 1) for Android which allows users to selectively disable permissions assigned to an application. To preserve functionality a static analysis infers the permissions required during execution from the bytecode. For efficiency we exploit existing knowledge about the permission requirements of Android API calls, resource access, intent broadcasting etc. Detected permissions are compared with the permissions requested in the application manifest to detect excessive permissions etc. Additionally, we scan the bytecode for plugins using a pre-generated database of API methods and classes used in popular adware. They define context for each bytecode instruction. This allows us to infer the permissions exclusively required for plugins or for the application hosting the plugins. We communicate this information to the user. Depending on his needs, the user can enable or disable permissions for specific application contexts.

Disabled and excessive permissions can be completely removed from the manifest. However, removing an effectively required permission will trigger a security exception during runtime. If these exceptions are unhandled the application will terminate. Therefore, URANOS additionally adapts the application bytecode and replaces the API calls in the respective call context by feasible wrappers.

This combination of analysis and rewriting allows a user to generate operational applications compliant with his security needs. Unfortunately, compliant but rewritten Android applications are neither directly installed nor are they updated by Android. Therefore, URANOS also delivers an application manager service, replacing applications with their rewritten counterparts and ensuring their updates.

3 Background

This section gives a short overview of the structure of Android applications, their execution environment, and the permission system in Android.

3.1 Android Applications

Applications (Apps) are delivered in zipped packages (`apk` files). They contain multimedia content for the user interface, configuration files such as the manifest, and the bytecode which is stored in a Dalvik executable (`dex` file). Based on the underlying Linux, Android allots user and group IDs to each application.

Four basic types of components can be used to build an App: *activities*, *services*, *content providers*, and *broadcast receivers*. *Activities* constitute the user interface of an application. Multiple activities can be defined but only one activity can be active at a time. *Services* are used to perform time-consuming or background tasks. Specific API functions trigger remote procedure calls and can be used to interact with services. Application can define *content providers* to share their structured data with other Apps. To retrieve this data, so called *ContentResolvers* must be used. They use URIs to access a provider and query it for specific data. Finally, *broadcast receivers* enable applications to exchange *intents*. Intents express an *intent* to perform an action within or on a component. Actions include the display of a picture or the opening of a specific web page.

Developers usually use these components defined in the Android API and the SDK to build, compile, and pack Apps. Their `apks` are signed with the private developer key, distributed via the official Android market, other markets, or it is delivered directly to a Smartphone.

3.2 Dalvik Virtual Machine

Bytecode is stored in Dalvik executables (`dex` files) and is executed in a register based virtual machine (VM) called Dalvik. Each VM runs in its own application process. The system process Zygote starts at boot time and manages the spawning of new VMs. It also preloads and preinitializes the core library classes.

`Dex` files are optimized for size and data sharing among classes. In contrast to standard Java archives, the dex does not store several class files. All classes are compiled into one single `dex` file. This reduces load time and supports efficient code management. Type-specific mappings are defined over all classes and map constant properties of the code to static identifiers, such as constant values, class, field, and method names. The bytecode can also contain developer code not available on the platform, e.g. third-party code, such as plugins (see Figure 2).

Bytecode instructions use numbered register sets for their computations. For method calls, registers passed as arguments are simply copied into new registers only valid during method execution.

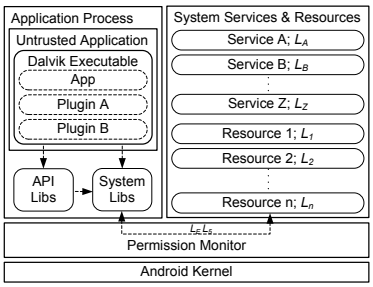

Fig. 2. Plugins and Permissions in Android Applications

3.3 Android Permissions

Android permissions control application access to resources. Depending on their potential impact Android distinguishes three levels: *normal*, *dangerous*, *signature* and *signatureORsystem*. Unlike normal permission which do not directly cause financial harm to users, dangerous and system permission control access to critical resources and may enable access to private data. Granted signature or signaturesORsystem permission grant access to essential system services and data. During installation permissions are assigned as requested in the manifest. The user only approves dangerous permissions. Normal permissions are granted without notification and signature or signatureORsystem permissions verify that the application requesting the permissions has been signed with the key of the device manufacturer.

