Context-Aware Security Framework for Android

Contantin-Alexandru Tudorică
Automatic Control and Computers Faculty
University Politehnica of Bucharest
constantin.tudorica1305@cti.pub.ro

Laura Gheorghe
Research and Development Department
Academy of Romanian Scientists
Bucharest, Romania
laura.gheorghe@cs.pub.ro

***Abstract*—The popularity of mobile devices has increased and therefore they have become major targets for attacks, especially the ones that involve stealing private information. On Android, security policies are approved at install time and can't be changed afterwards. This behavior gives the decision power to the developer, although the user should be in charge of what is happening on their device. Users mostly find a need to change permissions depending on their context (location, network, time of day, etc.). This paper proposes a method of implementing Context-Awareness security policies on Android through an extensible Context-Awareness Security Framework that works with policy files and third party providers of contextual information.**

***Index Terms*—Android, Context-Aware, security, framework, permissions**

I. Introduction

Due to the increasingly evolving role that mobile devices have in our lives, they now store sensitive and private information that needs to be protected from malevolent actors that use malicious applications in order to leak private information.

Android Permissions are grouped into Android permission groups. Whenever an application is installed you are granting the application access to a group of permissions. Subsequent updates of that application will not trigger an approve prompt if the new permissions are part of preapproved groups. This allows a certain degree of privilege escalation. Also some APIs are pretty coarse, like the one that allows Internet access. You can't specify which domains or IP addresses the application is allowed to connect to.

There is a need for a finer grained control over Android permissions. Previous work, by M. Conti et al. [1], proved that a context aware security policy mapped on Android's existing permission system can be made. Rather than creating a generic way of defining contexts, they focused more on implementing flexible policies for permissions access control and only treated a single type of IPC (Interprocess communication) that can take place between two Android applications and did not look at securing the Android System Libraries.

Another approach was done by Arena et al. [2], where they provided a different experience by requesting the user's confirmation when a certain permission was used for the first time, mimicking the way permissions are granted on iOS.

This paper aims to provide a method for integrating context-aware security policies into the Android Framework, thus creating a security framework for Android. The reason for choosing context-aware security is that it greatly improves information security without having to involve the user and has the ability to create dynamic security policies.

The solution proposed in this work tries to take into consideration the vast ecosystem and uses frameworks and methods that are minimally invasive and have been proven to work on a large number of devices and Android versions.

There are a couple of problems with the existing solutions outlined above. The first problem is that they don't have an encompassing protection for all the existing Android API (Application Programming Interface) calls and can't allow fine grained access to certain permissions. The second problem is that they each define a restrained set of contextual information that their system supports and do not look at ways of allowing the contextual information to be extended.

Our solution takes another approach by constructing a framework that allows for the easy integration of third party applications that provide context information and designs a policy specification which delivers great flexibility by combining all the contextual information into permissions across the Android OS.

Specifying rules for context-aware security on Android devices is similar to specifying a set of Access Control Lists. One of the standards that are used to define ACLs in the industry is the Extensible Access Control Markup Language (XACML) standard [3]. SecuDroid[2] presents a way of extending the language in order to define context rules in it. The markup extension used by our solution, is based upon their work and extends it in order to be able to define specific contexts based upon our extensible context architecture.

The rest of the paper is structured as follows. Section II presents the existing work related to the design of context-aware solutions and Android security frameworks. Section III describes the architecture of a context-aware security framework and the policy definition mechanism. Section IV presents the implementation decisions and the frameworks that were used. Section V shows the test scenarios and results. Section VI presents the conclusions and possible ways of further extending our framework.

II. Background

A. Android Permission System

Android represents a privilege-separated operating system, therefore each application functions with a different system identity (Linux user ID and group ID)and also elements of the system are divided into distinct identities. In this manner, Linux separates the applications from each other and from the system.

Security features that are more refined are provided using a "permission" mechanism that implements restrictions on the particular tasks that a specific process can perform, and per-URI permissions for offering ad hoc access to specific pieces of data.

1) Security Architecture: One important design feature belonging to the Android architecture is the fact that no application has the authorization to run any operation that could affect any other one, including reading or writing the personal data of the user or another's application's files.

Every Android application functions in a process sandbox and must share resources and data by declaring the authorization they require for further abilities that are not offered by the basic sandbox. When statically declaring the permission they need, the Android system asks for the user's consent.

The application sandbox is independent from the technology that was used in order to build the application. For example, the Dalvik VM is not a security edge, any application having the possibility to run native code (see the Android NDK). Java, native, and hybrid applications are sandboxed in a similar manner and have the same degree of security from each other.

2) Application signing: All `.apk` files must be signed using a developer's certificate. This way, the author of the application is identified and it is possible for the system to approve or deny the request of an application to receive the same Linux identity as another one. It isn't necessary for the certificate to be signed by a certificate authority, being allowed for Android applications to use self-signed certificates. This type of permissions are named `signature-level` permissions.

