Security Knowledge Base for Android Applications

Nicolas Noël

Research Master Thesis

Research Master Thesis

Security Knowledge Base for Android Applications

Author:
Nicolas Noel

Supervisor:
David Aspinall
Nassim Seghir
Laboratory for Foundations of
Computer Science

Abstract

During the last years, the number of mobile devices has been growing rapidly as well as the number of dedicated applications. Such devices are exposed to a variety of threats: hackers, malware, etc.

For these reasons, several tools have been developed to assess and analyse the security of Android applications. However, there is no data base regrouping the results of these tools which is publicly available. Some security data bases exist, but they are too focused on the malware side of security.

The goal of this internship is to develop a security knowledge base for Android regrouping applications, analysis tools and results. It will allow different kind of users (developers, researchers, etc.) to invoke a variety of tools and compare the results and understand unwanted or malicious behaviour of applications. It will also provide ranking an prioritisation of analysis results based on metrics proposed in this report.

Contents

1 Introduction

Modern smartphones allow the installation and the execution of third-party applications. These applications are at the centre of a business model created by mobile operating system providers. Android is the leader in this domain with 77.78% of market share for the fourth quarter of 2013. For this reason and because of its openness, Android is a perfect choice for research purposes.

With a number of activated devices which exceed one billion in 2013, the sector of mobile applications became very attractive for hackers. Several kind of malware have been appeared during the last years like DroidKungFu[1]. Moreover, mobile applications can also have vulnerabilities in the code which can be exploited. This gives the possibility to hackers to steal information, take control of the device, etc.

To detect malwares and vulnerabilities, several research as well as commercial tools have been developed. These tools (analysers) provide information about potential malicious behaviour or the presence of a vulnerability in an application. The goal of this work is to develop a security knowledge base for Android applications regrouping different alerts generated by analysers, information about the analyser itself and the application analysed.

One of the biggest challenges for the construction of a knowledge base containing security information about Android applications is the lack of a norm or a standard to describe vulnerabilities of applications together with the result (audit) of the analysis tools.

It will be also interesting to have a global view which includes the result coming from different tools.

During the development of the security knowledge base, several questions have emerged. As we may have many versions of the same application, is there a correlation between the application evolution and raised alerts ? For this, we want to study the evolution of alerts found by the different tools and see if they are fixed in the successive versions of an application.

Several applications developed by the same (team of) developer(s) can be present in our security knowledge base. The errors made by developer in an application could be present in some other applications as well. We would like to find a relation between the results of the analysis of different applications written by the same developer. The errors will not be syntactically the same, but it seems possible to categories such alerts. For example, a developer who does not know how to handle the API related to some technology could repeat the same mistakes for all applications. It is possible to track the evolution of the developer skills through the evolution of his applications based on the errors found. We can then create a reputation model for developers. This also support the developers in improving the quality of their applications.

Another point of interest is the existence of a correlation between the results of the different tools. For example, the same private data leaks could be raised by two different tools in different forms. Having such correlation will increase our confidence in a result if it is reported by more than one tool.

Our contribution is the proposal of a standard to describe the result of different analysis. We have also defined a scoring function to rank the application according to the types of alerts raised by the analysis tools. We have also developed the security knowledge base composed of 4 different tools and more than 1000 applications. We have also defined a metric to compare applications in terms of alerts they contain. Finally, we have proposed a reputation model for developers based on the categorisation of errors they make in their applications.

[1] http://www.f-secure.com/v-descs/trojan_android_droidkungfu_c.shtml

This master thesis is organized as follow: Section 2 illustrates use cases of the security knowledge base and provides some possible usage scenarios. Section 3 describes some aspects of Android infrastructure and exposes the state-of-the-art in terms of vulnerability representation standard available in the literature. Section 4 introduces our proposed standard for alerts classification for Android as well as a metric for application ranking. In Section 5, we describe our implementation of the knowledge base and we present an experimental study in Section 6. Finally, Section 7 concludes the thesis.

2 Usage scenarios for the security knowledge base

The security knowledge base (S.K.B.) will be potentially used by diverse communities:

- Researchers can use it to compare the different analysis tools
- It will offer a standard output model to guide tool developers. It will also serve for regression test purposes.
- It will allow average users to evaluate the security of applications they attempt to install

We describe use case scenarios for these categories of users.

2.0.1 Tool developers

Tool developers must test their tools and compare their results to the existing tools. For this reason, it is very important to have a standard output format which facilitates the comparison. In this respect, the security knowledge base will offer a regression test suite for developer to test the soundness and precision of their tools.

The standard format will also be useful for the developers to export the results of their tools. Actually, there is no straight way to compare the results of different tools. Standard test sets to compare results are available in other research domains. For example, the TRECVID (workshop about information retrieval in videos) provides a large dataset to compare results of tools and a uniform scoring metric.

The security knowledge base could also provide a benchmark set for performance evaluation and comparison.

Hence, to exploit the S.K.B., the main requirements for a tool developer are:

- A clear API to integrate their tools in the security knowledge
- A standard output format to export their results
- A way to insert the new results in the Knowledge Base

2.0.2 Android security researchers

The security knowledge base will help the researchers in many ways. It will help them to extract the different security properties of an application. It will also offer the possibility to extract a subset of applications from the S.K.B. based on some security properties.

For example, when writing a paper, a researcher could use the security knowledge base to perform a triage to only keep application exhibiting the wanted security aspect. He can them compare the results of a tool with other tools.

Moreover the security knowledge base can help the researcher to develop new ideas by observing and analysing weaknesses of the existing tools with respect to some given properties.

To summarize, the requirements for a researcher are mainly:

- An interface to explore the knowledge base
- A standard output for the analysis tools

2.0.3 Average User

An average user would like to test an application downloaded from the Internet to be sure that it is not a malware and that it is free from critical vulnerabilities. For this, he would like to submit an APK file to the security knowledge base or redirect the base to analyse an application from a compatible store. At the end, he needs a feedback from the analysis which explains to him, without all details, the result of the analysis in a understandable way.

Hence, he mainly needs:

- An interface to submit new applications
- A natural and intuitive representation of the analysis result
- A score to sort the application according the alerts raised

3 State of the Art

A presentation of the Android system is necessary to understand the requirements and the specificities of the project. The study of the different standards for alert or vulnerabilities will also help us to bridge the gap between what already exists and our requirements.

It is very important to study the different existing tools and the possible output they can provide. Indeed, by understanding and classifying the different tools, it will be possible to extract profiles for the standard output.

In what follows, we first start by presenting the Android system. We them give an overview of the existing security bases and the standards for expressing weaknesses and vulnerabilities. We finally give a presentation of several existing tools for Android analysis.

3.1 Android platform

Android is an operating system equipping several types of devices (Smartphone, Tablets, TV ...). It is based on a Linux kernel and maintained by the Open Handset Alliance led by Google. Android's business model is based on an online application store similar to the one of Apple. The main store which is also the default store for Android devices is the Play Store develop by Google. It contained 1 Million applications in July 2013[2] that the users can download. Google analyses the applications before releasing them for the public. However this analysis is not complete and some malware could

[2] www.technobuffalo.com/2013/07/24/google-70m-android-tablets-1-million-android-apps/

circumvent it. Benign applications could also contain vulnerabilities due programming errors or bugs in the source code of the application or in third-party libraries included in the application.

It exists other stores for Android devices. For example, the Amazon appstore[3] or Aptoide[4]. It also exists some manufacturer stores like Samsung Apps which offers some applications that use hardware specific features of Samsung devices (like SPen[5]).

None of these stores offer full immunity against malware or vulnerabilities. Some stores do not perform a security analysis at all. It is for these reasons that a knowledge base including the security alerts raised by analysis tools is necessary. It will allow the user to spot the vulnerabilities and see if they are fixed in newer versions of the application. It will also allow the user to make better decisions before downloading applications. Eventually, it could allow to detect and delete some malware from stores.

Applications are distributed as a single file called Android Application Package or APK. During installation, a unique identifier is assigned to the application which is executed in isolation from other applications thanks to a sandbox mechanism implemented by Android. The unique identifier used by Android is based on the package name of the application. Each application is signed with the developer's certificate to allow Android to detect modification by a third party. However this security does not prevent the repackaging of the application by signing it with a different certificate than the original one.

Applications are developed using Java language but native code (C, C++) can be used via JNI(Java Native Interface). The Java source code is compiled to a custom bytecode which is executed on an associated virtual machine called Dalvik. This bytecode is different from the standard JVM bytecode. For example the Dalvik bytecode uses 16 bit instructions and the standard bytecode uses 8 bit instructions.

Android offers an API to exploit the resources present on the device. The API functions are protected by permissions. An application must declare the list of permissions it requires and theses permissions are displayed to the user during the installation. The user can then decide about granting or denying these permissions to a given application. In the latter case, the application will not be installed. Android will block any application which tries to use a protected function if it does not have the appropriate permission.