Resource access can be obtained through API calls, the processing of intents, and through access to content providers and other system resources such as an SD card. Thus, permission enforcement varies with the type of resource accessed. In general, permission assignment and enforcement can be described using a label model as depicted in Figure 2. Each system resource or system service is labeled with the set of permissions it requires to be accessed. An application uses the public API to trigger resource access. This request is forwarded to the system. The system libraries, the binder, and the implementation of the libraries finally execute the resource access. We abstract from the details of the binder-library pair and call this entity a central *permission monitor*. It checks whether an application trying to access a resource with label L_x has been assigned this label. If not, access is forbidden and an appropriate security exception is thrown.

Android also places permission checks in the API and RPC calls [9]. Thus, security exceptions may already occur although the access requests have not reached the permission monitor, yet. As such checks may be circumvented by reflection the actual enforcement happens in the system.

4 The URANOS Framework

This section explains our system in more detail. To ease the understanding we complement our description with Figure 3.

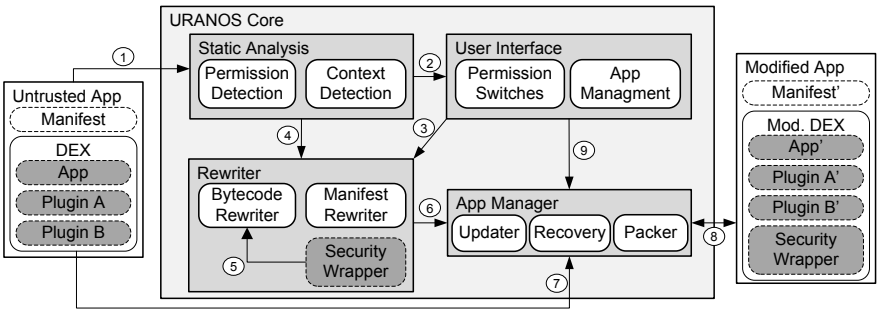

Fig. 3. System Overview

4.1 Application Processing

To process manifest and bytecode of the application, URANOS must obtain access to the `apks`. Depending on how the developer decides how to publish an APK, it is stored in different file system locations: the regular application storage, the application storage on an SD card, or storage which prevents the forwarding (forward-lock) of the application. The `PackageManager` API offered by Android can be used to retrieve the path and filename of the `apks`.

Regular applications are able to obtain reading access to other `apks`. Thus, as a regular application, URANOS can copy `apks` to a local folder and process them. With root permissions, it can also process forward-locked applications.

`Apks` are extracted to obtain access to the manifest and the `dex` file. We enhanced the `dex-tools` from the Android source tree. It directly operates on the bytecode and can extract information required for our analysis. Thus, we avoid intermediate representations. Handles to manifest and bytecode are forwarded (1) to the static analysis and rewriting components of our framework.

4.2 Permission Detection

Next, we parse the manifest and retrieve the set P_{apk} of permissions requested by the App. Afterwards, we scan the bytecode to find all invoke instructions and determine the correct signature of the methods invoked. Invoke instructions use identifiers pointing to entries of a management table which contains complete method signatures. From this table we derive the set I of methods potentially invoked during execution. As this is a syntactical process set I may contain methods which are never called.

We then use function π to compute $P_M = \bigcup_{\forall m \in I} \pi(m)$, i.e. π maps method m to a set of permissions required to invoke m at runtime. Thus, P_M reflects the permissions required by the application to execute all methods in I. Function π is based on the results of Felt et al. [9] which associate actions in an Android App with its required permissions, e.g. method calls.

The use of content providers or intents may also require permissions. However, specific targets of both can be specified using ordinary strings. To keep our

analysis process simple we search the `dex` for strings which map the pattern of a content provider URI or of a activity class name which is defined in the Android API. If a pattern is matched, we add the respective permission to the set P_P of provider permissions or to the set P_I of intent permissions, respectively.

At the end of this process we intersect the permissions specified in the manifest with the permissions extracted from the bytecode, i.e. $P_{val} = P_{apk} \cap (P_M \cup P_P \cup P_I)$ to obtain the validated permissions likely to be required for the execution of the application. Our heuristics induce an over-approximation in this set. Section 5 explains why it does not influence the security of our approach.

