Specifying the Android Permission System
A Z approach

Mohammed El Amin TEBIB Pascal ANDRE

July 22, 2021

This document present a first attempt to specify with the Z notation the permission access control of Android using Z. The case study is not detailed here. The proof obligations will be explored using the Z/Eves tool.

1 Specification principles and method

Developping android applications is based on Java programing language using the android software developpement kit (SDK). Once developed, Android applications are compiled into Dalvik bytecode, then be packaged through a app store such as Google Play. The android permission model significantly evolved since the 4 version released on 2016.

The goal of this section is to specify the main concepts of an Android application and the permission management. We formalize the concepts through a formal Z description presented in Section 2.1. This formal presentation is followed by a security meta model that will be used in Section 2.2 to extract the security features from the android application under analysis.

2 Formal specification

The goal of this section is to figure out the main concepts composing an Android application and to formalize these concepts in order to set the properties we ought to verify in Section 3 and check on Android apps by implementing the rules in OCL. We assume the reader being not familiar to the Z notation [9]. First, we define the Z schema for data and state modelling.

2.1 App Part specification

We consider Android applications made of *Components* that communicate via *Intents*. The following Z basic types assume the existence of abstract sets (the values are not known)

$$[COMPONENT, APPLICATION, INTENT]$$

Basically, components are categorized into two families : 1) foreground components such as activities and 2) background components such as Service, Broadcast Receivers, and Content Providers.

- Activity *Act*. An activity is the starting point of an Android application (like the main class in Java). During its execution, it represents the graphical interface that the user will interact with to perform different tasks.
- Service *Serv*. Services are used to handle background processing.

- Broadcast Receiver BrC. Communications between applications and OS are managed by Broadcast Receiver, it could also be used to manage communications between components of the same application. The messages transmitted between components in an Android application are called Intents.

- Content Provider $CProv$: It manages the data layer of an Android application, and shares between applications a content that is previously defined. The access to this content is secure and well controlled.

Based on their role during the communication. We distinguish two kinds of $Components$: (1) Active Component $CompAct$ that can send and receive communication requests such as: Act, BcR, and $Serv$ and (2) Passive Components $CompPas$, which can only receive communication requests like $CProv$. Some components can be exported ($expComp$).

SComponents

$$Components : \mathbb{P}\, COMPONENT$$
$$CompAct, CompPas : \mathbb{P}\, COMPONENT$$
$$FgComp, BgComp : \mathbb{P}\, COMPONENT$$
$$Serv, Act, BcR, CProv : \mathbb{P}\, COMPONENT$$
$$expComp : \mathbb{P}\, COMPONENT$$

$$\langle CompAct, CompPas \rangle \text{ partition } Components \land$$
$$\langle FgComp, BgComp \rangle \text{ partition } Components$$

$$CProv \subseteq CompPas \land Serv \subseteq CompAct \land$$
$$Act \subseteq CompAct \land BcR \subseteq CompAct \land expComp \subseteq Components$$

$$Serv \cap Act = \emptyset \land Serv \cap BcR = \emptyset \land Act \cap BcR = \emptyset$$

A partition $\langle A, B \rangle$ partition AB means $A \cup B = AB \land A \cap B = \emptyset \land A \subseteq AB \land B \subseteq AB$

Intents are the main Android concepts to perform inter-app communications, when the interacting components share the same application context and intra-app communications to allow the interaction between components of different applications. Intents are said to be implicit ($IntentImpl$) when the callee is not specified once the intent is broadcast or explicit ($IntentExpl$) when the callee is explicitly specified.

SIntents

$$Intents : \mathbb{P}\, INTENT$$
$$implIntent, explIntent : \mathbb{P}\, INTENT$$

$$implIntent \subseteq Intents \land explIntent \subseteq Intents$$

$$implIntent \cap explIntent = \emptyset$$

Next schema provides relations between applications, components and intents. Applications are made of components ($cApp$), which can be part of several applications, and intents ($iApp$). Components can be source or target of intents ($csrcInt$, $ctargInt$).

$$
\begin{array}{|l}
\textit{AndroidApps} \\
\hline
\textit{SComponents} \\
\textit{SIntents} \\
\textit{Applications} : \mathbb{P}\, \textit{APPLICATION} \\
\textit{cApp} : \textit{Applications} \leftrightarrow \textit{Components} \\
\textit{iApp} : \textit{Applications} \leftrightarrow \textit{Intents} \\
\textit{csrcInt} : \textit{Intents} \leftrightarrow \textit{Components} \\
\textit{ctargInt} : \textit{Intents} \leftrightarrow \textit{Components} \\
\textit{interactions} : \textit{CompAct} \leftrightarrow \textit{Components} \\
\hline
\mathrm{dom}\, \textit{ctargInt} \subseteq \mathrm{dom}\, \textit{csrcInt} \\
\forall\, i : \textit{INTENT} \mid i \in \mathrm{dom}\, \textit{ctargInt} \bullet \\
\quad (\forall\, c : \textit{COMPONENT} \mid c \in \textit{ctargInt}(\!|\, \{i\} \,|\!) \bullet \\
\qquad c \notin \textit{csrcInt}(\!|\, \{i\} \,|\!))
\end{array}
$$

$R : A \leftrightarrow B$ means that R is a binary relation between sets A and B. The domain dom (resp. range ran) of R is the subset of A (resp. B) which is involved by the relation R and $\mathrm{dom}\, R \subseteq A$ (resp. $\mathrm{ran}\, R \subseteq B$). The expression $R(\!|\, subA \,|\!)$ denotes the set of images of $subA$ a subset of a ($subA \subseteq A$) by the relation R.