An Android application is composed of four different components: activity, service, broadcast receiver and content provider. An activity is a component with a graphical interface. It is similar to a window in desktop applications, this component handles the different interactions with the user. A service is a component working in background with no graphical interface. A service can be used to download a document or calculate some GPS coordinates in background. The broadcast receiver handles the different messages sent by the operating system or applications. The content provider is an interface giving access to a persistent dataset (generally a SQLite database). All components have different states describing their lifecycles. Each component is *created* and after creation it can be stopped, so it can be in an *active* or *inactive* (for activities, we say *running* or *stopped*) state. An activity can also be *paused*. All components can also be *destroyed*. The Figure 1 shows the lifecycle of an Android activity[6] with the different states and callbacks used by the system. These callbacks can be overridden by the developer to provide treatment for specific actions. Android

[3]http://www.amazon.com/mobile-apps/b?node=2350149011

[4]http://m.aptoide.com/

[5]http://techlife.samsung.com/top-10-apps-samsung-s-pen-1357.html

[6]http://developer.android.com/guide/components/activities.html

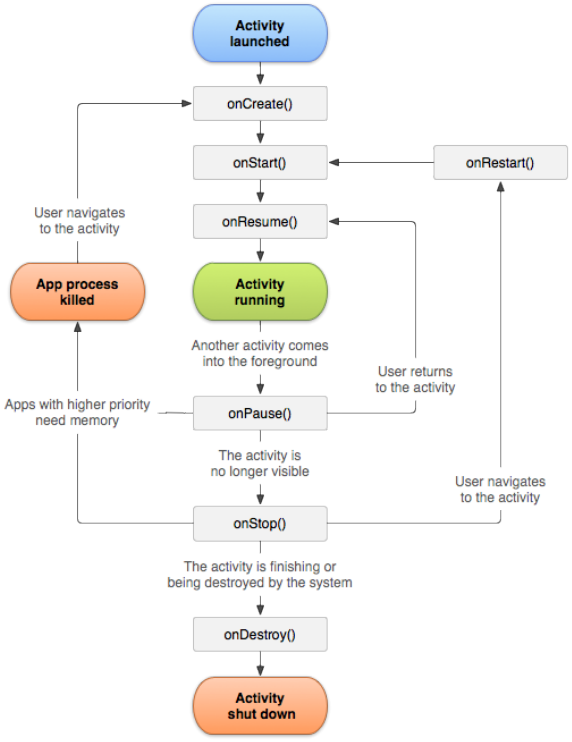

Figure 1: Actvity Lifecycle

provides also mechanisms for inter-process communication through high-level objects called intents which can influence components' state.

3.2 Analysis tools

To propose a standard and generic representation, it is important to have an idea about the different existing tools for the analysis of Android applications. The goal of the knowledge base is not to store just the result of the analysis but also to create a report supporting the analysis outcome. For example, if a tool finds an illegal data flow (according to a policy), it could be important to store information about the way it occurs as well. With standardized results, it becomes possible to compare the results of different tools.

Analysis tools could be generally classified into two categories: static and dynamic tools. In what follows, we organise our presentation according to this classification.

3.2.1 Static analysis

Static analysis tools check properties of applications without having to execute them. This prevents triggering potential malicious behaviour. Static analysis is either applied to source code of the program (high-level language) or the compilation result (binary or bytecode). Standard analysis tools for the JVM bytecode cannot be directly applied to Android applications.

Hence, some tools translate the JVM bytecode to Dalvik bytecode to re-use existing tools. This approach hides some parts of the Android system and increases the complexity of the analysis.

Another technique is to translate the bytecode in another language like FlowDroid [8] do to perform a static dataflow analysis. The tool tries to detect the leaks of private data like the user's location in Android applications by detecting a data flow between a source of private data and a sink. A sink is a method where data could exit from the secure environment (device, application, etc.). Flowdroid is based on the Java optimisation framework called Soot[7]. Flowdroid allows to customize the list of private data sources and the list of sinks for the analysis.

SCanDal [10] is also an analyser which uses a static data flow analysis to enforce the protection of the confidentiality of user's data. Unlike Flowdroid, SCanDal is not available for download. Another dataflow-based tool is SCanDroid[9]. It uses an intermediate representation of the code and WALA[8], which is a library for Java analysis, as backend. Like Flowdroid, here also it is possible to customize the list of sources and sinks used for the analysis. All these data flow analysis tools return a list of flows as result. For example, if one of these tools finds a flow between the function *getSimSerialNumber()* from *android.telephony.TelephonyManager* which returns serial numbers of the device SIM card and *sendTextMessage()* from the class *android.telephony.TelephonyManager*, it means that the application could send the serial number of the SIM card via a SMS. It is necessary for our security knowledge base to store these methods in the alert and eventually the entire path between the two functions.

Other tools use static analysis for different goals with different techniques. PScout[1] looks for unnecessary permissions declared by an application. PSCout parses the bytecode file of an Android application to construct a call graph and find protected functions, i.e., functions associated with permissions. It then runs a reachability analysis to find out if a protected function can be called. If this is the case, the permission is necessary.

Indeed, it could be dangerous for the system to leave an application run with unnecessary permissions. Android has a sandbox mechanism which isolates each application from other and gives a unique Linux user ID to each sandbox. But it is possible for an application to authorize other applications to share the same sandbox. All applications in the same sandbox share the same set of permissions. The permissions of a sandbox is the union of the permissions of all applications sharing it. A malware which targets a particular vulnerable application authorizing sandbox sharing could use the permissions declared by other applications to run discreetly malicious behaviours without having to declare any permission by itself.

MalloDroid [7] uses static analysis to find SSL communication misuse in applications. It tries to find specific programming errors. This tool parses the code of an application and finds the classes related to SSL usage. It tries then to find some vulnerabilities like the non verification of certificate. The tool generates alerts which can be classified in different categories according to the type of the alert. There is also a confidence indicator for the alert.

It is important for our knowledge base to save the name of the class and the name of the method incriminated. MalloDroid identifies several possible causes of misuses. It is very important to save them too.

Comdroid[5] is another tool which uses static analysis to find vulnerabilities related to communication between components in Android applications. Like explain in Section 3.1, intents are used to communicate between the Android components. Comdroid tries to detect programming errors or malicious behaviour relative to these objects and relative to the communication inter components.

For example, an activity can be started using an intent destined to it. In this case, it is necessary

[7]https://github.com/Sable/soot

[8]http://wala.sourceforge.net/wiki/index.php/Mainpage

to protect such components to avoid malicious applications to hijack them for malicious actions. An example is the usage of a login activity from a banking application to steal the login and password of a user. It could be also possible to hijack a service and do an unauthorized money transfer.

In case of a possible vulnerability, it is necessary to save in our knowledge base the information about the type of attack which concerns that vulnerability and the component which is vulnerable.

Evicheck is a tool developed in the context of the Appguarden project of the University of Edinburgh. This tool used static analysis to verify if an application is valid relative to a security policy. It is based on Androguard[9] (a tool for forensic analysis and retro engineering specialised in Android applications) to statically analyse the bytecode of the application. If the application respects the security policy, a certificate is generated.

3.2.2 Dynamic analysis

Dynamic analysis tools monitor applications during execution time on an emulator or a device. One of the most popular emulator is Qemu[10]. This emulator gives access to data of the emulated system during the usage of applications. Some tools like Copperdroid [13] uses Qemu to emulate a device and to track the system calls and analyse the IPC and RPC messages during application execution.

This tool offers a good way to extract the behaviour of applications but not to detect if an application is malicious. The aim of some tools like Mallodroid it to detect that a method or a class is suspicious but it cannot confirm it. In this case, Copperdroid can be useful to support a manual investigation by providing the list of system calls done during the execution of the application.

There are other categories of dynamic analysis tools. The data flow analysis by taint values is one of them and is used by Taindroid [6]. It is based on the modification of the Dalvik Virtual Machine to monitor applications. It assigns a taint (a value) to some data considered as private. For example, the user location or the user's contacts are considered as private data. A variable will be tainted if the application saves the user location in it. For each instruction of bytecode, a taint propagation logic is associated with it to propagate (or not) the taint. If the taint reaches a function considered as a sink (an exit point for the data), an alert is displayed to the user. Like for FlowDroid (Section 3.2.1), it is important for us to save the source, the sink and the entire path of the flow. It is more complicated to save these information in a dynamic analysis context because all of this is done on the device in real-time and not all information might be available.

The advantage of dynamic analysis and monitoring system is that they offer a better accuracy than static analysis as they rule out false positives.

However, one problem that dynamic analysis tools have is the redundancy of the results. Indeed, tools like TaintDroid monitor applications and the Android system. An alert could be raised again and again if the application is used in the same way. It is important to distinguish them to avoid saving the same alert several times and waste space and time in our security knowledge base.

Aurasium [14] repackages applications by inserting some code to monitor the interaction between the system and the application. Aurasium accepts a policy specification and raises an alert if the policy is violated. One problem for the security knowledge base is to store the policy used by the tool. Indeed, each tool could have a specific formalism to declare a security policy as there is no standard to describe it. Hence, it will be necessary for our knowledge base to allow the user to

[9] https://code.google.com/p/androguard/
[10] www.qemu.org

Type of analysis	Informations to save	Tool
Data flow analysis	Type of flow Class and method of the source Class and method of the sink	SCanDroid Flowdroid Taintdroid
Unnecessary permission	Permission	PScout
SSL misuse	Status of the error (confidence indicator) Error type Class name Method name	Mallodroid
Component and Intent hijacking	Type of attack Class involved	ComDroid
Security policy violation	Security policy Details of the violation (IP adress, command, telephone number ...)	Evicheck Aurasium

Table 1: Categories of information extracted by Android application analysers

create an analyser composed of a tool and a configuration (security policy ...) and offers to users the possibility to declare several instances of the same tool but with different configurations.

Another tool which uses security policies and monitors the system to track violation os a security policy is AppGuard [2]. AppGuard inserts code into the application to monitor it without changing the system like Aurasium.

A limit for tools based on dynamic analysis is that the result could be incomplete as we run the application and only analyse the executed paths. Hence, parts of the application could remain unexplored. In addition to this, a malware can delay the malicious behaviour if it detects that it runs on an emulator.

To conclude, Figure 1 shows the information wanted for each category of tools. This allows us to build a standard to represent the results of an analysis by using these specificities.