4.3 Context Detection

Based on P_{val} we now determine the App components in which the methods requiring these permissions are called. For this purpose we define the *execution context* for an instruction. It is the signature of the method and its class in which the instruction is executed. This definition is generic and can be applied to various detection problems. We focuse on widely distributed plugins for Android.

To give users a better understanding on the possible impact of the plugins hosted by the analyzed Apps we manually assign each plugin to the following four categories: *passive*, *active*, *audio advertising*, and *feature extensions*.

Passive advertising plugins display advertisements as soon as an activity of the hosting application is active. They are usually integrated into the user interface with banners as placeholders.

Active advertising plugins are similar to pop-up windows and do not require a hosting applications. They use stand alone activities or services, intercept intents, or customize events to become active.

Audio advertising is a rather new plugin category which intercepts control sequences and interferes with the user by playing audio commercials or similar audio content, e.g. while hearing the call signal on the phone.

Feature extensions include features in an application a user or developer may utilize. Among many others, they include in-app billing or developer plugins easing the debugging process.

To detect plugins in an application, we perform the same steps required for archiving the signatures. We scan the application manifest and bytecode for the names listed above and investigate which libraries have to be loaded at runtime. From this information we build a signature and try to match it against our plugin database. This process also uses fuzzy patterns to match the strings inferred from the application. We assume that plugins follow common naming conventions. So, full class names should start with the top Internet domain name, continue with the appropriate company name, and end with the class names. If we do not find matches on full class names, we search for longest common prefixes. If they contain at least two subdomains, we continue searching for the other names to refine the plugin match. In this way we can account for smaller or intentional changes in class or package naming and prevent a considerable decline of the detection rate.

The ability to detect classes of plugins allows us to determine execution contexts. During the bytecode scanning, we track the context C. As soon as our analysis enters the namespace of a plugin class, we change C. It is defined by the name of the plugin N_{Plugin} or by the name N_{apk} of the application if no plugin matches. We generate a map for each method call to its calling context. Together with the function π, this implicitly defines a map γ from permissions to calling contexts. We can now distinguish four types of permissions:

Dispensable permissions $p \in P_{apk} \setminus P_{val}$ are not required by the application,
Application only permissions $p \in P_{apk}$ are exclusively required for the hosting application to run, i.e. $\gamma(p) = \{N_{apk}\}$,
Plugin only permissions $p \in P_{apk}$ are exclusively required for the execution of a plugin, i.e. $\gamma(p) \cap \{N_{apk}\} = \emptyset$, and
Hybrid permissions $p \in P_{apk}$ which are required by both, the hosting application and the plugin, i.e. $\gamma(p)$ does not match the conditions for the other three permission types.

This result is communicated to the user in step (2). He gets an overview of the types of permissions and the context in which they are required. The user can enable or disable them in the entire application, only in the plugin, or only in the hosting application. The next section shows how to support this feature with the help of bytecode rewriting and without modifying Android.

4.4 Rewriter

In general, dispensable permissions are not required for the execution and don't need to be assigned to the application. They can removed from the manifest. The same holds for permissions which should be disabled for the entire application. Thus, the first rewriting step is performed on the application manifest. It revokes the permissions either not required or not desired.

However, withdrawing permissions from an application may render it unusable. Calls to methods which require permissions will throw exceptions. If they are not handled correctly, the runtime environment could finally interrupt execution. To avoid this problem, enable the deactivation of permissions in only specific application components, and to retain an unmodified Android core, the activation or deactivation of permissions triggers a rewriting process (3). It is guided by the results of the syntactical analysis (4). The rewriter, described in this section, adapts the bytecode in such a way that the App can be executed safely even without the permissions it originally requested.

API Methods For each method whose execution requires a permission, we provide two types of wrappers (5) to replace the original call. Regular API method calls which require a permission, can be wrapped by simple `try` and `catch` blocks as depicted by `WRAPPER1` in Listing 1.1. If the permission required to execute the API call has been withdrawn, we catch the exception and return a feasible default value. In case the permission is still valid, the original method

is called. In contrast, the second wrapper `WRAPPER2` (Listing 1.2) completely replaces the original API call and only executes a default action.

