KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Computer Science

Second cycle, 30 credits

# Context-aware security testing of Android applications

## Detecting exploitable vulnerabilities through Android model-based security testing

**IVAN BAHEUX**

Stockholm, Sweden, 2023

# Context-aware security testing of Android applications

## Detecting exploitable vulnerabilities through Android model-based security testing

IVAN BAHEUX

Master's Programme, Computer Science, 120 credits
Date: April 4, 2023

Supervisors: Karl Palmskog, Oum-El-Kheir Aktouf
Examiner: Roberto Guanciale
School of Electrical Engineering and Computer Science
Host company: Laboratoire de conception et d'intégration des systèmes (LCIS)
Swedish title: Kontextmedveten säkerhetstestning av androidapplikationer
Swedish subtitle: Upptäckande av utnyttjingsbara sårbarheter genom Android modellbaserad säkerhetstestning

# Abstract

This master's thesis explores ways to uncover and exploit vulnerabilities in Android applications by introducing a novel approach to security testing. The research question focuses on discovering an effective method for detecting vulnerabilities related to the context of an application.

The study begins by reviewing recent papers on Android security flaws affecting application in order to guide our tool creation. Thus, we are able to introduce three Domain Specific Languages (DSLs) for Model-Based Security Testing (MBST): Context Definition Language (CDL), Context-Driven Modelling Language (CDML), and Vulnerability Pattern (VPat). These languages provide a fresh perspective on evaluating the security of Android apps by accounting for the dynamic context that is present on smartphones and can greatly impact user security.

The result of this work is the development of VPatChecker[1], a tool that detects vulnerabilities and creates abstract exploits by integrating an application model, a context model, and a set of vulnerability patterns. This set of vulnerability patterns can be defined to represent a wide array of vulnerabilities, allowing the tool to be indefinitely updated with each new CVE.

The tool was evaluated on the GHERA benchmark, showing that at least 38% (out of a total of 60) of the vulnerabilities in the benchmark can be modelled and detected.

The research underscores the importance of considering context in Android security testing and presents a viable and extendable solution for identifying vulnerabilities through MBST and DSLs.

## Keywords

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

# Sammanfattning

Detta examensarbete utforskar vägar för att hitta och utnyttja sårbarheter i Android-appar genom att introducera ett nytt sätt att utföra säkerhetstestning. Forskningsfrågan fokuserar på att upptäcka en effektiv metod för att detektera sårbarheter som kan härledas till kontexten för en app. Arbetet inleds med en översikt av nyliga forskningspublikationer om säkerhetsbrister som påverkar Android-appar, vilka vägleder utvecklingen av ett verktyg. Vi introducerar tre domänspecifika språk (DSL) för modellbaserad testning (MBST): CDL, CDML och VPat. Dessa språk ger ett nytt perspektiv på säkerheten för Android-appar genom att ta hänsyn till den dynamiska kontext som finns på smarta mobiltelefoner och som kan starkt påverka användarsäkerheten.

Resultatet av arbetet är utveckling av VPatChecker[1], ett verktyg som upptäcker sårbarheter och skapar abstrakta sätt att utnyttja dem i en programmodell, en kontextmodell, och en mängd av sårbarhetsmönster. Denna sårbarhetsmönstermängd kan defineras så att den representerar ett brett spektrum av sårbarheter, vilket möjliggör för verktyger att uppdateras med varje ny CVE.Verktyget utvärderades på datamängden GHERA, vilket visade att 38% (av totalt 60) av alla sårbarheter kunde modelleras och upptäckas.Arbetet understryker vikten av att ta hänsyn till kontext i säkerhetstestning av Android-appar och presenterar en praktisk och utdragbar lösning för att hitta sårbarheter genom MBST and DSLs.

## Nyckelord

Android-applikationssäkerhet, Upptäckt av sårbarheter, Kontextmedvetenhet, Modellbaserad säkerhetstestning, Domänspecifikt språk

# Résumé

Ce mémoire de maîtrise explore les moyens de découvrir et d'exploiter les vulnérabilités des applications Android en introduisant une nouvelle approche des tests de sécurité. La question de recherche se concentre sur la découverte d'une méthode efficace pour détecter les vulnérabilités liées au contexte d'une application.

L'étude commence par l'examen de documents récents sur les failles de sécurité des applications Android afin de guider la création de notre outil. Nous sommes ainsi en mesure d'introduire trois Langages dédié (DSL) pour des Tests de Sécurité Basés sur les Modèles (MBST) : Langage de Définition de Contexte (CDL), Langage de Modélisation Déterminée par le Contexte (CDML) et Motif de Vulnérabilité (VPat). Ces langages offrent une nouvelle perspective sur l'évaluation de la sécurité des applications Android en tenant compte du contexte dynamique présent sur les smartphones et qui peut avoir un impact important sur la sécurité de l'utilisateur.

Le résultat de ce travail est le développement de VPatChecker[1], un outil qui détecte les vulnérabilités et crée des exploits abstraits en intégrant un modèle d'application, un modèle de contexte et un ensemble de modèles de vulnérabilité. Cet ensemble de modèles de vulnérabilité peut être défini pour représenter un large éventail de vulnérabilités, ce qui permet à l'outil d'être indéfiniment mis à jour avec chaque nouveau CVE.

L'outil a été testé sur le benchmark GHERA[2] et montre qu'un total d'au moins 38% (sur un total de 60) des vulnérabilités peut être modélisé et détecté.

La recherche souligne l'importance de prendre en compte le contexte dans les tests de sécurité Android et présente une solution viable et extensible pour identifier les vulnérabilités par le biais de MBST et DSLs.

## Mots-clés

Sécurité des Applications Android, Détection de Vulnérabilités, Sensibilité au Contexte, Tests de Sécurité Basés sur les Modèles, Langage Dédiés

# Acknowledgments

I could not have undertaken this journey without Oum-El-Kheir Aktouf, my LCIS Supervisor, thank you for trusting me with this project, for your guidance during these past 5 month and also for your patience. I am also extremely grateful to Karl Palmskog, my KTH supervisor, for his help in editing and writing this report, as well as for his general guidance in making my scientific project complete.

I am thankful for the very helpful criticisms of the LIG, LCIS and LS2N laboratory representatives which made it possible to direct the project towards what it is today. I would also like to thank Roberto Guanciale, my examiner, who introduced me to Karl Palmskog and took the time to read and give feedback on my work.

I had the pleasure of collaborating with the members of the LCIS lab, listening to their stories and chilling with them, thank you for making this experience so fun.

Lastly, I would like to thank my family. My partner, parents and sisters who listened to me and gave me moral support when I needed it most.

Valence, France, April 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. |
| FlowDroid | FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. |
| 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.. |
| Sink | A sink is an open output of a system that acts as the exit of data in a data flow. |

# List of acronyms and abbreviations

| | |
|---|---|
| ABI | Application Binary Interface. |
| APK | Android Package Kit. |
| ART | Android Runtime. |
| CA | Certificate authority. |
| CDL | Context Definition Language. |
| CDML | Context-Driven Modelling Language. |
| CIA | Confidentiality, Integrity and Availability. |
| CVE | Common Vulnerabilities and Exposures. |
| CWE | Common Weakness Enumeration. |
| DEX | Dalvik Executable Format. |
| DOS | Denial Of Service. |
| DSL | Domain Specific Language. |
| FSM | Finite State Machine. |
| HAL | Hardware Abstraction Layer. |
| HFSM | Hierarchical Finite State Machine. |
| IoT | Internet of Things. |
| IPC | Inter-Process Communication. |
| MBST | Model-Based Security Testing. |
| MBT | Model Based Testing. |
| MitM | Man in the Middle. |
| OS | Operating System. |
| SQL | Structured Query Language. |
| SSL | Secure Sockets Layer. |
| SuT | System under Test. |
| TLS | Transport Layer Security. |
| VPat | Vulnerability Pattern. |

# 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 [3]. In 2020 over 200 billion mobile applications have been downloaded [4] 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 [5], 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 [5]. 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. [6] 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 [7]. 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 [8], 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 [9], 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 foundation for this master's thesis.

## 1.2 Problem

As seen in section1.1, with the amount of time spent on their phone, there is a need for users to be able to trust the tools t hey are using. For this reason, 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 tremendous amount of downloads a single application can have, the executable is run in different Operating Systems (OSs), in different devices with different capabilities and limitations.

In terms of development, security is still fairly uncommon as a

consideration and not considered a priority by developers [10]. 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.

## 1.3 Threat Model

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: Denial Of Service (DOS)

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

More specifically, in the context of Android applications, the adversary will either want to steal the user's private data or misuse the application to get more rights than what should be allowed.

## 1.4 Goals

In 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 can they be modeled 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 [11] concerning context-aware testing of Android applications, Adwan developed a model for Android apps and their context. The host laboratory LCIS 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 an application code.

## 1.5 Research methodology

The methodology will be an implementation of MBST. If we follow the definition by Schieferdecker et al. [12], 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, etc.)
- Expand on the testing model previously created by the lab with the information needed to detect the selected vulnerability.

- Model the vulnerability with a specifically crafted Domain Specific Language (DSL).
- Link both models via a penetration test generator on a vulnerable application.
- 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.
- Expand and generalise our process to related vulnerabilities.

## 1.6 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.

### 1.6.1 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.

## 1.7 Structure of the thesis

- Chapter 2 introduces the concepts of Android, context and a study on current vulnerabilities.
- Chapter 3 presents the design goals for the project, and the methodology for analysing the results obtained.
- Chapter 4 presents the tools created during the thesis, *ConTest* a code coverage tool and *VPatChecker* an exploit generator.