The predicate $\mathrm{dom}\, \textit{ctargInt} \subseteq \mathrm{dom}\, \textit{csrcInt}$ means that each communication must have a sender. The second predicates means that the receiver cannot be the sender.

The first two lines *SComponents* and *SIntents* are Z inclusions of the (named) schema declarations. Inclusion enables complex definitions in a modular way. It can be seen as a kind of inheritance in object orientation.

We use Z shortcut notations to simplify the predicate part of the Z schemas. For example, $\textit{interactions} : \textit{CompAct} \leftrightarrow \textit{Components}$ is equivalent to

$$
\begin{array}{|l}
\textit{interactions} : \textit{COMPONENT} \leftrightarrow \textit{COMPONENT} \\
\hline
\mathrm{dom}\, \textit{interactions} \subseteq \textit{CompAct} \land \\
\mathrm{ran}\, \textit{interactions} \subseteq \textit{Components}
\end{array}
$$

2.2 Security & Permission Part specification

The sensitive data transmitted between apps and components via intents are protected through Permissions. The Android permission system verifies if the app or the service is allowed to access the requested functionality. This decision is defined by the developers at coding stage.

In Android applications, *Permissions* are declared in the manifest.xml configuration file, and required at different stages: (1) system APIs interactions, (2) database access, (3) message passing system via intents, (4) invocation of specific protected methods in public APIs and (5) content provider data access. They also have different protection levels:

- A *normalPerm* permission is the default value. It defines a permission with low risk to the system or other applications. Permissions with protection level normal are automatically granted

without requiring user confirmation

- A *dangPerm* (dangerous) permission gives access to user data or some form of control over the device. e.g: SMS, Camera, Location...
- A *sigPerm* (signature) permission is only granted to applications that are signed with the same key as the application that declared the permission.
- A *sosPerm* Signature or system permissions are similar to signature permissions, but are also granted to packages in the Android system image.

In the following Z schema, we enrich the *AndroidApps* schema with permissions according to the above textual description.

The Z basic types assumes the existence of an abstract set of permissions and the free type provides an enumeration of permission categories.

$$[PERMISSION]$$
$$Category ::= normal \mid dang \mid sig \mid sos$$

Permissions are declared per application (*permApp*), components (*permComp*) and intents (*permIntent*). Component permissions are declared by components (*cpermDec*) and required by active components (*cpermReq*). Passive components provide read and write access permissions (*rpermDec*, *wpermDec*).

AndroidPerm

$$AndroidApps$$
$$Permissions : \mathbb{P}\, PERMISSION$$
$$decPermLevel : PERMISSION \pfun Category$$
$$permLevel : PERMISSION \fun Category$$
$$permApp : Applications \rel Permissions$$
$$permComp : Components \rel Permissions$$
$$permIntent : Intents \rel Permissions$$
$$cpermDec : Components \rel Permissions$$
$$cpermReq : CompAct \rel Permissions$$
$$rpermDec : CompPas \rel Permissions$$
$$wpermDec : CompPas \rel Permissions$$

$$\operatorname{dom} decPermLevel \subseteq Permissions \land \operatorname{dom} permLevel \subseteq Permissions$$
$$permLevel = (Permission \fun \{normal\}) \oplus decPermLevel$$
$$\operatorname{ran} permApp \cup \operatorname{ran} permComp = Permissions \land$$
$$\operatorname{ran} permIntent \subseteq Permissions$$
$$permComp = cpermDec \cup cpermReq$$
$$(rpermDec \cup wpermDec) = (CompPas \rhd cpermDec)$$

Each permission belongs to one *Category* by *permLevel* ($\fun$ is a total function). This category can be explicitly defined by *decPermLevel* (it is optional since $\pfun$ is a partial function) and only the considered permissions have one $\operatorname{dom} permLevel \subseteq Permissions$. If no category is explicitly given, the level is *normal* ($(Permission \fun \{normal\}) \oplus decPermLevel$).

All considered *Permissions* are associated to applications or components by ($permApp \cup \operatorname{ran} permComp = Permissions$). Intents permissions are related to them by $permIntent \subseteq Permissions$.

Component permissions are declared or required ($permComp = cpermDec \cup cpermReq$) but only active components require permissions. Passive components provide read and write access permissions ($rpermDec, wpermDec$) of the declared permissions of passive components ($(rpermDec \cup wpermDec) = (CompPas \lhd cpermDec)$).

> Recall $R : A \leftrightarrow B$ is a binary relation between A and B. The expression $subA \lhd R$ denotes the restriction of R to a sub domain $subA$ ($subA \subseteq A$); it results in the set of couples $(a, b) \in R$ such that $a \in subA$.

The above formal description of the specification system covers a minimal number of concepts that should be extended in a future work. However, it is sufficient to formally specify security properties as described in the next section.

3 Properties

In this section we present the specification of security properties in the context of the abstract Model presented in Section **??**. This is definitely not an exhaustive set of properties but an excerpt to illustrate our approach. The verification of these properties will help developer to identify the potential permission violation at an early stage of development. We define static security properties in Section 3.1 and discuss dynamic ones in Section 3.2. The properties are presented as theorems but must be stored as invariants to be checked by Z-EVES. We provide both representations.