3.3 Security Knowledge Base and Malicious Behaviour Representation

Section 3.2 presents different existing tools used to analyse and detect vulnerabilities or malicious behaviour in Android application. These tools display the alerts or export them in a file. This section describes the state of the art regarding security knowledge bases to store security alerts related to malicious behaviour or vulnerabilities. The goal is to figure out the advantages and the disadvantages of the different solutions. Concerning Android applications, there is no security knowledge base regrouping their security properties. The only existing base which is close to our goal is the one called Andrototal[11] developed by NECSTLab in Milan [11]. This base regroups applications submitted by users and the results of anti-virus analysis. The details of analysis are not always available. We want that our security knowledge base includes the results of several analysers developed by the research community and also correlation metrics for these results and not results from anti-viruses analysis. Unlike Andrototal which only detect malware, our knowledge base will try to detect vulnerabilities in benign applications too.

[11] http://andrototal.org

Others bases exists which regroup Android applications. The Android Observatory is a base regrouping applications coming from different sources. No analysis are made on the application but several information like the permissions declared or the developer's certificate are extracted from the application and shown to the users. The Android Malware Genome [15] is a security base regrouping the different families of malware which exists on the Android platform. The goal is to understand the different existing malware and not to analyse applications.

3.3.1 The IDMEF representation

There is no standard way specific to Android to express alert raised by analysers. In the domain of intrusion detection in general, there is the Intrusion Detection Message Exchange Format (IDMEF) defined in the RFC 4765[12]. This RFC defines a standard way to describe intrusions in an environment. The term *environment* means a place regrouping several analysers and resources (devices for IDMEF, applications for the S.K.B.). This RFC shows the difficulties to find a way to express the alerts raised by a tool. This RFC has been submitted in 2007 and is still experimental, it means that it is not clear if this will be widely adopted. If the IDMEF is widely adopted, this RFC will become a standard (at the RFC sense).

Each IDS (intrusion detection system) may come from a different manufacturer and each manufacturer may use a different way to export alerts. In addition to that, the IDS present in an environment could be duplicated to avoid mistakes or to detect sophisticated attacks. IDMEF uses the XML language to express the alerts and the RFC contains the Domain Type Definition (or DTD) file which describes the structure of XML files used to express alerts. One goal of the IDMEF is to allow the user to have a good GUI to visualize the data. The IDMEF representation is object-oriented which makes easier the understanding of the model but also decreases the possible consequences of modifications.

The RFC gives a good description of all the different possible parts of an IDMEF message. In IDMEF, it exists several types of alerts, the Alert object is the most general message for an alert. Some specific objects could be used in IDMEF to express a specific alert. Such classes have new information that the generic alert class does not contain. To keep a trace of different alerts raised in an environment, the date and time of alert creation are associated with the alert.

The goal of IDMEF is not to provide a universal alert but rather an alert destined to a particular environment. This implies that each analyser or alert is identified with a unique identifier for a given environment.

3.3.2 Malware Attribute Enumeration and Characterization

The MITRE corporation works on the representation of malware behavior by developing the Malware Attribute Enumeration and Characterization (MAEC)[13]. MAEC is an on-going project which is still in a experimental phase which defines a file format to keep the information related to a security analysis. Several informations like the command line used or the time of execution are saved. They keep the identity of the technician who runs the analysis. Except some human readable text, the majority of information in the output of the analysis is not stored in the MAEC file. The file saves more the context than the result of the analysis. This is to avoid that MAEC file will be too precise and not usable for all types of malware.

[12] http://tools.ietf.org/html/rfc4765
[13] http://maec.mitre.org

Unlike MAEC, we are focus on a subset of malware: the Android malware. It means that several things will be the same for each malware found like the operating system. These informations are not necessary for the description of an Android malware. It means also that it is possible to include more information in our standard output to be more precise and give more information.

3.3.3 Malware MetaData Exchange Format

The Industry Connection Security Group (ICSG) of the IEEE standards develops, inside the Malware Metadata Exchange Format Working Group, the Malware MetaData Exchange Format (MMDEF)[14] to describe the metadata used by a malware. This group aims at creating a format file which describes in a standard way the different informations used by a malware like the files, addresses, etc.

This standard is useful to describe malware and to understand its actions but it is not helpful to describe the vulnerabilities of an application. All these previously mentioned standards are not sufficient to characterise our alerts. Nonetheless, we can borrow some ideas for our own standard.

3.4 Weaknesses and Vulnerabilities representation

Several standards exist to express the different possible weaknesses present in a piece of software. The most known and used is Common Weakness Enumeration (CWE)[15] developed and maintained by the MITRE corporation.

3.4.1 Common Weakness Enumeration

The goal of this standard is to have a catalogue of software weaknesses which could be present in a piece of software. The Comprehensive CWE Dictionnary contains actually 2 000 sorts of weaknesses and is one of the several dictionaries developed by the CWE project. One dictionary has a research scope and another one has a development scope. This separation between research and development allows to specify the weaknesses and be more precise. Each of these sets is evolving like explained in [12]. The CWE has a system of inheritance for the weaknesses which allows an eventual refinement of the different weaknesses.

For example, the weakness identified by the number 502 of the Comprehensive CWE dictionnary is about the deserialization of untrusted data. Each weakness has a description which gives more information about it. It also provides informations about the parent of this weakness and the possible children. Figure 2 represents the weakness 502 and its inheritance.

Some weaknesses are specific to one or several programming languages. For example a SQL injection is not associated with Java or C language. Some weaknesses present in CWE can be applied to Android ecosystem. For example, the weakness 402 defines the vulnerability 'Transmission of Private Resources into a New Sphere' which could be regarded as a private data leak. This kind of vulnerability can be present in an Android application. It is possible to use the CWE enumeration to classify the results in our knowledge base. The CWE enumeration is actually used in some systems to classify the vulnerabilities found. But this system is not sufficient for our case, because it describes the types of vulnerabilities but do not give information about how to describe the

[14] https://standards.ieee.org/develop/indconn/icsg/mmdef.html
[15] http://cwe.mitre.org

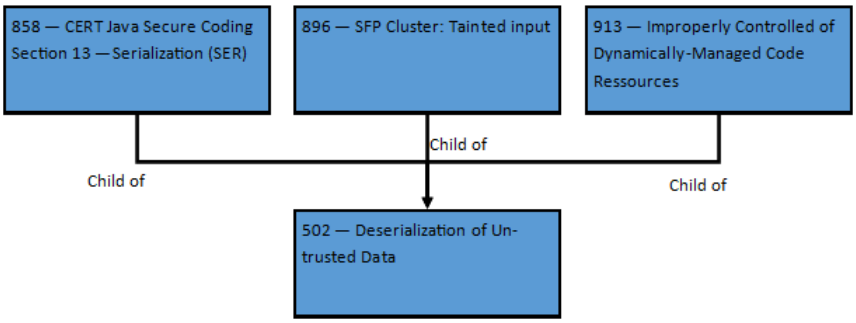

Figure 2: CWE 502 relationship

application which has the vulnerabilities. This standard has been made to categorize alerts (errors or vulnerabilities).

3.4.2 Common Vulnerabilities and Exposures

A practical usage of the CWE could be found in the Common Vulnerabilities and Exposures (CVE [16]) developed and maintained by the MITRE corporation. The CVE is a public list of vulnerabilities. Each vulnerability in the CVE has an unique identifier of the form $CVE-YYYY-NNNN$ where $YYYY$ is the year where the vulnerability has been discovered. The $NNNN$ part is an incremental counter which gives a unique identifier within the current year of the vulnerability.

One of the biggest vulnerability bases using the CVE standard is the National Vulnerability Database[17] (NVD) financed by the U.S. government.

Each CVE item is very specific to a piece of software and sometime to a specific version. Descriptions and links to complementary informations are provided with each CVE entry in the NVD. The vulnerabilities could be found by a manual inspection or by a tool. The creation of entries in our base will be automatically made by the analysis tools.

The NVD contains some vulnerabilities found in Android applications but no easy means are available to query them. There are also no information about the methods or tools used to find these vulnerabilities. An example of NVD entry concerning Android is the entry CVE-2013-7372[18] which has 310 (cryptographic issues) as CWE identifier and concerns a bad usage of a cryptographic algorithm.

To assign a CVE identifier to a vulnerability, it is necessary to be a CVE authority. For example, Apple is a CVE authority but can only publish CVE entries related to Apple vulnerabilities. An alternative is to submit the vulnerability to the Forum of Incident Response and Security Team (FIRST) to validate it.

Our security knowledge base must work in a decentralised way to allow any tool which raises an alert to submit and save it. It means that we will not have a central organisation which validates the alerts.