Listing 1.1. Wrapper pattern one

```
public static WRAPPER1 {
  try {
    API_CALL_ACTION;
  } catch (SecurityException se) {
    DEFAULT_ACTION;
  }
}
```

Listing 1.2. Wrapper pattern two

```
public static WRAPPER2 {
    DEFAULT_ACTION;
}
```

Evidently, rewriting could be reduced to only `WRAPPER2`. But, `WRAPPER1` reduces the number of events at which an application has to be rewritten and reinstalled. Assume that a user deactivates a permission for the entire application. The permission is removed from the manifest and all methods requiring it are wrapped. Depending on the wrapper and the next change in the permission settings a rewriting may be avoided because the old wrapper already handles the new settings, e.g. the reactivation of the permission.

Wrappers are static methods and apart from one additional instance argument for non-static methods, they inherit the number and type of arguments from the methods they wrap. This makes it easy to automatically derive them from the API. Additionally, it simplifies the rewriting process as follows.

URANOS delivers a `dex` file which contains the bytecode of all wrappers. This file is merged with the original application `dex` using the `dx` compiler libraries. The new `dex` now contains the wrappers but does not make use of them, yet. In the next step we obtain the list of method calls which need to be replaced from the static analysis component (4). The corresponding invoke instructions are relocated in the new `dex` and the old method identifiers are exchanged with the identifiers of the corresponding wrapper methods.

Here, the rewriting process is finished even if the wrapped method is non-static. At bytecode level, the replacement of a non-static method with a static one simply induces a new interpretation of the registers involved in the call. The register originally storing the object instance is now interpreted as the first method argument. Thus, we pass the instance register to the wrapper in the first argument and can leave all other registers in the bytecode untouched. We illustrate this case in Listing 1.3. It shows bytecode mnemonics for the invocation of the API method `getDeviceId` as obtained by a disassemblers.

Listing 1.3. Regular API call

```
invoke-virtual {v0},
  Landroid/telephony/
      TelephonyManager;
    .getDeviceId:()
       Ljava/lang/String;
```

Listing 1.4. Rewritten API call

```
invoke-static {v0},
  Lde/wrapper/Wrapper;
    ._getDeviceId:(Landroid/telephony/
         TelephonyManager;)
        Ljava/lang/String;
```

The instruction `invoke-virtual` calls the method `getDeviceId` on an instance of class `TelephonyManager`. It is rewritten to a static call in Listing 1.4 and passes the instance as an argument to the static wrapper method.

Reflection Android supports reflective method calls. They use strings to retrieve or generate instances of classes and to call methods on such instances.

These operations can be constructed at runtime. Hence, the targets of reflective calls are not decidable during analysis and calls to API methods may remain undetected. Therefore, we wrap the methods which can trigger reflective calls, i.e. `invoke` and `newInstance`. During runtime, these wrappers check the `Method` instance passed to `invoke` or the class instance on which `newInstance` is called. Depending on its location in the bytecode the reflection wrapper is constructed in such a way that it passes the invocation to the appropriate wrapper methods (see above) or executes the function in the original bytecode. This does not require dynamic monitoring but can be integrated in the bytecode statically. Reflection calls show low performance and are used very infrequently. Thus, this rewriting will not induce high additional overhead.

Content Providers Similar to reflective calls, we handle content providers. Providers must be accessed via content resolvers (see Section 3) which forward the operations to be performed on a content provider: `query`, `insert`, `update`, and `delete`. They throw security exceptions if required read or write permissions are not assigned to an application. As these methods specify the URI of the content provider we replace all operations by a static wrapper which passes their call to a monitor. It checks whether the operation is allowed before executing it.

Intents In general, intents are not problematic as they are handled in the central monitor of Android, i.e. the enforcement does not happen in the application. If an application sends an intent to a component which requires permissions an exception in the error log is generated if the application does not have this permission. The corresponding action is not executed but the application does not crash. Thus, our rewriting must cover situations in which only some instructions in specific execution contexts must not send or receive intents. The control over sending can be realized by wrappers handling the available API methods such as `startActivity`, `broadcastIntent`, `startService`, and `bindService`. The wrappers implement monitors which first analyse the intent to be sent. Depending on the target, the sending is aborted. By rewriting the manifest, we can control which intents a component can receive. This excludes explicit intents which directly address a application component. Here, we assume that the direct access of a system component to an application can be considered legitimate.