- Chapter 5 will demonstrate the capabilities of the aforementioned tools and discuss their range of usage.
- Lastly, chapter 6 will discuss the limitations, possible enhancements that could be made in the future, as well as conclude this thesis.

# Chapter 2

# Background

## 2.1 Security models and security policies

In the following sections, we will be reviewing Android application vulnerabilities. We consider vulnerabilities from several perspectives, but one particularly important perspective is based on *information flow*. In the idealized form originally described by Goguen and Meseguer in 1982 [13], information flow security is attained by achieving *noninterference*: no matter what public inputs are given to a system, the public outputs of this system will not be unduly influenced by the private (secret) inputs to the system. In particular, the publicly available outputs of a secure program (or, mobile app) is not allowed to contain data deemed private. Noninterference is a form of *hyperproperty* [14] that imposes requirements over *sets* of executions of a system and is difficult to establish in practice.

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. Since hyperproperties such as noninterference are too strong for our purposes, we will limit our scope to direct information flow (from high to low) meaning private data leakage. 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. [15] 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 [9]. 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 [16].

The development of Android is mostly done by Google but has open-source parts (mostly the base OS) and they are open to outside contributions. For example, the Google vulnerability reward program [17] rewards users for vulnerability reports with digital trophies and paid rewards.

### 2.2.1 Android architecture

As explained in section 2.2, Android is a mobile operating system build on the Linux kernel. According to the official documentation of Android [18], the OS is built of several layers that are called software stacks, as can be seen on 2.1*.

*Source:developer.android.com/

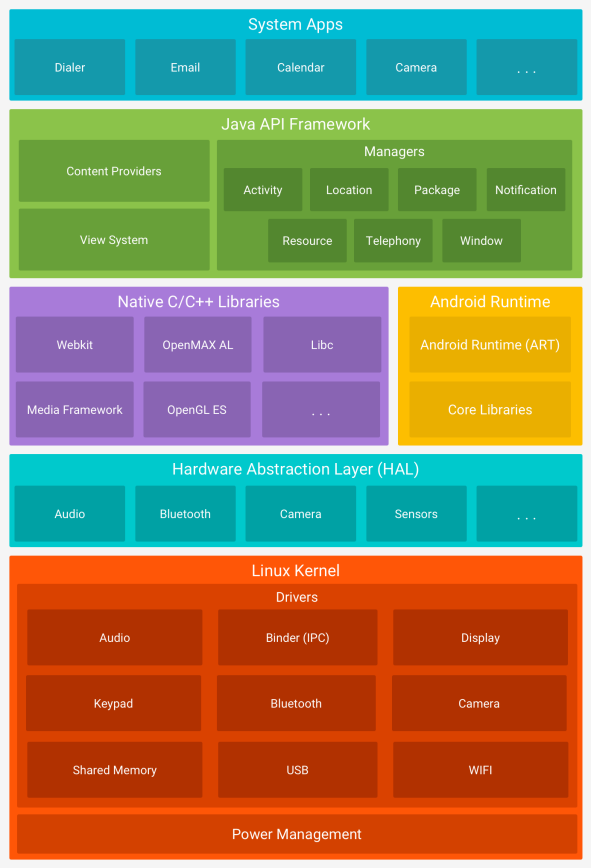


Figure 2.1: Android general architecture.

#### 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 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 [19] (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 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 [18]:

- 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 the scope of the app's UI.
- Activity Manager: That controls the application's 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 comprises 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

A lot of security features are implemented in Android at the application level. Arguably the most important feature is the permission system. According to the official documentation of Android [20], 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 Wi-Fi 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) [21] these permissions are asked to the user either when he installs the application or during runtime, when before it was only on installation.

#### 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 Wi-Fi 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 demands 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 choose between accepting a runtime permission forever or only accepting once the permission. This means that when the application is not used any more, 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 user's 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 are too coarse [22]: 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 [22].
- Too complicated for developers: Leading to overprivileged applications, as developers don't want to select every permission specifically [23].

## 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. [24] separated mobile applications into two different categories:

- *App4Mobile* are apps that are simply translated software for mobile applications, and
- *MobileApps* are apps that were 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. [25] 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. [26] 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. [27] 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. [28] 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 over speed?" 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 application's execution
- Dynamic Context: Context that may evolve with time during the application's execution
- Derived Context (or situation): High level contexts acting as an agglomeration of contexts or specific context values.

## 2.4 Android vulnerability survey

In order to answer to the requirement RQ1 defined in section 1.4, an analysis of current Android vulnerabilities has been made. This survey is designed to be clarifying the current ecosystem of application security and be used to conceptualize the vulnerability model introduced in section 1.5.

### 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 supervise 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.

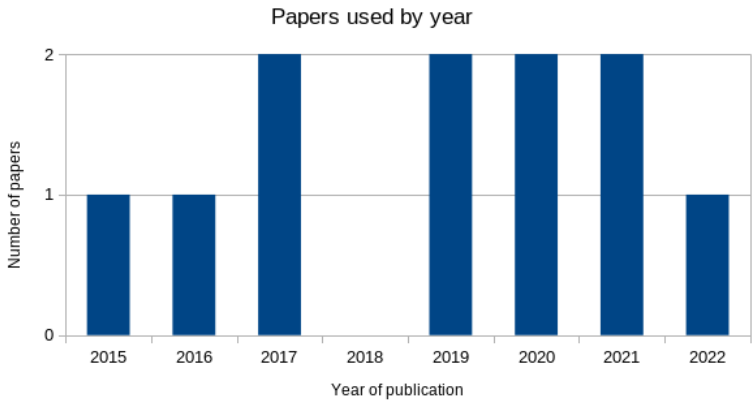


Figure 2.2: Graph of research papers selected by year

We can see in 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 of android vulnerabilities relating to application security can be seen on figure 2.3.

Lastly, we linked each vulnerability to an existing mobile Common Weakness Enumeration (CWE) [29]. The goal of an 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.

The analyzed documents and authors are summarized in the table 2.1.

Table 2.1: Vulnerability papers explored in the survey

| **Author.s** | **Paper title** | **Year** | **Type of paper** |
|---|---|---|---|
| S. Bojjagani et al. | *STAMBA: Security Testing for Android Mobile Banking Apps*[30] | 2015 | Tool showcase |
| V. Jain et al. | *Detection of SQLite Database Vulnerabilities in Android Apps*[31] | 2016 | Partial Survey |
| F. Tabassum et al. | *Vulnerability testing in online shopping android applications*[32] | 2017 | Partial Survey |
| F. H. Shezan et al. | *Vulnerability detection in recent Android apps: An empirical study*[33] | 2017 | Survey |
| S. Almanee et al. | *Too Quiet in the Library: A Study of Native Third-Party Libraries in Android*[34] | 2018 | Partial Survey |
| P. Bhat et al. | *A Survey on Various Threats and Current State of Security in Android Platform*[35] | 2019 | Survey |
| A. Nirumand et al. | *VAnDroid: A framework for vulnerability analysis of Android applications using a model-driven reverse engineering technique*[36] | 2019 | Tool showcase |
| L. Gonzalez-Manzano et al. | *Impact of injection attacks on sensor-based continuous authentication for smartphones*[37] | 2020 | Vulnerability showcase |
| T. Liu et al. | *MadDroid: Characterizing and Detecting Devious Ad Contents for Android Apps*[38] | 2020 | Tool showcase |
| M. A. El-Zawawy et al. | *Vulnerabilities in Android webview objects: Still not the end!*[39] | 2021 | Partial Survey |
| J. Gao et al. | *Understanding the Evolution of Android App Vulnerabilities*[40] | 2021 | Survey |
| P. Sun et al. | *VenomAttack: automated and adaptive activity hijacking in Android*[41] | 2022 | Vulnerability showcase |

