KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Computer Science and Engineering

Second cycle, 30 credits

Context-aware security testing of Android applications

Detecting exploitable vulnerabilities through Android models

IVAN BAHEUX

Stockholm, Sweden, 2023

Context-aware security testing of Android applications

Detecting exploitable vulnerabilities through Android models

IVAN BAHEUX

Bachelor's Programme in Information and Communication Technology
Date: January 26, 2023

Supervisors: A. Busy Supervisor, Another Busy Supervisor, Third Busy Supervisor
Examiner: Roberto Guanciale
School of Electrical Engineering and Computer Science
Host company: Företaget AB

Abstract

Write an abstract that is about 250 and 350 words (1/2 A4-page) with the following components:

- What is the topic area? (optional) Introduces the subject area for the project.
- Short problem statement
- Why was this problem worth a Bachelor's/Master's thesis project? (*i.e.*, why is the problem both significant and of a suitable degree of difficulty for a Bachelor's/Master's thesis project? Why has no one else solved it yet?)
- How did you solve the problem? What was your method/insight?
- Results/Conclusions/Consequences/Impact: What are your key results/ conclusions? What will others do based upon your results? What can be done now that you have finished - that could not be done before your thesis project was completed?

Keywords

Android, Model-Based Security Testing, Domain Specific Language, Context-Awareness

Sammanfattning

Enter your Swedish abstract or summary here! Alla avhandlingar vid KTH **måste ha** ett abstrakt på både *engelska* och *svenska*.
Om du skriver din avhandling på svenska ska detta göras först (och placera det som det första abstraktet) - och du bör revidera det vid behov.

If you are writing your thesis in English, you can leave this until the draft version that goes to your opponent for the written opposition. In this way you can provide the English and Swedish abstract/summary information that can be used in the announcement for your oral presentation.
If you are writing your thesis in English, then this section can be a summary targeted at a more general reader. However, if you are writing your thesis in Swedish, then the reverse is true – your abstract should be for your target audience, while an English summary can be written targeted at a more general audience.
This means that the English abstract and Swedish sammnfattning or Swedish abstract and English summary need not be literal translations of each other.

Nyckelord

Cybersäkerhet, Kannelbulle, Mjölk

Résumé

C’est ici que je peux mettre des idioties sans que personne ne s’en rende compte.

Mots-clés

Android, Tests de sécurité basés sur des modèles, Langage Specifique au contexte, Sensibilité au contexte

Acknowledgments

Författarnas tack

It is nice to acknowledge the people that have helped you. It is also necessary to acknowledge any special permissions that you have gotten – for example getting permission from the copyright owner to reproduce a figure. In this case you should acknowledge them and this permission here and in the figure's caption.

Note: If you do **not** have the copyright owner's permission, then you **cannot** use any copyrighted figures/tables/.... Unless stated otherwise all figures/tables/... are generally copyrighted.

I detta kapitel kan du ev nämna något om din bakgrund om det påverkar rapporten på något sätt. Har du t ex inte möjlighet att skriva perfekt svenska för att du är nyanländ till landet kan det vara på sin plats att nämna detta här. OBS, detta får dock inte vara en ursäkt för att lämna in en rapport med undermåligt språk, undermålig grammatik och stavning (t ex får fel som en automatisk stavningskontroll och grammatikkontroll kan upptäcka inte förekomma)

En dualism som måste hanteras i hela rapporten och projektet

I would like to thank xxxx for having yyyy. Or in the case of two authors: We would like to thank xxxx for having yyyy.

Valence, France, January 2023
Ivan BAHEUX

Contents

List of Figures

List of Tables

Listings

Glossary

Android — Android is a mobile operating system based on a modified version of the Linux kernel. xiii, 2, 4, 7–12, 14, 17–21, 23–25, 27, 29, 32–34, 36, 38, 44

Flowdroid — FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. 38

Phishing — Phishing is a type of attack, not specific to Android, that consists of faking a real system to steal data from a user. It is often used with login pages to steal credentials.. 22

Sink — A sink is an open output of a system that acts as the exit of data in a data flow. 21

List of acronyms and abbreviations

ABI	Application Binary Interface. 9
APK	Android Package Kit. 38
ART	Android Runtime. 9, 10
CA	Certificate authority. 18
CDL	Context Definition Language. 33, 36, 37, 41, 43
CDML	Context-Driven Modelling Language. 32, 38, 42, 43
CIA	Confidentiality, Integrity and Availability. 3, 4
CWE	Common Weakness Enumeration. 15, 19, 21, 22
DEX	Dalvik Executable Format. 10
DOS	Deny Of Service. 21
DSL	Domain Specific Language. 27, 29, 31, 33, 36, 43
FSM	Finite State Machine. 31–36, 40, 41
HAL	Hardware Abstraction Layer. 9, 10
HFSM	Hierarchical Finite State Machine. 31, 32, 40
IoT	Internet of Things. 2
IPC	Inter-Process Communication. 7
MBST	Model-Based Security Testing. 3, 4, 27, 40
MitM	Man in the Middle. 18
SQL	Structured Query Language. 20
SSL	Secure Sockets Layer. 18
SUT	System Under Test. 39
TLS	Transport Layer Security. 18, 21
VPat	Vulnerability Pattern. 43

Chapter 1

Introduction

This chapter presents a short introduction to the subject of the master's thesis. We will set the context of the research field and discuss the problems and scientific issues brought up.

1.1 Motivation

Over 5 billion people use a mobile phone today, meaning that 2/3 of the population is prone to an attack via a mobile application [1]. In 2020 over 200 billion applications have been downloaded [2] and an average user has spent 3h39 browsing the internet on his smartphone daily. With this amount of usage, the impact of a single vulnerability can have disastrous effects. From an attacker's perspective mobile devices are a gold mine, since the creation of the first smartphone a couple of decades ago they grew in usefulness and contain nowadays our bank accounts, mail services and social medias. This amount of data gives incentive to be attacked through different methods, and while security has evolved tremendously we still find vulnerabilities each day.

To handle this demand in security researchers and engineers have conceptualized and created different tools and methods to handle different steps of an applications cycle. Scanning, either statically (without executing) or dynamically (with execution), is part of this process that consists of giving more tools to developers, that sometimes is not aware of the security issues he may bring to his code. According to Veracode 12th volume on the state of software security [3], the number of applications scanned has tripled from 2011 to 2021 and the cadence of scan has been multiplied by 20. But, at the same time, approximately 75% of applications still contain a flaw of some sort with almost 25% of applications containing high severity flaws.

An important distinctive parameter of mobile security is context and context-aware applications. Context-awareness is a methodology of taking into account context in the analysis of the program, the context itself is a broad subject that was already defined in 2008 by Hong et al. [4] as "[...] any information that can be used to characterize the situation of an entity". While the definition is vague and there is no set consensus among researchers, the concept is easy to grasp. In smartphone applications, this could be the location, orientation, time or even smartphone model just to cite a few.

The importance of context-awareness comes from the rapid evolution of Internet of Things (IoT) and the need for tools to be intuitive and easy to use. In this context, our smartphones have evolved to change automatically depending on the context. Whether it is simply accessing the user's location to give him feedback on the weather or even automatically setting energy saving mode under a certain battery percentage, a lot of data affect our applications [5]. The ubiquity of Android smartphones brings the need to test applications in different contexts to ensure security. Here is where our work comes in, we will aim to bring a new method for developers to prevent vulnerabilities on their applications through context-aware security testing before release of their application. While a lot of work already exists in terms of application testing [6], security testing, i.e. testing for vulnerabilities, has only very little work in terms of context-awareness (also called adaptive). Simply enough, this thesis aims to bring more security to users by testing applications whose context may change during their execution.

Our work will be primarily oriented at Android applications as, according to statista.com [7], Android holds more than 70% of the market share and we want to be able to apply our project to the most systems we possibly can.

This situation is the foundations for this master's thesis.

1.2 Problem

Bringing new ways for developers to give users trust in their systems is a priority. While protecting the system in which the code is executed is possible, developers are not always in control of it. This is due to the widespread of today's software, the executable is run in different OSes, in different devices with different capabilities and limitations.

In terms of development, security is still fairly uncommon and not considered a priority by developers[8]. This is due to a multitude of factors: Lack of money, short development deadlines or simply lack of knowledge. To solve that companies have started including security as an extra step in the

development cycle using large scale tools that can assess the security of a project. Due to the amount of new vulnerabilities each year and even simply of new features and technologies developed it is important that research be oriented in the conception of new techniques following recent vulnerabilities and technologies, or even try to foresee the future evolution of hackers.

One method to detect vulnerabilities is using a model of the general code and running a set of algorithms that infer if a vulnerability may be/is present on the source code. Then it generates a penetration test to assess if the vulnerability truly exists. This method is called Model-Based Security Testing (MBST) and is useful in giving a more abstract representation of the code to run faster tests. Through the rise of smartphones, ubiquity is a real problem that is hard in terms of time analysis. Indeed, checking every possible combination of context is not realistic as such models and their level of abstraction can be a way to solve this new problem.

For this exact reason and, in order to simplify the development process, we aim to bring a larger tool set for software developers focusing on automatic vulnerability detection through context-aware MBST.

1.2.1 Adversary

In the context of security in computer science we choose to protect against specific types of adversaries, each having specific goals and means of attack.

During our project we will consider that the attacker wants to break one of the components of the Confidentiality, Integrity and Availability (CIA) triad of our vulnerable application. This can be :

- Confidentiality: Data theft.
- Integrity: Data injection or Tampering.
- Availability: Deny of Service or logic bombs.

We will assume that the attackers victim is always an application and ultimately the user. We will also assume that any cryptography protocol is perfect if used correctly, an exception would be if the protocol is known for not being safe.

1.3 Goals

During this master's thesis we will aim at answering the following questions:

RQ1. What Android permission system vulnerabilities could be related to the context of an application and how to model them in order to test them?

RQ2. What security testing process could be applied to detect vulnerabilities related to the context (threat models, attack vectors, security tests, analysis of results)?

Through an earlier master's thesis work by Abdallah Adwan [9] concerning context-aware testing of Android applications, Abdallah developed a model for Android apps and their context. The laboratory is interested in expanding the project to allow for model based security testing, in other words expanding the aforementioned model to detect possible vulnerabilities within a application code.

1.4 Research methodology

The methodology will be an implementation of MBST. If we follow the definition by Schieferdecker et al. [10], MBST is a broad subject that is important in that it aims to test security requirements (CIA) in an efficient manner. In our case, we will model the security mechanism of an application to generate security tests for a specific vulnerability and then, try to generalize to more vulnerabilities.

We will follow the following steps:

- Find a specific vulnerability that is defined by the context of the application.
- Analyse the extent of the vulnerability. (How many devices can be affected, is it still relevant nowadays..)
- Expand on the testing model previously created by the lab with the information needed to detect the selected vulnerability.
- Model the vulnerability.
- Link both models via a penetration test generator on a vulnerable application.
- Then, we will measure the performance of our process via a set of applications. This set will either be found free online or be generated during the master's thesis.

- Lastly, we will expand and generalise our process to related vulnerabilities.

1.5 Delimitation

We will not be able to model every existing vulnerability during the degree project, but only give the ability to extend the model to further research. As such, we will choose a set of vulnerabilities during the pre-study on which we will focus the thesis work.

1.5.1 Ethics and Sustainability

While the project does not directly address questions of ethics or sustainability it is, by design, at least partly a software process consuming energy. We do not need to precisely measure the power usage but it will be developed with this issue in mind in order to limit at our scale the impacts on the environment.

In order to explain the studied vulnerabilities, we may need to explain in what way they make the application vulnerable. In order not to share exploits, we will try to have an artificial scenario and clearly separated from real world applications.

1.6 Structure of the thesis

Chapter 2

Background

2.1 Security models and security policies

In the following sections we will be reviewing Android application vulnerabilities and for that we will consider security from the point of view of information flow. As introduced in 1982 by Goguen and Meseguer [11], the policy model of non-interference can be described as simply as no matter what public inputs are given to a system, the public outputs of said system will not be influenced by the private inputs of the system. More formally, the inputs of an application will not allow it's output to contain private data if not explicitly allowed to.

In the field of Android applications, we find different types of data depending on their criticality for the owner. Some data can be viewed as public or non-sensitive like data not permission-protected, for example the API version. On the other hand, some data are considered critical (or sensitive) when it affects the user's privacy like accessing the location of the device, although more application-specific sensitive data exists like login credentials for banking or social media applications.

In our case, the context of the device is important and not to be blindly trusted by the application. Our adversary may be a malicious user or a malicious application running in the same device as such inputs, and Inter-Process Communication (IPC) can be vectors of attack and must be analyzed.