4.5 Application Management

We realize permission revocation by repackaging applications. First, our App manager obtains the manifest and `dex` (6) from the rewriter. For recovery, we first backup the old `dex` file and its corresponding manifest. All other resources, such as libraries, images, audio or video files, etc. are not backed up as they remain untouched. They are extracted from the original `apk` (7), signed with the URANOS key together with the new bytecode and manifest. The signed application is then directly integrated into a new `apk`. This process is slow due

to the `zip` compression of the archive. In the end, the application manager assists the user to deinstall the old and install the new application (8,9).

In the background we also deploy a dedicated update service. It mimics the update functionality of Android but also operates on the applications resigned by URANOS. We regularly query the application market for updates, inform the user about them, and assists the update process by deinstalling the old App, rewriting the new App, and installing it. Similarly, the App manager provides support for deinstallation and recovery.

5 Discussion

5.1 Performance

To assess the performance of our approach we downloaded over 180 popular applications from the Google Play Store. The URANOS App was adjusted in such a way that it automatically starts analysing and rewriting newly installed applications. Our benchmark measured the analysis time, i.e. the preprocessing of the `dex` (pre) and the execution context detection (det), and the rewriting time, i.e. the merging of wrappers (wrap), the rewriting of the resulting `dex` (rew), and the total time require to generate the final `apk` (tot). The analysis and rewriting phase were repeated 11 times for each App. The first measurement was ignored as memory management and garbage collection often greatly influence the first measurements and hard to reproduce as they heavily depend on the phone state. For the rewriting process, we always selected three random permissions to be disabled. If there were less permissions we disabled all. All measurements were conducted on a Motorola RAZR XT910, running Android 4.0.4 on a 3.0.8 kernel. Due to space restrictions this contribution only discusses a selection of applications and their performance figures. An overview of the complete results, a report on the impact of our rewriting on the App functionality, and the App itself are available at http://web.sec.uni-passau.de/research/uranos/.

Apart from the time measurements mentioned above Table 1 enumerates the number of plugins the application contains (#pl), the number of permissions requested (#pm), the number classes (#cl) in the `dex` and the size of the `apk`. In particular the `apk` size has a tremendous impact on the generation of the rewritten application due to APK compression. This provides potential for optimization in particular if we look at the rather small time required to merge the wrapper file of 81 kB into the complex `dex` structure and redirecting the method calls. This complexity is also reflected in the time for pre-processing the `dex` to extract information required to work on the bytecode.

We can also see that the number of classes and permissions included in an application influence the analysis time. Classes increase the number of administrative overhead in a `dex`. Thus their number also increases the effort to search for the appropriate code locations. Here, Shazam and Instagram are two extreme examples. In turn, the number of permissions increase the number of methods which have to considered during analysis and rewriting.

App	#pl	#pm	#cl	apk[MB]	pre[ms]	det[ms]	wrap[ms]	rew[ms]	tot[ms]
100 Doors	4	3	757	14.4	1421	356	1690	4277	9073
Angry Birds	10	6	873	24.4	1863	640	2308	5767	50408
Bugvillage	13	8	1127	3.1	1819	1214	3425	6832	18092
Coin Dozer	11	6	855	14.7	2028	788	2605	6457	56749
Fruit Ninja	8	8	1472	19.2	2520	1197	3657	7955	144374
Instagram	7	7	2914	12.9	5168	1906	8114	17031	39908
Logo Quiz	3	2	232	9.7	553	96	701	1939	7729
Shazam	8	13	2822	4.4	4098	3214	7837	15182	27263
Skyjumper	3	4	292	0.9	772	257	1222	2991	4106

Table 1. Selection of analyzed and rewritten applications

In our measurements, we do not include execution overhead. The time required for the additional instructions in the bytecode are negligible and within measuring tolerance. Thus, although the generation of the final `apk` is slow, our measurements certify that the analysis and rewriting on Android bytecode can be implemented efficiently on a Smartphone. While other solutions run off-device and focus on precision, such as the web interface provided by Woodpecker [9], URANOS can deliver timely feedback to the user. With this information he can decide about further countermeasures also provided by our system.

5.2 Limitations

As we have already stated above, our analysis uses approximations. In fact, P_M is an overapproximation of the permissions required by method calls, e.g. there may be methods in the bytecode which are never executed. Thus, the mere existence of API calls does not justify a permission assignment to an application. On the other hand $P_P \cup P_I$ is an underapproximation as we only consider strings as a means to communicate via intents or to access resources. There are numerous other ways for such operations, our heuristic does not cover.