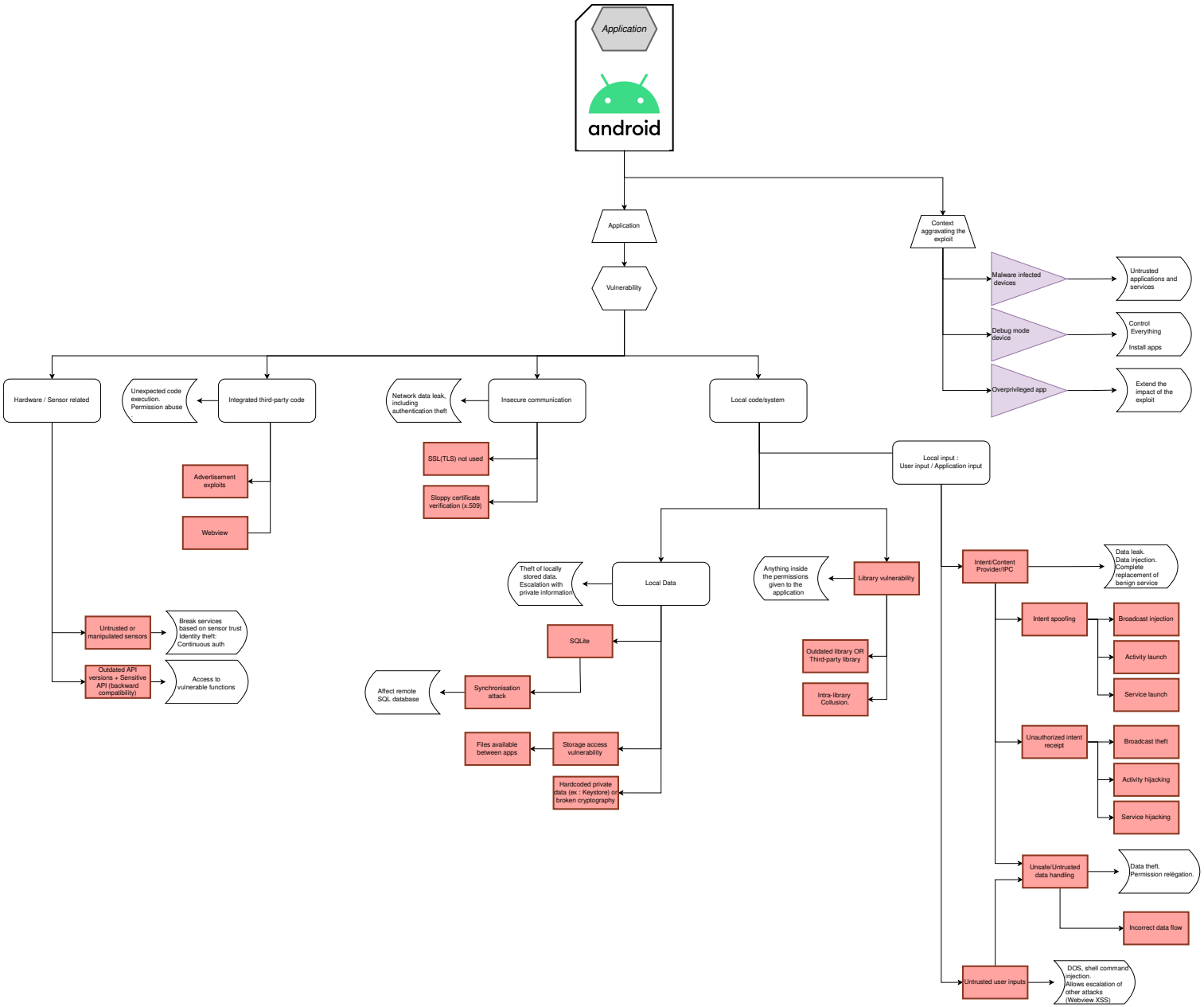


Figure 2.3: Interconnection of Android application vulnerabilities

### 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 [33]. This case is studied in depth by Jonathan Crussel et al. [38].

#### 2.4.2.2 Webview

This vulnerability vector has a lot of literature on it. Webview is a component that allows developers to open webpages inside their app. As explained by Faysal et al. [33], while webview is not vulnerable by themselves, it allows for execution of JavaScript [32] and more generally arbitrary third-party content. While important vulnerabilities have been removed after API level 17 [40]. Recent work shows that a lot of vulnerabilities still exist [39] 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 [42]. 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 [40] [32] [30], basically interception of the data sent through the network. While HTTPS is activated by default since API level 23 [43] 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 pointed out by Jun Gao et al. [40], 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 [32] (application signing and TLS handshakes) but if your code accepts every possible certificate to communicate than the TLS means nothing. Simply enough, if an application developer gives trust to anyone then the certificate has no reason to exist, thus configuring the server with an 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. [34], 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 is 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. [35] 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 [40]. The first one we can find is Structured Query Language (SQL) related. SQL is a Structured Query Language designed by IBM in 1974 [44]. SQL is a DSL designed to create, modify, delete and classify data stored in table, these tables are often high impact for data thefts [32]. 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. [31].

From a more general perspective, unsafe data storing in Android is a common concern. As explained in the document by Shezan et al. [33], 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, 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[40]. 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. [33], this type of vulnerability is one of the most common in Android. In their work on Vandroid [36], 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 [36]. 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. [41] 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 application's UI (usually a login screen) then transforms it into real code. Through a hot patch 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. [45] 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. [37] 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 [33]. According to a study by Jun Gao et al. [40], 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. [45] it can be misused to install overprivileged services or just to simply install root applications.

#### 2.4.7.3 Overprivileged applications

Lastly, giving too many 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

# Methods

## 3.1 Security Approach

The project is based on software testing, a methodology consisting of designing and executing tests, and observing the results. In their book, Myers et al. [46] give the following definition of testing;

> *"Testing is the process of executing a program with the intent of finding errors"*
>
> **Glenford J. Myers**

Testing was already an important step of the development phase in 1979 when the book [46] came out and is still as of writing these lines.

An important advantage of software testing is that it is not expensive to just test the tool, in comparison to using more complex methodologies like formal verification. That said, the low-cost of this methodology comes with the drawback that testing is based on executing cases, and it is mostly impossible to execute every possible case of a software. We are still able to control the amount of tests executed by following specific coverage criteria, these criteria define what a full test is and allow limiting the amount of tests needed to validate the program.

For our project we decided to use White-Box testing, that means that we will be testing the tools with the knowledge of the code of the tool itself and not only by controlling inputs and observing outputs. White-box testing allows for more fine-grained and cheap tests, as we already have a lot of information on the System under Test (SuT).

### 3.1.1 Model-Based Testing

One of the subcategories of testing is called Model Based Testing (MBT). This methodology uses a partial description of the SuT called a model. A lot of methods exist that fall in the category of Model-based testing, but the one that will interest us is finite-state machine MBT.

The advantage is that we are able to simplify the SuT and the testing mechanism by using a stripped-down, ideal set of conditions, thus having a more efficient tool with the drawbacks that we decide to ignore parts of the SuT.

Other methodologies could have been chosen, we could cite:

- Fuzz testing: The SuT is tested with pseudo-random inputs to find bugs or unexpected outputs. This is very close to what we do, but can be very expensive in terms of energy or CPU usage.

- Boundary value analysis: This methodology is based on building the topology of an SuT input/outputs and testing only the boundaries/extreme cases that are usually more error-prone. This methodology works perfectly on simple algorithms, but can be hard to implement on more complex cases.

### 3.1.2 Application security testing

This project has a more specific focus on security testing, this means defending against't a specific adversary (see section 1.3). The security testing equivalent of MBT is MBST. We introduced MBST briefly in section 1.5 as *a methodology to test security requirements efficiently*.

More specifically, we use MBST to carefully model the context of an application and abstracting most of the application in order to gain efficiency and create a process that is able to cover different contextual events.

One of the difficulties in analysing context when trying to detect vulnerabilities is that due to the situation of mobile phones, there exists way more contextual data than classic software, so the cost is very high. Using models specifically for context representation allows us to test the context while keeping the cost low.

### 3.1.3 Security properties

As introduced in section 1.3, we focus our definition of vulnerability and our threat model with the CIA triad. Confidentiality, Integrity and Availability. In

particular, we focus on Confidentiality and integrity of data. Private data is an important part of security, and protecting applications against data theft is crucial.

The main part of security testing of this project is the analysis of data flows, whether it's private or public data. This analysis was important as it allows to properly detect vulnerabilities and not "features". The idea is that we define a vulnerable feature when we find private flows of data on public channels.

It is important to note that we are not specifically studying data flows, but merely using them to have a more precise model of the SuT.

## 3.2 Base work

As introduced in section 1.4, we will be expanding on the work of Adwan and build upon his theory with our contribution.

During his master's thesis [11], Adwan designed a methodology to generate tests that take into account the context of an application. The general idea he brought was to define two 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.

Adwan goes on by defining algorithms for different coverage criteria 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 were 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.

## 3.3 Design objectives

In terms of design, we have a few constraints:

- Base our work on Adwan's internship work.

- Cover most vulnerability types with a focus on vulnerabilities linked with context.
- A maximum of 5 months are available for this project

The goals we aimed during the design of the architecture of the solution 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** The project's time is limited to 5 months 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 alter the code.

## 3.4 Evaluation methodology

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

During the research process, the GHERA [2] repository has been found. This repository gives 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.

Table 3.1: GHERA and the represented vulnerabilities

| Vulnerability Type | Crypto | ICC | Networking | NonAPI | Permission | Storage | System | Web |
|---|---|---|---|---|---|---|---|---|
| Number of examples | 5 | 17 | 8 | 2 | 2 | 7 | 7 | 12 |

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 its activities. This vulnerability shows that we can call a specific action to query the "database" it contains from outside 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.

In terms of evaluation, we will be using GHERA to measure the detectability of every vulnerability:

*Is the vulnerability X possible to detect using the tool Y?*

As explained, due to the limitations of GHERA we will try to use vulnerabilities that allow for data leakage. If needed, we will add the data flow through the presented vulnerability.

**Limitations** Our project aims at detecting vulnerabilities in specific contexts as much as giving a Model-Based security testing tool. Due to time limitations and lack of existing benchmarks, we kept GHERA. Due to that, we could not specifically measure the efficiency in terms of context-specific vulnerability detection or how well our tool compares to non context-aware security testing tools.

# 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 give a conclusion to our vulnerability study. Then, section 4.2 will introduce the general architecture of the process we created. After that, in section 4.3, will be reviewed the first part of the architecture on the subject of behavioural test generation. Lastly, we will present in section 4.4 the exploit generator, its design and philosophy, and its architecture.

## 4.1 Vulnerability focus

In section 2.4 we have described the state of the art of Android application vulnerabilities. The table 4.1 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 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 (Wi-Fi, 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 4.1: 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 | Sometimes | 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.