Information flow security must be true in any execution of an application with any type of input and additionally, in our case, in any type of execution context as will be defined by section 2.3. The type of policy model brought by information flow security is defined by Clarkson et al. [12] as a hyperproperty. In their paper, they model non-interference as a hyperproperty which, in our

thesis, we will attempt to reveal violation of. To disprove a property, we will try to generate a proof of broken information flow security with a specifically crafted execution of an application.

Lastly, because our model only takes into account input and output from an application we knowingly exclude side-channels as external communication outputs. This is a simplification that allows for a more restricted model but it should be noted that this means that any conclusion drawn from our vulnerability detection cannot be complete. The work by Alqazzaz et al. [13] provides insight on the confidentiality problems that exist in the Android environment.

2.2 Android

Android is a mobile operating system that owns more than 70% of the marketshare at the time of writing [7]. Originally released in September 2008 with Android 1.0, the operating system owned mostly by Google has released, on the 15th of august 2022, Android 13 the current most recent Android version.

Android's core components are completely free and open-source. Based on a modified version of the Linux kernel and other open-source software, it is used by mobile device vendors around the world as a base to proprietary versions [14].

The development of Android is mostly done by Google although being open-source they are open to outside contributions. For example the Google vulnerability reward program[15] rewards users for vulnerability reports with digital trophies and paid rewards.

2.2.1 Android architecture

This section is based on the Wikipedia page of Android [14] and the official documentation of Android [16]

As explained above, Android is a mobile operating system build on the linux kernel. The OS is built of several layers that are called software stacks as can be seen on 2.1.

2.2.1.1 Hardware level

The first stack is the kernel, this stack handles most of the hardware operations, low-level memory management and threading. Most of this stack is written is

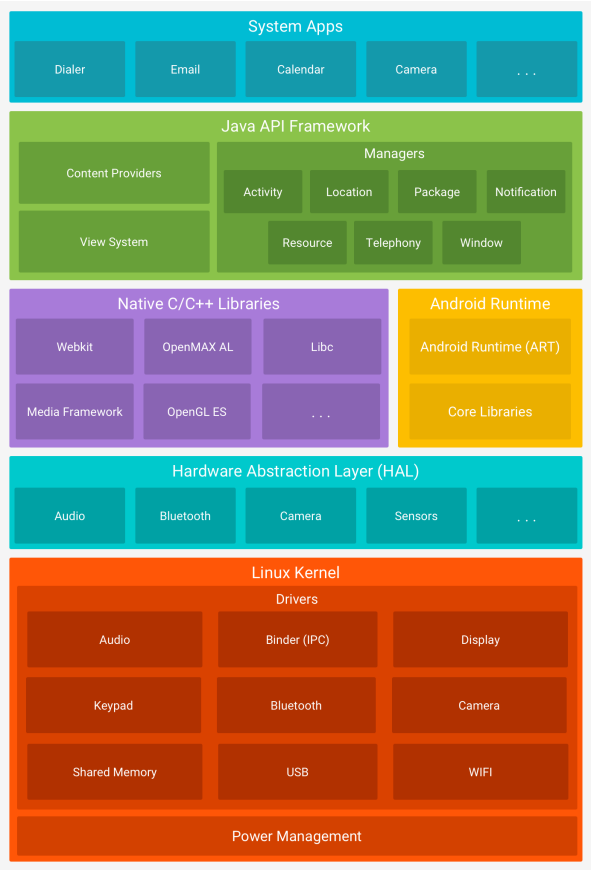

Figure 2.1: Android general architecure. Source:developer.android.com/

in C language and assembly code as it is closer to the physical components of the device.

2.2.1.2 Hardware Abstraction Layer

The next stack is the Hardware Abstraction Layer (HAL) that acts as a set of Application Binary Interface (ABI) to enable communication between higher-level java layers and lower-level (usually) C-level code. The HAL consists of multiple library modules used as an interface of hardware components. Android API that need sensor values call this stack.

2.2.1.3 Android Runtime

The Android Runtime (ART) stack allows for sandboxing of android applications.

As a security measure but also per design choice, Android application are executed in a virtual machine through a format called Dalvik Executable Format (DEX) a custom bytecode that is not java bytecode but a custom made language. This bytecode was executed on the Dalvik Virtual Machine until Android 5. Two effects come from this sandboxing :

- Optimisation : ART has optimised DEX that is better than java bytecode
- Complete separation per application : When an application is executed it runs on a completely separate environment from other applications. It basically runs with a different user each application.

2.2.1.4 Native C/C++ Libraries

This part is separated from the HAL in that it does not act as interface with hardware, but sometimes optimised native libraries are needed. For example the ART and HAL may need some functions from the Bionic libC [17] (a Android specific GNU C library designed for less processor power). Other graphical libraries are also found here like the OpenGL implementation available to applications through the a Java API.

2.2.1.5 Java API Framework

This stack is the main one used by application developers, it contains most of the libraries/API that allows usage of Android features as well as the services (called managers) that provide different features to applications.

This stack contains the following[16] :

- View system : That controls UI's, views, buttons etc..
- Resource manager : That gives access to resources like images and XML files
- Notification manager : That allows applications to send notification to the user outside of the scope of the app's UI
- Activity Manager : That controls the applications execution and lifecycle. For example an application can call another app's activity to access some feature he does not provide like taking a photo for example. It also controls what is in foreground and background etc...
- Content providers : That allows to access information from other applications or to share the app's information to others.

2.2.1.6 Application / System applications

Lastly, this stack is comprised of most of the user experience : Applications.

In the base Android, this stack has only the default system applications but nothing separates application installed by default and others. These applications use the framework API to communicate with the other applications, with sensors or even with the network.

2.2.2 Android permission system

This section is based on the official documentation of Android [18]

A lot of security features are implemented in Android, at the application level, arguably the most important would be the permission system. This permission system defines what the application may do and use at the system level, whether it is accessing data on the device like reading the state of the wifi or even being able to use the camera.

To do this Android separates the permissions into 3 categories normal, signature, and dangerous. Since Android 6 (API version 23) [19] these permissions are asked to the user either when he installs the application or during runtime when before it was only on install.

2.2.2.1 Install time permissions

The permissions that are asked from the user before he installs the application fall under the categories of normal or signature.

Normal permissions are general permissions that allow the application to access information that are not critical to the user privacy like allowing access to Bluetooth, changing the wifi state, and accessing internet. These permissions are automatically granted to an installed application and the application store takes care of informing the user that an application may use these permissions.

Signature permissions are more specific permissions custom made by application developers to allow applications made by the same developer to access a specific feature. For example a developer may create an application *A* that has a feature F_a and that asks for a permission $P(F_a)$ if another application wants access to the feature. If the developer wants he can restrict this permission $P(F_a)$ so that it is signed by him, meaning that only his applications or applications signed with the same certificate may have access to this permission.

Specifically, a certificate may be shared between groups and not only be restricted to a single developer.

2.2.2.2 Runtime permissions

The permissions that are asked from the user before he installs the application fall under the category of dangerous. They are requested specifically when using the app's feature that needs the permission and can be refused by the user. The permission request is done by the developer and can technically be done at any point during the app's execution, it is good practice to wait until the last moment to ask that permission. Since Android 11 (API version 30) users may chose between accepting a runtime permission for ever or only accepting once the permission. This means that when the application is not used anymore, the permission will be revoked. This is called *one-time permission*.

Dangerous permissions are permissions that give access to critical information or features. This means private data or sensors like the camera and the location. These permissions have a great impact on the users privacy and should be kept to a minimum by developers.

2.2.2.3 Criticisms of the permission system

The permission system has been an ongoing debate between researchers for a long time and has evolved a lot to try to find solutions. The base problem is that it is difficult to have a system that combines :

- Great security measures
- Simple/straightforward developer implementation of security
- Clear user understanding of the risks and fast UI

For this reasons design choices have been made, that are criticised among those :

- Permissions too coarse [20]: Meaning that permissions are too broad in regard to what is needed by applications. Which meant that the user could not know precisely what was going to be used.
- Too static in nature: While there are now runtime permissions, it is clear that some contexts may need different permissions. For example an application may ask for Bluetooth during install time but never actually use it, depending on how the user uses the app [20].

- Too complicated for developers: Leading to overprivileged applications as developers don't want to select every permission specifically [21].

2.3 Context and Context-Awareness

Context and context-awareness has evolved a lot in the last decade. This field of study has gained importance with the appearance of mobile applications for their ubiquity in comparison to standard wired equipment. Muccini et al. [22] separated mobile applications into two different categories:

- *App4Mobile* are apps that are simply translated software for mobile applications, and
- *MobileApps* are apps that where designed with context and adaptiveness in their core.

This difference showed a shift in paradigm between software development and mobile app development.

A base definition appears in the widely cited paper on context computing by Abowd et al. [23] as *any information that can be used to characterize the situation of an entity*.

In terms of definition, nowadays context is often defined in two different ways :

- A list of elements considered as context
- A list of subcategories of context

The first definition is useful for simplification but is limited when building models. As an example, in his document, Brown et al. [24] considers that the context is what the computer sees and understands of the user's environment. This includes the location, the time, what is near the computer et cetera.

For subcategories we can find examples that separate static and dynamic context, this is the case in the document by Gomez et al. [25] in which they separate static properties and dynamic properties. This definition is useful in defining when each property has to be surveyed. The static properties being detected at boot and the dynamic surveyed during the execution.

With the evolution of mobile applications the need for a third more abstract or high level categories has raised, often inferred from other categories. As explained by Almeida et al. [26] low-level context is what we referred as static and dynamic context and high-level context are situations that have an impact

on the system and exists in order to simplify analysis. As an example they explain the situation "high-speed vehicle" which aggregates :

- GPS coordinates to detect the highway the vehicle is in
- Accelerometer/gyroscope values to detect the current speed of the vehicle
- Internet to check maximum speed accepted in this vehicle

This allows for a simple check "is the context overspeed?" to trigger events for an application.

For the continuation of this paper we will be using the terms :

- Static Context : Context that does not evolve with time during the applications execution
- Dynamic Context : Context that may evolve with time during the applications execution
- Derived Context (or situation) : High level contexts acting as an agglomeration of contexts or specific context values.

2.4 Android vulnerability

2.4.1 Survey methodology

The research for Android vulnerability papers has been made with the usage of common research papers engines like Google Scholar with snowballing of references. In order to stay relevant to future research we have filtered papers that were older than 2015. It is also important to note that our goal is to list vulnerabilities linked to applications that can be detected and solved through modification of the application code.

For the research a total of 12 papers have been selected to base our survey on, we tried to find big research papers that tried to survey Android vulnerabilities. We also selected a few papers that explained some very specific or sometimes overlooked (either because too specific or too recent) vulnerabilities that remained relevant to our topic.

We can see on figure 2.2 the distribution of papers selected for section 2.4. Each document will be cited for the specific vulnerability they relate to.

A visual representation android vulnerabilities relating to application security can be seen on figure 2.3

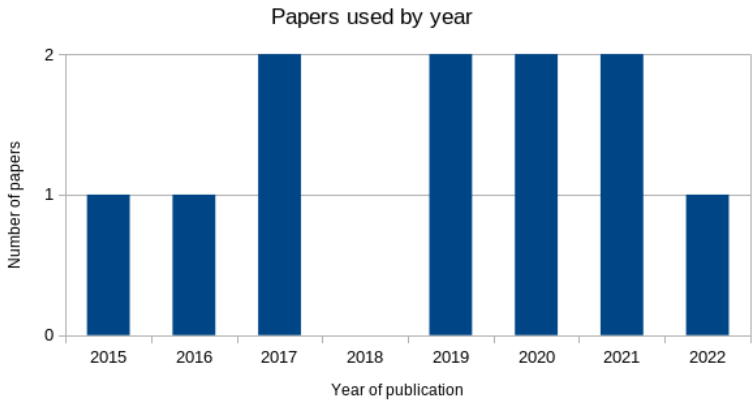

Figure 2.2: Graph of research papers selected by year

Lastly we linked each vulnerability to an existing mobile Common Weakness Enumeration (CWE) [27]. The goal of a CWE is to serve as a common baseline for any vulnerability in order to be able to classify specific vulnerabilities faster. This will allow us to measure the effectiveness of our tool by class to then lead further research more carefully into what is missing with our work.