Several vulnerability bases use the NVD as reference and the CVE syntax as a way to store the different vulnerabilities found. An example of this is the Japan Vulnerability Notes (JVN) which

[16] http://cve.mitre.org
[17] http://nvd.nist.gov
[18] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-7372

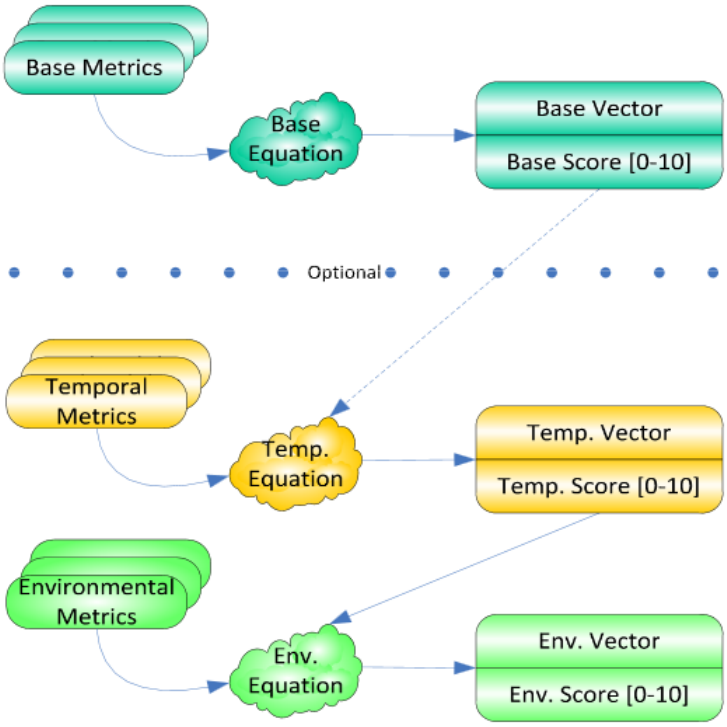

Figure 3: CVSS function

uses CVE syntax. The difference between the two bases is the replacement of *CVE* by *JVNDB* in the identifier.

3.4.3 Common Vulnerability Scoring System

A score about the criticality of the vulnerability is given in CVE . The Common Vulnarability Scoring System (CVSS) is used[19]. This metric allows users to sort vulnerabilities by severity. The score ranges from 0 to 10, a vulnerability which has a score in the interval $[7.0, 10]$ is considered as critical. If the score is in $[4.0; 6.9]$, the vulnerability is major otherwise the vulnerability is minor.

To construct the score, the CVSS function takes a vector of arguments and provides a base result. It is possible to refine the base result by using the environment and temporal equations which will increase or decrease the result according the arguments given to these functions. As illustrated in Figure 3, the temporal and environmental parts are not mandatory and the base score is sufficient to have a judgement about the criticality of a vulnerability. The NVD entry (CVE-2013-7372) used as example in Section 3.4.2 has a CVSS base score of 5.0.

To conclude this part, we have seen that some knowledge bases for Android exist but they are not adapted to our case as they are mainly focused on malware and Antivirus analysis. Also, they do not offer correlation of the different results. In our case, the security knowledge base will contain applications coming from different sources and these applications will be analysed by different kind of security analysers. Our security knowledge base will try to save all security related informations

[19] http://www.first.org/cvss

provided by these analysers to offer a centralized source of information about Android applications and their security properties to the different users.

It is important to have an idea about the different way in the literature for representing vulnerabilities. This help us to assess their limits. For example, one of the problems of bases like NVD is the lack of information about the application or the tools used to detect the vulnerabilities. Indeed, the vulnerabilities found in the NVD related to Android applications are not detailed. They only give the package name and the version number to identify an application, but no information about the developer's certificate or the source of the application. This is important because repackaging attacks exist and malwares usurp generally the package name and the name of an application to distribute a hijack version on third party stores[20].

For these reasons, we think that the existing solutions are not sufficient and must be improved. The definition of our model will be a first step to the construction of a standard to express vulnerabilities and errors in Android applications.

4 Design of unified output format and metrics

4.1 Standard output for analysis results

In this part, we illustrate an approach for the normalization of the output of different Android analysis tools.

4.1.1 Requirements

As explained in Section 3 there are several vulnerability data bases and alerts representation. There are all concerned with identifying and classifying different alerts. To resolve this problematic, several propositions have been made by the research world like the CVE for vulnerabilities or the IDMEF for the intrusion detection system but none of these standards are compatible with our security knowledge base and the Android ecosystem. We define a new standard to describe alerts raised by Android application analysers present in our Security Knowledge Base. The goal is to describe the alerts and the reasons of these alerts in a common file which could be read by different tools (analysers ...). Several requirements have been found and are explained underneath.

The first requirement for our standard output is to be object oriented. This facilitates the definition of the model and future modifications by limiting the impact to one or more objects and not to the entire norm.

The model will face several problem like the heterogeneity architecture of the different analysers. This heterogeneity implies different outputs and formats according to the type of the analyser like explain in Section 3.2. For two tools doing the same type of analysis, the outputs could be different from a tool to another. One example could be imagined with two tools which use static dataflow analysis. The first tool could write the all path between the source and the sink (exit point) of the dataflow found (FlowDroid for example). The second could just gives the sink. It implies a flexibility for the norm to do not reject some results because they are incomplete. It means that almost all fields of the standard are not mandatory to be flexible with the different possible output coming from different tool of the same category. These fields must be filled when it is possible.

Another requirement is to be flexible relative to the future updates. It is really necessary to avoid that our standard is valid for a few months or years and unusable after. The object-oriented syntax

[20] http://www.androidauthority.com/android-malware-genome-project-legitimate-apps-repackaged-89045

chosen for our standard facilitates the updates but it also implies to create several mechanisms in our standard to presage the creation of new tools categories to have a backward compatibility. Each new versions of the standard must be able to propose mechanism to use the older versions of our standard (even if we loose some informations). For these reasons, generic object must be created to support backward compatibility.

It is necessary that the standard could describe the three part of an environment: the analysers, the application and the alerts. Like explain above, it is necessary to have enough information about the application to identify it in a set of application. Several information are necessary to describe the application (certificate, package name). It is very important to do the same thing for the tools to compare them or to compare several versions of the same tool. Each category of tools must have a way to describe their results. Several categories have been identified:

- Dataflow analysis for the tools which use the dataflow analysis or taint analysis to find leaks of private data.
- Programming error analysis. This category is used for tools which try to find some errors in the source code. It could be misuse of SSL technology or bad protection of components which allow hackers to hijack them.
- Permission analysis. It refers to analysis which try to find some properties related to permissions like a bad combination of permissions which allows data leaks or malicious behaviour.
- Security policy analysis. This category is used for tools which use a security policy and report violation of this security policy.

This list is not comprehensive and must be completed to handle more categories of analysis. The following section will describe the definition of this standard.

4.1.2 Definition

As explained in Section 4.1.1, our standard is based on an object oriented representation to increase the modularity and the flexibility in case of eventual modifications. This work presents a new approach and needs more time to be improved.

Figure 4 describes the actual definition of the Standard Output for Android Application Analysis (SO3A). A more complete definition of the standard (using the RelaxNG syntax) is available in Annex A.

The standard is composed of three parts: result, application and analyser. Each part is represented by an object which gives a good description of the corresponding 'real' object. The analyser part describes the tool by giving it an identifier, a name, a version code and a type which is 'static' or 'dynamic'. The goal is to differentiate static tools which run on a server from dynamic tools which run on an emulator or a device. Unlike IDMEF and CVE, several tools and results could be aggregated in the same file. The goal behind this is to create an unique file with all the alerts raised by different tools for the APK.

The application part of the standard file should describe an application (permissions ...). Like explain in Section 3.1, the package name is used by Android to identify an application on the device but it is not enough to uniquely identify an application. The version code and the certificate could be used to have an unique identifier for the application. In our case, these informations are saved in the application part of SO3A but a simpler identifier can be used to retrieve faster an application

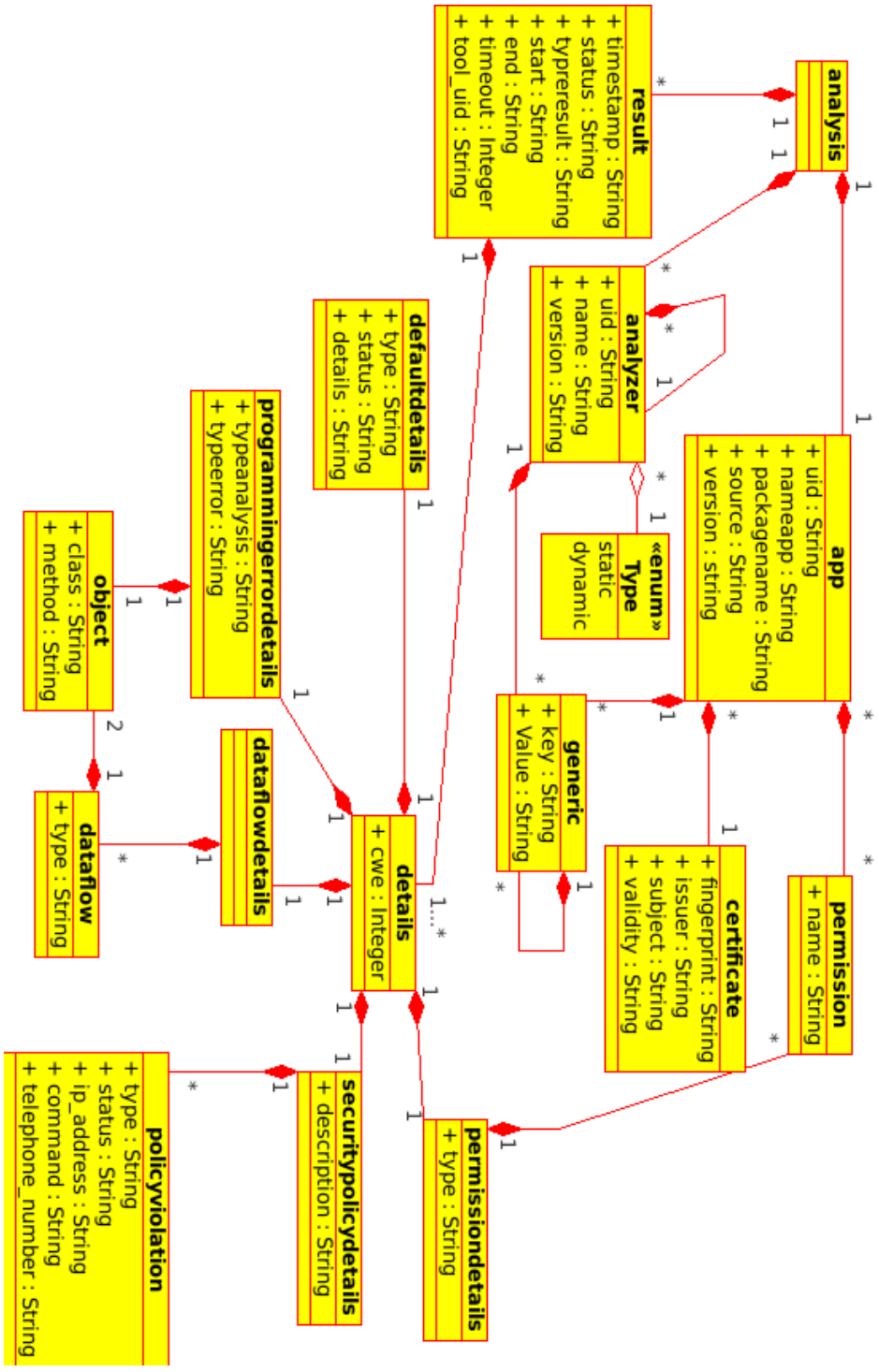

Figure 4: Class Diagram of the Standard Output for Android Application Analysis(S03A)