3) Permission definition: Application permissions are defined in the application's manifest file named `AndroidManifest.xml`. Android permissions can be requested and also specified in the `AndroidManifest.xml` file. The operating system maintains a cache of these policies in the `/data/system/packages.xml` file which is used by the Package Manager Android system service.

Android permissions are of four types: normal, dangerous, signature and signatureOrSystem. The normal permissions are permissions that any application can request and are granted by default. Some of these permissions are `INTERNET` and `VIBRATE`, which control access to the Internet and respectively, access to the vibration function of a device. Dangereous permissions on the other hand, are the ones that could permit an application access to private information, or that could have an impact on the data stored on the device.

Dangerous permissions are grouped into permission groups. When the user approves a permission it actually sees the permission group name and icon. If in an update an application requires another permission from the same group, this permission is automatically granted. For example an application can request the `RECEIVE_SMS` permission that is the `SMS` category. The application can then be updated to also require the `SEND_SMS` permission since they are in the same `SMS` permission group the package installer will not request the approval of the new permission.

Applications can also define their own permissions. In order to do so an application must define a `permission-tree` which is a reverse domain path letting the OS know that all the permissions under that path are managed by the application. The application can also define permissions dynamically by calling `PackageManager.addPermission()`. After this, it must define a `permission-group`. The `permission-group` is similar to the dangerous permissions groups that are defined in Android. This is what the user will see in the installer application, this means that it is recommended to have an icon and label specified, besides the name. Finally a permission is defined using the `permission` tag. The permission's protection level can have one of the four values `normal`, `dangerous`, `signature` and `signatureOrSystem`. The `signature` protection level specifies that the permission is automatically granted to an application signed with the same developer certificate, while the `signatureOrSystem` also grants the permission to a system application.

B. XPosed Framework

For security reasons Android System Services are installed on a read-only partition that can be only updated through a Android system update.

The normal way of modifying Android Services is that of getting the open-source code and recompiling an new custom Android version. Another less invasive way of doing this is through the XPosed framework. The XPosed framework once installed allows a developer to write hooks into any Android Java application. The XPosed framework works by modifying the `/system/bin/app_process` and injecting itself into the Zygote process. From there, due to the nature of the Zygote and the way Android applications are created, inject hooks into any application that is started, including Android System Services. This is done by changing the hooked methods type to native and linking the method to a special native method that is part of XposedBridge, named `hookMethodNative`. This method passes the parameters it was called with, to the XposedBridge which then invokes our code.

III. Related work

The development of context-aware services has accelerated academic interest toward context-aware policies, especially the ones focused on mobile devices.

A. Security Policies

In order for access control systems to integrate context data into their policies they need a way to support it in their specification. The current language used for access control systems is named eXtensible Access Control Markup Language (XACML)[3]. Cheaito et al. [4] presented a specification that is based upon XACML and that extends it in order to add support for contextual information.

Another work treating access control policies was proposed by Li et al.[5]. It deals with integrating context and role into the access control of web services. Their model focuses on the

context in which the device is used in order to define access control policies. The context is defined as the environment and resources of the user and can be used to disable specific rights. For example, if the user logs in from a public WiFi he can view documents, but not modify them. Systems like this add an extra security layer on top of the already implemented security measures.

B. Context

Although Context-Aware Computing is a term used in the academic world, there is not one agreed upon definition of what the word context means. The Merriam-Webster dictionary defines it as "the interrelated condition in which something exists or occurs". There are multiple definitions of context depending on the domain it is used. Dey [6] defines context as "any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves". This is a proper definition that relates better to the mobile context that can be extracted from an Android device.

The factors that can make up a context are classified by Chen et al.[7] into four categories:

1) Computing context: network connectivity, communication cost and bandwidth, nearby resources
2) User context: user profile, user role, location, social situation
3) Physical context: lighting, noise, traffic condition, temperature
4) Time context: time of day, day of the week, timezone, season of the year

Location can be determined in multiple ways. Global Positioning System (GPS) can only be used for outdoor locations. For indoor locations there are multiple ways of identifying user location, either through Ultrasonic and radio signals, RADAR or WiFi signals and Bluetooth signals. Since there is no way to track the user both indoors and outdoors using the same method, multiple methods must be used in parallel which can lead to conflicts.

C. Android Security

The need for a secure mobile device has forced companies to create alliances and projects that aim to produce secure mobile devices. Some examples of these projects are OpenMoko [8] and OMTP [9]. A lot of the academic effort went into securing mobile phones by creating certification mechanisms for applications and implementing a permission system at the runtime level, for example Java MIDP 2.0 security model restricts application permissions.