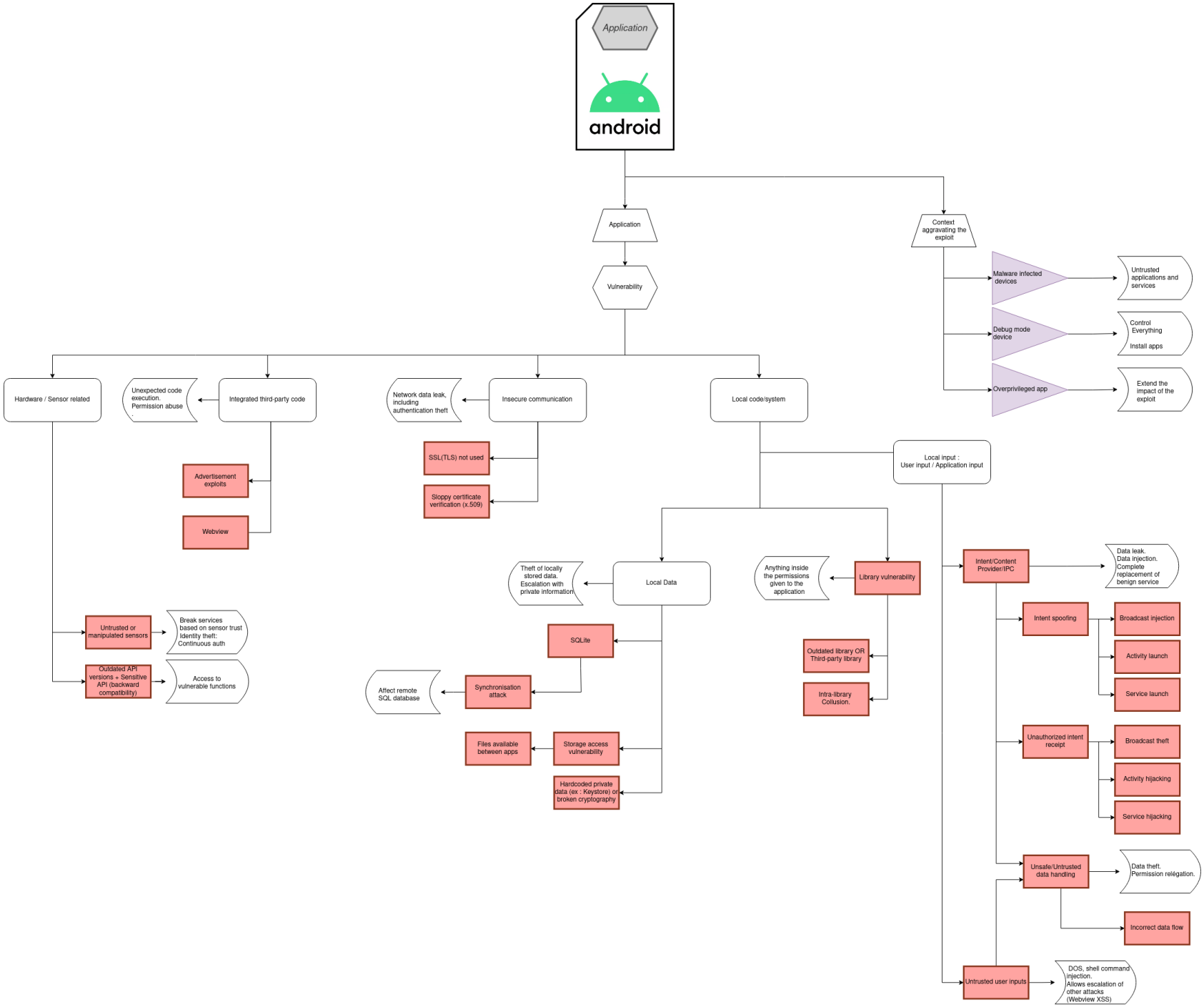

Figure 2.3: Interconnection of android application vulnerabilities

2.4.2 Integrated third-party code

This type of vulnerability relates to external code inclusion that directly accepted by the developer.

These vulnerabilities can be classified as :

- CWE-200: Exposure of Sensitive information to an unauthorized actor
- CWE-359: Exposure of Private Personal Information to an Unauthorized Actor

In the survey two possibilities emerge.

2.4.2.1 Advertisement code exploits

The Android revenue system for applications has 3 main ways of working :

- Pay to use : The user needs to pay to download and use the application.
- Pay to upgrade : The user can download the application for free but can pay to upgrade or get bonuses in the app.
- Advertisement inclusion : The user sees adds in the application, this generates revenue for the developer.

The third method uses the inclusion of ad libraries provided by ad providers. Ad provider can pay more than others and ad providers can check more or less the ad they are including in an application.

This means that it can lead to insecure code being run on the device through an application, whether it's malware, adware or ad that redirects to malicious websites [Vuln1]. This case is studied in depth by Jonathan Crussel et al. [Vuln2].

2.4.2.2 Webview

This vulnerability vector has a lot of litterature on it. Webview is a component that allows developers to open webpages inside their app. As explained by Faysal et al. [Vuln1], while webview is not vulnerable by themselves, it allows for execution of javascript [Vuln3] and more generally arbitrary third-party content. While important vulnerabilities have been removed after API level 17 [Vuln4]. Recent work shows that a lot of vulnerabilities still exist [Vuln5] that allow for data theft without the user's/developer's consent.

2.4.3 Insecure communication

Mobile applications are very deeply tied to network usage for multiple design reasons. The misuse of communication channels due to misunderstanding or mistakes can lead to very serious confidentiality leaks. Two vulnerabilities groups have been identified relating to network communications.

2.4.3.1 SSL(TLS) not used

Secure Sockets Layer (SSL) (Replaced by Transport Layer Security (TLS) since 1999, and completely replaced in 2014) is a security protocol for network communications ensuring server authentication, confidentiality and integrity of exchanged messages [28]. When application developers make use of network channels without using proper cryptography (like the simple usage of HTTPS) they expose their code to Man in the Middle (MitM) attacks [Vuln4] [Vuln3] [Vuln6], basically interception of the data sent through the network. While HTTPS is activated by default since API level 23 [29] a misuse by the developer can lead to data theft.

These SSL-related vulnerabilities can be classified as :

- CWE-200: Exposure of Sensitive information to an unauthorized actor
- CWE-359: Cleartext Transmission of Sensitive Information

2.4.3.2 Sloppy certification usage

As seen by Jun Gao et al. [Vuln4], even if SSL usage is good a bad verification of the certificate leads to MitM attacks. The idea is that in order to work, Android applications are required to have a valid x.509 certificate for multiple reasons [Vuln3] (application signing and TLS handshakes) but if your code accepts every possible certificate to communicate then the TLS means nothing. Simply enough, if an application developers gives trust to anyone then the certificate has no reason to exist, thus configuring the server with a Certificate authority (CA) is primordial to only accept trusted websites. These certificate-related vulnerabilities can be classified as :

- CWE-295: Improper Certificate Validation
- CWE-297: Improper Validation of Certificate with Host Mismatch
- CWE-940: Improper Verification of Source of a Communication channel

2.4.4 Local code/system

The following subsection lists vulnerabilities related to the local environment of the application.

2.4.4.1 Library vulnerabilities

Libraries are fully part of an application they are used and trusted without thinking too much about it. According to Sumaya Almanee et al. [Vuln7], application developers take, on average, 3 times more time to patch native libraries than the time it takes for the library to be updated. This means that even if a library is updated, the developers don't make directly the choice to release a new version of their code using this new library. This happens because they are afraid their code will break or because they think of functionality before utility.

Another aspect of libraries are libraries that come from third-party providers, this is the case for example with ad providers but also with any code libraries that a user may find on the internet. This leads the application to be prone to misuse of permissions provided to these libraries. According to Parnika Bhat et al. [Vuln8] third-party libraries hold more than 60% of Android application code. This means that the quantity of vulnerabilities are very large as no security verification is necessary on any third-party library. Vulnerabilities related to libraries are really wide as they open to any kind of code exploit : Logic/time bombs, data leak, power consumption.

- CWE-511: Logic/Time Bomb
- CWE-200: Exposure of Sensitive Information to an Unauthorized Actor
- CWE-359: Exposure of Private Personal Information to an Unauthorized Actor

Another type of library related vulnerability is intra-library collusion. The idea of this vulnerability is that, due to how libraries are designed, libraries have the union of every permission that applications using the libraries have. This means that they end up being over privileged and can be attacked this way. We can link this to the following CWE.

- CWE-250: Execution with Unnecessary Privileges

2.4.4.2 Local data

Vulnerabilities relating to how the local data is handled is a large part of vulnerabilities. They mainly relate to vulnerabilities appearing due to low encryption settings when storing data or trusting that no one will tamper available data on the device [Vuln4]. The first one we can find is Structured Query Language (SQL) related. SQL is used a lot to store data that can be quickly stored, classified and modified, this is often high impact for data thefts [Vuln3]. One of the usages of SQLite is to act as a copy of the SQL stored in a server, this allows to not have to exchange a message between server and client every time. The problem is that if the local database is stored without proper encryption then it can be tampered leading to a synchronization attack. A modification of the local database can have repercussions on the server database meaning a possible SQL injection. This type of attack is presented in more detail by V. Jain et al. [Vuln9].

From a more general perspective, unsafe data storing in Android is a common concern. As explained in the document by Shezan et al. [Vuln1], even with a sandboxed application isolation, we can use permissions to be able to share our files between applications. This opens up possible security problems if developers misuse these permissions. In the same way rooted devices make every file available to any applications opening up even more risks that the developer should be aware of.

Lastly, we can find local vulnerabilities in files with encryption when the key or encryption mechanism is left open by the developers, this can be the case with hardcoded encryption keys or a misunderstanding between encryption and encoding that has happened. Even leaving the hashed data in the local files could be flagged as a security failure as they can possibly be stolen. We can relate these vulnerabilities to :

- CWE-200: Exposure of Sensitive Information to an Unauthorized Actor
- CWE-312: Cleartext Storage of Sensitive Information
- CWE-359: Exposure of Private Personal Information to an Unauthorized Actor
- CWE-921: Storage of Sensitive Data in a Mechanism without Access Control

2.4.5 Application input

This section relates to the input/output of the application. Should it be from the user or from other entities, communication should be wary of inputs received and of the privacy of data sent.

2.4.5.1 Untrusted user input

Untrusted user input can be separated into two parts, first inputs that are not sanitized leading to further attacks, Deny Of Service (DOS), code execution and more, depending on the rest of the code and of the permissions given to said vulnerable application.

Another well studied vulnerability is insecure data flow[Vuln4]. The principle is that an application can have private/sensitive data exiting the application to a public Sink. This can happen explicitly, for example when data is sent to the server without TLS or implicitly, like when an event happens only when the private data is in a certain state allowing us to infer the private data.

The CWE relating to the above vulnerabilities are :

- CWE-200: Exposure of Sensitive Information to an Unauthorized Actor
- CWE-359: Exposure of Private Personal Information to an Unauthorized Actor

2.4.5.2 Inter process communication

This part relates to vulnerabilities linked to inter process communication (Content provider, explicit/implicit intents, broadcasts...). In a device multiple application are able to communicate with each other, in Android the main way is by using content providers/receivers and intents. But if the developer is not careful, it is easy to either leak data or allow for third-party usage of the benign app's permissions.

2.4.5.2.1 Intent spoofing This subtype of vulnerability concerns benign applications that may receive data from other applications. According to Faysal .H et al. [Vuln1], this type of vulnerability is one of the most common in Android. In their work on Vandroid [Vuln10], Atefeh N. et al. classified intent spoofing into three categories :

- Broadcast injection: When a receiving application trusts blindly any type of broadcast intent sent by other applications. This can lead to using the data received as if it was trusted data.
- Activity launch: When an activity is launched by an intent from untrusted applications. With this we can control the applications data or even do Phishing-type exploits as an attacker.
- Service launch: A service can be understood as a background activity. Therefore, like activity launch attacks, we are able to control the data or even start new tasks. This is even more powerful as the former as it can stay up even if the user changes the foreground application.

We can relate these vulnerabilities to the following CWE.

- CWE-925: Improper Verification of Intent by Broadcast Receiver
- CWE-926: Improper Export of Android Application Components

2.4.5.2.2 Unauthorized intent receipt This subtype of vulnerability concerns benign applications that are too noisy when sending data. The most common mistake when sending data is to not be sufficiently specific about who is able to receive data. When sending data through a broadcast, virtually any application is allowed to register as a receiver to read the data. This type of vulnerability is called broadcast theft [Vuln10]. Another more complex type of intent reading concerns hijacking attacks. The general idea of a hijacking attack is to be able to either:

- Replace an app's service or activity
- Redirect an app's service or activity to a malicious app

A recent work by Pu SUN et al. [Vuln11] has shown the effects of hijacking attacks. The idea is to detect/infer when an application is starting an activity/service/component and to find a way to hijack the flow of execution to inject their own code or redirect to their own application.

Venomattack makes a screenshot of the applications UI (usually a login screen) then transforms it into real code. Through a hotpatch technique they update the malicious application to contain the login page of the victim application then they hijack and redirect to their own application. The hijacking attacks are both system and application vulnerabilities. We can relate these vulnerabilities to the following CWE.