3.1 Static Security Properties

In this work we focus on static security features including the component invocation, content access, permission naming. Referring to standards such as CIA (Confidentiality, Integrity, Availability) or AAA (Authorization, Authentication, Accounting) [3]. We target confidentiality and authorization properties. We refer mainly to the experimentation study of Jha et al. [7] that investigated the mistakes committed by developers.

3.1.1 Potential violation related to permission naming

Potential violation is related to a well known security issue so called *Naming Conflicts*, which was always indicated as common mistake committed by developers and classified as unintentional case due to omission or ignorance reasons [7].

Definition 3.1 (Type Conflict (P1)) *A type conflict occurs when the same permission is classified twice in the same application, even by default classification.*

This issue was pointed out by [5] following a formal analysis of the android system based on High Level Petri Nets [1]. Since the release of version 6 of Android, *Normal* permissions are granted automatically during the installation time, whereas dangerous ones are granted dynamically during the execution time after a checking process. Having a permission *perm* which is declared as a normal and dangerous in the same application could lead to private data ex-filtration. when an application could have a direct access to such resource protected by a dangerous level permission without users awareness and approval.

[1] https://www.techfak.uni-bielefeld.de/ mchen/BioPNML/Intro/pnfaq.html

theorem P1

$$\forall\, AndroidPerm \bullet permLevel \rhd \{normal\} \cap permLevel \rhd \{dang\} = \emptyset$$

*P*1

$$AndroidPerm$$

$$permLevel \rhd \{normal\} \cap permLevel \rhd \{dang\} = \emptyset$$

This is proved by the invariant preservation since *permLevel* is a function.

3.1.2 Component Invocation

This property aims to check the correctness of components invocation in order to protect sensitive data manipulated by a component. As a component could perform sensitive actions, it should be protected against unauthorized accesses.

Definition 3.2 (Permission Consistency of Component Invocation (P2)) *Two interacting components must have compatible permissions. Every required permission of a called component must be fulfilled by the caller component.*

theorem P2

$$\forall\, AndroidPerm;\ ca, cp : COMPONENT \bullet \forall (ca, cp) \in interactions \bullet cpermReq(\!|\ \{co\}\ |\!) \subseteq cpermDec(\!|\ \{ca\}\ |\!)$$

*P*2

$$AndroidPerm$$
$$ca, cp : COMPONENT$$

$$(ca, cp) \in interactions \Rightarrow cpermReq(\!|\ \{cp\}\ |\!) \subseteq cpermDec(\!|\ \{ca\}\ |\!)$$

3.1.3 Unprotected components - Privilege escalation

As a component can perform sensitive actions, it should be protected against unauthorized access. Exported components could be accessed by other applications including malicious apps. Referring again [7], a significant number of non protected exported components was found in on-line Android open source projects (19039 components overall 8749 application).

Definition 3.3 (Unprotected components - Privilege escalation (P3)) *An exported component must declare permissions to be protected.*

theorem P3

$$\forall\, AndroidPerm \bullet cpermDec(\!|\ expComp\ |\!) \neq \emptyset$$

*P*3

$$AndroidPerm$$

$$cpermDec(\!|\ expComp\ |\!) \neq \emptyset$$

3.1.4 Permission Over-privilege

Application over-privileged issue is widely studied by the research community. It occurs when the application declares more permission than those really used. The main reason of this issue comes from developers as they are responsible to declare these permissions and manipulate them. To evade this issue, we have to formally ensure that all the declared permissions in the manifest file are used in the app code.

Definition 3.4 (Permissions Over-privilege (P4)) *At any time, a component cannot use an undeclared permission. All declared permission should be used at some time.*

This is time-bound property usually written using temporal logic. We simplify here by asserting it as an invariant of an execution schema.

> We do not care about the caller components here, just on which permission of one component has been called (used).

Suppose a set of executions and for each execution we assume *permUse* the set of permissions that are used for each component (this set can be filled dynamically during executions or just simulations if we want to prove the property or monitor the implemented Android permission system).

$$
\begin{array}{|l}
\textit{RunExecution} \\
\hline
\textit{AndroidPerm} \\
\textit{cpermUse} : \mathbb{P}\, \textit{Components} \leftrightarrow \textit{Permissions} \\
\hline
\forall\, \textit{exec} : \textit{Components} \leftrightarrow \textit{Permissions} \mid \textit{exec} \in \textit{cpermUse} \bullet \textit{exec} \subseteq \textit{cpermDec} \\
\hline
\end{array}
$$

In this invariant, for each execution *exec* of *permUse*, the permission used are declared: each member of *exec* is a subset of the declared permissions. Only declared permissions are invocable. For each permission used for one component, this permission has been declared by the component ($\forall\, c \mapsto p \in \textit{exec} \bullet c \mapsto p \in \textit{cpermDec}$).

Considering now all (possible) executions, we state that all declared permissions should be used at least once.

theorem P4

$$\forall\, \textit{RunExecution} \bullet \bigcup \textit{cpermUse} = \textit{cpermDec}$$

$$
\begin{array}{|l}
P4 \\
\hline
\textit{RunExecution} \\
\hline
\bigcup \textit{cpermUse} = \textit{cpermDec} \\
\hline
\end{array}
$$

In this invariant theorem, the whole set of *permUse* is computed by the multi-set union for all executions; it must be equal to *permDec*, meaning that all declared permissions can be invoked. A similar reasoning can apply to application and intents.