## 4.2 Global Architecture

During the master's thesis, we designed and created a two part process. First, a context-aware code coverage generator tool called ConTest. Then an abstract vulnerability generator tool called 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 [47]. This tool allows building a call graph 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 and I, to generate a set of tests by combining it with a generic definition of Android context. This combination allows generating 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 preventing vulnerabilities during the development process by telling the developer how we can exploit his specific code and allowing him to fix his code.

## 4.3 Test Generation

As explained in section 3.2, we based our work on a work by Adwan called "Context-dependent Model-based Testing of Mobile Apps" [11] (or ConTest). In order to be clear about the participation of Adwan in this base, each part will detail the contributions.

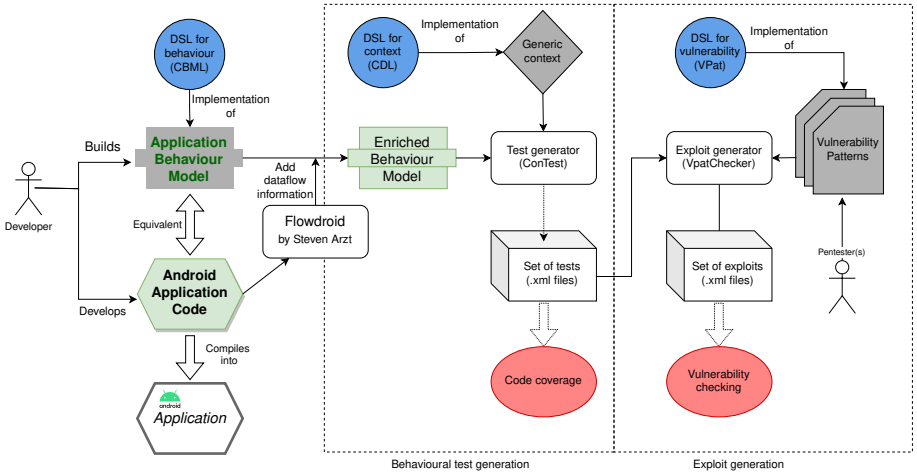


Figure 4.1: Global architecture of test generation in ConTest

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.

### 4.3.1 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 [48], a part of the Eclipse Modelling 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).

A popular framework for creating DSLs is Xtext which is part of the Eclipse Modelling 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 colouring and error highlighting

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

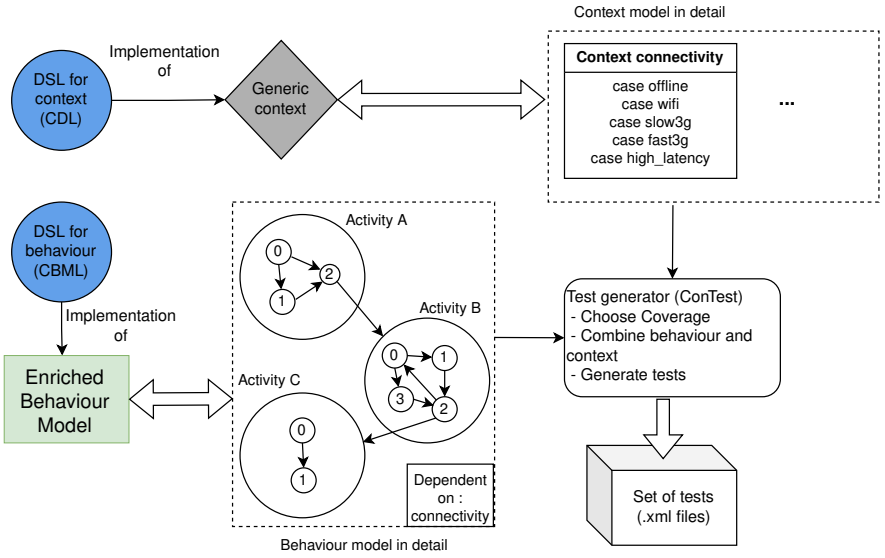


Figure 4.2: Detailed architecture of test generation in ConTest

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, Wi-Fi, slow3g or more.

### 4.3.2 ConTest - Implementation

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

#### 4.3.2.1 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 [49] 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 in the Android developer's webguide [50], each component is an entry point through which the system or a user can enter your app. The developers are 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 state machine: 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 listing 4.1.

```
model TranslationApp {

    contexts {
        INTERNET_CONNECTIVITY
    }
```

Listing 4.1: Example of dynamic context model

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 particularly 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.

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

Listing 4.2: Example of static context model

In the listing 4.2, 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
    }
```

Listing 4.3: Example of situation model

In the listing 4.3, 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 state machines for each component. As we can see in figure 4.4, each state machine will announce its permission type, these values are usually written by the developer in its 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 4.4: 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
}
```

Listing 4.4: 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 listing 4.4, *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 the same FSM.

```
statemachine SEND_MESSAGE_ACTIVITY_SM exported {
    state SEND_MESSAGE awareof INTERNET_CONNECTIVITY {
        transition on SEND_MESSAGE_CLICKED -> SHOW_ANSWER
    }

    state SHOW_ANSWER
}
```

Listing 4.5: Statemachine containing the context aware state Send_message and the atomic state Show_Answer

As seen in listing 4.5, 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.

```
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
    }
}
```

Listing 4.6: Adaptation of the state SEND_MESSAGE in the situation INTERNET_DISCONNECTED

In this example 4.6, 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.

#### 4.3.2.2 Context DSL

The second component of ConTest is the Context Definition Language (CDL) DSL. This language allows defining 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, 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 [11], 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 (Eg: Location is given by the GPS sensor or internet)
- Properties: The set of value types that define a context (Eg: 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]
}
```

Listing 4.7: 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 listing 4.8.

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

Listing 4.8: 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 listing 4.9.

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

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

Listing 4.9: Situations as defined by CDL

#### 4.3.2.3 Model enrichment

During the study on Android vulnerabilities seen on figure 2.3 and later on the table 4.1, 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 [47]. FlowDroid is a very powerful tool created by Steven Arzt et al. that allows the creation of call graphs 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 modifications. 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 kinds 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 black box 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 state machine Send_message_activity_SM (listing 4.5 this will update the into listing 4.10.

```
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
}
```

Listing 4.10: Statemachine containing the context aware state Send_message and the atomic state Show_Answer enriched with the private data Internet_Status

On another part of the program we find that a function *log.d* outputs that same *internet status,* this then updates the model from listing 4.6 to listing 4.11.

```
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
                ↪ )
        }
    }
```

Listing 4.11: Adaptation of the state SEND_MESSAGE in the situation INTERNET_DISCONNECTED enriched with the public sink log.d

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 it is in a specific context only (Internet_disconnected).

Sinks can have multiple parameters, in listing 4.11 log.d only has a single parameter, but it could have more.

It is important to note that this section 4.3.2.3 is not used by the test generation process (information is kept but not used) but only by the vulnerability detection process of section 4.4.

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

A coverage criterion is defined by the context of the SuT and in the case of MBST:

- All-States: All states have been executed at 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. [51], 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 aimed 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 criteria 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 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 listing 4.4, we are able to generate the following paths (table 4.2).

Table 4.2: 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 (listing 4.10) with the adaptation of listing 4.11 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 4.7, 4.9 and 4.8, it allows us to generate the following paths 4.3.

Table 4.3: 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 possibility 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 4.2 and 4.3, taking Abstract_SM as the starting node, we get the table 4.4.

In the table 4.4, we only see a limited version, as the final number of paths is 40. This is due to the original subpaths of abstract sm 4.2 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.

Table 4.4: 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 |

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.

## 4.4 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 4.4.1 will be introduced the architecture of the solution. Then, section 4.4.2 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*.

### 4.4.1 VPAT - Architecture

The definition of this contribution has been an extension of our design goals of section 3.3. 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 4.5, we see represented the detailed process of VPatChecker, that uses abstract tests in XML form and patterns to detect vulnerabilities and generate exploits.

### 4.4.2 VPAT - Implementation

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

#### 4.4.2.1 Vulnerability patterns

As explained in section 4.4.1, 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 its name and description. We are then able to separate it's characteristics into two main parts:

- Context: Defines the specific state the application/host should have to be vulnerable.

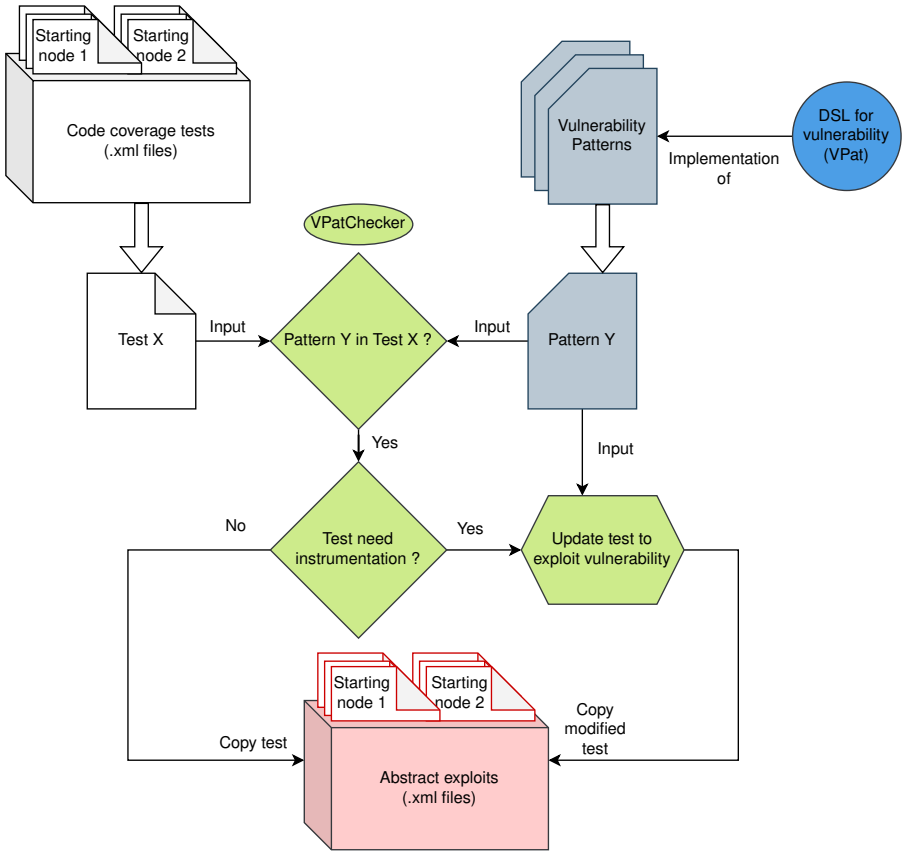


Figure 4.5: Detailed high level architecture of exploit generation (VPatChecker)

- 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 Pie), 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 4.12.