Attackers or regular programmers can achieve underapproximations by hiding intent or provider access with various implementation techniques. In this case URANOS will alert the user that a specific permission may not be needed. The user will deactivate the respective permission and no direct damage is caused. Overapproximation can be achieved by simply placing API calls in the bytecode which are never executed. In this case, our analysis does not report the permission mapping to those *dead* API calls to be dispensable. Thus, the overapproximation performed in this round may give the user a wrong sense of security concerning the application. Therefore, URANOS also allows the deactivation of permissions in the hosting application and not only in the plugin.

Attackers may also hide plugin code by obfuscating it, e.g. by renaming all plugin APIs. In this case, URANOS will not detect the plugin. This will prevent the user from disabling permissions for this plugin. In this case, it is still possible to remove permissions for the whole application. Plugin providers which have an interest in the use of their plugins will not aim for obfuscated APIs.

5.3 Legal Restrictions

If software suffices the copyright law's fundamental requirement of originality it is protected by international and national law, such as the Universal Copyright Convention, the WIPO Copyright Treaty grant protection, Article 10 of the international TRIPS agreement of the WTO agreement, and the European Directive 2009/24/EC. In general, these directives prohibit the manipulation, reproduction, or distribution of source code if the changes are not authorized by its rights holder. No consent for modification is required if the software is developed using an open source software licensing model or if *minor* modifications are required for *repair or maintenance*. To achieve *interoperability* of an application even reverse engineering may be allowed. However, any changes must not infringe with the regular *exploitation* of the affected application and the *legitimate interest* of the rights holder.

URANOS cannot satisfy any of the conditions mentioned above. First of all, all actions are performed automatically. Thus, it is not possible to query the rights owner for his permission to alter the software. One may argue that URANOS rewrites the application in order to ensure correct data management. Unfortunately, the changes described above directly infringe with the interest of the rights holder of the application.

On the other hand one argue that a developer must inform the user how the application processes and uses his personal data as highlighted in the *"Joint Statement of Principles"* of February 22^{nd} 2012 and signed by global players like Amazon, Apple, Google, Hewlett-Packard, Microsoft and Research In Motion. However, current systems only allow an informed consent of insufficient quality. In particular when using plugins, a developer would need to explain how user data is processed. But developers only use APIs to libraries without knowing internal details. To provide adequate information about the use of data a developer would have to understand and/or reverse engineer the plugin mechanisms he uses. So, for most plugins or libraries, the phrasing of a correct terms of use is impossible. Yet, this fact does not justify application rewriting. The user can still refuse the installation. If, despite deficient information, he decides to install the software he must stick to the legal restrictions and use it as is.

In short: URANOS and most security systems which are based on application rewriting conflict with international and most national copyright protection legislation. This situation is paradoxical as such systems try to protect private data from being misused by erroneous or malicious application logic. Thus, they try to enforce data protection legislation but are at the same time limited by copyright protection laws.

6 Related Work

This section focuses on recent work addressing permission problems in Android. We distinguish two types of approaches: Analysis and monitoring mechanisms.

6.1 Permission Analysis

One of the first publications analysing the Android permission system is Kirin [8]. It analyzes the application manifest and compares it with simple and pre-defined rules. In contrast to URANOS, rules can only describe security critical combinations of permissions and simply prevent an application from being installed.

The off-device analysis in [4] is more sophisticated. It defines attack vectors which are based on design flaws and allow for the misuse of permissions. Chin et al. describe secure coding guidelines and permissions as a means to mitigate these problems. Their tool, ComDroid, can support developers to detect such flaws but it does not help App users in detecting and mitigating such problems.

This lack of user support also holds Stowaway [9]. This tool is focused on permissions which are dispensable for the execution of an application. Comparable to URANOS, Stowaway runs a static analysis on the bytecode. However, this analysis is designed for a server environment. While it provides better precision through a flow analysis, it can not correct the detected problems and the analysis times exceed those of URANOS by several magnitudes.

Similar to Stowaway, AndroidLeaks [11] uses an off-device analysis which detects privacy leaks. Data which is generated by operations which are subject to permission checks are tracked through the application to data sinks using static information flow analysis. AndroidLeaks supports the human analyst. The actual end user can not directly benefit from this system.