Even though the above solutions solve the problem of filtering applications at install time, they don't implement any additional security enforcement at runtime.

CRePE [1] introduces a system that works by using the Android permissions and enforces context aware policies. It bases its context on different scenarios (e.g. the connection of a mobile phone to a workplace WiFi connection, a mobile device present in a certain geographical position using GPS), each one representing a context. Their solution modifies the `ActivityManagerService` and intercepts only `startActivity` Intents, which they consider as being the most used Intents for communicating between two applications. While this might be true, their solution does not enforce context-aware policies in the communication between an application and the Android System. They do consider the implications of securing system services by using their `ActionPerformer` service which, if the policy specifies, can shut down system services on the user's device.

SecureDroid [2] is another work that treats the problem of changing permissions after the application is installed. They present a system that asks the user for a confirmation the first time a permission is used, giving him the option to accept or deny the permission. The answer is remembered for the lifetime of the application. The user experience is similar to that of the iOS operating system and Android 6 when running applications developed for this version of Android specifically. Context information is not used in order to change the policies. They define their policies by extending XACML in a specific manner similar to our system. They also thought about implementing a priority between multiple actors that are interested in enforcing security policies on the device, thus they can have multiple policies which are evaluated in this order: the device manufacturer's policy, the network carrier's policy, third party policies like the ones a workplace might enforce on the device and then the user's policy. When it comes to enforcement SecureDroid injects itself in the PackageManager's `checkPermission` method in order to verify every call that is made to this endpoint. This looks like a good approach, just that it is still implemented on top of Android permissions, thus it lacks the granularity of control.

Apex [10] is a system that extends the Android permissions with conditional checks. Apex doesn't detect a user's context but allows limitations for a permission depending on the number of uses or time of day. For example, a limit can be imposed on the number of SMS messages that the application can send or limit an application's access to GPS information for just a specific period of the day. The only time these restrictions can be created is at install time, therefore not allowing the user to modify these permissions afterwards. They also created a domain specific policy language in order to implement more complex security policies.

All the previously mentioned solutions don't take into consideration that the majority of applications will crash if a permission is revoked after install. Even in Android 6 they request the permissions dynamically just for applications that are made specifically for that version of Android.

Our solution handles these cases by responding with fake data instead of denying the API call.

IV. Architecture

Our solution involves implementing an engine that dynamically modifies the permission database according to a

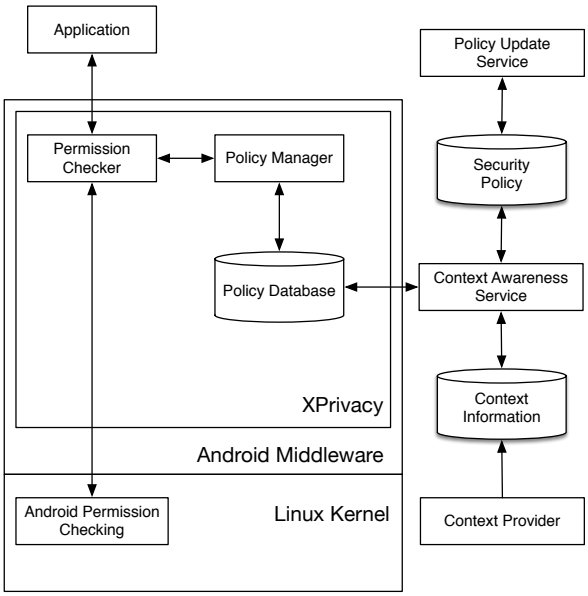

Figure 1: Architecture for a Context Aware Security Framework

security policy described in XACML. This engine uses context providing applications which are installed on the device and are used to determine the context of the device.

The architecture is comprised of the following components (as it can bee seen in Figure 1):

- **Policy Update Service** - a service that maintains up to date the Security Policy and the Context Providers specified in the policy
- **Security Policy** - a security policy written in XACML
- **Context Provider** - an application that provides contextual information to the Context Awareness Service
- **Context Awareness Service** - the service that modifies the Policy Database whenever it is necessary
- **Policy Database** - the database with security policies
- **Policy Manager** - the service that answers queries coming from the Permission Checker
- **Permission Checker** - the part of the system that checks a permission each time an API is called

In order to intercept all the calls made to the Android APIs we needed to modify all the API endpoints in the Android Middleware. Instead of modifying the Android code we choose to inject our code into the Android API processes with the help of the XPosed module, which allows dynamic code injection in process at start time. The XPrivacy framework already does this for all the hundreds of Android API endpoints including the ones that are protected under the same Android permission. It is compatible with Android API versions 15 to 19 which translates to Android 4.0.3 to 4.4.4. According to Google's own dashboard [11] this makes up to 60% of the Android devices at this time.