3.1.5 Implicit Intent Protection

In a previous experience conducted by Xu et al. [10] it has been shown that 80% of Android applications extracted from F-Droid[2] use implicit intent, leading to a common vulnerability called activity hijacking. That is why, it is preferable for developers to use explicit intents especially when the intent use sensitive data. However, if it is not possible to use the explicit type, the developers must have strong permissions to protect the data in the intent.

Definition 3.5 (Implicit Intent Protection (P5)) *Implicit intents should be protected by a permission having dangerous protection level.*

In the following assertion, the level of implicit intent permisssion is dangerous ($dang$).

theorem P5
$\forall\, AndroidPerm \bullet$
$permlevel(\!|\ (permIntent(\!|\ implIntent\ |\!))\ |\!) = \{dang\}$

where $permIntent(\!|\ implIntent\ |\!)$ is the set of permissions of the implicit intents.

P5
$AndroidPerm$

$permlevel(\!|\ (permIntent(\!|\ implIntent\ |\!))\ |\!) = \{dang\}$

We presented in this section, five security properties formalized using the Z notation. To perform verifications, we use the Z/EVES System [8]. It enables not only syntax and type checking verification, but also the proof of operation assertions (pre/post conditions vs the state schema invariants) and theorems. The effective verifications are not provided here but the formalisation is an input for the implementation. These properties will be translated in Section 5 into the formal constraint language OCL to implement a support IDE solution allowing their automatic verification.

3.2 Dynamic Security Properties

The 5 security properties specified above analyse static features related to the structure of a single application during development. We don't focus on dynamic analysis like many related research studies [2, 4, 1] that explore the dynamic behaviour coming from the interaction between different applications at the system environment level, and that with the goal to find permission related security flaws in the Android permission framework. There it becomes very important to verify such properties related to (1) the modelling of the application execution, (2) studying events related to the application install, (3) uninstall, and (4) the dynamic permission granting and revoking between applications. But that does not mean that dynamic analysis is not useful for our case. Dynamic analysis would be very helpful to verify some properties for developers to improve the accuracy of over-privilege application property. Based on our static analysis we could only 1) determine the requested permissions declared in the manifest configuration file, 2) generate permissions used (PUs) by the application through inspecting the permission related APIs, 3) inspect methods involving sending and receiving intents, (4) and methods involving the management of content provider. Whereas dynamic analysis could assist in 1) handling the dynamic loading of classes from embedded .jar and .apk files, 2) handling Java refection which is used by more than 57% (2013) of Android

[2] https://f-droid.org/fr/

apps [6] which provides a program the ability to inspect classes, methods, interfaces and fields at runtime without knowing the names of the classes and methods in prior. which helps to enrich the *Uperm* set.
We presented in this section our Z specification of security properties in the context of the abstract Model presented in Section **??**. In the next Section, we describe the design and the implementation of the proposed model based approach enabling the verification of these properties.

4 Z-EVES verification

Several syntactic modifications have to be done to be accepted as a correct syntax in Z-Eves.

- Type shortcuts are not allowed (see note 3).
- Power sets must be finite.
- Declarations of free types must be separate from basic types.
- etc.

We rewrited Z specifications by respecting these conventions (`androidPerm.zed` file). Fig. 1 shows en screen shot of the specification state.

5 Details of PermDroid Design and Implementation

We formalised in Section 3 five specific security related permissions issues coming from: permissions naming violations, unprotected components, components invocation, over-privileged permission use, and unprotected implicit intents. To assist developers (especially third party ones) to automatically prevent these issues in their implementations, We propose in this section a model-based approach (See Fig. 3) that takes as a centric element the security meta-model of Fig. 2 which is equivalent to the formal model presented in Section 2 with some additional technical aspects.

We started by extracting the application model from a real Android application following a MDRE process. The resulted model will help to summarize only the security aspects related to permissions as a target model instead of checking the whole application code, which is difficult and time consuming. The security model is obtained through a ***model** (Android application) to **model** (representing the security aspects)* transformation process. The followed steps and the proposed approach are presented in Fig. 3

5.1 Proposed Approach: Main Steps

As presented in Fig. 3, our approach is based on three steps: (1) A reverse engineering step to perform our security study on the model level instead of the source code level. (2) A model to model transformation step to get out the security model allowing to (3) perform in the 3rd step, the analysis of the security properties that we specified using Z then implemented using OCL[3].
Reverse Engineering Phase (Model Discovery). We mean by reverse engineering here, having abstract models with XMI notation that exactly conform to the real source code of the application. Due to the several facilities that it proposes, we used MoDisco tool [**?**] to perform the reverse engineering process. Basically, MoDisco[4] provides an extensible framework to develop model-driven tools to support use-cases of existing software modernization. It provides a graphical representation

[3] https://projects.eclipse.org/projects/modeling.mdt.ocl
[4] https://wiki.eclipse.org/MoDisco