in the S.K.B. David Barrera uses as identifier the hash of the developer's certificate and of the APK files [3]. The user is free to choose another way to identify the applications. The only requirement is to choose a method which will not give the same identifier to two different applications and does not give two identifiers to the same application (in case of re-upload of the same application). The cryptographic hash functions meet these requirements. These functions map datas of arbitrary size (input called message) to data of arbitrary size (output called hash). These functions do not allow to modify the message without any changes in the hash. And it is infeasible to find two different messages with the same hash. In our case, we recommend to use the output of SHA256 function as identifier with the APK file as input. It avoids the extraction of the certificate of the APK and it is easily computable.

The result part is composed of a time stamp to identify a result coming from the same tool at different point in time. The 'status' and the 'typeresult' attributes are used to describe the result. In our case, the 'status' field describes the analysis in a human readable sentence and the 'typeresult' field is for classifying the result (e.g., dataflow, security policy ...). The 'start' and 'end' attributes are used to declare the time interval of the analysis for a dynamic tool. For the static tool, the 'timeout' attribute stores the possible timeout used.

Each result has detail which contains a CWE identifier. It is a fast and easy way to classify the alerts. Indeed, in case of alert describing a vulnerability, it will give more information. In case of alert describing malicious behaviour, the CWE identifier gives informations about the vulnerabilities used by the malware. We recommend to expand the CWE list by adding negative values to indicate some error during the analysis which prevent the security knowledge base to have pertinent informations coming from this tool for this application. It is possible that the tools present some malfunction during the analysis (timeout error ...). The zero value could be used by an analyser to declare that nothing suspicious has been detected during the analysis. It is also possible to refine CWE by adding entries which inherit from existing CWE. An example will be the creation of entries according the types of information which leak. These entries will be children of the entry 402 (Transmission of private resources into a new sphere) presented in Section 3.4.1.

Each category of detail tries to handle a maximum of information for each alert. Of course, this standard will not replace the original output but the goal is to have a good summary of the alert. Each result has the unique identifier of the analyser too. Several analysers could be described in the file and each analyser could have raised several alert. It is necessary to keep the link between the result and the analyser used.

For each category, by using the existing tools, it has been possible to create detailed fields. The different categories identified and implemented are :

- Policy security violation
- Programming error
- Unnecessary permissions
- Illegal dataflow

The unnecessary permission is a easy category because it consists of a list containing the unnecessary permissions declared by the application. The data flow details are very close to the characteristics identified in Section 3.2. The policy violation part is more difficult to handle because of the different way to declare a security policy. Like explain above, there are no standard

way to describe them. The policy must be saved entirely. The best way to provide a maximum of information about the reason of the alert is a sentence which explains the reasons and different characteristics linked to this alert like IP address or telephone number ... Our study of tools using policy shows that rule are often used to describe the security policy. In this case, it will be great to show the violated rule as reason of the alert.

This standard gives a first answer to the standardisation of the output for the Android application analysers. The lack of standard for the alerts prevents the easy reuse of the existing results.

4.2 Scoring function

In this part, we describe the scoring function which gives a global score to the application according the alerts raised like CVSS gives a score to vulnerabilities as explained in section 3.4.3. It will allow the user to rank the different applications of the knowledge base. As mentioned previously in section 3.4.3, it is not possible to use the CVSS function directly. Unlike our scoring, which must be designed to score an application with several alerts, CVSS is designed to characterize only a vulnerability. The CVSS function could be used but only to characterize each alerts of an application separately.

The alerts could be classified in different categories like private data leak, programming errors, etc. A analyser could declare several type of alerts. For example, Flowdroid, presented in section 3.2.1, declares that it can raise two types of alerts: 'personnal information leakage', 'device information leakage'. Each alert raised must be associated to a category. Several analysers could raised alerts in the same category. By aggregating all categories declared by all tools, we will have the set of possible categories. For each category C_i of alerts, a category set called Cs_i regrouping all alerts associated exists. The set Cs regroups all existing category sets. Each alert has a score which gives it criticality. The idea to build the global score of the application is to add the maximum score of each category set like explain in (1).

$$Final\_score = \sum_{c \in Cs} max(c) \tag{1}$$

Indeed, we want to give more importance to applications which have several types of vulnerabilities. If an application has several type of vulnerabilities, it will allow a larger set of possible actions for the hacker to exploit the device or the application. It is for these reasons that the diversity of vulnerabilities is more important for us than the frequency of the same type of vulnerabilities.

This method requires to have the list of tools used in the security analysis to contrast the results of an application which has been analysed by one tool with another which has been analysed by several tools. It allows also the user to abstract away from the categories of vulnerabilities that he does not want to consider. For example, if the user does not care about the leak of his location, it will have the option to disable this category.

4.2.1 Example

Let us consider a simple example of application and vulnerabilities to illustrate our metric. We have two tools: analyser A which detects private data leaks and another one called B which detects programming errors. The first analyser could detect two types of illegal flows and raises two types of alert :

- User information leak of category C1
- Device information leak of category C2

The second tool raises one type of alert:

- Wrong usage of API of category C3

Each alert raised by one of these tools is linked to one category (C1, C2 or C3). The application is first analysed by the analyser B which detects two vulnerabilities vul_1 and vul_2. The first one (vul_1) has a CVSS score of 2.3 and the second one (vul_2) has 3.5 as CVSS score. These two vulnerabilities are in the category C3. As the global score is the sum of the highest scores of each category, the application get the global score corresponding to vul_2 because there are results only in one category (C3) and the highest score of C3 is vul_2 whether is 3.5.

It is possible to have a global score for an application without the need of the results from all analysis. This score must be associated with the list of analysers which have effectively provided a result.

A few time later, the analyser A raises two alerts : Vul_3 in category C1 and Vul_4 in category C2. Vul_3 has a score of 1.7 and Vul_4 has a score of 6.5. It means that the application has a global score of 11.7. This score is obtained as follows.

$$Final\_score = max(2.3, 3.5) + max(1.7) + max(6.5) = 3.5 + 1.7 + 6.5 = 11.7$$

The CVSS function can be replaced by any scoring function which describes a vulnerability and which use an increasing scale to describe the severity of the vulnerability, i.e., the more the vulnerability is critic, the higher is the score. This scoring function allows the users to sort several application analysed by the same tools.

5 Implementation of the knowledge base

5.1 General design

The first ingredient that the S.K.B. must offer is downloading applications from stores. No store offers an API to browse its content or to download applications. We decide to have the Google Play Store as first provider of applications. As it does not offer API for downloading applications and related information such as rating, etc., we decided to use the same protocol use by the mobile version of Play Store to communicate with the Google servers. The applications present in the Google Play Store are classified by category. We have collected information about categories of applications as well as other information available such as the application description for research purposes. All these information are specific to the Play Store. Other store will not have necessarily the same information.

The second part of the S.K.B. concerns the analysers. We abstract the tools and represent them by an object (called instance) such that the S.K.B. does not see any difference between the tools. The instance is a black box which has an entry point and an return point for the analysis of Android applications. The goal is to build an adapter for each tool which runs the analysis with the right parameters and return the results in the appropriate format. Through this mechanism, it is possible to have several instances based on the same tool. It is useful for tool with large possibilities of customisation like those using security policies for example. The instance takes several arguments

to run an analysis: the APK file, the output path (to save a copy of results in a file) and a timeout value to prevent long runs.

We have a client which downloads applications and analyses them automatically with the different instances of tools available. Several ways to submit new APK to the S.K.B. exist. The first one is by giving a package name. The S.K.B. finds on the Play Store the up-to-date version of the application and the associated APK file. The second method is by directly submitting an APK file to the S.K.B. In both cases, a test is carried out to check the presence of the APK file in the S.K.B. For this, we identify an APK file with its SHA256 hash and check if the identifier does not already exist in the S.K.B.

5.2 Tools used for the analysis

Several requirements have limited our choices in adopting existing tools. The tool must be available to public and it should be possible to run it on a server. Some tools are no longer maintained. Others are not publicly available. For these reasons, many tools have been excluded.

A more detailed description of the functionality and results of the chosen tools is given below.