The contextual information supplied by a Context Provider is made out of facts. Each fact can be true or false. Each fact's name is prefixed with the component name of it's Context Provider, hence avoiding namespace collisions.

XACML is successfully used on RFID physical security locks[12]. In that case, the locks which represent the Policy Enforcement Point (PEP), can't query a Policy Decision Point (PDP) because they need to be able to operate in a distributed manner which means that each lock gets a set of cards that are allowed to access the door. The PDP system needs to compute this for every lock and then program the information into them. This system is similar to our case in Android, where XPrivacy maintains a cache of the current policy in each Android API endpoint which is refreshed every 15 seconds. The elimination of the cache is impractical because of performance reasons, otherwise each Intent would trigger another Intent to the XPrivacy service. Hence, we can assume that each endpoint of a system service has a list of applications which are allowed to talk to that endpoint. In order to achieve this we must pre-compute all the possible combinations of API categories and applications for the current context and update XPrivacy's database.

The architecture of the solution (presented in Figure 1) is centered around a Context Awareness Service that evaluates a policy based on contextual information, named facts, which is updated by Context Providers. The Security Policy is updated using the Policy Update Service. The Policy Update service is also responsible for keeping the Context Providers up-to-date considering the definitions they have in the security policy. The security policy is an XML file based on an extended XACML[3] specification.

V. Implementation

A. XPrivacy

Denying access by enforcing the policies at the Android Permission level will break a lot of applications. Work done by Marcel Bokhorst in XPrivacy [13] showed us that in order to enforce the policies and deliver a good user experience these applications need to receive data, just not real data.

For example, if you don't want to give access to the contacts to an application, you can't block the response, since this will make the application unresponsive or crash it. XPrivacy solves this problem by returning an empty list of contacts or a list of predefined fake contacts.

The way we can communicate with XPrivacy is by modifying the permission database, that it keeps in the `/data/system` folder. In order to access the database, our Context Manager must have elevated privileges.

XPrivacy's databases structure represented in Figure 2 is simple enough to be modified using basic information about the applications, like what User ID (UID) the Android Operating System assigns to each application at install time, which is obtained from the XML file `/data/system/packages.xml`.

The Context Awareness Service is responsible for updating XPrivacy's database. Because of the location of the database, the process needs to have the SUPERUSER privilege in order to access it. This is the only process of the system that requires elevated privileges.

B. System applications

There are two types of system applications in Android: applications that are installed on the system partition, these have access to privileged APIs in the Android, and applications that are run under the `system` user, which have higher privileges at the operating system level. Just a couple of system applications run under that user. Functioning under the `system` user requires the application to be signed with the same key as the Android image. This would hinder distribution of the application by requiring each user to install a custom version of Android in order to use the system.

In order to circumvent the requirement, all the modifications done to the XPrivacy database require a root user shell. The database is modified using the `sqlite3` helper from the command line.

C. Context Providers

Context Providers are normal or system applications that register facts and update their facts status to the Context Awareness Service.

Context Providers must have a BroadcastReceiver that answers to the `UPDATE_FACTS` Intent that is broadcasted by the Context Awareness Service in order to discover what facts are still available and which are not.

Context Providers must request the `com.tudalex.fingerprint.permission.CONTEXT_PROVIDER` permission in order to be able to communicate with the Context Awareness Service.

This allows for modularity and extensibility of the architecture, enabling the Context Providers to be normal Android applications.

All the facts are identified by the name and the package name of the application that provides the facts so that there can't be a collision made in the fact namespace.

In the policy oriented approach Context Providers are registered in the policy file and the facts that are exported specified in that file. This approach means that they don't have to implement a broadcast receiver in order to handle the `com.tudalex.fingerprint.UPDATE_FACTS` Intent.

In our implementation we created a couple of Context Providers. The demo context provider has a set of checkboxes each representing a fact that it exports. This Context Provider was used to simplify development and testing of the framework.

The other Context Provider has a list of trusted Wi-Fi SSIDs and monitors any change to the network status in order to update it's fact. This is done by listening, using a BroadcastReciever, for the `android.net.wifi.STATE_CHANGE` Intent. The Intent it receives carries with it the current network information, from which you can extract the status of the network which can be connected, connecting, disconnecting and disconnected. It checks that the state is connected and then compares the SSID to the ones it has stored. For storing trusted SSIDs the Context Provider uses the SharedPreferences Android API. This API allows for simple key-value storage. Because accessing these preferences takes a long time since they are stored on disk, a cache was created around the SharedPreferences API. We are aware that a WiFi can't be trusted just by it's SSID, but we consider this provider as being a practical example of the extent in which the system can be used.