```
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 *
    }
}
```

Listing 4.12: 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 4.13, 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 data"

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

Listing 4.13: 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.

#### 4.4.2.2 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. As shown in figure 4.6

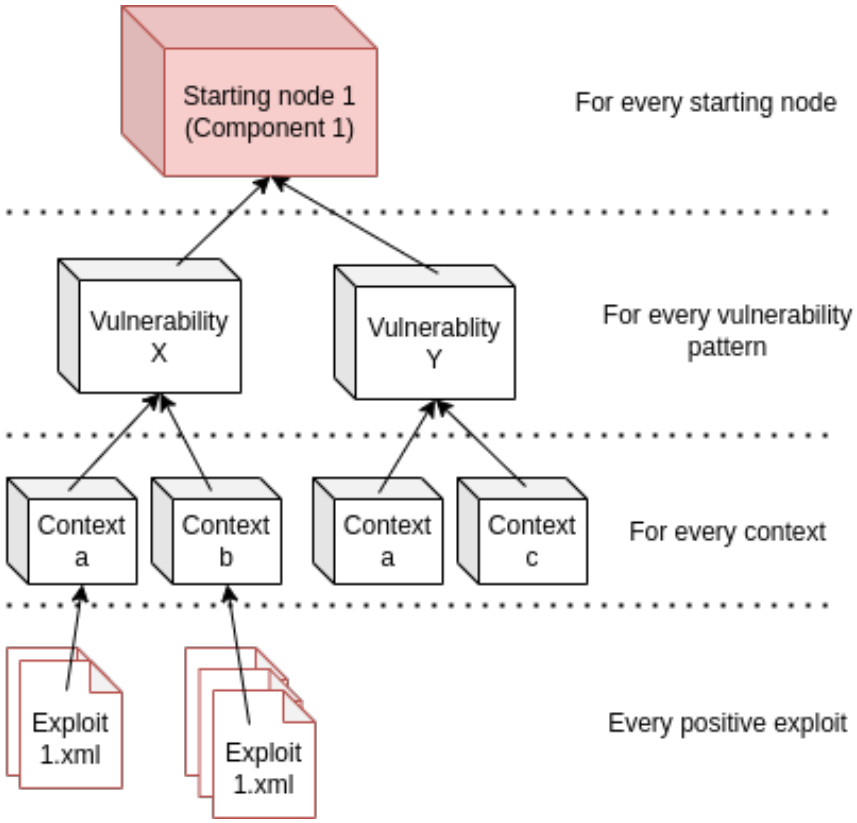


Figure 4.6: Reporting created by VPatChecker

#### 4.4.2.3 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 "vulnerableFunc EXPLOITME" {
    description "The function vulnerableFunction leaks data when the second
        ↪ parameter is EXPLITME in android version 31"

    context {
        apiversion "31"
    }

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

Listing 4.14: Example of pattern that needs a specific value in its second parameter

In the pattern seen in listing 4.14, 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 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 4.7, we can see in the state *DISPLAY_WARNING* that the dataflow *vulnerableFunc* takes as parameters:

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

When building the exploit for the specific pattern 4.14, 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 shown in figure 4.8.

```
<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"/>
</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 4.7: Excerpt from a test generated by ConTest, input value is not set

```
<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 4.8: Excerpt from an exploit generated by VPatChecker, input value is set to the value EXPLOITME

# Chapter 5

# Results and Analysis

## 5.1 GHERA as a test database

As explained in section 3.4, to test the project, an interesting dataset is the GHERA [2] dataset. GHERA is a paper and open-source project with the idea to give researchers a set of Android applications carefully designed to demonstrate a vulnerability.

In section 6.2 will be introduced that the tool is not yet automated, making any test a long process (that for now includes implementing the vulnerability pattern for the specific vulnerability), this limited our testing capabilities.

In this chapter, we will show the capabilities of *ConTest* and *VPatChecker* through a series of tests, then a discussion on the theoretical range of vulnerabilities we are able to detect and what parts should be enhanced to broaden this range.

### Limitations of GHERA

As introduced in section 3.4, GHERA has three main limitations:

- It has not been updated since 2019, which means that the test set does not include recent vulnerabilities.
- The tests are made for demonstration. With only a single demonstration per vulnerability. Meaning, we cannot test against complex cases with only GHERA.
- The vulnerabilities depicted are not always really considered as a vulnerability, this is due to our definition of adversary in sections 1.3 and 2.1.

### 5.1.1 Detail of the analysis process

In this section we will show a full detailed usage of the contribution to vulnerable code of GHERA.

As example, we will be taking showing that a dataflow failure (high to low) is detected and a test leading to the specific situation will be generated. The base example will be on *WeakPermission-UnauthorizedAccess-Lean* [52] of the permission category.

#### 5.1.1.1 Base code

The main idea of this vulnerability is to show off that an exported content provider that demands a *normal* permission can be called by anyone, as a normal permission is accepted by default. In the test case, the vulnerable app has a query that can be called by another app, we will modify this query so a real vulnerability is exploited.

In order to have an interesting test, a dataflow from high to low has been added when the query operation of the content provider is called. The added dataflow is a simple log of a check to see if internet is available, which gives the information on connectivity to the attacker.

```
@Override
public Cursor query(Uri uri, String[] projection, String selection,
                    String[] selectionArgs, String sortOrder) {

try {
    ConnectivityManager cm = (ConnectivityManager) this.getContext().getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo nInfo = cm.getActiveNetworkInfo();
    boolean connected = nInfo != null && nInfo.isAvailable() && nInfo.isConnected();
    String connectedString = "Connectivity : Phone is " + new Boolean(connected).toString();
    Log.d(TAG, connectedString);
}
catch (Exception e) {
    Log.e("Connectivity Exception", e.getMessage());
}
    return null;
}
```

Figure 5.1: Added part to the GHERA code example

The figure 5.1 checks if the network is available through a check whether the NetworkInfo created is null or not and prints the results (true or false) with log.d. For an attacker, reading the logs is simple and often forgotten by developer, which allows giving a pretty common vulnerability as example.

#### 5.1.1.2 Model creation

The first part of the process is to convert the chosen code to a CDML model. This example being simplistic it is easy to create:

- Two different statemachine: One for MainActivity.java the other one for MyContentProvider.java
- No dynamic context condition, and a simple static context API version from 22 to API version 27 and default network
- Mostly empty states (States that have no transition)

This gives us the following model 5.1.

```
model WeakPermission {

    static contexts {
        minSdk = "22",
        maxSdk = "",
        targetSdk = "27"
    }

    statemachine MainActivity_SM {
        state onCreate
    }

    statemachine MyContentProvider_SM exported normal "edu.ksu.cs.benign.MYCP_ACCESS_PERM"{
        ↪
        state receiveIntent {
            transition on DELETE_INTENT -> delete
            transition on GETTYPE_INTENT -> getType
            transition on INSERT_INTENT -> insert
            transition on QUERY_INTENT -> query
            transition on UPDATE_INTENT -> update
            transition on ONCREATE_INTENT -> onCreate
        }

        state delete

        state getType

        state insert

        state query

        state update

        state onCreate
    }
}
```

Listing 5.1: CDML model of WeakPermission-UnauthorizedAccess-Lean from GHERA

The content provider *MyContentProvider* is exported with the normal permission *"edu.ksu.cs.benign.MYCP_ACCESS_PERM"* which has been added to the model.

#### 5.1.1.3 Model Enrichment

For the model enrichment after using the given script, we get the graph 5.2.

As we can see, a flow from a *NetworkInfo* source to a *log.d* sink was detected. We can then extend the model with the red and blue information of the graph 5.2.

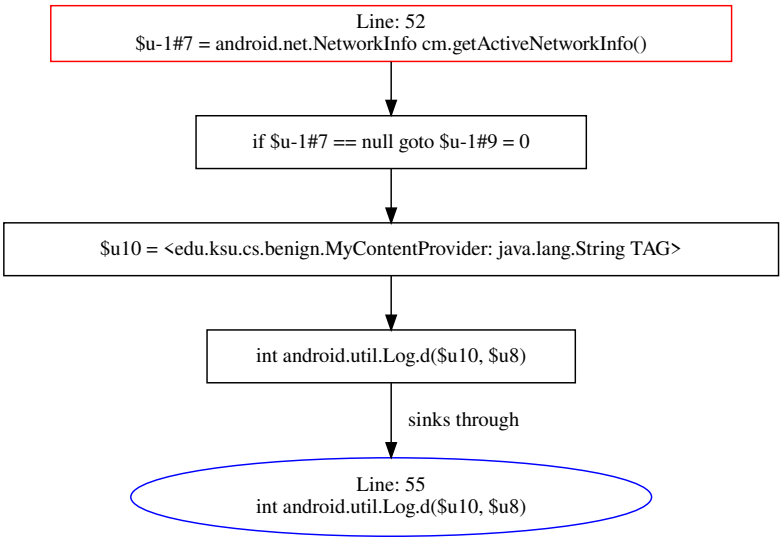


Figure 5.2: Flowgraph given by the enrichment script, from FlowDroid's output

This gives the following enriched model 5.2