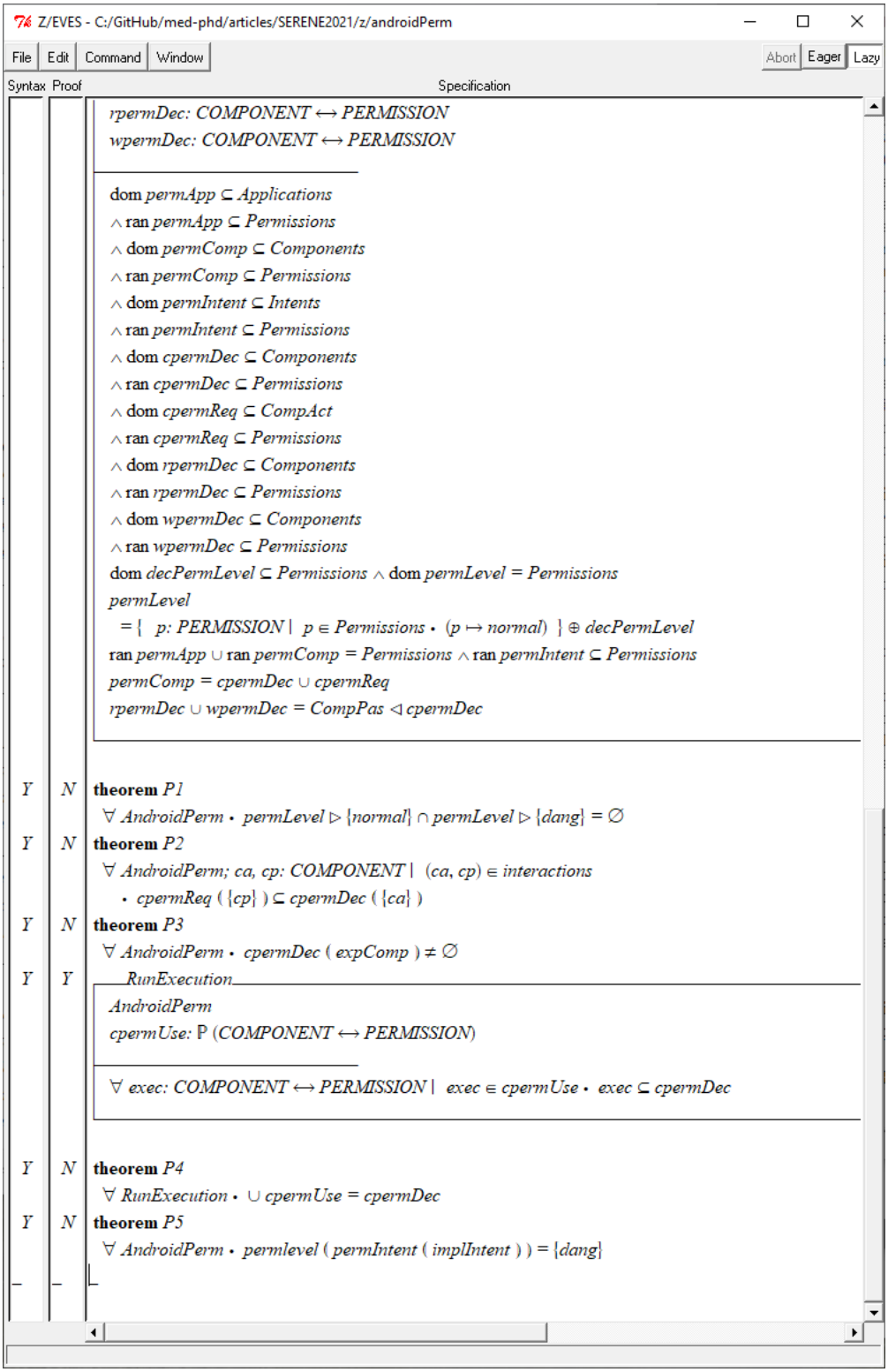

Figure 1: Z-Eves GUI screenshot

of the program Abstract Syntax Tree AST[5] which makes the corresponding models it generates