D. XACML policy

In order to include contextual information in the XACML policy, it needs to be extended with tags that contain this information. Based upon the work on SecuDroid we apply several modifications to a XACML policy. An example of an extended XACML policy can bee seen in Listing 1.

```
<policy-set>
  <policy combine="deny-overrides">
    <target>
      <subject attr="id" match="com.company.
    exampleApp"></subject>
    </target>
    <rule effect="deny">
      <condition>
        <resource-match attr="api-category" match="
    internet"/>
        <context-match attr="id" match="com.
    contextProvider.Wifi"/>
      </condition>
    </rule>
    <rule effect="permit"></rule>
  </policy>
  <context-set>
   ...
  </context-set>
</policy-set>
```

Listing 1: Sample policy markup

The policy set has one `policy` tag which uses the `deny-overrides` algorithm to combine the rules, this algorithm signifies that if a rule with deny can be applied then that one takes precedence over any permit rule that is also valid. It also contains a target specified by an application ID, which must match `com.company.exampleApp`. The first rule has a set of conditions which are evaluated using a logical AND operator between them. The first condition matches all APIs in the Internet category, while the second rule matches the context `com.contextProvider.Wifi`. So that if the application `com.company.exampleApp` tries to access the Internet while being connected to Wifi, it will be blocked. The second rule specifies that in other cases the application should have access to the Internet APIs.

```
<context-set>
<context name="com.company.TestingEnviroment">
<rule attr = "com.company.Wifi.RestrictedWifi" match
    ="true"/>
<rule attr = "com.company.Exploit.CVE-2015-6638"
    match="false"/>
</context>
```

```
<context-provider package-name="com.company.Wifi"
    signature="ab823f..." source-url="https://....">
</context-set>
```

Listing 2: Sample context markup

In Listing 2, there is the description of a `context-set` which specifies the contexts defined in a `policy-set`. Each context is made out of multiple rules which are evaluated with a logical AND between them. Each rule matches a specific fact that is provided by a Context Provider and a specific value that each one has to have (*true* or *false*). Each fact is named by appending it to the Context Provider's application ID. A `Context-Provider` specifies a Context Provider, with its application ID, its signature and the URL where the Update Service can fetch the latest version of the Context Provider in case it is not installed or the signatures don't match.

Our extensible way of accepting contexts in which no verification is carried on the Context Providers leaves the system exploitable, because another application can be installed which has the same package name, since there is nothing in Android to enforce and validate, just certain developers. In order to check if a Context Provider is trusted, each Context Provider must be validated by suppling a package signature. If the package signature does not match the provider, it is treated as if it was not installed.

E. Context Awareness Service

The Context Awareness Service manages the Context Providers. The fact handling was taken out of it so that it doesn't require to be a system application. Also this allows for an extensible architecture in which any application can provide facts to be included in context definitions.

The Context Awareness Service is responsible of also processing the policy. It uses a simple boolean algebra to parse rules and does not handle conflicts. Taking into consideration the amount of processing necessary to handle conflicts in policies and due to the model of XPrivacy's database (presented in Figure 2) and architecture, we can't have an online PDP engine, therefore the Context Awareness Service must evaluate all the rules in the XPrivacy database against the policy for each change in the contextual information it has.

If the Context Provider is not installed on the device, the policy must specify a URL from which the Context Provider can be obtained.

If a Context Provider can't be obtained and it is not installed, then the policy can't be processed. We can't make any safe assumptions regarding the context, because policies can depend on the presence or absence of a context. Therefore, the old policy remains in place.

The reason why the policy is not stored in memory is because there is no reliable way of doing it in Android and also because of a design decision that was done in the beginning, of having the Context Awareness Service be an IntentService. This means that each time the service is called, a process is spawned and after it finishes processing the Intent it gets destroyed. This is the recommended way[14] of implementing a service which responds to Intents in Android and also helps the framework consume less memory since it doesn't have a running service in the background all the time. If we were to implement the Context Awareness Service as a normal Android service there are no guarantees that Android will keep the service running if it needs more memory.

Because of the extension of the language and the fact that Android runs on an older version of Java, other implementations of an XACML PDP[15] are not easy to modify for our use case, so we wrote an evaluating engine ourselves.

The evaluation engine makes a few assumptions:

- A rule contains at least one `rule-match` element or no elements.
- A `rule-match` element can only match the `api-category` attribute, on one of the supported API categories.
- Policies do not conflict.

These assumptions are made such that the engine doesn't have to deal with corner cases such as a policy that doesn't specify any API to restrict. Hence, also speeding up the execution by not having the evaluation engine verify if each API category is in fact a valid XPrivacy API category.

After parsing the policy the first thing the engine does, is that it triggers and waits for all the Context Providers to be available. A Context Provider is an Android application that implements a specific API which provides the Context Awareness Service with its context information.

In order to bypass the normal installation screen that an Android device has, the applications are installed using adb from a root shell (e.g. `adb install -r app.apk`).