```
model WeakPermission {

    static contexts {
        minSdk = "22",
        maxSdk = "",
        targetSdk = "27"
    }

    statemachine MainActivity_SM {
        state onCreate
    }

    statemachine MyContentProvider_SM exported normal "edu.ksu.cs.benign.MYCP_ACCESS_PERM"{
        ↪
        state receiveIntent {
            transition on DELETE_INTENT -> delete
            transition on GETTYPE_INTENT -> getType
            transition on INSERT_INTENT -> insert
            transition on QUERY_INTENT -> query
            transition on UPDATE_INTENT -> update
            transition on ONCREATE_INTENT -> onCreate
        }

        state delete
        state getType
        state insert
        state query {
            dataflows {
                source NetworkInfo,
                sink "log.d" ( source MyContentProvider_SM.query.NetworkInfo )
            }
        }
        state update
        state onCreate
    }
}
```

Listing 5.2: Enriched CDML model of WeakPermission-UnauthorizedAccess-Lean from GHERA

As we can see in figure 5.2, the model's state *query* has two dataflows: A sink and a source.

#### 5.1.1.4 Code coverage

In order to get a code coverage of this model, we use the *ConTest* tool which is able to generate us:

- 6 Tests for MainActivity: Only one final state (onCreate) in 6 different static contexts (1 per Android API versions)
- 36 Tests for MyContentProvider: 6 Final states in 6 different static contexts

Of course, this test is made simple for presentation. A more complex test was run on the introduction to *ConTest* and *VPatChecker* in chapter 4 as seen in the tests of annex C.

These generated tests are abstract and can be translated by the developer, or we can analyse them to check for vulnerabilities.

#### 5.1.1.5 Vulnerability detection

For the vulnerability detection, we tested with 3 different vulnerability patterns. One of those is seen in listing 4.13 a simple flow from a source to the sink log.d.

By checking every generated test, we get the report shown in figure 5.3

In this report, we see that every test was checked against every pattern and gives us the following:

- For MainActivity_SM: Every pattern was negative
- For MyContentProvider_SM: The pattern 4.13 was positive for 6 out of 36 tests

Note that the pattern *log.t* is only checked on tests with the static context API version 26.

Our tool shows us that the 6 different exploits exist when an exterior intent sent to the MyContentProvider_SM. This exploit exists when we simply call query, and in every static context.

A more complex full test can be found in annex E.

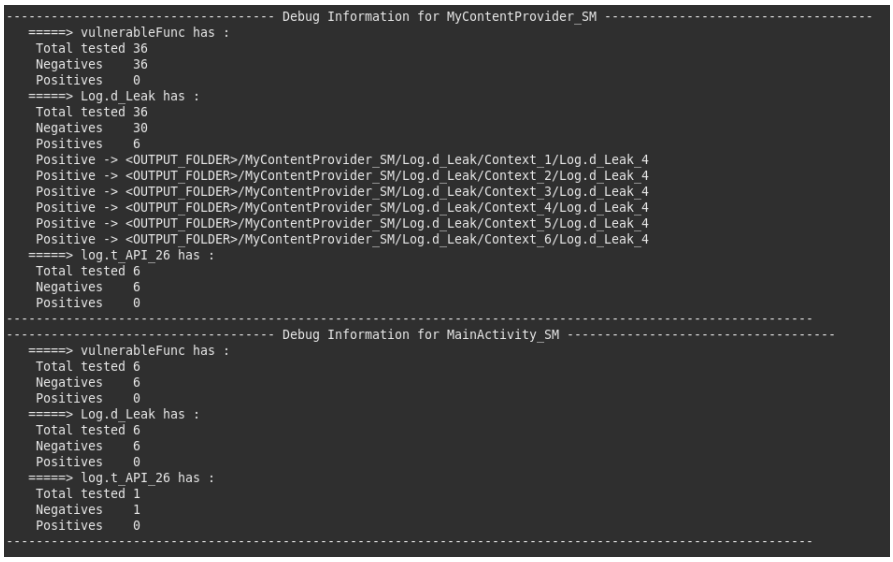


Figure 5.3: Report generated by VPatChecker for WeakPermission-UnauthorizedAccess-Lean from GHERA

## 5.2 Detectability analysis on GHERA's database

In this section, we will detail the analysis of this project on the test bench GHERA.

### Permission subgroup

Table 5.1: Result of analysis for Permission-type vulnerabilities of GHERA

| **Vulnerability Subtype** | **Detectable** | **Explanation** *(Optional)* |
|---|---|---|
| WeakPermission-UnauthorizedAccess-Lean | Yes | |
| UnnecessaryPerms-PrivEscalation-Lean | No | An application can have libraries that contain unused functions that could be called through the application by a malicious application |

## NonAPI subgroup

Table 5.2: Result of analysis for NonAPI-type vulnerabilities of GHERA

| **Vulnerability Subtype** | **Detectable** | **Explanation** *(Optional)* |
|---|---|---|
| NonFunctional-OutdatedLibrary-Lean | Yes/No | It is possible to detect this specific instance of the vulnerability but as we do not model API versions used we are not able to correctly model the vulnerability of this type |
| MergeManifest-UnintendedBehavior-Lean | No | Technically the same vulnerability as **UnnecessaryPerms-PrivEscalation-Lean** but we are not able to model API versions used to safely model vulnerable libraries |

## Crypto subgroup

Table 5.3: Result of analysis for Crypto-type vulnerabilities of GHERA

| **Vulnerability Subtype** | **Detectable** | **Explanation** *(Optional)* |
|---|---|---|
| BlockCipher-ECB-InformationExposure-Lean | Yes | |
| BlockCipher-NonandomIV-InformationExposure-Lean | No | It is possible to model the vulnerability, but the PoC checker does not allow for multiple functions to be detected |
| PBE-ConstantSalt-InformationExposure-Lean | Yes | |
| ConstantKey-ForgeryAttack-Lean | Yes | |
| ExposedCredentials-InformationExposure-Lean | Yes | |

## Storage subgroup

Table 5.4: Result of analysis for Storage-type vulnerabilities of GHERA

| **Vulnerability Subtype** | **Detectable** | **Explanation** *(Optional)* |
|---|---|---|
| ExternalStorage-DataInjection-Lean | Yes | |
| ExternalStorage-InformationLeak-Lean | Yes | |
| InternalStorage-DirectoryTraversal-Lean | Yes/No | We are able to detect that there may be a vulnerability as we have the information that an input was sent to a specific function, but we do not handle modification of the data |
| InternalToExternalStorage-InformationLeak-Lean | Yes | |
| SQLite-execSQL-Lean | Yes/No | We are able to detect that there may be a vulnerability as we have the information that an input was sent to a specific function, but we do not handle modification of the data |
| SQLlite-RawQuery-SQLInjection-Lean | Yes | |
| SQLlite-SQLInjection-Lean | Yes | |

## System subgroup

Table 5.5: Result of analysis for System-type vulnerabilities of GHERA

| **Vulnerability Subtype** | **Detectable** | **Explanation** *(Optional)* |
|---|---|---|
| CheckCallingOrSelfPermission-PrivilegeEscalation-Lean | Yes | |
| CheckPermission-PrivilegeEscalation-Lean | Yes | |
| ClipboardUse-InformationExposure-Lean | Yes | |
| DynamicCodeLoading-CodeInjection-Lean | Yes | |
| EnforceCallingOrSelfPermission-PrivilegeEscalation-Lean | Yes | |
| EnforcePermission-PrivilegeEscalation-Lean | Yes | |
| UniqueIDs-IdentityLeak-Lean | Yes | |

## Network subgroup

Table 5.6: Result of analysis for Networking-type vulnerabilities of GHERA

| **Vulnerability Subtype** | **Detectable** | **Explanation** *(Optional)* |
|---|---|---|
| CheckValidity-InformationExposure-Lean | No | The vulnerability model do not allow modelling non-called functions |
| IncorrectHostNameVerification-MITM-Lean | No | Same as **CheckValidity-InformationExposure-Lean** |
| InsecureSSLSocket-MITM-Lean | Yes | |
| InsecureSSLSocketFactory-MITM-Lean | Yes | |
| InvalidCertificateAuthority-MITM-Lean | No | Same as **CheckValidity-InformationExposure-Lean** |
| OpenSocket-InformationLeak-Lean | No | Too complex |
| UnEncryptedSocketComm-MITM-Lean | No | Same as **CheckValidity-InformationExposure-Lean** |
| UnpinnedCertificates-MITM-Lean | No | Same as **CheckValidity-InformationExposure-Lean** |

## Web subgroup

The result of the web subgroup can be summarized as it being out of scope of our project. The main reason being that we are unable to model JavaScript code in our vulnerability model.

Vulnerabilities tied to JavaScript inside webview will never be detected. Of course, vulnerability "**JavaScriptExecution-CodeInjection-Lean**" tied to accepting the execution of JavaScript is detectable.

## ICC subgroup

Table 5.7: Result of analysis for ICC-type vulnerabilities of GHERA

| **Vulnerability Subtype** | **Detectable** | **Explanation** *(Optional)* |
|---|---|---|
| DynamicRegBroadcastReceiver-UnrestrictedAccess-Lean | No | Dynamically registered broadcasts components are not tracked by the program model |
| EmptyPendingIntent-PrivEscalation-Lean | Yes | |
| FragmentInjection-PrivEscalation-Lean | Yes | |
| HighPriority-ActivityHijack-Lean | No | We do not track **priority** in the models |
| ImplicitPendingIntent-IntentHijack-Lean | No | The vulnerability model do not allow modelling non-called functions |
| InadequatePathPermission-InformationExposure-Lean | Yes/No | The model does not track **pathPrefix** thus we are able to detect the vulnerability |
| IncorrectImplicitIntent-UnauthorizedAccess-Lean | Yes/No | The model does not track **Intent Filters** thus we are able to detect the vulnerability |
| NoValidityCheckOn BroadcastMessage-UnintendedInvocation-Lean | Yes/No | Same as **IncorrectImplicitIntent-UnauthorizedAccess-Lean** |
| OrderedBroadcast-DataInjection-Lean | No | Too complex |
| StickyBroadcast-DataInjection-Lean | No | Too complex |
| TaskAffinity-ActivityHijack-Lean | No | We do not handle taskAffinity |
| TaskAffinity-LauncherActivity-PhishingAttack-Lean | No | We do not handle taskAffinity |
| TaskAffinity-PhishingAttack-Lean | No | We do not handle taskAffinity |
| TaskAffinityAndReparenting-PhishingAndDoSAttack-Lean | No | We do not handle taskAffinity |
| UnhandledException-DOS-Lean | No | Not a confidentiality vulnerability |
| UnprotectedBroadcastRecv-PrivEscalation-(Fat\|Lean) | Yes | |
| WeakChecksOnDynamicInvoation-InformationExposure-Lean | Yes | |

## 5.3 Summarized analysis results

Here we can see the complete result of the analysis found in tables 5.1, 5.2, 5.3, 5.4, 5.5 and 5.7 summarised in a single table 5.8. The results which showed a vulnerability detected but missing the real vulnerability were counted as "**Not detected**" in the resulting table.

Table 5.8: Summarised result of vulnerabilities detectable by VPatChecker on the GHERA benchmark

| **Vulnerability type** | **Number of total vulnerabilities** | **Number of detectable vulnerabilities** | **Percentage of detectability** |
|---|---|---|---|
| Permission | 2 | 1 | 50% |
| NonApi | 2 | 0 | 0% |
| Crypto | 5 | 4 | 80% |
| ICC | 17 | 4 | 24% |
| Networking | 8 | 2 | 25% |
| Storage | 7 | 5 | 71% |
| System | 7 | 7 | 100% |
| Web | 12 | Out of scope | Out of scope |
| Total | 60 | 23 | 38% |

## 5.4 Range of tests outside GHERA

In this section we discuss the capabilities and limitations of the VPat patterns in order to measure this we will compare with our specifications, more specifically the vulnerabilities we want to tackle as shown by table 4.1.

#### 5.4.0.1 Storage Access Vulnerability

As explained in earlier sections, we define this vulnerability by the misuse of certain permissions concerning data written. With VPat we are able to detect when a wrong setting is sent to a specific function, thus we are able to detect the "vulnerability" that consists in making a specific file world_readable (although that can be a choice).

A lot of different functions exist for this, although most are deprecated as this is not the best way to share a file between apps. Figure 5.3 shows an

example of pattern for the function "file.setReadable".