[5]https://www.vogella.com/tutorials/EclipseJDT/article.html

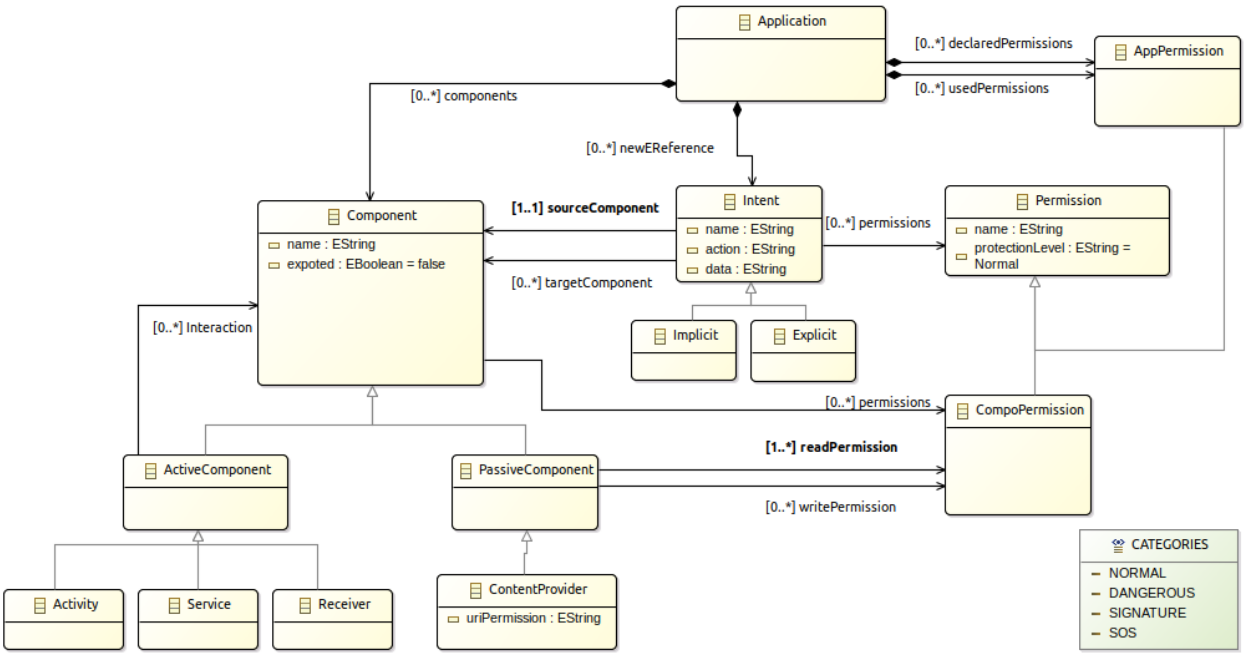

Figure 2: Android Permission Security MetaModel

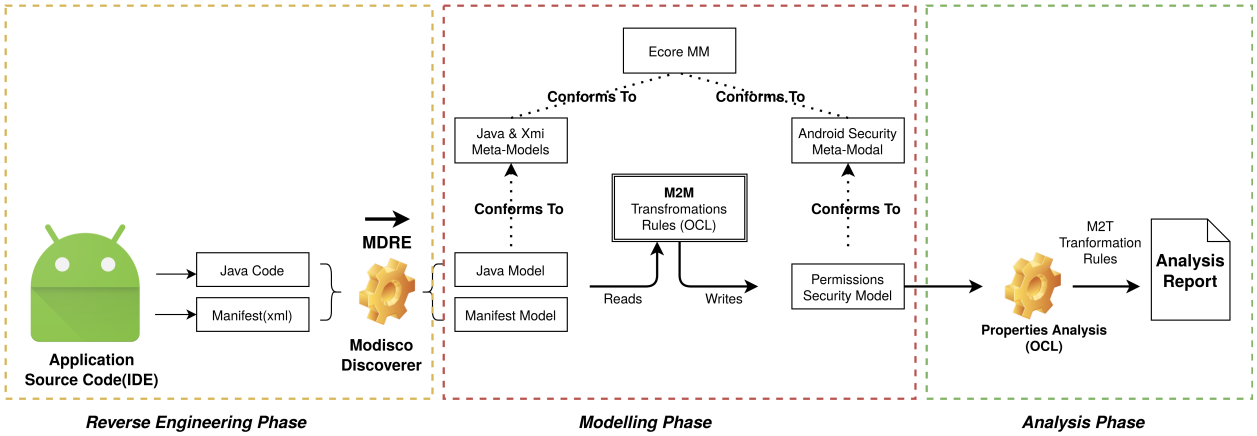

Figure 3: PermDroid Architecture

correct and perfectly conform to the source code. In addition, MoDisco supports this process for Java and XML based applications which is relevant to study Android applications. For this aim, we started in phase 1 (orange part) of Fig. 3 by the *manifest.xml* configuration file. We created a script with Java using Modisco offered interfaces that takes as input the manifest file and generates as output the xmi corresponding model. (Fig. 4 shows an example of the manifest model generated for an Android open-source application published in github[6]).

The same steps are followed to generate the corresponding models for Java classes.

Model Transformation Phase. Once both Java and Manifest corresponding models are generated during the first step, we apply to them a M2M transformation process using ATL[7] tool

[6] `https://github.com/Telegram-FOSS-Team/Telegram-FOSS`

[7] https://www.eclipse.org/atl/

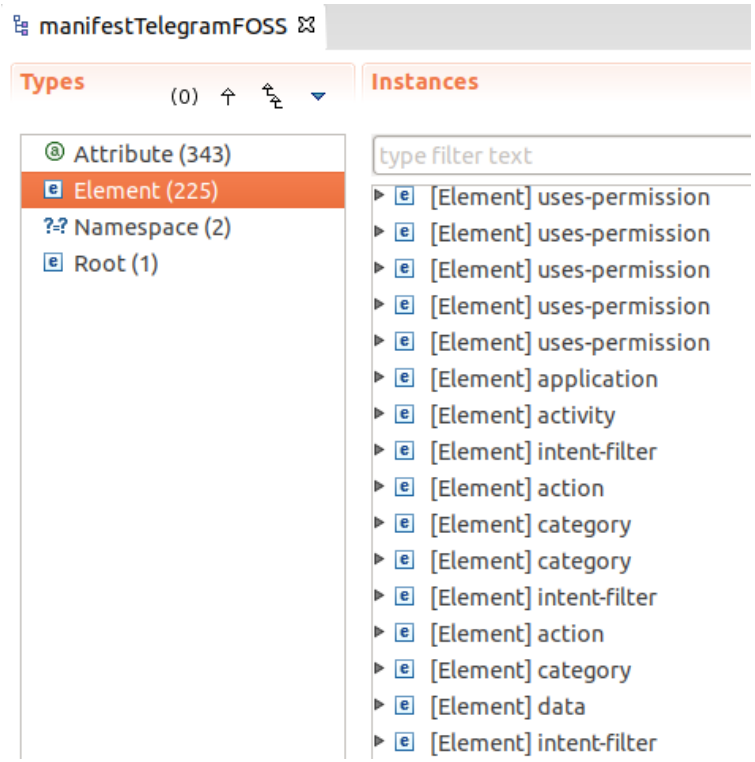

Figure 4: Example of a generated manifest model

which is based on the OCL formal language. ATL could be used in Eclipse but it also provides the possibility to run its transformations on VM mode. This solution is flexible and makes it easier to integrate PermDroid tool into other development IDE for Android applications, like IntellIJ and Android Studio[8]. As a result of the model transformation process, we obtain the Android security model used for analysis phase. A schematic view showing the whole model-driven chain is presented in phase 2 of Fig. 3.