- CWE-927: Use of Implicit Intent for Sensitive Communication

2.4.6 Hardware / Sensor related vulnerabilities

With the common usage of a lot of sensors, some vulnerabilities arise.

2.4.6.1 Untrusted sensors

In 2016, a paper by Manar Mohamed et al. [30] demonstrated the ability to inject sensor values with the usage of commonly installed apps (at the time) with only the need to have an internet API and the right device context. While this specific vulnerability is rarer, injection attacks still exist and are studied. In a recent paper by Gonzalez-Manzano Lorena et al. [Vuln12] we see that injection attack can potentially lead to identity theft through the exploitation of continuous authentication. Nowadays mobile applications like banking apps use continuous authentication to keep the user connected for a short time before disconnecting them. This is done either by reading time, reading the touchscreen usage or even by inferring the user's presence via the light sensor. Being able to inject might help an attacker to perform further attacks on the now authenticated application.

2.4.6.2 Sensor API and backward compatible API

With the complexity and number of devices running Android, it is nearly impossible to have every device using the same up-to-date version. This means that sometimes developers have to create backwards compatible code. Thus when working with different versions of Android a developer may use a deprecated package with real, very well known vulnerabilities [Vuln1]. According to a study by Jun Gao et al. [Vuln4], applications tend to stay on one API level and not necessarily upgrade.

2.4.7 Settings aggravating an application vulnerability

This section will not define more vulnerabilities but will contribute to determine *bad* contexts. We will list device settings that very importantly aggravate the possible vulnerabilities or even open more vulnerabilities on an application.

2.4.7.1 Malware infected devices

The most straightforward setting would be when an attacker is present in the device when the application is installed or executed. Depending on the malware some usually trusted mechanisms may be broken or replaced (For example, a fake camera application may be present).

2.4.7.2 Debug mode

Then, we may also have a debug mode activated. This mode is part of the developers options that allow for testing of Android application, sadly and as seen in the paper by Manar Mohamed et al. [30] it can be misused to install overprivileged services or just to simply install root applications.

2.4.7.3 Overprivileged applications

Lastly, giving too much privileges to an application is an extremely dangerous action. This is an ongoing research topic but it is a fact that developers tend to give too many privileges (intentionally or not). This leads to different situations, attackers who get to control the app's whereabouts may execute unwanted actions on the benign app, libraries can do unwanted actions if not trusted among other things.

Chapter 3

Method or Methods

The purpose of this chapter is to provide an overview of the research method used in this thesis. First, in section 3.1 will be reviewed the factors of vulnerability via the interpretation of the vulnerability survey of section 2.4. Then, section 3.2 gives an explanation of the methodology followed to design the architecture of our solution. Lastly we will focus on the techniques used to evaluate the tool in section 3.3.

3.1 Vulnerability focus

In section 2.4 we have described the state of the art of Android application vulnerabilities. The table 3.1 transcribes the features that need to be used to detect each vulnerability. Which will be used as a guide for the design of our contribution.

The features are :

The application code

- Yes : The application code is enough to detect the vulnerability.
- No : The application code is not enough to detect and an additional information is needed (context).
- Not applicable : The vulnerability cannot be detected through the application code and other means must be used.

Context

- Dynamic : The dynamic context can (or must if application code is not enough) be used to detect the vulnerability. This can refer to interactions with other applications or sensor values (wifi, location..)
- Static : The static context can (or must if application code is not enough) be used to detect the vulnerability. This can refer to the configuration of the phone and application (API version, network configuration...)

Table 3.1: Vulnerabilities and their detectable features

Vulnerability	**Is application code enough.**	**Context**	**Explanation**
Untrusted or Manipulated Sensors	Not applicable	Dynamic	Detected with unusual context modifications. May be detected with unusual sensor patterns
Outdated API version, Sensitive API	No	Static	Detected through application calls to vulnerable functions or old API configurations.
SQLite	Yes		Detected through code review.
Storage Access Vulnerability	Not applicable	Static	Detected by checking configuration of readable content.
Hardcoded private data or broken cryptography	Yes	Static	Detected through bad cryptographic function written, hardcoded values or bad libraries.
Outdated library or third-party library	Yes		Detected through bad library usage.
Intra library collusion	Not applicable	Dynamic	Detected through contextual checking of other applications using the same library. Also checking library code.
Intent spoofing	Yes	Static	Detected through bad configurations of the activities/services or code that gives too much rights to incoming intents.
Unauthorized intent receipt	No	Dynamic	Detected through bad coding practices when writing broadcasts. Detected through strange activity overlap between applications.
Untrusted user inputs	Yes		Detected through input sanitizing.
Incorrect data flow	No	Dynamic	Detected by checking application code or private data leakage on public channels

This table highlights the value of handling context when designing a tool for vulnerability detection, as it is mandatory for at least 6 vulnerabilities to not limit to source code our research but also as it helps most vulnerabilities possibly allowing us to detect wider versions of the cited vulnerabilities.

3.2 Design methodology

In terms of design we have a few constraints :

- Base our work on A. Abdallah's internship work.
- Cover most vulnerability types with a focus on vulnerabilities linked with context.

- Aim at trying to avoid the limitation of the master's thesis duration (5 months)

3.2.1 Base work

As introduces in section 1.3, we will be expanding on the work of A. Abdallah and build upon his theory with our contribution.

During his master's thesis [9], Abdallah designed a methodology to generate tests that takes into account the context of an application. The general idea he brought was to define two Domain Specific Language (DSL) and apply a MBST process to generate tests. The first DSL defines the context that a generic application can have and the values that each context may have and the second DSL defines the behaviour a specific application has and which context may affect it.

By combining both DSL we are able to have the information on what an application does, and which contexts change the behaviour of the application.

Abdallah goes on by defining algorithms for different coverage criterias that allows his architecture to generate tests.

During our project we will be reusing his theory, expanding on the two DSL so that we have enough information to detect vulnerabilities. It is important to note that while the two DSL he created exist the algorithms where never implemented so our work will also contain this part. His project also asks the developer to write the model of his application manually and the tests generated also need to be converted into java (or kotlin) code to be used as real tests.

3.2.2 Design goals

The goals we aimed during the design of the architecture of our project are as follows :

Make the most out of the models Building upon models is really useful as it allows to abstract specific parts of the code, this is often used to detect vulnerabilities over multiple types of language or vulnerabilities tied to behaviour. Our project being specifically on Android we will want to have the possibility to detect vulnerabilities up to function specific but still keeping a high level of abstraction.

Time is a constraint As explained in section 3.2 the project's time is limited to 5 month and while it is a limitation, it also gives us a reason to make use of

open-source and try to find a way to provide something that can be expanded easily.

A tool for developers or testers Our contribution aims to be a helping guide to developers that want to check their code. It would also be useful for a tester to add a layer of vulnerability checking to the application process

Separate the work Our contribution will aim at separating the knowledge needed to use the tool. If developers want to use the tool then they don't need to understand the vulnerabilities they check. This way we will need to generate reports to guide the developers in removing the vulnerability, either by automatically removing the vulnerability or by giving the information needed to change the code.

3.3 Data collection and testing methodology

In order to measure the efficiency of our work we will need to build a testbench as none is currently available at the laboratory.

During the research process the ghera [31] repository has been found. This repository gives a examples of vulnerabilities, how to exploit them and the solution to the vulnerability. We will base some of our tests on this repository and build the rest in the same manner as they do as the limitations of this website is that it has not been updated since 2019.

A problem that arises is that the ghera project gives "fake" examples of vulnerabilities. To give an example, the vulnerability *WeakPermission-UnauthorizedAccess-Lean* features an application that did not limit who can access and launch it's activities. This vulnerability shows that we can call a specific action to query the "database" it contains from outside of the application. To show this they simply print the string *"query MyContentProvider for sensitive information"* which is in my design not a vulnerability as it does not print a private value nor does it allow for any misuse of the application. To solve this we will partly modify the code so that it actually prints a private value if needed.

Chapter 4

Software contribution

In this section we will be reviewing the theory and technology built during the master's thesis. The first section, section 4.1 will introduce the general architecture of the process we created. Then, in section 4.2, will be reviewed the first part of the architecture on the subject of behavioral test generation. Lastly we will present in section 4.3 the exploit generator, its design and philosophy, and its architecture.

4.1 Global Architecture

During the master's thesis we designed and created a two part process called ConTest and VPatChecker. In figure 4.1 we can see the global design of this process.

In terms of usage, the developer of a specific Android application creates a high level model of his application using a defined DSL.

The resulting model is enriched with the information on dataflow using FlowDroid [32]. This tool allows to build a callgraph of an application using the specified functions as input and the specified functions as output.

This allows to:

- See the flows from private data to public output
- See the effects on public inputs to specific functions (in the case of vulnerable functions)

The model is then used by ConTest, the test generator built by Adwan Abdallah and myself, to generate a set of tests by combining it with a generic definition of Android context. This combination allows to generate a set of tests

taking into account the multiplicity of dynamic context that may or may not change during execution of the application. The generated tests are behaviour test exported as ".xml" files, these test provide coverage information on the application.

Lastly the generated tests are used by VPatChecker comparing each test with every vulnerability pattern written in order to see if a specific pattern can be applied to one of the tests. If a positive comparison is found, VPatChecker makes the necessary modifications to the tests in order to transform it into an exploit (like changing the input values for example). The output is exported as ".xml" files, these provide information on vulnerabilities present in the code.

The entire process allows to prevent vulnerabilities during the development process by telling the developer how we can exploit his specific code and allowing him to fix his code.

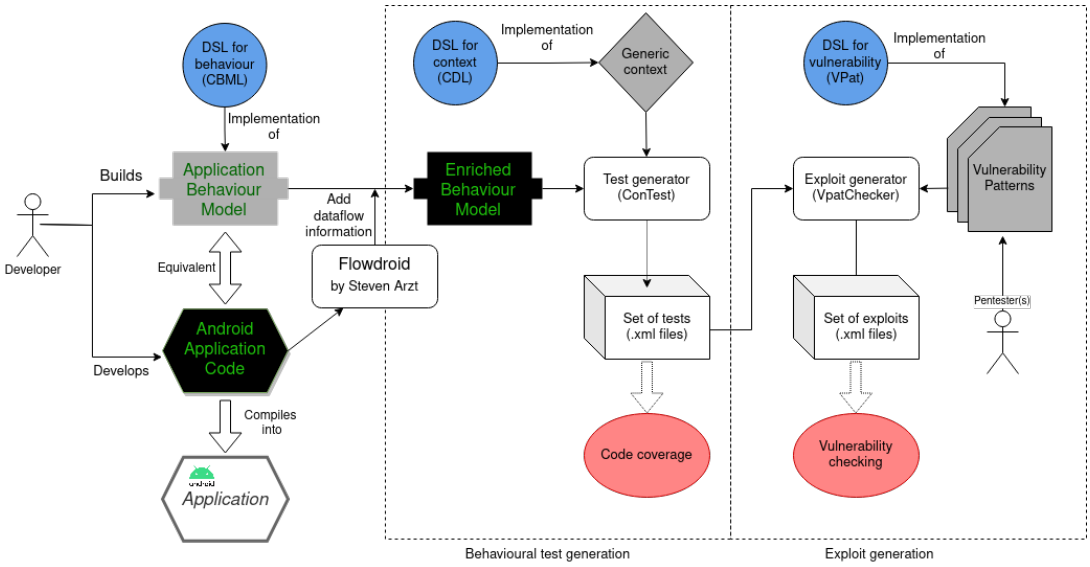

Figure 4.1: Global architecture design

4.2 Test Generation

As explained in section 3.2.1, we based our work on a work by A. Abdallah called "Context-dependent Model-based Testing of Mobile Apps"[9] (or ConTest). In order to be clear about the participation of A. Abdallah in this base, each part will detail the contributions.

ConTest was designed as a tool for code coverage that in its design allows for a multiplicity of coverage criteria. Because generating test is a complex

problem we limit the number of tests we generate with test objectives, this can be for example : Checking that we executed each line of code at least once or checking that we executed each function once.

4.2.1 ConTest - Architecture

The ConTest process design is defined around a model of an application. This model is an implementation of a DSL designed using Xtext[33], a part of the Eclipse Modeling Framework. Xtext gives a tool to describe a language via metamodels or directly writing the code ourselves. The power of xtext is that it generates a parser and multiple other tools from the definition of the language we gave. In ConTest it allows us to create languages that are easy to write for someone but also easy to parse by a program and thus allows us to exploit the model with algorithms (for code coverage for example).