DroidChecker [3] and Woodpecker [12] use inter-procedural control flow analyzes to look for permission vulnerabilities, such as the confused deputy (CD) vulnerability. However, DroidChecker additionally uses taint tracking to detect privilege escalation vulnerabilities in single applications while Woodpecker targets system images. Techniques applied in Woodpecker were also used to investigate the functionality of in-app advertisement [13]. URANOS is based on an extended collection of advertisement libraries used in this work. Similar analytical work with a less comprehensive body has been conducted in [20].

6.2 Enhanced Permission Monitoring

An early approach which modifies the central security monitor in Android to introduce an enriched permission system of finer granularity is Saint [17]. However, Saint mainly focuses on inter-application communication. CRePE [5] goes one step further and extends Android permissions with contextual constraints such as time, location, etc. However, CRePE does not consider the execution context in which permissions are required. Similar holds for Apex [16]. It manipulates the Android core implementation to modify the permissions framework and also introduces additional constraints on the usage of permissions.

Approaches such as QUIRE [7] or IPC inspection of Felt et al. [10] focus on the runtime prevention of CD attacks. QUIRE defines a lightweight provenance system for permissions. Enforcement in this framework boils down to the discovery of chains which misuse the communication to other apps. IPC inspection solves this problem by reinstantiating apps with the privileges of their callers.

Both approaches require an OS manipulation and consider an application to be monolithic. This prevents them from recognizing execution contexts for permissions. The same deficiencies hold for XManDroid [2] which extends the goal of QUIRE and IPC inspection by also considering colluding applications.

Similar to IPC inspection and partially based on QUIRE is AdSplit [19]. It targets advertisement plugins and also uses multi-instantiation. It separates the advertisement and its hosting application and executes it in independent processes. Although mentioned in their contribution Shekhar does not aim at deactivating permissions in one part of the application or at completely suppressing communication between the separated application components.

Leontiadis et al. also do not promote a complete deactivation of permissions and separate applications and advertising libraries to avoid overprivileged execution [15]. A trade-off between user privacy and advertisement revenue is proposed. A separated permission and monitoring system controls the responsible processing of user data and allows to interfere with IPC if sufficient revenue has been produced. URANOS could be coupled with such a system by only allowing the deactivation of permissions if sufficient revenue has been produced. However, real-time monitoring would destroy the lightweight character of URANOS.

Although developed independently, AdDroid [18] realizes a lot of the ideas proposed by Leontiadis et al. AdDroid proposes specific permissions for advertisement plugins. Of course, this requires modifications to the overall Android permission system. Further, to obtain a privilege separation, AdDroid also proposes a process separation of advertisement functionalities from the hosting application. Additionally, the Android API is proposed to be modified according to the advertisement needs. It remains unclear how such a model should be enforced. The generality of URANOS could contribute to such an enforcement.

Two approaches which are very similar to URANOS are I-Arm Droid [6] and AppGuard [1]. Both systems rewrite Android bytecode to enforce user defined policies. I-Arm Droid does not run on the Android device and is designed to enforce developer defined security policies enforced by inlined reference monitors at runtime. The flexibility of the inlining process is limited as all method calls are replaced by monitors. Selective deactivation of permissions is not possible. The same holds for AppGuard. While it can be run directly on the device the rewriting process replaces all critical method calls. AppGuard compares to URANOS as it uses a similar resource and user friendly deployment mechanism which do not require root access on the device.

7 Conclusions

The permission system and application structure in today's Smartphones do not provide a good foundation for an informed consent of users. URANOS takes a first step into this direction by providing enhanced feedback. The user is able to select which application component should run with which set of permissions. Thus, although our approach can not provide detailed information about its functionality the user benefits from a finer granularity of permission assignment.

If in doubt, he is not confronted with a all or nothing approach but can selectively disable critical application components. The execution contexts we define in our work are general and can describe many different types of application components. Further, we neither require users to manipulate or root their Smartphones. Instead we maintain the regular install, update, and recovery procedures.

Our approach is still slow when integrating the executable code into a fully functional application. However, this overhead is not directly induced by our efficient analysis or rewriting mechanisms. In fact, we highlighted the practical and security impact of a trade-off between a precise and complete flow analysis and a lightweight but fast and resource saving syntactical analysis which can run on user device without altering its overall functionality.