The transformation engine we implemented is composed from a set of rules. Each one serves to *read* a specific part from the source model "Java or manifest models" (*like the manifest "MM!Root" element in Fig. 5*) and *write* the corresponding element (*like the "MM1!Application" element in Fig. 5*) in the target model following a well defined semantic. Fig. 5 show the main rule of the transformation script that aims to generate an instance of *application* element structure presented in the meta-model of Fig. 3 from the *root* xml element of the manifest file.

Security Properties Analysis Phase. For the analysis phase, we implemented using ATL a set of rules called *helpers* (like methods in Java but with a formal constraint format) to implement the security properties we specified in Section 3. To make clear the idea of how using OCL for the analysis phase. We present in the following a simple part of the implemented helpers, but we do not put the whole rules due to a presentation and formatting issue.

```
helper def: permission_name_conflict():
  Boolean =
  if ((thisModule.normalPermissions -> intersection(thisModule.dengerousPermissions)
      )-> isEmpty()
     )
```

[8] https://plugins.jetbrains.com/androidstudio

```
1  -- @nsURI MM=http://www.eclipse.org/MoDisco/Xml/0.1.incubation/XML
2  -- @path MM1=/safetyProperties/metamodels/androidapp.ecore
3
4  module application2securitymodel;
5  create OUT : MM1 from IN : MM;
6
7
8  rule mainRule {
9      from
10         manifest : MM!Root (manifest.name="manifest")
11     to
12         securityModel : MM1!Application (
13             name <- manifest.name,
14             version <- manifest.version,
15             permissions <- MM1!Permission.allInstances(),
16             components  <- MM1!Component.allInstances()
17         )
18 }
```

Figure 5: M2M Transformation engine: The Main Rule

```
    then true
    else false
    endif;
```

The permission_name_conflict() is an example of helper that is used to implement the permission property (P1). It parses the instance security model representing the Android application under analysis, to inspect if there is such permission with normal protection level that is also declared for a second time as dangerous permission.

```
helper def: MM!Implicit :
     have_dangerous_permissions(
       permissions: Sequence(MM!Permission)
     ) : Boolean =
if(permissions.forAll(perm |
   perm.protectionLevel = 'Dangerous'))
then  true
else false
endif;

helper def:
   implicit_intent_protection(
     intent: MM!Implicit
   ):
   Boolean =
if(intent.have_dengerous_permissions(
          inten.permissions
       )
   )
then true; else false; endif;
```

In the same line, both have_dangerous_permission() and implicit_intent_protection() helper implement the property (P5) to be applicable to each "Implicit Intent" element. If the return type is true the tool PermDroid will proceed to notify the developer. We give an example at the end of Section 6.

6 Details of Experimentation details

To experiment the ongoing work on PermDroid tool, we selected on a simple open source application called ***Telegram*** extracted from F_Droid[9], a famous repository for Android open source applications. It represents a messaging app with a main focus on speed and security. We add some modifications to the application in order to design the security flaws we want to raise at the analysis phase. The goal is to validate the followed steps in our approach such as: model discovery with MoDisco, modelling , and analysis phases; and that based on a simple case study.

Reverse Engineering Phase (Model Discovery). As mentioned before, the first step of PermDroid processing is the model discovery step, where the tool will have as an input the Android application code files (manifest.xml and Java classes) and provides as an output the models representing them. Fig. 6 presents *xmi* the generated model representing the manifest configuration file of ***Telegram*** application. As can be seen MoDisco provides an abstract representation of the

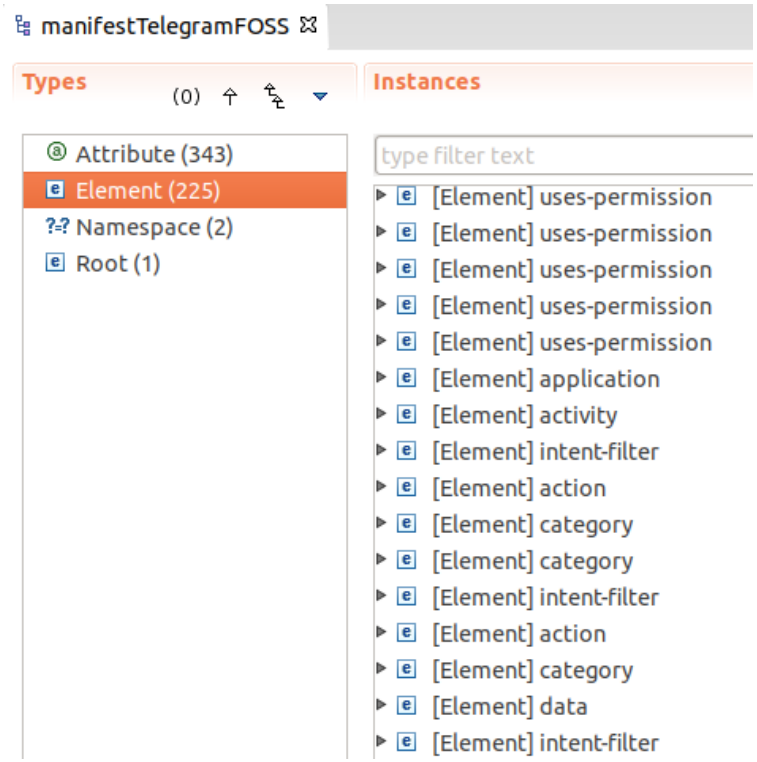

Figure 6: Modisco discoverer for ***Telegram*** application

manifest file. It exposes the *root* instance that represents the composite element by which we can access children items such us: components, intents, permissions and their attributes.