```
Vulnerability "File.setReadable" {
    description "File.setReadable(true, false); - Local storage made world
        ↪ readable is vulnerable"

    function {
        main Sink "File.setReadable" {
            parameter {
                static "true",
                static "false"
            }
        }
    }
}
```

Listing 5.3: Example of pattern detecting the use of File.setReadable which would allow anyone to read the specified file

#### 5.4.0.2 Intent spoofing

This specific vulnerability is related to bad configurations in the manifest file. More specifically, it is about leaving a component available for others to start or use when the developer did not intend to.

For detection, it is easy to detect, with VPatChecker, that a component is callable by an outsider. The problem we have is the same we had with GHERA [2], an application may have its components open for others for good reasons.

Thus, we could consider that this is not a vulnerability per se, even though it is considered this way in the literature.

In its design, VPatChecker (and ConTest) find every path from every starting node in our project, and we define a starting node as the first line of code that we are able to execute. In this case, open components are defined as starting nodes if they did not ask for permissions that are signatures (See section 2.2.2).

#### 5.4.0.3 Unauthorized intent receipt

This vulnerability can be considered as the opposite of intent spoofing, this time we try to prevent other applications from being able to read the intents we sent.

In the same way, in section 5.4.0.2 something unauthorized is to be defined by a developer. We can still imagine a pattern that tells the developer that a broadcast has been sent, for example by simply detecting the use of sendBroadcast(...).

```
Vulnerability "Broadcaster intent with private data" {
    description "sendBroadcast is dangerous, it can leak data"