DSLs is Xtext which is part of the Eclipse Modeling Framework . Xtext can be used to create DSLs either by supplying a metamodel or by defining a grammar for the language. Furthermore, Xtext automatically generates a parser and a text editor supporting syntax coloring and error highlighting

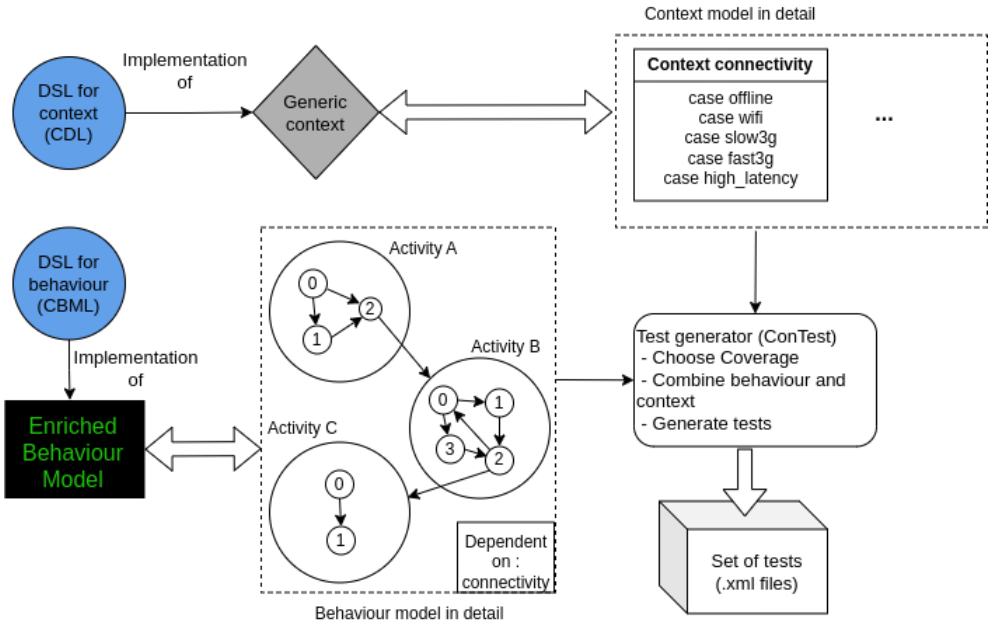

Figure 4.2: Detailed architecture of test generation

In figure 4.2 we see the design of the test generation process. As a whole, the project takes a Finite State Machine (FSM) (more precisely an Hierarchical

Finite State Machine (HFSM)) of a specific application. The behaviour knows that it is dependent on a specific number of dynamic and static contexts. Then using the context definition we generate the combinations of tests using the different context values.

In the example 4.2, the model is dependent on the connectivity. In this case we generate tests that follow the coverage criteria, taking into account that context can either be : offline, wifi or slow3g...

4.2.2 ConTest - Implementation

In this section we will explain in more detail the inner workings of ConTest following the 3 parts of figure 4.2.

4.2.2.1 CDML : Behaviour DSL

General idea : As we can see in figure 4.2, an HFSM is used to represent the model. This particularity, although originally designed for web applications [34] fits perfectly in the context of Android development.

To understand why we have to understand that an Android application is a agglomeration of different components :

- Activities
- Services
- Broadcast receivers
- Content providers

As explained the Android developers webguide[35], each component is an entry point through which the system or a user can enter your app. The developers is then able to limit who can access which components through different permissions or filters.

In terms of model we can thus separate each component in its own FSM and study it separately. This means that we would have a model with two level of abstractions : HFSM defining the model as seen from the exterior (an aggregation of components jumping from one to another), and a set of FSM defining the inner behaviour of each component.

With this in mind we define the behaviour model (Context-Driven Modelling Language (CDML)) as an .xtext file. The defining file can be found at annex A.

```
Cdml:
    'model' name=EString '{'
        ((contexts+=Contexts)?) &
        ((staticContexts+=StaticContexts)?) &
        ((situations+=Situations)?) &
        (statemachines+=Statemachine+) &
        (adaptations+=Adaptation*)
    '}'
;
```

Figure 4.3: Main description of CDML

In figure 4.3 we can see the main parts of the model. A model is described by :

- A set of dynamic contexts : The type of contexts its code depends on
- A set of static contexts : The type of static contexts its code depends on
- A set of situations : As defined by section 2.3. A set of specific contexts.
- At least one statemachine : Each defining a specific component of our application

Lastly, in order to carefully represent states that happen in a specific situation, like for the case of error handling (internet disconnected), we define a set of FSM called *adaptations*. Adaptations adapt from a specific state in a specific FSM when a specific situation triggers.

Context : As explained context is separated in three parts, each of them defining a specific part of the code.

In the *dynamic context* section, we define context names that affect specific states and situations. The names relate this model to the DSL for context Context Definition Language (CDL). The CDL model will then give us the real values that this context can have during an execution of the application. This separation helps the developer not having to specify the values itself, removing mistakes and improving model readability. An example can be seen in figure 4.4.

In the *static context* part, we define context values that affect the start of the application, these values reign the execution process of an application and do not change after boot. This information is particularily interesting for vulnerability detection as it contains configurations like Android versions this app may run on or network configuration, allowing detection of retro-compatibility vulnerabilities or more.

```
model TranslationApp {

    contexts {
        INTERNET_CONNECTIVITY
    }
```

Figure 4.4: Example of dynamic context model

```
    static contexts {
        minSdk = "26",
        maxSdk = "",
        targetSdk = "32"
    }
```

Figure 4.5: Example of static context model

In the figure 4.5, the model defines as static values the version on which the application may run. Notably minsdk defines that an application may not run on an Android phone running an API version older than minsdk. This specific example will allow us to generate tests on specific Android versions and possibly find unexpected results.

Lastly, *situations* define specific dynamic context events that may affect a state in a very specific way. Situations define specific context values during which a state will jump to an adaptation.

```
    situations {
        INTERNET_DISCONNECTED : INTERNET_CONNECTIVITY,
        INTERNET_SLOW : INTERNET_CONNECTIVITY
    }
```

Figure 4.6: Example of situation model

In the example 4.6, a situation exists when internet is disconnected. This allows to later define adaptations for cases when internet is disconnected (like to handle errors or disconnections).

FSM : As explained above, we defined statemachines for each component. As we can see in figure 4.7, each statemachine will announce it's permission type, these values are usually written by the developer in it's manifest file but are needed in order to determine if an application has permission flaws or if it's accessible from another application. The exported value indicates that the component may receive broadcasts from an exterior application, the permission

value shows which permission a specific component requires the caller to have to be called.

```
/*
 * FSM defining a specific component
 */
Statemachine:
    'statemachine' name=EString (exported?='exported' (permission=Permission)?)?'{'
        states+=State*
    '}'
;

State:
    (AtomicState | SuperState)
    (
        '{'
            transitions+=Transition*
            ('dataflows' '{' dataflows+=DataFlow* '}')?
        '}'
    )?
;
```

Figure 4.7: Description of statemachine

Each FSM (or statemachine) is defined as an agglomeration of states. Each state being either an atomic state or a super state.

```
statemachine ABSTRACT_SM {

    state START {
        transition on APP_STARTED -> SEND_MESSAGE_ACTIVITY
    }

    super state SEND_MESSAGE_ACTIVITY abstracts SEND_MESSAGE_ACTIVITY_SM {
        transition on TERMINATE_BUTTON_CLICKED -> EXIT
        transition on SUCCESS -> HANDLE_SUCCESS
    }

    state HANDLE_SUCCESS {
        transition on BACK_BUTTON_CLICKED -> SEND_MESSAGE_ACTIVITY
    }

    state EXIT
}
```

Figure 4.8: Statemachine (SM) containing the atomic state Start, Handle_Success and Exit, and the super state Send_Message_Activity

We define super states as states that abstract another FSM (in simpler words, links to another statemachine). In figure 4.8, *Send_Message_Activity* is a super state that links to the FSM *Send_Message_Activity_SM*. On the other hand, atomic states are what we could call normal states containing transitions to other states inside of the same FSM.

As seen in figure 4.9, an atomic state may be context dependent, which indicates that transitions may differ in different contexts. If a specific transition happens in a specific situation, a developer may model that with an adaptation.

```
statemachine SEND_MESSAGE_ACTIVITY_SM exported {

    state SEND_MESSAGE awareof INTERNET_CONNECTIVITY {
        transition on SEND_MESSAGE_CLICKED -> SHOW_ANSWER
    }

    state SHOW_ANSWER {
    }
}
```

Figure 4.9: Statemachine containing the context aware state Send_message and the atomic state Show_Answer

```
adaptation for INTERNET_DISCONNECTED at SEND_MESSAGE {

    state SEND_MESSAGE {
        transition on SEND_MESSAGE_CLICKED -> HANDLE_ERROR
    }

    state HANDLE_ERROR{
        transition on BACK_BUTTON_PRESSED -> external SEND_MESSAGE_ACTIVITY_SM.SEND_MESSAGE
    }

}
```

Figure 4.10: Adaptation of the state SEND_MESSAGE in the situation INTERNET_DISCONNECTED

In this example, the adaptation happens on state *SEND_MESSAGE* when situation *INTERNET_DISCONNECTED* is true. In this case the adaptation will act as if it was more states existent in the original FSM.

4.2.2.2 Context DSL

The second component of ConTest is the Context Definition Language (CDL) DSL. This language allows to define the context values of an Android application. Originally thought to be implemented by the developer for each application, during this master's thesis the design changed so that only a single implementation of CDL is needed for every application keeping it as a DSL so that it's easily extendable by a tester

It is also important to note that no really significant change has been made to CDL since the work of Adwan Abdallah, I will thus only briefly present the language to simplify the understanding of the test generation for the reader.

As explained by Adwan in his paper[9], CDL was created to capture the concepts of our context representation (Dynamic and Situation). This DSL contains information surrounding the contexts, separated from the application itself. The language itself, as seen in the figure 4.2 is set as a list with a certain

amount of data.

For each dynamic context we have can have the following data :

- Provider : Source of the context information (Ex : Location is given by the GPS sensor or internet)
- Properties : The set of value types that define a context (ex : Location is defined by longitude and latitude)

```
context INTERNET_CONNECTIVITY {
    providers: [WIFI_ADAPTER, CELL_ADAPTER],
    properties: [connectivity: Connectivity]
}

context LOCATION {
    providers: [GPS_SENSOR, CELL_ADAPTER],
    properties: [availability: Availability, longitude: double, latitude: double]
}
```

Figure 4.11: Context for location and internet connectivity as defined by CDL

The types of the properties can either be java native ones (double, int, string..) or defined by the user in lists, as seen in the figure 4.12.

```
type Connectivity {offline, wifi, slow3G, fast3G, _4g, high_latency}
```

Figure 4.12: Example of new type definition in CDL

CDL also defines situation values, meaning that for a specific situation it gives the values that the context needs to have to validate the situation. This can be seen in figure 4.13.

```
situation INTERNET_DISCONNECTED {
    INTERNET_CONNECTIVITY.connectivity == offline
}

situation INTERNET_SLOW {
    INTERNET_CONNECTIVITY.connectivity == slow3G
}
```

Figure 4.13: Situations as defined by CDL

4.2.2.3 Model enrichment

During the study on Android vulnerabilities seen on figure 2.3 and later on the table 3.1, a large part of vulnerabilities are based on incorrect/unsafe data flow. For this very reason we needed a way to track information flow through our applications.

While this is impossible in the high-level oriented design of CDML, it felt like an incredible limitation as most vulnerabilities detected would have been linked to the behaviour. One important design choice added to CDML was dataflows.

The idea of adding dataflow information in the model was guided by research papers on the subject, most notably the open-source project Flowdroid[32]. Flowdroid is a very powerful tool created by Steven Arzt et al. that allows the creation of callgraphs of the function we give it as input.

Flowdroid takes as input the Android Package Kit (APK) of an application, a list of input functions and output functions. The tool then outputs a graph of links from these input to these output functions as a full graph of function calls and data motifications. The main goal of Flowdroid is to link private data (location, sensor values..) to public channels (logs, prints, non-encrypted communication..). Indirectly we can hijack the process to also follow public inputs in any function we set as output.

This possibility means we can track these kind of dataflow :

- Public input data not sanitized in specific vulnerable functions.
- Private data sent on public channels.

Flowdroid works directly on the APK which will allow the enrichment process to be used on a possible future blackbox version of ConTest.

To give an example of the enrichment process, let's say that we use the private source *internet status* in the state *Send_Message* of the statemachine Send_message_activity_SM (figure 4.9 this will update the figure as figure 4.14.

On another part of the program we find that a function *log.d* outputs that same *internet status* this then updates the model from figure 4.10 to figure 4.15.

This example already gives an information to the developer by itself by its simplicity, we find a direct flow from a private source (data) to a public sink (channel), although its in a specific context only (Internet_disconnected).