The engine then goes over all the possible combinations of application and category ID's and evaluates if access to that category of endpoints is permitted or not. This information is written to the XPrivacy database, thus enforcing the new security constraints.

Making sure that an attacker can not cause a Denial of Service attack by blocking the possibility of obtaining a Context Provider does not make the scope of this paper.

F. Known limitations

One of the limitations of XPrivacy is that in order to prevent excessive use of its database, it has two layers of cache: a layer in the XPrivacy Service and another one in the code that is injected in each application.

The XPrivacy Service acts as a proxy for the database and it is called by all the injected code. This cache can be flushed by sending an Intent with the action `biz.bokhorst.xprivacy.action.FLUSH`.

The second cache present in the injected code can't be flushed and it has a cache time hard coded to 15 seconds.

This means that it might take up to 15 seconds for the modifications to be seen. This applies only to applications that are already running.

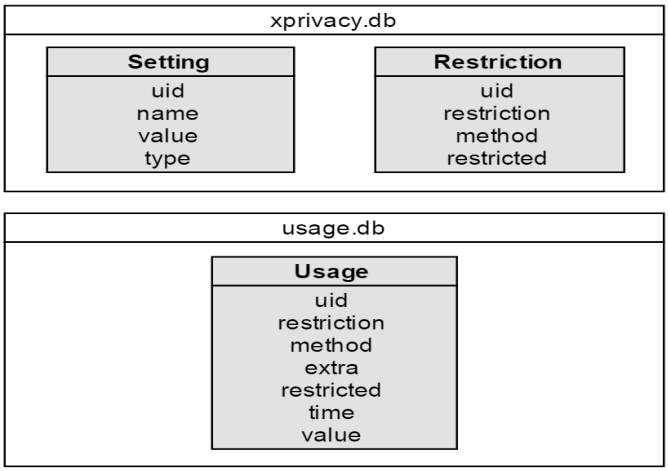

Figure 2: XPrivacy databases

G. Secure Intents

The problem of communicating using Intents in Android is that they can't be verified. Intents, in the form that they arrive in Java do not contain information pertaining to who sent them that can be verified and not spoofed. Even declaring an `intent-filter` attribute will not make Android filter explicit intents, just the implicit ones. Since the permissions can be easily approved by the user, we don't consider this a safe approach to securing the communication between a trusted Context Provider and the Context Awareness Service, hence we investigate other ways of verifying the sender. Some of the possible Intents also lack reliable authentication that can't be spoofed.

1) startActivityForResult: The most simple way to verify if an Intent comes from the corresponding application is to only accept Intents that require a result. These Intents have a method `getComponentName`, that returns the name of the Android component that sent the Intent. A simple check can validate an Intent against a known list of components that are allowed to access this BroadcastReceiver.

This can be easily implemented starting from the policy, since the policy needs to explicitly define each Context Provider as it can be seen in Listing 2.

Considering the way that our Context Awareness Service is created as an IntentService this method does not work for us, and we don't want to make a `startActivityForResult` call that will launch a new Android Activity on the screen.

2) TrustedIntents: TrustedIntents [16] is a library created by the Guardian Project that aims at fixing exactly our problem. TrustedIntents builds upon the `startActivityForResult` solution and adds an extra verification, by comparing the calling application's signature to a list of pinned signatures.

Though useful in some use cases, TrustedIntents is not useful for our approach since we don't want to use the `startActivityForResult` call for communicating with the Context Awareness Service which is implemented as an IntentService, not as an Activity.

3) Signing Intent data using HMAC: A simple solution of verifying an Intent is to sign it. By applying a HMAC (keyed-Hash Message Authentication Code) to the name of the Context Provider concatenated with name of the fact and its value, we can create a signature of the Intent, which can be easily verified at the other end. This does not require the use of `startActivityForResult`, thus leaving us free to implement the Context Awareness Service as an Intent Service [14], which is only present in memory when it has Intents he needs to respond to. In order to implement this, another attribute must be added to the XACML specification: a key for each Context Provider. An example specification can be seen in Listing 2.

The only way an Intent can be compromised is if a malicious application has the ability to intercept the security policy and extract the secret keys for each application.

This can be improved by using Public-Private Cryptography and delivering only the public key in the security policy. This allows for data to be signed using the Context Provider's private key and verified by the receiver using the public key that is listed in the security policy.

But even this way of signing Intents does not prevent Intent sniffing. The only way to truly prevent it is for the Context Provider to have the public key of the Context Awareness Service and initiate a Diffie-Hellman Key Exchange in order to agree on an encryption key that will be used to encrypt Intent data.

```
<context-set>
 ...
<context-provider package-name="com.company.Wifi"
     signature="ab823f..." source-url="https://...."
     secret_key="A8F6D..." >
</context-set>
```

Listing 3: Sample context-provider markup that uses a signature attribute

H. Rule evaluation

The rules are evaluated using a backtracking approach, looking into all the possible conditions. More complex rule evaluation can be done with a constraint solver, like Z3. Running Z3 on Android proved to be a challenge due to differences in the Java Native API which is different in Android compared to pure Java.