Acknowledgements

The research leading to these results has received funding from the European Union's FP7 project COMPOSE, under grant agreement 317862.

References

1. Backes, M., Gerling, S., Hammer, C., Maffei, M., von Styp-Rekowsky, P.: AppGuard - Real-time policy enforcement for third-party applications. Tech. Rep. A/02/2012, Saarland University (2012)
2. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R.: XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks. Technical Report TR-2011-04, Technische Universität Darmstadt (Apr 2011)
3. Chan, P.P., Hui, L.C., Yiu, S.M.: Droidchecker: analyzing android applications for capability leak. In: Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks. pp. 125–136. WISEC '12, ACM, New York, NY, USA (2012)
4. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in android. In: Proceedings of the 9th international conference on Mobile systems, applications, and services. pp. 239–252. MobiSys '11, ACM, New York, NY, USA (2011)
5. Conti, M., Nguyen, V.T.N., Crispo, B.: CRePE: context-related policy enforcement for android. In: Proceedings of the 13th international conference on Information security. pp. 331–345. ISC'10, Springer-Verlag, Berlin, Heidelberg (2011)
6. Davis, B., Sanders, B., Khodaverdian, A., Chen, H.: I-arm-droid: A rewriting framework for in-app reference monitors for android applications. In: IEEE Mobile Security Technologies (MoST). San Francisco, CA
7. Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., Wallach, D.S.: QUIRE: Lightweight Provenance for Smart Phone Operating Systems. CoRR abs/1102.2445 (2011)
8. Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. In: CCS '09: Proceedings of the 16th ACM conference on Computer and communications security. pp. 235–245. ACM, New York, NY, USA (2009)
9. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: ACM Conference on Computer and Communications Security. pp. 627–638 (2011)

10. Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission re-delegation: attacks and defenses. In: Proceedings of the 20th USENIX conference on Security. pp. 22–22. SEC'11, USENIX Association, Berkeley, CA, USA (2011)
11. Gibler, C., Crussell, J., Erickson, J., Chen, H.: AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale. In: Proceedings of the 5th international conference on Trust and Trustworthy Computing. pp. 291–307. TRUST'12, Springer-Verlag, Berlin, Heidelberg (2012)
12. Grace, M., Zhou, Y., Wang, Z., Jiang, X.: Systematic Detection of Capability Leaks in Stock Android Smartphones. In: Proceedings of the 19th Network and Distributed System Security Symposium (NDSS) (Feb 2012)
13. Grace, M.C., Zhou, W., Jiang, X., Sadeghi, A.R.: Unsafe exposure analysis of mobile in-app advertisements. In: Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks. pp. 101–112. WISEC '12, ACM, New York, NY, USA (2012)
14. 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: Proceedings of the 18th ACM conference on Computer and communications security. pp. 639–652. CCS '11, ACM, New York, NY, USA (2011)
15. Leontiadis, I., Efstratiou, C., Picone, M., Mascolo, C.: Don't kill my ads!: balancing privacy in an ad-supported mobile application market. In: Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications. pp. 2:1–2:6. HotMobile '12, ACM, New York, NY, USA (2012)
16. Nauman, M., Khan, S., Zhang, X.: Apex: extending android permission model and enforcement with user-defined runtime constraints. In: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security. pp. 328–332. ASIACCS '10, ACM, New York, NY, USA (2010)
17. Ongtang, M., McLaughlin, S.E., Enck, W., McDaniel, P.D.: Semantically Rich Application-Centric Security in Android. In: ACSAC. pp. 340–349. IEEE Computer Society (2009)
18. Pearce, P., Felt, A.P., Nunez, G., Wagner, D.: AdDroid: Privilege separation for applications and advertisers in Android. In: Proceedings of AsiaCCS (May 2012)
19. Shekhar, S., Dietz, M., Wallach, D.S.: AdSplit: separating smartphone advertising from applications. In: Proceedings of the 21st USENIX conference on Security symposium. pp. 28–28. Security'12, USENIX Association, Berkeley, CA, USA (2012)
20. Stevens, R., Gibler, C., Crussell, J., Erickson, J., Chen, H.: Investigating user privacy in android ad libraries. In: IEEE Mobile Security Technologies (MoST). San Francisco, CA
21. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: IEEE Symposium on Security and Privacy. pp. 95–109 (2012)