5.2.1 Mallodroid

Mallodroid deals with SSL errors in Android applications. Such vulnerabilities are mainly programming errors, like the non-audit of certificates by the application. These errors allow a hacker to usurp the identity of a server. Mallodroid performs a shallow analysis at the syntaxic level. It means that the analysis does not take into account the relation between the functions. As explained in Section 3.2.1, it looks at the inheritance relation between different classes and the redefinition of some methods and the usage of dangerous classes too. This is the list of the different errors that Mallodroid can detect:

- **Trustmanager error:** this alert is raised when an error appears in the verification of the signer and the subject of the certificate.
- **Insecure SSL socket:** this alert is raised when bad settings of SSL has been made making the connection insecure.
- **Hostname verifier Error:** this alert is raised when errors have been found in the verification of the hostname.
- **SSL Error:** bad usage of SSL (i.e., mix of secure and unsecure connections, etc)

5.2.2 Flowdroid

Flowdroid is a static analysis tool which detects private data leaks in Android applications by static dataflow analysis. For each flow found between a source and a sink (an exit point for the data), Flowdroid returns the source, the sink and the complete path between them. This analysis takes into account the language semantics. It means that Flowdroid takes into account the relations between the function and not only the syntax of the source code. An example of detected flow is presented below.

```
Found a flow to sink staticinvoke <android.util.Log: int i
(java.lang.String,java.lang.String)>("SR", $r15), from the following sources:
    - virtualinvoke $r1.<android.os.Bundle: java.lang.String getString
    (java.lang.String)>("xml") (in
    <com.adknowledge.superrewards.ui.activities.SRPaymentMethodsActivity:
    void onCreate(android.os.Bundle)>)
        on Path [virtualinvoke $r1.<android.os.Bundle: java.lang.String ...]
```

This example describes a flow where data coming from intent leak through logs functions. The entire path between the source and the sink is described in the result but has been shortened to display it.

5.2.3 Evicheck

Evicheck is a tool based on Androguard developed in the context of the AppGuarden project at the University of Edinburgh. It accepts a security policy specifying antimalware rules for an application and generates a certificate if the policy is valid.

The security policy can express prohibitions and obligations. If a rule which expresses a prohibition is violated, only the element (function) violating the rule is given. If an obligation rule is violated, the violated rule is provided.

An example of rule is given below. this rule tries to prevent the sending of contacts by SMS. The first part defines the context of the rule, i.e., the objects of the application concerned by the rule. And the second part describes a behaviour (forbidden or mandatory).

$$\textit{ACTIVITY\_METHOD: } \neg \textit{ READ\_CONTACTS } \neg \textit{ SEND\_SMS}$$

ACTIVITY_METHOD defines the context of the rule. In this case, the rules concerns only the activities methods. Several tags are used in the security policy to define the usage of protected API. The tags, like *SEND_SMS*, are given to a list of method which uses protected function. For *SEND_SMS*, Androguard uses a map between permissions and functions to put the tag to functions which can send SMS. The security policy used in the S.K.B. has 36 rules to prevent mainly data leaks and eavesdropping (audio and video).

5.2.4 Comdroid

As explained in Section 3.2.1, Comdroid tries to find the programming errors in the communication between different components. This tools works at a syntaxic level and will check some specific functions and see if the development guidelines are respected. It means that Comdroid parses the code to see if the different communication between them is well developed. Indeed, if the possibilities to interact with a component are not secured, it gives the possibility for the hackers to hijack them or to launch it for malicious actions. Several types of errors could be found by Comdroid and are described below:

- **Action misuse:** bad usage of component and intents functions.
- **Possible Hijacking or Sniffing:** possibility for another application to call a component to use it.

Category	Number of APKs
Arcade	2
Books and reference	84
Business	54
Comics	58
Communication	108
Education	71
Entertainment	69
Finance	48
Game music	1
Health and fitness	60
Libraries and demo	53
Lifestyle	1
Media and video	43
Medical	56
Music and audio	54
News and magazines	93
Personalization	92
Photography	55
Productivity	75
Shopping	24
Tools	78
Travel and local	2
Weather	59

Table 2: Categories of downloaded applications

- **Possible malicious component launch:** it describes a possible malicious action which try to launch a component to hijack this component.

These alerts could be classify in two categories: good and malicious. The possible malicious component launch alerts that Comdroid could raise alert the user of a possible malicious behaviour. The others describe vulnerabilities or errors which could also happen in a benign behaviour.

5.3 Applications and analysis

The security knowledge base is actually composed of 1240 APK files coming from Google Play Store. We keep a maximum of different versions of the same application. Several information like rating are extracted from Play Store and saved for research purposes. We store the APKs and assign a unique identifier to each file. We use the hash function SHA256 as identifier generator. An application could be present in two different stores. In our base, if the generated identifier of a new apk file is the same as the identifier of an already saved APK file, it means that the two applications are the same and the new file will not be saved. The repartition of these APKs file according the category they declared on the Play Store is shown in Table 2.

	MalloDroid	Flowdroid	Evicheck	Comdroid
Alert	354	174	300	609
Benign	298	13	341	16
Aborted	5	264	7	32

Table 3: Analysis results by tool

Our base extracts the certificate of the application and save it to track the versions of an application (identified by a package name) signed with the same certificate and to find the different applications developed by the same developer. Indeed, it is possible to use the same certificate for several applications. For example, we have 24 APK files signed with the same certificate. These 24 APK files represent 14 different applications. The summary of analysis made by the tools present in the security knowledge base is shown in Table 3. All applications have not yes been analysed. We define three categories of analysis result: 'benign' for the analysis completed without alerts, 'errors' for the aborted analysis and 'alert' for the analysis with at least one alert raised.

More detailed results by tools are available in Table 4. For each tool, we classify the alerts corresponding to the types of alerts raised. For Mallodroid and Comdroid, the categories are similar to the categories of errors these tools could detect described in section 5.2. For Flowdroid, the categories are related to the type of data which could leak. And for Evicheck, the category is related to the tag present in the rules.

There is a high number of application with at least one alert raised for Comdroid and more particularly in three categories of alerts (*Possible broadcast theft*, *Possible activity hijacking* and *Possible malicious broadcast injection*. It is possible to consider that the detection of these alert is not well handled by Comdroid and a lot of false positive alert exist.

6 Experiments and evaluation

6.1 Is there a correlation between application evolution and alerts ?

An interesting point to study is the relation between the evolution (or updates) of an application and the number of alerts raised.

We first need to define what is an update of an application. We want to study the evolution of an application developed by the same developer and not an application which has been repackaged. An update of an application is always an application which uses the same package name as the old version and has been signed with the same certificate. The new version must have a version code (an integer present in the APK file) higher than the one of the current version.

If an application fulfils these conditions, it is considered as an update. An important thing about the version code is that there are no guidelines in Android documentation to describe it. It is possible to increment 1 the version code for each new version but it is also possible to use the date as version number. The only requirement is to have a version code higher than the one of the current version.

Our first contribution in this part is the definition of a metric giving the difference of alerts raised between two versions of an application. For this, we define a metric called Δ which represents the difference of alerts between two versions of the same application. This value is divided in two

Mallodroid	Application with at least one alert raised
SSL error	214
Insecure SSL Socket	177
Hostname verification error	131
Trustmanager error	245

Comdroid	Applications with at least one alert raised
Possible broadcast theft (sniffing)	477
Possible activity hijacking	600
Possible service hijacking	376
Possible action misuse	312
Possible malicious service launch	103
Possible malicious activity lunch	235
Possible malicious broadcast injection	524

Flowdroid	Applications with at least one alert raised
Phone state	47
Databases	21
Bundle	210
Handler	15
Intent	223
Location	47
Components	29
Internet	61
Audio	0
Browser	0

Evicheck	Applications with at least one alert raised
Phone state	15
Contacts	186
Storage	77
Camera	65
Location	220
Audio	156

Table 4: Analysis results by tools

	MalloDroid	Evicheck	Flowdroid	Comdroid
$\Delta < 0$	14	29	5	41
$\Delta = 0$	199	161	15	83
$\Delta > 0$	21	40	7	91
$\Delta_{del} = 0$	141	212	27	39
$\Delta_{del} > 0$	93	18	23	176
$\Delta_{add} = 0$	140	189	27	38
$\Delta_{add} > 0$	94	41	23	177
$\Delta_{add} > 0$ & $\Delta_{del} = 0$	10	37	3	1
$\Delta_{add} = 0$ & $\Delta_{del} > 0$	9	8	3	0
$\Delta_{del} = \Delta_{add} > 0$	68	0	3	45

Table 5: Evolution of alerts number through the versions

parts Δ_{add} and Δ_{del}. The difference Δ is computed as shown in (2). It allows us to see if there are less alerts in the new version than in the current one. It gives a global view of the evolution of alerts in the application.

$$\Delta = \Delta_{add} - \Delta_{del} \tag{2}$$

6.1.1 Alerts difference between two version of an application

We performed experiments on a set of 1240 APK files representing 1005 different applications. From these 1005 applications, 164 have multiple versions which are present in the knowledge base. The maximum number of different versions for the same application is 4.

In our study, we use Evichek, Mallodroid, Comdroid and Flowdroid to analyse the applications. In each of these tools, we found some bugs which prevent us from analysing all the applications. We divided the results in different sets.

The first one concerns updates which delete vulnerabilities (with negative Δ), the second one concerns updates which add vulnerabilities (with positive Δ) and the last one is about updates which do not change the alert number (with a null Δ). We also count the number of updates with positive values for Δ_{add} and Δ_{del}. We identify also the updates with only addition or deletion of alerts and, to finish, we count the number of applications which add and delete the same number (higher than 0) of alerts.