VI. Results

A. Setup

The development has been done on a Nexus 7 device running Android 4.4.2. This represents the latest Android version supporting the XPosed framework. In order to install the XPosed framework, the Android device must have the Android.SUPERUSER permission. The XPosed framework requires it so that it can inject the modules in the Android Zygote.

In order to automate tests on the Android device, the `monkeyrunner` framework was used. The monkeyrunner framework simplifies interaction with the device.

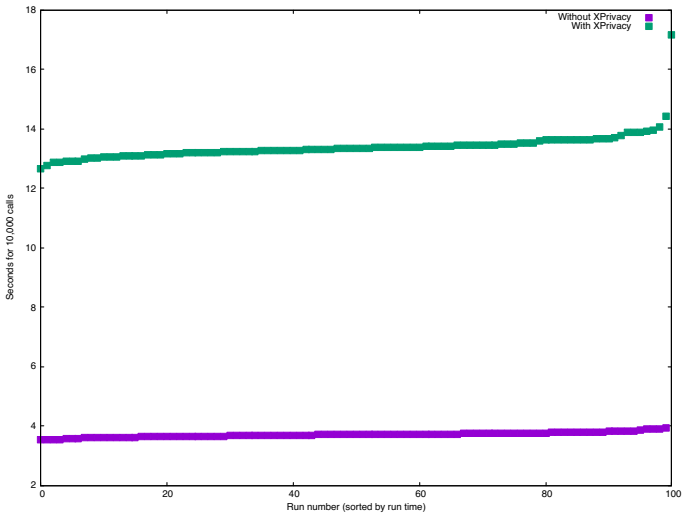

Figure 3: Intent benchmark

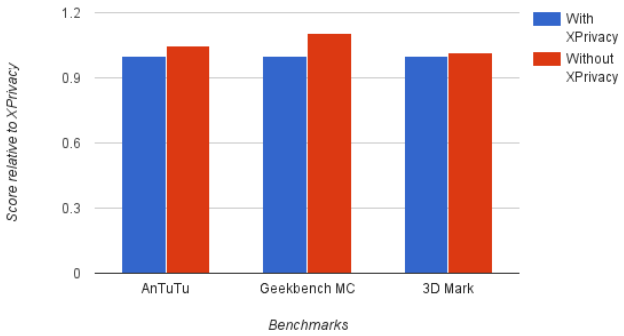

Figure 4: Popular benchmarks, average over 10 runs

B. Scenarios

Using the `monkeyrunner` framework, we have created a real life testing scenario of normal usage of a device that consisted of opening the GMail application, clicking on the first email, then opening the Chrome browser and navigating to `www.facebook.com`. The main problem was that the framework did not have any possibility of waiting for an action to happen. The only metric we could obtain is whether the final image after passing through each of the steps was the desired one or not. Given the dynamic nature of an Android device this benchmark was pretty flaky, for example notifications being received would make the image to be different from the reference. We decided to not purse this method any further.

Since the framework does not have a direct impact on compute performance, a good benchmark would be to evaluate added power consumption over a period of time of normal use. In this case, the `monkeyrunner` can come in handy. We created a scenario of looping through four applications: GMail, Chrome, Facebook and 3DMark, changing the application every minute while keeping the screen awake and starting with a full powered battery. Due to the high variation in results, we did not include the results of this benchmark in the paper.

The most important aspect of any security framework is the impact it has on the system. In order to measure the impact our framework has on the system, a simple scenario is constructed in which an application requests access to network settings. Because accessing the network settings takes less than a millisecond, hence being hard to measure, this approach evaluates how long it takes to request them 10.000 times.

C. Results

As it can be seen in Figure 3, the XPrivacy framework increases the time it takes by a factor of 3.6. This is much more than it was expected and might be a significant problem that needs to be overcome in order for this approach to be viable.

Since the previous benchmark only measures the time it takes to do an Intent to a system service, it shows a skewed view of how well an Android application run, because this penalty is only applied to calls made to system services which should not be as many as in our benchmark in a real application. The sole purpose of the benchmark is to show the overhead involved in communicating with system services when using our solution.

There was no change in policies so there was no overhead of updating the XPrivacy database while the benchmark was running.

During benchmarking, every 15 seconds our benchmark would have a spike in execution speed. This spike is due to the way XPrivacy caches information. XPrivacy injects into the Zygote process, therefore it then can inject custom code for each Android system service. Each injected code maintains its policies in a 15 second cache in order to prevent it from querying a SQLite database for each Intent. This has the effect that any policy change will take up to 15 seconds to be implemented on the device.