Modelling Phase. Once the corresponding ***Telegram*** demo app models are generated, PermDroid launch the m2m transformation process to generate the application security model displayed in Fig. 7 where types specified on our security meta-model are presented on the left side, and the existing instances for each type are presented on the right side. Based on these representation, we notice that the demo app is composed only from background components: 15 services, 18 broadcast receivers and 3 content providers

[9] https://f-droid.org/app/org.telegram.messenger

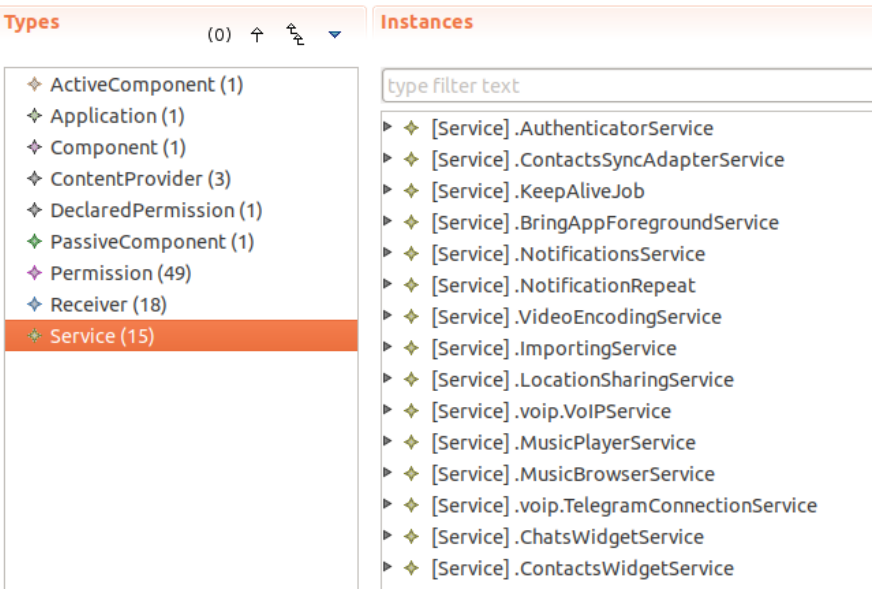

Figure 7: Generated Security Model of ***Telegram*** demo application

Analysis Results. The final step of the tool is to launch the ATL script implementing the formulated security properties. Fig. 8 shows the analysis result raised by PermDroid tool. The analysis

Figure 8: Analysis Results of *Telegram* demo app

report raises the unsatisfaction of the security propety related to naming conflicts. It indicated that *perm1="android.permission.BIND _TELECOM_CONNECTION_SERVICE"* with *Nrml_perm* is duplicated in: "Content Provider":"$applicationId.notification_image_provider" with: "DANGEROUS" protection level. Note that all this steps including: reverse engineering, modelling and analysis will be executed on the background by the PermDroid tool.

References

[1] Gustavo Betarte, Juan Campo, Maximiliano Cristiá, Felipe Gorostiaga, Carlos Luna, and Camila Sanz. Towards formal model-based analysis and testing of android's security mechanisms. In *2017 XLIII Latin American Computer Conference (CLEI)*, pages 1–10. IEEE, 2017.

[2] Gustavo Betarte, Juan Campo, Carlos Luna, and Agustín Romano. Formal analysis of android's permission-based security model. *Scientific Annals of Computer Science*, 26(1), 2016.

[3] W.J. Buchanan. *Introduction to Security and Network Forensics*. Taylor & Francis, 2011.

[4] Elli Fragkaki, Lujo Bauer, Limin Jia, and David Swasey. Modeling and enhancing android's permission system. In *European Symposium on Research in Computer Security*, pages 1–18. Springer, 2012.

[5] Xudong He. Modeling and analyzing the android permission framework using high level petri nets. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 232–239. IEEE, 2017.

[6] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. Slicing droids: program slicing for smali code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1844–1851, 2013.

[7] Ajay Kumar Jha, Sunghee Lee, and Woo Jin Lee. Developer mistakes in writing android manifests: An empirical study of configuration errors. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 25–36. IEEE, 2017.

[8] Mark Saaltink. The Z/EVES system. In Jonathan P. Bowen, Michael G. Hinchey, and David Till, editors, *ZUM '97: The Z Formal Specification Notation, 10th International Conference of Z Users, Reading, UK, April 3-4, 1997, Proceedings*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–85. Springer, 1997.

[9] J. Michael Spivey. *Z Notation - a reference manual (2. ed.)*. Prentice Hall International Series in Computer Science. Prentice Hall, 1992.

[10] Guosheng Xu, Shengwei Xu, Chuan Gao, Bo Wang, and Guoai Xu. Perhelper: Helping developers make better decisions on permission uses in android apps. *Applied Sciences*, 9(18):3699, 2019.