Table 5 summarizes our results for the different tools depending on different values of Δ. We can see that we have different classes of results, e.g., when $\Delta_{del} = \Delta_{add} > 0$ represents the case where the number of alert which are added is equal to the number of alerts which are deleted. This situations represents 34% of the updates for Mallodroid and 0% for Evicheck.

The low number of results for Flowdroid is due to its intense consumption of resources which where beyond we could provide to run it in a reasonable amount of time.

The important thing to notice is that the number of alerts raised is the same for a large majority of updates. A part of these updates (66% for Mallodroid, 100% for Evicheck, 80% for Flowdroid and 46% for Comdroid) has exactly the same number of alerts in the previous and the new version. A possible explanation of this could be that developer is probably not aware of these vulnerabilities.

The number of updates which add alerts is very important for Comdroid. This could be ex-

plained by the nature of these updates. Comdroid looks for possible component misuse or hijacking. Some updates are here to add content (i.e., code) to the application and others to fix bugs present in the source code (not necessary security bugs). By adding content to the application, the developer increases the number of components and consequently the possibility to hijack them if they are not protected. For tools like Mallodroid, the situation will not change if the update does not affect the part of the application which handles the internet connection. For Evicheck, unless a service or an activity is modified by adding or deleting API functions related to some permissions, the number of alerts will not change as Evicheck rules are expressed over these ingredients.

6.1.2 Alert evolution vs. application similarity

The intention behind developers updates may be to fix some vulnerabilities. We want to keep track of such updates by taking into account the ratio of the percentage of alerts deleted over the percentage of code changes in the application. If this ratio is bigger than one, the updates are likely to be security fixes. The intuition is that important changes in the alerts number without significant modifications in an application may suggest a desire for the developer to fix the vulnerabilities and not to change the code to add some new features or functionalities. We define $\Delta_{ratio} = \frac{\Delta}{S}$ with S the number of alert raised for the current version of the application.

This experimentation is possible with Androsim[21], an application included in Androguard, which computes the method similarities between two applications and allows us to have informations about the modifications caused by an update. The tool gives us a global rating of modifications but also counts the number of methods added, deleted or modified. These measures will allow us to see if the modification in the set of alerts raised is linked to the similarity between applications.

The results for Evicheck, Comdroid and Mallodroid are displayed in Figure 5. It illustrates updates according Δ_{ratio} and the percentage of similarity.

The first thing we notice is that the differences between two versions of an application are very limited and a majority of updates produces a new version which has at least 85% of similarity with the current version. There are also several updates which do no modify methods (100% of similarity). These updates changes resources included in the APK and do not influence the evolution of alerts raised.

It seems that the repartition of results does not follow any logic. One explanation for this could be the no existence of correlation between syntactic changes and their semantic impact with regards to security.

Some updates, in Figure 5, have a particular profile which deserves further investigation. Some update completely remove alerts in the new version of the application. These are the updates with -100% as Δ_{ratio}. We can find 5 of such updates with the respect to alerts raised by Evicheck and 4 with the respect of alerts raised by Mallodroid as we can see from the respective plots (Figure 5). On the other hand, several updates add a high percentage of alerts in the new version. These represent the updates giving raise to a percentage higher than 100% of alerts number between the two versions. At the same time, the updates representing these two extreme cases could have the same percentage in similarity.

[21] http://code.google.com/p/elsim/wiki/Similarity

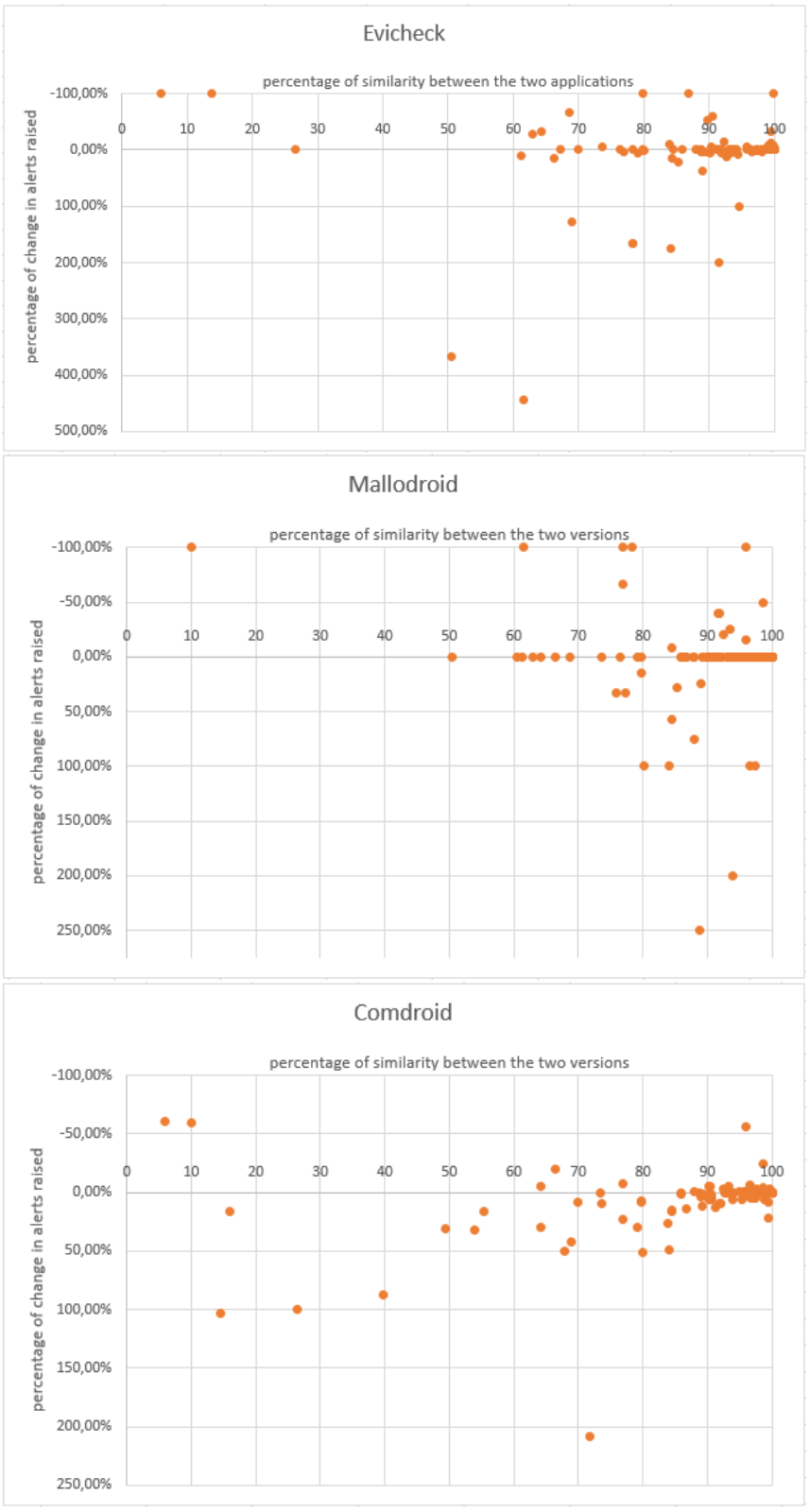

Figure 5: Comparison of alerts raised

6.2 Do the same developers commit the same mistakes ?

Actually, no reputation model based on the alerts raised by several analysis tools exists to have an idea of the developer skills.

We study the creation of such a model based on the alerts raised in the applications the developer has written. Our goal is to improve the quality of the code produced by developers by drawing their attention to issues which are present in their applications and about which they have little knowledge.

Two applications are developed by the same author if they have two different package names but are signed with the same certificate.

The first step is to categorize the alerts raised for all applications created by the same developer. With this categorisation (possibly with the CWE identifier of the alerts present in the SO3A file), we will have the list of errors made by the developer in the application. By computing the sum of the categories of errors made by the developer in the applications present in the S.K.B., it is possible to compute $Total\_Categories$ which is the reputation of the developer. To take into account the improvements made by the developer, we use the most recent version of each application.

The higher is the score and the worse is his reputation. A score of 0 means that no alert has been raised for any of the applications written by the developer.

It is possible to focus the score on the errors that a developer commits in every application he wrote. By using the previously mentioned classification of alerts, it is possible to know the categories of alerts shared by all the applications. We call the number of these categories $Frequent\_Categories$. This number together with $Total\_Categories$ provides a score about the developer reputation.

6.2.1 Example

Let us assume that Bob developed two different applications: Application 1 and Application 2. The first one raised alerts in two categories called: Category A and Category B. The second one raised alerts in two categories: Category B and Category C.

Bob will have a $Total\_Categories$ value of 3, because there are alerts in three different categories. The $Frequent\_Categories$ will be 1 because only one category of alerts is shared between every application written by Bob.

Such model could help users with informations which allow them to compare different developers.

6.3 Is it possible to correlate the results coming from different tools ?

A main goal for us is to increase the confidence in the global analysis result by aggregating results coming from the different tools. The scoring function described in Section 4.2 is a first answer to this question.

Actually, no system exists to correlate the results coming from different analysers. We carry out an experimental study to check if a the categorisation of alerts is useful to correlate them.

In our case, the correlation is possible between Flowdroid and Evicheck which have the same kind of alerts. This is not obvious for Mallodroid and Comdroid as they deal with other types of vulnerabilities which are not handled by the other tools of the S.K.B. and vice versa. The first step is to find exactly what kind of alerts could be raised by Flowdroid and Evicheck. For example, They both try to detect leaks of private such as phone state or user location. To filter out the

alerts by only keeping the one related to the previous information (phone state and user location), we map the Evicheck rules to the sources used of Flowdroid.