Another approach of testing our solution was through the use of popular benchmark software already available for the Android platform. Each benchmark was run 10 times with our system and 10 times without, an average of their scores was calculated. Because the benchmarks represent their results in their own defined metric, we took the results without XPosed as a baseline for each benchmark and only calculated the difference as a percentage to the baseline.

In Figure 4, we can see that the 3Dmark benchmark is less than 2% slower when using XPosed and XPrivacy and AnTuTu benchmark is only 4% slower. These tests were obtained as the average scores over 10 consecutive runs of the benchmarks. An interesting fact is that AnTuTu benchmark obtained the same scores after each run, denoting the fact that it probably already does multiple runs internally before coming up with the score. The outlier is GeekBench which is around 10% slower.

The 3DMark benchmark used is called Ice Storm. IT uses OpenGL ES2.0. It includes two Graphics tests to measure GPU

performance and a Physics test to stress test CPU performance.

The Geekbench version we used only tests single core performance of the device. Further testing is necessary to figure out if the results are reproducible on other more powerful hardware or what does the Geekbench benchmark specifically do in order to be affected by XPosed and XPrivacy.

AnTuTu benchmark similar to the 3DMark benchmark tests both the GPU and CPU performance of the device.

VII. Conclusion and Further Work

In this paper we have proposed a solution for implementing a context-aware security framework in Android. The framework's engine can implement a XACML policy into rules for the XPrivacy framework. The XACML policy has been extended to contain constraints related to contextual information. We also investigated a user oriented approach in which the user has to manage the security policy.

This approach minimizes the changes that need to be done to the operating system and has an extensible design by offering a way to define a context aware policy. Such a provider can be used by third party applications that are authorized to configure contexts and policies.

As more and more companies adopt a Bring your own device (BYOD) policy there will come a need for centralized control of application privileges and ways to enforce them on the managed devices. As it could be seen in the results section the performance of the system is quite good, which allows for this approach to be used also on low end devices.

The system could be further pursued to be integrated directly in Android version 6 and newer where we would not be forced to use XPrivacy and should have little impact on the system.

Acknowledgment

The work has been funded by the program Partnerships in priority areas – PN II carried out by MEN-UEFISCDI, project No. 47/2014.

References

[1] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. CRePE: Context-related policy enforcement for android. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6531 LNCS, pages 331–345, 2011.

[2] Valerio Arena, Vincenzo Catania, Giuseppe La Torre, Salvatore Monteleone, and Fabio Ricciato. SecureDroid: An Android security framework extension for context-aware policy enforcement. *2013 International Conference on Privacy and Security in Mobile Systems (PRISMS)*, pages 1–8, June 2013.

[3] OASIS. Xacml. http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html. Accesed: 2014-12-30.

[4] Marwan Cheaito, Romain Laborde, François Barrère, and Abdelmalek Benzekri. An extensible XACML authorization decision engine for context aware applications. In *2009 Joint Conferences on Pervasive Computing, JCPC 2009*, pages 377–382, 2009.

[5] Xu Feng, Lin Guoyan, Huang Hao, and Xie Li. Role-based access control system for web services. In *Computer and Information Technology, 2004. CIT'04. The Fourth International Conference on*, pages 357–362. IEEE, 2004.

[6] Anind K. Dey. Understanding and using context. *Personal and Ubiquitous Computing*, 5:4–7, 2001.

[7] Guanling Chen and David Kotz. A Survey of Context-Aware Mobile Computing Research. Technical report, Dartmouth College Hanover, 2000.

[8] Openmoko Project. Openmoko. http://www.openmoko.org. Accessed: 2014-12-30.

[9] OMTP Project. Omtp. http://www.omtp.org. Accessed: 2014-12-30.

[10] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex : Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. *ASIACCS '10 Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 328–332, 2010.

[11] Google. Dashboards — Android Developers. https://developer.android.com/about/dashboards/index.html. (Visited on 02/03/2015).

[12] DS Kim, TH Shin, and JS Park. Access control and authorization for security of RFID multi-domain using SAML and XACML. ... *Intelligence and Security, 2006* ..., pages 587–590, 2006.

[13] Marcel Bokhorst. Xprivacy. http://www.xprivacy.eu. Accesed: 2014-12-30.

[14] Google. Running in a Background Service — Android Developers. http://developer.android.com/training/run-background-service/index.html.

[15] AT&T. AT&T XACML 3.0 Implementation. https://github.com/att/XACML. Accessed: 2016-02-22.

[16] Hans-Christoph Steiner. Improving trust and flexibility in interactions between Android apps. https://guardianproject.info/2014/01/21/improving-trust-and-flexibility-in-interactions-between-android-apps/, 01 2014.