Sinks can have multiple parameters, in figure 4.15 log.d only has a single parameter but it could have more.

```
statemachine SEND_MESSAGE_ACTIVITY_SM exported {

    state SEND_MESSAGE awareof INTERNET_CONNECTIVITY {
        transition on SEND_MESSAGE_CLICKED -> SHOW_ANSWER
        dataflows {
            source internet_status
        }
    }

    state SHOW_ANSWER {
    }
}
```

Figure 4.14: Statemachine containing the context aware state Send_message and the atomic state Show_Answer enriched with the private data Internet_Status

```
adaptation for INTERNET_DISCONNECTED at SEND_MESSAGE {

    state SEND_MESSAGE {
        transition on SEND_MESSAGE_CLICKED -> HANDLE_ERROR
    }

    state HANDLE_ERROR{
        transition on BACK_BUTTON_PRESSED -> external SEND_MESSAGE_ACTIVITY_SM.SEND_MESSAGE
        dataflows {
            sink "log.d" ( source SEND_MESSAGE_ACTIVITY_SM.SEND_MESSAGE.internet_status )
        }
    }

}
```

Figure 4.15: Adaptation of the state SEND_MESSAGE in the situation INTERNET_DISCONNECTED enriched with the public sink log.d

It is important to note that this section 4.2.2.3 is not used by the test generation process (information is kept but not used) but only by the vulnerability detection process of section 4.3.

4.2.2.4 Generator

The main process of generation is made by combining the two models (Context and Behaviour) with a set of coverage algorithms.

A coverage algorithm is ruled by a coverage criteria, a set of rules that define when the set of tests is complete. A coverage criteria exists to simplify the highly complex problem of test generation as well as to allow simple validation of the tests generated.

A coverage criteria is defined by the context of the System Under Test (SUT)

and in the case of MBST :

- All-States : All states has been executed att least once
- All-Transitions : All transitions have been gone through.
- All-Transition pairs : All pairs of adjacent transitions have been executed.

These are taken from the document by Dawood et al. [36], in the case of graphs comparable to FSMs.

In the case of contextual operations we define a new coverage criteria called All-Situation. This coverage criteria aims at validating that every context-aware state is tested against every possible value of context it depends on.

The developer is free to choose the context criterion that will be used during the test generation, each criterion has different complexity values. During the tests and for the rest of the report we will be using the criterion "All-transition and All-Situation"

Generator is a small plugin of 2400 lines of code that allows in its design to add as many coverage criterias as we want. In its current design it follows the following steps :

- Load CDML and CDL models
- Depending on the coverage criteria selected, initialize the end goal of test generation.
- For each statemachine we generate the subtests following the selected criteria
- If a statemachine has "exported" or is the main, it is counted as a starting node.
- We aggregate the tests from each super state to the statemachines generated tests
- Print/export all of the tests that come from starting nodes

The advantage of an HFSM model is the possibility to separate the generation algorithms in two parts. Generating the paths (or tests) in a single FSM (or statemachine). Then aggregating the tests together using the super states.

SubPath generation The first part, subpath generation takes every statemachine separately and build the paths found inside.

If we take as example the figure 4.8, we are able to generate the following paths (table 4.1).

Table 4.1: Subpaths of abstract_sm; Transitions between square brackets

Path #	Path description
Path 1	Start [APP_STARTED] → SEND_MESSAGE_ACTIVITY [TERMINATE_BUTTON_CLICKED] → EXIT
Path 2	Start [APP_STARTED] → SEND_MESSAGE_ACTIVITY [SUCCESS] → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE_ACTIVITY [TERMINATE_BUTTON_CLICKED] → EXIT

In a less trivial example, if we take the statemachine Send_Message_Activity_SM (figure 4.14) with the adaptation of figure 4.15 then we have to generate subpaths after having handled the contexts for the context-aware state SEND_MESSAGE. The values of Internet connectivity are found in the CDL model 4.11, 4.13 and 4.12, it allows us to generate the following paths 4.2.

Table 4.2: Subpaths of send_message_activity_sm; Transitions between square brackets; Context in green; Situation in purple

Path #	Path description
Path 1	SEND_MESSAGE [SEND_MESSAGE_CLICKED] {offline} {Internet_Disconnected} → HANDLE_ERROR [BACK_BUTTON_PRESSED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] → Show_Answer
Path 2	SEND_MESSAGE [SEND_MESSAGE_CLICKED] {wifi} {} → SHOW_ANSWER
Path 3	SEND_MESSAGE [SEND_MESSAGE_CLICKED] {slow3g} {} → SHOW_ANSWER
Path 4	SEND_MESSAGE [SEND_MESSAGE_CLICKED] {fast3g} {} → SHOW_ANSWER
Path 5	SEND_MESSAGE [SEND_MESSAGE_CLICKED] {_4g} {} → SHOW_ANSWER
Path 6	SEND_MESSAGE [SEND_MESSAGE_CLICKED] {high_latency} {} → SHOW_ANSWER

Test Path Aggregation Now that we have every subpath for every FSM, the process of aggregating every subpath from a specified top level of the hierarchy is called *aggregation*. The goal here is to take every starting node selected (the main activity and every activity we can call from the outside) and to fill every super state with the subpath we generated earlier.

One of the limitations of this technique is infinite calls. There is the possiblity that a model contains an activity **A** that calls an activity **B** that calls an activity **A**. While this is not realistic we decided to ignore loops in our code when a case like this one happens. Another debatable method is to limit the

CDML model directly, but this could have led to problematic limitations so the solution was ignored.

Once we apply this process to the tables 4.1 and 4.2, taking Abstract_SM as starting node we get :

Table 4.3: Aggregated paths of abstract sm (incomplete version)

Path	Path description
Path 1	Start [App_Started] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {offline} {Internet_Disconnected} → HANDLE_ERROR [BACK_BUTTON_PRESSED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] → Show_Answer [TERMINATE_BUTTON_CLICKED] → EXIT
Path 2	Start [App_Started] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {wifi} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 3	Start [App_Started] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {slow3g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 4	Start [App_Started] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {fast3g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 5	Start [App_Started] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {_4g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 6	Start [App_Started] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {high_latency} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path X	... Shortened, complete version at annex C

In the table 4.3, we only see a limited version as the final number of paths is 40. This is due to the original subpaths of abstract sm 4.1 having two times the super state Send_Message_Activity_SM. This means that we generated the combinations of paths.

The resulting tests are generated as xml documents, in terms of usage the main goal of using xml is to be able to both easily read the document if needed for manual translation in java/kotlin and also to be able to be able to easily parse the document for a possible automatic generation or other software. As is the case for the exploit generation that directly uses these output tests.

Lastly, we chose xml over any other type of serialising tool as it is native in java.

4.3 Exploit generator

As seen in figure 4.1, if the previous part allows us to generate tests for code coverage this section will introduce the process that allows vulnerability detection and exploit generation. In section 4.3.1 will be introduces the architecture of the solution. Then, section 4.3.2 will give a better understanding of how this solution is assembled and works by detailing the vulnerability

pattern language Vulnerability Pattern (VPat) and the vulnerability checker *VPatChecker*.

4.3.1 VPAT - Architecture

The definition of this contribution has been an extension of our design goals of section 3.2.2. The solution is technically completely separate from ConTest as it only requires tests as an input that could be generated by another tool as long as the information contained in these data packets are of the same nature as the one we implemented.

Two parts define this process :

- A set of vulnerability patterns not tied to a specific application
- A tool that compares these patterns with a set of tests and modifies the tests if they could be vulnerable

The general idea of the project is to separate the work between two distinct groups, the first one being the developer that creates his application (and potentially the model of his application). The second actor being a pentester whose job is to analyse new vulnerabilities and design their pattern with our provided language and editor VPat. VPat is a DSL written with xtext that provides a way to define a vulnerability in a simple way, with the objective of building an open-source database of vulnerabilities written abstractly to be used by security testing tools.

In figure 4.16, we see represented the detailed process of VPatChecker, that uses abstract tests in xml form and patterns to detect vulnerabilities and generate exploits.

4.3.2 VPAT - Implementation

In this section we will detail the way VPatChecker and VPat are built.

4.3.2.1 Vulnerability patterns

As explained in section 4.3.1, VPat was created like CDL and CDML using xtext. Here the advantage is very clear, we are able to separate the tool to check vulnerabilities, from the vulnerabilities and from the actual application. Complete xtext language can be found in annex D.

A vulnerability pattern is defined by it's name and description. We are then able to separate it's caracteristics into two main parts :

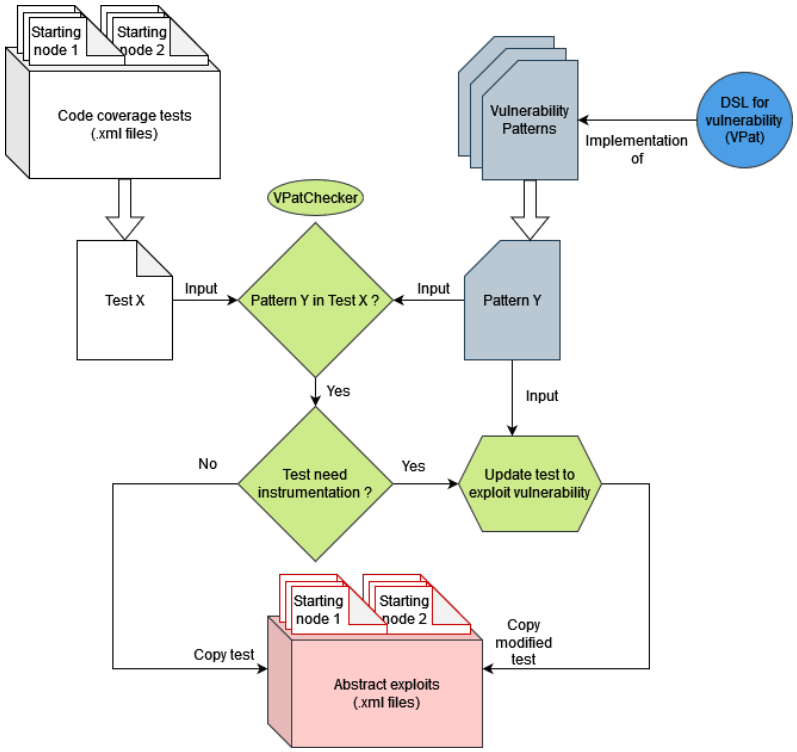

Figure 4.16: Detailed high level architecture of exploit generation (VPatChecker)

- Context : Defines the specific state the application/host should have to be vulnerable.
- Function : Defines the specific function and/or dataflow the application should have to be vulnerable.

Context This can be seen as the static context component of the application or activity. This component aims at allowing contextual vulnerabilities to be defines. Whether it's specific Android versions, network configurations or permission configurations a vulnerability can be present on a specific situation.

To give a specific example, in Android versions lower than API 28 (which corresponds to Android 9 Pïe), the default network configuration is to accept plaintext HTTP communications. This means that in the context of a default network configuration and Android API versions lower than 28, a request to a website is considered a completely public channel. Whether the communication is encrypted otherwise meaning that the communication is private (or at least as private as HTTPS is..). See example 4.17.

```
Vulnerability "HTTP_API27" {
    description "Cleartext communication is accepted on api version < 28 when network configuration is default"

    context {
        apiversion "27",
        network default
    }

    function {
        main Sink "Example_HTTP_func" {
            parameter {
                private
            }
        },
        Source private *
    }
}
```

Figure 4.17: Example of private data sent to example http_function

function This is the component that defines the dataflow we should be tracking. In a very simple way we can write the pattern 4.18, in this pattern we describe that the flow of a private value to the public sink *log.d* should be detected.

```
Vulnerability "Log.d Leak" {
    description "Log.d kept in code makes it vulnerable to leakage of private data"

    function {
        main Sink "log.d" {
            parameter {
                private
            }
        } ,
        Source private *
    }
}
```

Figure 4.18: Example of private data sent to log.d

In more generic way the function component is a sink (or final function) called with specified parameters. This sink can either be a literal sink to a public channel or simply a vulnerable function. The specified parameters are either a sink, an inflow or a static value (string, int...). Inflow is a term defining both a private source (a function or set of functions giving private information) or an input.

4.3.2.2 Vulnerability detection

The vulnerability detection process compares every test generated to every vulnerability a build a report containing the resulting information. In terms of implementation the code allows for easy addition of a different checker allowing the code to be updated with new methods using the model that has been created.

A checker makes use of the defined test model and pattern to find possible

vulnerabilities in a test. For our project we designed a simple checker that does not optimise the checking process, meaning that it checks every test on every pattern. This simple checker takes the main sink of the pattern as base and tries to match the pattern sink to the test sink. The checker builds a report folder of positive and negative reports that will later be used to generate reporting and real folders with the tests.

The report classifies the exploits per starting node, then vulnerability and finally execution context. This classification is used to simplify real code generation and for readability.