Concretely speaking, we map the rules in Evicheck containing the tags *location* and *phone_state* to sources in Flowdroid which are reading the phone state or the location. For Evicheck, 12 rules prevent the leak of data (user location or device state). For Flowdroid, six methods (sources) could be associated with the category *location* and seven other methods identified as sources could be associated with the category *phone_state*.

For each application which has been analysed by the two tools, we count the number of alerts which could be associated with these categories. For both categories, there are 47 applications with at least one alert raised by Flowdroid. None of these 47 applications raises an alert for the subset of rules choosen for Evicheck. At a first sight, it seems that no correlation exists between the results of the two tools.

After deeper inspection of these results. We have found out that the alerts raised by Flowdroid and relative to the categories declared earlier, 22 different sinks have been identified. 11 from them are related log of Android operating system that applications could use. Three sinks are related to the application preferences (dictionary of value saved on a file and used by the applications), 3 are related to OutputStreaming methods and only one sink (method) present in an alert is relative to the Internet usage. Its presence is involved in 2 flows. Only them could be detected by Evicheck. The reason is that no rule takes into account the log methods as a possible sink for data. The rules present in the security policy prevents the device leaking the device state or the user location via Internet or SMS. The other way to write or send data are not handle by the security policy in Evicheck. The conclusion of this experimentation is that for the correlation of results concerning private data leaks, the sink is as important than the source in our categorisation and the security policy used by Evicheck must be improved to handle more possibilities of data leakage.

It is important to understand why the two alerts related to Internet usage have not been detected by Evicheck. As explained in Section 5.2.3, Evicheck uses a map between APIs and permissions. This map is not complete and the method used is not present in that map. To resolve this, it is possible to use a tool like PSCout, presented in Section 3.2.1, as back-end to automatically generate the map between APIs and permissions.

To conclude this section, the correlation between tools seems possible if the analysis concerns the same kind of properties. The experiments have shown no correlation between our two tools, this is no due to the analysis that each tools tries to implement but it is rather due to some technical details in the implementation. This also show us the importance of combining tools as we can improve Evicheck by using PSCout to generate the map it uses.

7 Conclusion

The goal of this internship is to develop a security knowledge base for Android applications. For this, we have addressed several questions. Is there a standard output format for Android analysers ? Is there a metric to rank applications according to the analysis results ? Is there a relation between alert evolution and different versions of the same application? Is it possible to correlate alerts coming from different tools ? Finally, can we create a reputation model for developers based on analysis results ?

The popularity of Android and the high proliferation rate of malware targeting it make such knowledge base very valuable to researchers, developers as well as the standard users.

For the creation of such a knowledge base, we needed to propose a formalism which describes the results of different analysers. We have also presented a metric to rank applications based on the categorisation of the different alerts.

By focusing on the developer side, we have described a reputation model for developers based on the frequency of the type of mistakes they commit. We have also addressed the question of correlating results coming from different tools by comparing the two most related tools we have: Evicheck and Flowdroid.

As final product, the Appguarden project team of the university of Edinburgh has now a security knowledge base which includes 4 tools and more than 1000 applications. For each application the results of analysis performed by the different tools is available.

Some obstacles have been faced during this internship. One of this is the choice of tools which is quite limited as most of them are not publicly available. In addition to that, several available tools are still in a development phase. Moreover, the continuous changes in the Android platform requires a continuous maintenance in the associated analysis tools to keep them up-to-date. The availability of more tools in the future could help us to have more qualitative as well as quantitative results. We have proposed SO3A as standard output to take into account the results of tools we are using as well as eventual new tools. We need to integrate more type of results. As a part of our metric, we have used the CVSS function however our method is independent from the CVSS function, it means that we can use any other function which characterizes a vulnerability.

References

[1] Kathy Wain Yee Au. Pscout: Analyzing the android permission specification, 2012. Master Thesis, University of Toronto.

[2] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. AppGuard - Real-time policy enforcement for third-party applications. Technical Report A/02/2012, Saarland University, 2012.

[3] David Barrera, William Enck, and P.C. van Oorschot. Meteor: Seeding a Security-Enhancing Infrastructure for Multi-market Application Ecosystems. In *Mobile Security Technologies Workshop (MoST)*. IEEE, 2012.

[4] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: Behavior-based malware detection system for android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 15–26, New York, NY, USA, 2011. ACM.

[5] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[6] William Enck, Peter Gilbert, Byung-gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Operating Systems Design and Implementation*, page 393 to 407, Vancouver, 2010. USENIX Association.

[7] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: An analysis of android ssl (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 50–61, New York, NY, USA, 2012. ACM.

[8] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves L. Traon, Damien Octeau, and Patrick McDaniel. Highly precise taint analysis for Android applications. Technical Report TUD-CS-2013-0113, TU Darmstadt, Darmstadt, Germany, May 2013.

[9] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. Scandroid: Automated security certification of android applications, 2009.

[10] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. ScanDal: Static analyzer for detecting privacy leaks in android applications. In Hao Chen, Larry Koved, and Dan S. Wallach, editors, *MoST 2012: Mobile Security Technologies 2012*, Los Alamitos, CA, USA, May 2012. IEEE.

[11] Federico Maggi, Andrea Valdi, and Stefano Zanero. Andrototal: A flexible, scalable toolbox and service for testing mobile malware detectors. In *Proceedings of the 3rd Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*. ACM, November 2013.

[12] Robert A Martin and Sean Barnum. Common weakness enumeration (cwe) status update. *ACM SIGAda Ada Letters*, 28(1):88–91, 2008.

[13] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *Proceedings of the 6th European Workshop on System Security (EUROSEC)*, Prague, Czech Republic, April 2013.

[14] Rubin Xu, Hassen Saidi, and Ross Anderson. Aurasium: practical policy enforcement for android applications. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, page 27 to 27, Berkeley, CA, USA, 2012. USENIX Association.

[15] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.

Annex

Annex A - Definition of the Standard Output for Android Application Analysis

The definition uses the Relax NG Compact Style[22] to describe the syntax of the file format used by our standard. This has been chosen because of the simplicity of the syntax but also for technical reasons (compatibility with Domain Type Definition for XML file).

```
default namespace rng = "http://relaxng.org/ns/structure/1.0"
namespace local = ""
datatypes xsd = "http://www.w3.org/2001/XMLSchema-datatypes"
start = Analysis
Analysis = element result {App, Analyzer, Result*}

# Description of an application
App = element app {
        attribute uid { text }, # The unique identifier is in our case the hash of the APK
        attribute nameapp { text },
        attribute packageName { text },
        attribute source { text },
        attribute version { text },
        element certificate { Certificate }, #Description of the certificate
        element permissions { Permission* }, # List of permissions
        element generic { GenericInformations }*
}

#Description of the certificate used to sign the APK
Certificate = element certificate {
        attribute fingerprint { text },
        attribute subject { text },
        attribute issuer { text },
        attribute validity { text }
}

#Description of a permission
Permission = element permission {
        attribute name { text }
}

#Description of an analyzer
Analyzer = element analyzer {
        attribute uid { text }, #Unique identifer of the tool in the environment
        attribute name { text }, #Name given to the analyzer (used for a GUI)
        attribute version { text }, # Version number (number, uid commit ...)
        element analyzer { Analyzer* }, #Used to create analyzer composed of several analyzer
```

[22] http://www.relaxng.org/compact-tutorial-20030326.html

```
            (ex: Several instace of the same tool with different security policies)
        attribute type { "static" | "dynamic" },
        element configuration { GenericInformations }*,
        element generic { GenericInformations }*
}

#This Element describes the result of an analysis
Result = element result {
        attribute timestamp { text }, #It is a way to have several version of an analysis and
            sort them by date
        attribute start { text }, #Beginning of the analysis
        attribute end { text }, #End fo the analysis
        attribute timeout { text }, #Timeout used for static tools
        attribute id_tool { text }, #Analyser which has raised this alert
        element status { text }, #The status is the sentence displays in the table of the
            GUI. It is an abstract of the result
        element typeresult { text }, #Example : Dataflow, ...
        element details { Details* } #More information about result
}

Details = element details {
        attribute cwe_id { integer }, #CWE extended with 0 and negative values
        element specificdetails { DataFlowDetails | SecurityPolicyDetails | PermissionDetails
            | ProgrammingErrorDetails | DefaultDetails }
}

#Each type of analysis will have a element to describe the results.
DataFlowDetails = element dataflow {
        element flows { Flow* }
}

Flow = element flow {
        element type { text },
        element source { ClassDetails },
        element sink { ClassDetails },
        element path { text }
}

#This element will describe an object (for example a class or a method which has a
    vulnerabilities
ClassDetails = element object {
        element class { text },
        element method { text }
}
```

```
SecurityPolicyDetails = element securitypolicydetails {
        element description { text },
        element policyviolations { PolicyViolation* }
}

PolicyViolation = element policyviolation {
        element type { text },
        element status { text },
        element ip_adress { text }*,
        element command { text }*,
        element telephone_number { text }*,
        element generic { text }*
}

PermissionDetails = element permissiondetails {
        element type { text },
        element permissioninvolved { text }*
}

ProgrammingErrorDetails = element programmingerrordetails {
        element typeanalysis { text },#SSl misuse, Hijacking of components ...
        element typeerror { text }, #allow all certificate , hijacking possible ....
        element object { ClassDetails }
}

DefaultDetails = element detauldetails {
        element type { text },
        element status { text },
        element details { text }
}

GenericInformations = element generic {
        element key { text },
        element value { text },
        element node { GenericInformations }
}
```