4.3.2.3 Update tests

The last process of the exploit generation process is transforming a simple behaviour test into an exploit. In our case the difference between behaviour test and exploit is thin, a behaviour test can be an exploit without change when the vulnerability is a simple design mistake, for example if the developer chose to let a logging function print the gps values then it leaks private data but no manipulation of input values may be needed as long as we get to the specific line of code of the function.

The other way around sometimes we may need to control the input values in a specific test to control a specific value in a function.

```
Vulnerability "vulnerableFunction EXPLOITME" {
    description "The function vulnerableFunction leaks data when the second parameter is EXPLOITME in android version 31"

    context {
        apiversion "31"
    }

    function {
        main Sink "vulnerableFunction" {
            parameter {
                private,
                static "EXPLOITME"
            }
        },
        Source private *
    }
}
```

Figure 4.19: Example of pattern that needs a specific value in its second parameter

In the pattern seen on figure 4.19, we can see that the function *vulnerableFunc* can be exploited when we have the possibility to set the value *EXPLOITME* on the second parameter. In our case a test may have a the sink *vulnerableFunc* with a second parameter that already has an static value in it but it also may be an input value.

In the excerpt 4.20, we can see in the state *DISPLAY_WARNING* that the dataflow *vulnerableFunc* takes as parameters :

```
<state name="SEND_MESSAGE">
    <transition name="SEND_MESSAGE_CLICKED">
        <contexts>
            <context origin="INTERNET_CONNECTIVITY">wifi</context>
        </contexts>
    </transition>
    <dataflows>
        <dataflow name="internet" type="Source"/>
    </dataflows>
</state>
<state name="SENDER">
    <transition name="EXCEPTION"/>
    <dataflows>
        <dataflow name="enter_value" type="Input" value="EXPLOITME"/>
    </dataflows>
</state>
<state name="DISPLAY_WARNING">
    <transition name="BACK_BUTTON_PRESSED"/>
    <dataflows>
        <dataflow name="vulnerableFunc" type="Sink">
            <parameters>
                <parameter origin="source">internet</parameter>
                <parameter origin="source">enter_value</parameter>
            </parameters>
        </dataflow>
    </dataflows>
</state>
```

Figure 4.20: Excerpt from a test generated by ConTest, input value is set to "*"

- The source *internet* from the state *SEND_MESSAGE*
- The source *enter_value* from the state *SENDER*

When building the exploit for the specific pattern 4.19, we will change the value of the input dataflow *enter_value* in the state *SENDER* to *EXPLOITME* to fit the pattern. This gives us the exploit modified 4.3.2.3.

```
<state name="SEND_MESSAGE">
<transition name="SEND_MESSAGE_CLICKED">
    <contexts>
        <context origin="INTERNET_CONNECTIVITY">wifi</context>
    </contexts>
</transition>
<dataflows>
    <dataflow name="internet" type="Source"/>
</dataflows>
</state>
<state name="SENDER">
<transition name="EXCEPTION"/>
<dataflows>
    <dataflow name="enter_value" type="Input" value="EXPLOITME"/>
</dataflows>
</state>
<state name="DISPLAY_WARNING">
<transition name="BACK_BUTTON_PRESSED"/>
<dataflows>
    <dataflow name="vulnerableFunc" type="Sink">
        <parameters>
            <parameter origin="source">internet</parameter>
            <parameter origin="source">enter_value</parameter>
        </parameters>
    </dataflow>
</dataflows>
</state>
```

Listing 4.1: Excerpt from an exploit generated by VPatChecker, input value is set to the value EXPLOITME

Chapter 5

Results and Analysis

5.1 Discussion

Chapter 6

Conclusions and Future work

Add text to introduce the subsections of this chapter.

6.1 Conclusions

6.2 Limitations

6.3 Future work

Describe valid future work that you or someone else could or should do. Consider: What you have left undone? What are the next obvious things to be done? What hints can you give to the next person who is going to follow up on your work?

6.3.1 What has been left undone?

6.3.1.1 Missing thing 1

6.3.1.2 Missing thing 2

6.3.2 Next obvious things to be done

6.4 Reflections

What are the relevant economic, social, environmental, and ethical aspects of your work?

References

[1] "Digital 2021: Global Overview Report." [Online]. Available: https://datareportal.com/reports/digital-2021-global-overview-report

[2] "Number of Mobile App Downloads in 2022/2023: Statistics, Current Trends, and Predictions," Mar. 2020. [Online]. Available: https://financesonline.com/number-of-mobile-app-downloads/

[3] V. team, "State of Software Security Volume 12," *Veracode*, 2022. [Online]. Available: https://www.veracode.com/state-of-software-security-report

[4] J.-y. Hong, E. Suh, and S.-J. Kim, "Context-aware systems: A literature review and classification," *Expert Syst. Appl.*, vol. 36, pp. 8509–8522, Jan. 2009.

[5] Zameer, "Understanding Mobile Context Awareness," Jan. 2021. [Online]. Available: https://medium.com/ascentic-technology/understanding-mobile-context-awareness-887a9d380d21

[6] D. R. Almeida, P. D. L. Machado, and W. L. Andrade, "Testing tools for Android context-aware applications: a systematic mapping," *Journal of the Brazilian Computer Society*, vol. 25, no. 1, p. 12, Dec. 2019. [Online]. Available: https://doi.org/10.1186/s13173-019-0093-7

[7] "statista.com : Global mobile OS market share 2012-2022." [Online]. Available: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

[8] "Secure Code Warrior Survey Finds 86% of Developers Do Not View Application Security As a Top Priority." [Online]. Available: https://www.securecodewarrior.com/press-releases/secure-code-warrior-survey-finds-86-of-developers-do-not-view-application-security-as-a-top-priority

[9] A. Adwan, "Context-dependent Model-based Testing of Mobile Apps," Master's thesis, University Grenoble Alpes, France, 2021.

[10] I. Schieferdecker, J. Grossmann, and M. Schneider, "Model-Based Security Testing," *Electronic Proceedings in Theoretical Computer Science*, vol. 80, pp. 1–12, Feb. 2012, arXiv:1202.6118 [cs]. [Online]. Available: http://arxiv.org/abs/1202.6118

[11] J. A. Goguen and J. Meseguer, "Security Policies and Security Models," in *1982 IEEE Symposium on Security and Privacy*, Apr. 1982, pp. 11–11, iSSN: 1540-7993.

[12] M. R. Clarkson and F. B. Schneider, "Hyperproperties," in *2008 21st IEEE Computer Security Foundations Symposium*, Jun. 2008, pp. 51–65, iSSN: 2377-5459.

[13] A. Alqazzaz, I. Alrashdi, R. Alharthi, E. Aloufi, and M. A. Zohdy, "An Insight into Android Side-Channel Attacks," in *2018 International Conference on Computational Science and Computational Intelligence (CSCI)*, Dec. 2018, pp. 776–780.

[14] Wikipedia contributors, "Android (operating system) — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Android_(operating_system)&oldid=1114389454, 2022, [Online; accessed 6-October-2022].

[15] "Google Bug Hunters." [Online]. Available: https://bughunters.google.com/

[16] "Platform Architecture." [Online]. Available: https://developer.android.com/guide/platform

[17] A. Mazuera-Rozo, J. Bautista-Mora, M. Linares-Vásquez, S. Rueda, and G. Bavota, "The Android OS stack and its vulnerabilities: an empirical study," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2056–2101, Aug. 2019. [Online]. Available: https://doi.org/10.1007/s10664-019-09689-7

[18] "Permissions on Android." [Online]. Available: https://developer.android.com/guide/topics/permissions/overview

[19] "Request app permissions." [Online]. Available: https://developer.android.com/training/permissions/requesting

[20] S. Kumar, R. Shanker, and S. Verma, "Context Aware Dynamic Permission Model: A Retrospect of Privacy and Security in Android System," in *2018 International Conference on Intelligent Circuits and Systems (ICICS)*, Apr. 2018, pp. 324–329.

[21] I. M. Almomani and A. A. Khayer, "A Comprehensive Analysis of the Android Permissions System," *IEEE Access*, vol. 8, pp. 216 671–216 688, 2020, conference Name: IEEE Access.

[22] H. Muccini, A. Di Francesco, and P. Esposito, "Software testing of mobile applications: challenges and future research directions," in *Proceedings of the 7th International Workshop on Automation of Software Test*, ser. AST '12. Zurich, Switzerland: IEEE Press, Jun. 2012, pp. 29–35.

[23] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles, "Towards a Better Understanding of Context and Context-Awareness," in *Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, ser. HUC '99. Berlin, Heidelberg: Springer-Verlag, Sep. 1999, pp. 304–307.

[24] P. J. Brown, "The stick-e document: a framework for creating context-aware applications," in *Proceedings of EP'96, Palo Alto*. also published in it EP–odd, January 1996, pp. 182–196. [Online]. Available: http://www.cs.kent.ac.uk/pubs/1996/396

[25] M. Gomez, R. Rouvoy, B. Adams, and L. Seinturier, "Reproducing Context-Sensitive Crashes of Mobile Apps Using Crowdsourced Monitoring," in *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, May 2016, pp. 88–99.

[26] D. R. Almeida, P. D. L. Machado, and W. L. Andrade, "Testing tools for Android context-aware applications: a systematic mapping," *Journal of the Brazilian Computer Society*, vol. 25, no. 1, p. 12, Dec. 2019. [Online]. Available: https://doi.org/10.1186/s13173-019-0093-7

[27] "CWE - CWE-919: Weaknesses in Mobile Applications (4.8)." [Online]. Available: https://cwe.mitre.org/data/definitions/919.html

[28] Wikipedia contributors, "Transport layer security — Wikipedia, the free encyclopedia," 2022, [Online; accessed 10-October-2022]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Transport_Layer_Security&oldid=1110721112

[29] "Network security configuration." [Online]. Available: https://developer.android.com/training/articles/security-config

[30] M. Mohamed, B. Shrestha, and N. Saxena, "SMASheD: Sniffing and Manipulating Android Sensor Data for Offensive Purposes," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 4, pp. 901–913, Apr. 2017, conference Name: IEEE Transactions on Information Forensics and Security.

[31] J. Mitra and V.-P. Ranganath, "Ghera: A Repository of Android App Vulnerability Benchmarks," in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE. New York, NY, USA: Association for Computing Machinery, Nov. 2017, pp. 43–52. [Online]. Available: https://doi.org/10.1145/3127005.3127010

[32] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594299

[33] "Xtext - Language Engineering Made Easy!" [Online]. Available: https://www.eclipse.org/Xtext/

[34] A. A. Andrews, J. Offutt, and R. T. Alexander, "Testing Web applications by modeling with FSMs," *Software & Systems Modeling*, vol. 4, no. 3, pp. 326–345, Jul. 2005. [Online]. Available: http://link.springer.com/10.1007/s10270-004-0077-7

[35] "Application Fundamentals." [Online]. Available: https://developer.android.com/guide/components/fundamentals

[36] Y. D. Salman, N. L. Hashim, M. M. Rejab, R. Romli, and H. Mohd, "Coverage criteria for test case generation using UML state chart diagram," *AIP Conference Proceedings*, vol. 1891, no. 1, p. 020125, Oct. 2017, publisher: American Institute of Physics. [Online]. Available: https://aip.scitation.org/doi/abs/10.1063/1.5005458

Vulnerability bibliography

[Vuln1] F. H. Shezan, S. F. Afroze, and A. Iqbal, "Vulnerability detection in recent Android apps: An empirical study," in *2017 International Conference on Networking, Systems and Security (NSysS)*, Jan. 2017, pp. 55–63.

[Vuln2] T. Liu, H. Wang, L. Li, X. Luo, F. Dong, Y. Guo, L. Wang, T. Bissyandé, and J. Klein, "MadDroid: Characterizing and Detecting Devious Ad Contents for Android Apps," in *Proceedings of The Web Conference 2020*, ser. WWW '20. New York, NY, USA: Association for Computing Machinery, Apr. 2020, pp. 1715–1726. [Online]. Available: https://doi.org/10.1145/3366423.3380242

[Vuln3] F. Tabassum and A. M. Faisal, "Vulnerability testing in online shopping android applications," in *2017 IEEE Region 10 Humanitarian Technology Conference (R10-HTC)*, Dec. 2017, pp. 654–657, iSSN: 2572-7621.

[Vuln4] J. Gao, L. Li, P. Kong, T. F. Bissyandé, and J. Klein, "Understanding the Evolution of Android App Vulnerabilities," *IEEE Transactions on Reliability*, vol. 70, no. 1, pp. 212–230, Mar. 2021, conference Name: IEEE Transactions on Reliability.

[Vuln5] M. A. El-Zawawy, E. Losiouk, and M. Conti, "Vulnerabilities in Android webview objects: Still not the end!" *Computers & Security*, vol. 109, p. 102395, Oct. 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404821002194

[Vuln6] S. Bojjagani and V. Sastry, "STAMBA: Security Testing for Android Mobile Banking Apps," vol. 425, Dec. 2015.

[Vuln7] S. Almanee, M. Payer, and J. Garcia, *Too Quiet in the Library: A Study of Native Third-Party Libraries in Android*, Nov. 2019.

[Vuln8] P. Bhat and K. Dutta, "A Survey on Various Threats and Current State of Security in Android Platform," *ACM Computing Surveys*, vol. 52, no. 1, pp. 21:1–21:35, Feb. 2019. [Online]. Available: https://doi.org/10.1145/3301285

[Vuln9] V. Jain, M. Gaur, V. Laxmi, and M. Mosbah, "Detection of SQLite Database Vulnerabilities in Android Apps," Dec. 2016.

[Vuln10] A. Nirumand, B. Zamani, and B. Tork Ladani, "VAnDroid: A framework for vulnerability analysis of Android applications using a model-driven reverse engineering technique," *Software: Practice and Experience*, vol. 49, no. 1, pp. 70–99, 2019, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2643. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2643

[Vuln11] P. Sun, S. Chen, L. Fan, P. Gao, F. Song, and M. Yang, "VenomAttack: automated and adaptive activity hijacking in Android," *Frontiers of Computer Science*, vol. 17, no. 1, p. 171801, Aug. 2022. [Online]. Available: https://doi.org/10.1007/s11704-021-1126-x

[Vuln12] L. Gonzalez-Manzano, U. Mahbub, J. M. de Fuentes, and R. Chellappa, "Impact of injection attacks on sensor-based continuous authentication for smartphones," *Computer Communications*, vol. 163, pp. 150–161, Nov. 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366420319095

Appendix A

CDML.xtext

```
grammar fr.lcis.castav.cdml.CDML with org.eclipse.xtext.common.Terminals

generate cDML "http://www.lcis.fr/castav/cdml/CDML"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Cdml:
    'model' name=EString '{'
        ((contexts+=Contexts)?) &
        ((staticContexts+=StaticContexts)?) &
        ((situations+=Situations)?) &
        (statemachines+=Statemachine+) &
        (adaptations+=Adaptation*)
    '}'
;

/*
 * Dynamic context whose applications depend on
 */
Contexts:
    'contexts' '{'
        contexts+=Context (',' contexts+=Context)*
    '}'
;

Context:
    name=EString
;

/*
 * Static Context that define the application
 */
StaticContexts:
    'static' 'contexts' '{'
        staticContexts+=StaticContext (',' staticContexts+=StaticContext)*
    '}'
;

StaticContext:
    name=EString '=' value=STRING
;

/*
 * Situations that exist in the context of the application.
 * Each situation links to which context changes said situation
 */
Situations:
    'situations' '{'
        situations+=Situation (',' situations+=Situation)*
    '}'
;

Situation:
    name=EString ":" context=[Context|ID]
;
```

```
/*
 * FSM defining a specific component
 */
Statemachine:
    'statemachine' name=EString (exported?='exported' (permission=Permission)?)?'{'
        states+=State*
    '}'
;

State:
    (AtomicState | SuperState)
    (
        '{'
            transitions+=Transition*
            ('dataflows' '{' dataflows+=DataFlow* '}')?
        '}'
    )?
;

AtomicState:
    'state' name=EString (contextAware?='awareof' contexts+=[Context] (',' contexts+=[Context])*)?
;

SuperState:
    'super' 'state' name=EString 'abstracts' abstracts=[Statemachine]
;

//External transition: source and target states do not belong to the same statemachine
Transition:
    {Transition} 'transition' ('on' on=Event)? '->' ((external?='external' target=[State|FQN] )? | target=[State])
;

Event:
    name=EString
;

/**
 * Permission defining what we need to start said component/FSM
 */
Permission:
    (
        normal?='normal'  permissionValues+=PermissionValue (',' permissionValues+=PermissionValue)* |
        dangerous?='dangerous' permissionValues+=PermissionValue (',' permissionValues+=PermissionValue)* |
        signature?='signature' |
        signatureOrSystem?='signatureOrSystem'
    )
;

PermissionValue:
    name=EString
;

/*
 * Enriched model part: Contains information on source/sink inside the model
 */

DataFlow:
    (Source | Sink)
;

Sink:
    'sink' name=EString '(' (parameters+=Parameter (',' parameters+=Parameter)*)? ')'
;

Parameter:
    (wildcard?='*') | value=ID | (source?='source') origin=[Source|FQN]
;

Source:
    (input?='input')? 'source' name=EString
;

/**
 * Defines states that happen in specific situations only
 */
Adaptation:
    'adaptation' 'for' situations+=[Situation] (',' situations+=[Situation])* 'at' state=[State]
    '{'
        states+=State*
```

```
    '}'
;

FQN hidden(): EString('.' EString)*;

EString returns ecore::EString:
    STRING | ID;
```

Appendix B

CDL.xtext

```
grammar fr.lcis.castav.cdl.CDL with org.eclipse.xtext.common.Terminals

generate cDL "http://www.lcis.fr/castav/cdl/CDL"

ContextModel:
    "contextModel" name=ID '{'
        Contexts+=Context* &
        Providers+=Providers* &
        Situations+=Situation* &
        Types+=DefinedType*
    '}'
;

Context:
    (static?='static')? 'context' name=ID (derived?='derives' derives+=[Context] (',' derives+=[Context] )*)? '{'
        ('providers' ':' '[' providers+=[Provider] (',' providers+=[Provider])* ']' ',')?
        'properties' ':' '[' properties+=Property (',' properties+=Property)* ']'
        (',' ('mappings') ':' '{'
            mappings+=ContextMapping (',' mappings+=ContextMapping)*
        '}')?
    '}'
;

ContextMapping:
    ref=[ContextValue|FQN] '->' expression=ContextExpression
;


Providers: 'providers' '{'
        providers+=Provider (',' providers+=Provider)*
    '}'
;

Provider:
    name=ID
;

Property:
    name=ID ':' type=(TypeRef|SimpleType)
;

TypeRef:
    ref=[DefinedType|ID]
;

DefinedType:
    'type' name=ID '{'
        values+=ContextValue (',' values+=ContextValue)*
    '}'
;

ContextValue:
    name=(STRING | ID)
;
```

```
SimpleType:
    StringType | IntegerType | BooleanType | DoubleType
;

StringType:
    {StringType} "string"
;
IntegerType:
    {IntegerType} "integer"
;
BooleanType:
    {BooleanType} "boolean"
;
DoubleType:
    {DoubleType} "double"
;

Situation:
    'situation' name=ID '{'
        expression+=ContextExpression
    '}'
;

ContextExpression:
    ref=[Property|FQN] ('<' | '>' | '>=' | '<=' | '==' | '!=') value=ContextValue (('and' | 'or') expr=ContextExpression)?
;

FQN hidden(): ID('.' ID)*;
```

Appendix C

Full table : Aggregation

Table C.1: Aggregated paths of abstract sm (complete version)

Path #	Path description
Path 0	Start [App_Started] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {offline} {Internet_Disconnected} → HANDLE_ERROR [BACK_BUTTON_PRESSED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] → Show_Answer [TERMINATE_BUTTON_CLICKED] → EXIT
Path 1	Start [App_Started] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {wifi} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 2	Start [App_Started] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {slow3g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 3	Start [App_Started] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {fast3g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 4	Start [App_Started] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {_4g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 5	Start [App_Started] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {high_latency} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 6	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {offline} {Internet_Disconnected} → HANDLE_ERROR [BACK_BUTTON_PRESSED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] → Show_Answer → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {offline} {Internet_Disconnected} → HANDLE_ERROR [BACK_BUTTON_PRESSED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] → Show_Answer [TERMINATE_BUTTON_CLICKED] → EXIT
Path 7	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {wifi} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {offline} {Internet_Disconnected} → HANDLE_ERROR [BACK_BUTTON_PRESSED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] → Show_Answer [TERMINATE_BUTTON_CLICKED] → EXIT
Path 8	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {slow3g} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {offline} {Internet_Disconnected} → HANDLE_ERROR [BACK_BUTTON_PRESSED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] → Show_Answer [TERMINATE_BUTTON_CLICKED] → EXIT
Path 9	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {fast3g} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {offline} {Internet_Disconnected} → HANDLE_ERROR [BACK_BUTTON_PRESSED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] → Show_Answer [TERMINATE_BUTTON_CLICKED] → EXIT
Path 10	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {_4g} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {offline} {Internet_Disconnected} → HANDLE_ERROR [BACK_BUTTON_PRESSED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] → Show_Answer [TERMINATE_BUTTON_CLICKED] → EXIT
Path 11	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {high_latency} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {offline} {Internet_Disconnected} → HANDLE_ERROR [BACK_BUTTON_PRESSED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] → Show_Answer [TERMINATE_BUTTON_CLICKED] → EXIT
Path 12	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {offline} {Internet_Disconnected} → HANDLE_ERROR [BACK_BUTTON_PRESSED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] → Show_Answer → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {wifi} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 13	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {wifi} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {wifi} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 14	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {slow3g} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {wifi} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 15	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {fast3g} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {wifi} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 16	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {_4g} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {wifi} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 17	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {high_latency} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {wifi} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 18	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {offline} {Internet_Disconnected} → HANDLE_ERROR [BACK_BUTTON_PRESSED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] → Show_Answer → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {slow3g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 19	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {wifi} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {slow3g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 20	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {slow3g} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {slow3g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 21	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {fast3g} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {slow3g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 22	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {_4g} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {slow3g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 23	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {high_latency} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {slow3g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 24	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {offline} {Internet_Disconnected} → HANDLE_ERROR [BACK_BUTTON_PRESSED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] → Show_Answer → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {fast3g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 25	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {wifi} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {fast3g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 26	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {slow3g} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {fast3g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 27	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {fast3g} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {fast3g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 28	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {_4g} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {fast3g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 29	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {high_latency} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {fast3g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 30	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {offline} {Internet_Disconnected} → HANDLE_ERROR [BACK_BUTTON_PRESSED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] → Show_Answer → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {_4g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 31	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {wifi} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {_4g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 32	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {slow3g} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {_4g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 33	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {fast3g} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {_4g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 34	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {_4g} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {_4g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 35	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {high_latency} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {_4g} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 36	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {offline} {Internet_Disconnected} → HANDLE_ERROR [BACK_BUTTON_PRESSED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] → Show_Answer → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {high_latency} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 37	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {wifi} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {high_latency} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 38	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {slow3g} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {high_latency} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 39	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {fast3g} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {high_latency} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 40	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {_4g} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {high_latency} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT
Path 41	Start [APP_STARTED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {high_latency} {} → SHOW_ANSWER → HANDLE_SUCCESS [BACK_BUTTON_CLICKED] → SEND_MESSAGE [SEND_MESSAGE_CLICKED] {high_latency} {} → SHOW_ANSWER [TERMINATE_BUTTON_CLICKED] → EXIT

Appendix D

VPAT.xtext

```
grammar fr.lcis.castav.vpat.VPAT with org.eclipse.xtext.common.Terminals

generate vPAT "http://www.lcis.fr/castav/vpat/VPAT"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore


Vulnerability returns Vulnerability:
    {Vulnerability}
    'Vulnerability'
    name=EString
    '{'
        ('description' description=EString)?
        ('context' '{' context+=Context ( "," context+=Context)* '}')?
        ('function' '{' ('main' mainFunction=Sink & (( "," function+=Function)*)) '}')?
    '}';

EString returns ecore::EString:
    STRING | ID;

Context returns Context:
    Permission?='android.permission.' value=Permission | Network?='network' value=Network | Version?="apiversion" value=Version
;

Version returns Version:
    name=STRING
;

Permission returns Permissions:
    {Permission} name=permissionID
;

//TODO : ADD every permission in android
permissionID returns PermissionID:
    name='ACCESS_MEDIA_LOCATION' |
    name='ACCESS_NETWORK_STATE' |
    name='ACCESS_WIFI_STATE' |
    name='INTERNET'
;

//TODO : ADD network configurations (Trusted CA...)
Network returns Network:
    {Network} 'default'
;

Function returns Function:
    Sink | Inflow
;


Sink returns Sink:
    'Sink' name=EString
    '{'
        ('parameter' '{'
            parameter+=Parameter ( "," parameter+=Parameter)* '}'
```

```
            )?
        '}'
;

Parameter returns Parameter:
        origin=[Function|FQN] |
        (static?='static' (anyValue?='*' | value=EString))
;

Inflow returns Inflow:
        Source |
        Input
;

Source returns Source:
        {Source} 'Source' name=EString (anyPrivate?='*' | method=EString)
;

Input returns Input:
        {Input} 'Input' name=EString method=EString
;

FQN hidden(): EString('.' EString)*;
```