    function {
        main Sink "sendBroadcast" {
            parameter {
                anyPrivate
            }
        },
        Source anyPrivate *
    }
}
```

Listing 5.4: Example of pattern detecting the use of sendBroadcast which would allow anyone to receive an intent

In figure 5.4, a pattern detects the usage of sendBroadcast with a payload containing private data, allowing the user to know data is leaked.

#### 5.4.0.4 Untrusted user inputs

For this vulnerability, the usage of a FlowDroid with custom input allows us to detect user inputs as being data that we should track, this way we are able to check if a user input is put inside a specific function. This specific function could be set as being vulnerable in a pattern when a specific parameter is set, this combination of features allow creating an exploit using specific user inputs.

An example of this would be the vulnerability linked to the API 28 update, that removed the possibility to use HTTP by default on Android. This means that we can create a pattern that checks if an application can be run in version 27 with default network configuration, listing 4.12 is an example of this.

# Chapter 6

# Conclusions and Future work

In this chapter we summarize the conclusion of the project, we then give possible future work tips to continue working on it. Lastly, we give a reflection on the ethical issues of the contribution.

## 6.1 Conclusions

Generally speaking, the project started out as an exploratory idea to complete and expand a previous work on application testing into a security testing tool. Through a first analysis on current Android vulnerabilities, it has been established that vulnerabilities cannot all be detected through the application's code and information on context is needed to tackle more specific vulnerabilities.

With the theory of Adwan [11] has been created a tool called *ConTest* (Context and Test) a process that, through a model of the application and the contexts, allows a developer to generate abstract tests by following a selected coverage criteria.

An extension of this work has been created, that instead of analysing an application, directly scans coverage tests for matches against patterns of vulnerabilities. This process has been named *VPatChecker* and is available on GitHub[1] (For Vulnerability Pattern Checker). The end goal of this contribution is to separate a developer's work from the security testers, this is allowed by designing the tool so that the patterns are separated from the application code.

In order to create patterns for vulnerabilities, a language has been built. This language aims at being simple to read, but also to parse. In a broader point of view, every part of the contribution has been designed to allow for

automatising in the future.

## 6.2 Limitations

The tools created, *ConTest* and *VPatChecker*, are not mature enough to join a proper development process. While the tools allow to safely detect a wide range of vulnerabilities two parts are missing:

- A database of vulnerability patterns: The project aims at using open-source processes to build up its vulnerability database, but no platform has been set up, and it will be left for a future contribution.
- An automation process: The project currently asks the user to rewrite the abstract tests into (or at least to read them), this is not something that a real developer may use (at least not an average developer).
- A complex matching algorithm: This should be created to detect complex cases that may be close to obfuscation. This is important for Boolean operations for example that may be more complicated than simply a *true* or *false* in the code.

## 6.3 Future work

As explained in 6.2, the project has a few key points that could prove to be interesting to examine in future works.

### 6.3.1 Automation process

This is the case for an automation process that converts the abstract tests generated by both *ConTest* and *VPatChecker* into real code. This would allow proving that vulnerabilities are detected and that no false positive was found. It would also allow for the project to reach usable maturity. Indirectly, this would allow a more specific analysis of the tool, giving more insights on its efficiency. A proposal was made by Adwan to translate our generated tests into Cucumber [53] tests, Cucumber uses the language gherkin to allow for human-readable tests to exist in the development chain. While it was first designed to allow non-developers to create tests, it is nowadays often used to automate test generation.

### 6.3.2 Vulnerability patterns

For the vulnerability patterns, a few key points can be enhanced, mainly taking into account regular expressions and Boolean operands. This would mainly allow having only one pattern for a multiplicity of different vulnerabilities working on the same principle. Extending the range of vulnerabilities that a pattern can take may also bring the need to alter its design in some parts. One of the ideas that was not implemented during the project was to bring the vulnerability fix into the vulnerability pattern. In simple words, have each pattern also define how you solve the vulnerability. This process could allow having a complete project directly solving a detected vulnerability, helping non-security oriented developers to quickly solve vulnerabilities.

#### Open-sourced patterns

A last future work idea concerns the open-sourcing of vulnerability patterns. If it is desired to allow anyone to add a vulnerability pattern (with moderation), there arises a need to classify the vulnerability other than by name. In the same fashion as during section 2.4, it could be interesting to classify every pattern by CWE and Common Vulnerabilities and Exposures (CVE).

## 6.4 Reflections

The main goal of this project is to give more tools for developers to defend their code from attacks or from attackers exploiting their code to steal the user's data. During the process, we generate abstract tests and exploits. While this may allow attackers to use this tool to generate exploits on other people's code, it is currently only possible if they already have the code. In a more mature version of the project, this could be a reason to stay with White-Box and not Black-Box. In its current form, the contribution does not allow an attacker to do any notable attack on an unknown code, we also only used fake applications (not real, distributed and open-source code) so we do not provide exploits in this master's thesis.

# References

[1] I. Baheux, "Vpatchecker," https://github.com/Myshtea/VPatChecker, 2023. [Pages i, iii, v, 67, 93, and 94.]

[2] 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 [Pages v, 30, 53, 65, and 94.]

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

[4] "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/ [Page 1.]

[5] V. team, "State of Software Security Volume 12," *Veracode*, 2022. [Online]. Available: https://www.veracode.com/state-of-software-security-report [Page 1.]

[6] 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. [Page 2.]

[7] Zameer, "Understanding Mobile Context Awareness," Jan. 2021. [Online]. Available: https://medium.com/ascentic-technology/understanding-mobile-context-awareness-887a9d380d21 [Page 2.]

[8] 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 [Page 2.]

[9] "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/ [Pages 2 and 8.]

[10] "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 [Page 3.]

[11] A. Adwan, "Context-dependent Model-based Testing of Mobile Apps," Master's thesis, University Grenoble Alpes, France, 2021. [Pages 4, 29, 34, 40, and 67.]

[12] 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 [Page 4.]

[13] 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. [Page 7.]

[14] M. R. Clarkson and F. B. Schneider, "Hyperproperties," in *2008 21st IEEE Computer Security Foundations Symposium*, Jun. 2008, pp. 51–65, iSSN: 2377-5459. [Page 7.]

[15] 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. [Page 8.]

[16] 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]. [Page 8.]

[17] "Google Bug Hunters." [Online]. Available: https://bughunters.google.com/ [Page 8.]

[18] "Platform Architecture." [Online]. Available: https://developer.android.com/guide/platform [Pages 8 and 10.]

[19] 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 [Page 10.]

[20] "Permissions on Android." [Online]. Available: https://developer.android.com/guide/topics/permissions/overview [Page 11.]

[21] "Request app permissions." [Online]. Available: https://developer.android.com/training/permissions/requesting [Page 11.]

[22] 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. [Pages 12 and 13.]

[23] 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. [Page 13.]

[24] 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. [Page 13.]

[25] 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. [Page 13.]

[26] 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 [Page 13.]

[27] 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. [Page 13.]

[28] 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 [Page 14.]

[29] "CWE - CWE-919: Weaknesses in Mobile Applications (4.8)." [Online]. Available: https://cwe.mitre.org/data/definitions/919.html [Page 15.]

[30] S. Bojjagani and V. Sastry, "STAMBA: Security Testing for Android Mobile Banking Apps," vol. 425, Dec. 2015. [Pages 16 and 19.]

[31] V. Jain, M. Gaur, V. Laxmi, and M. Mosbah, "Detection of SQLite Database Vulnerabilities in Android Apps," Dec. 2016. [Pages 16 and 21.]

[32] 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. [Pages 16, 18, 19, and 21.]

[33] 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. [Pages 16, 18, 21, 22, and 24.]

[34] S. Almanee, M. Payer, and J. Garcia, *Too Quiet in the Library: A Study of Native Third-Party Libraries in Android*, Nov. 2019. [Pages 16 and 20.]

[35] 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 [Pages 16 and 20.]

[36] 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 [Pages 16, 22, and 23.]

[37] 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 [Pages 16 and 24.]

[38] 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 [Pages 16 and 18.]

[39] 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 [Pages 16 and 18.]

[40] 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. [Pages 16, 18, 19, 21, 22, and 24.]

[41] 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 [Pages 16 and 23.]

[42] 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 [Page 19.]

[43] "Network security configuration." [Online]. Available: https://developer.android.com/training/articles/security-config [Page 19.]

[44] D. D. Chamberlin and R. F. Boyce, "SEQUEL: A structured English query language," in *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, ser. SIGFIDET '74. New York, NY, USA: Association for Computing Machinery, May 1974, pp. 249–264. [Online]. Available: https://doi.org/10.1145/800296.811515 [Page 21.]

[45] 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. [Pages 24 and 25.]

[46] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd ed. Wiley Publishing, 2011. [Page 27.]

[47] 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 [Pages 34 and 42.]

[48] "Xtext - Language Engineering Made Easy!" [Online]. Available: https://www.eclipse.org/Xtext/ [Page 35.]

[49] 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 [Page 36.]

[50] "Application Fundamentals." [Online]. Available: https://developer.android.com/guide/components/fundamentals [Page 37.]

[51] 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 [Page 43.]

[52] "secure-it-i / android-app-vulnerability-benchmarks / Permission / WeakPermission-UnauthorizedAccess-Lean — Bitbucket." [Online]. Available: https://bitbucket.org/secure-it-i/android-app-vulnerability-benchmarks/src/master/Permission/WeakPermission-UnauthorizedAccess-Lean/ [Page 54.]

[53] "BDD Testing & Collaboration Tools for Teams | Cucumber." [Online]. Available: https://cucumber.io/ [Page 68.]

# 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)*;
```

# Appendix E

# Complete Example Vulnerability Pattern Detection

In this annex, a complete example is shown, more complex than section 5.1.1. The enriched model we are going to analyse is the following:

```
model TranslationApp {

    contexts {
        INTERNET_CONNECTIVITY
    }
    static contexts {
        minSdk = "26",
        maxSdk = "",
        targetSdk = "32"
    }
    situations {
        INTERNET_DISCONNECTED : INTERNET_CONNECTIVITY,
        INTERNET_SLOW : INTERNET_CONNECTIVITY
    }

    statemachine "ABSTRACT_SM" {
        state START {
            transition on APP_STARTED -> SEND_MESSAGE_ACTIVITY
        }

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

        super state HANDLE_SUCCESS abstracts HANDLE_SUCCESS_SM {
            transition on END_HANDLE -> EXIT
        }

        state EXIT
    }
    statemachine HANDLE_SUCCESS_SM {
        state HANDLE_SUCCESS {
            transition on MESSAGE_PRINTED -> HANDLE
            transition on TIME_ELAPSED -> HANDLE_TIMEOUT
        }

        state HANDLE_TIMEOUT {
            transition on END_TIMEOUT -> HANDLE
        }

        state HANDLE
    }
```

Figure E.1: Enriched CDML of an example application - part 1

```
    statemachine SEND_MESSAGE_ACTIVITY_SM exported {
        state SEND_MESSAGE awareof INTERNET_CONNECTIVITY {
            transition on SEND_MESSAGE_CLICKED -> SENDER
            dataflows {
                source internet
            }
        }

        state SENDER {
            transition on GOOD_MESSAGE -> SHOW_ANSWER
            transition on WRONG -> DISPLAY_WARNING
            dataflows {
                input source enter_value
            }
        }

        state SHOW_ANSWER

        state DISPLAY_WARNING {
            transition on BACK_BUTTON_PRESSED -> SEND_MESSAGE
            dataflows {
                sink "log.t" ( source SEND_MESSAGE_ACTIVITY_SM.SEND_MESSAGE.internet, source SENDER.enter_value )
            }
        }

    }

    adaptation for INTERNET_SLOW at SEND_MESSAGE {

        state SEND_MESSAGE {
            transition on NONE -> HANDLE_SLOW_INTERNET
        }

        state HANDLE_SLOW_INTERNET{
            transition on NONE -> external SEND_MESSAGE_ACTIVITY_SM.SENDER
            dataflows {
                sink "log.d" ( source SEND_MESSAGE_ACTIVITY_SM.SEND_MESSAGE.internet )
            }
        }
    }
}
```

Figure E.2: Enriched CDML of an example application - part 2

With the *ConTest* tool, a total of:

- 245 tests starting from the main *abstract_sm*
- 49 tests starting from the exported activity *SEND_MESSAGE_ACTIVITY_SM*

The patterns used for the vulnerability detection are:

```
Vulnerability "log.t API 26" {
    description "log.t can be exploited by giving his first argument 'OUTPUsT'"

    context {
        apiversion "26"
    }

    function {
        main Sink "log.t" {
            parameter {
                private,
                static "OUTPUsT"
            }
        },
        Source private *
    }
}
```

Figure E.3: Annex: Pattern used 1

With this configuration, the output of figure E.6 shows the number of exploits generated for each starting node and each vulnerability pattern.

The resulting tests are not shown because of the sheer amount of space it would take on the report.

Here's a single example of a generated exploit in listing E.7.

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

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

Figure E.4: Annex: Pattern used 2

```
Vulnerability "vulnerableFunc" {
    description "vulnerableFunc can be exploited by giving his first argument 'evilPayload'"

    function {
        main Sink "log.t" {
            parameter {
                private,
                static "OUTPUsT"
            }
        },
        Source private *
    }
}
```

Figure E.5: Annex: Pattern used 3

```
---------------------------------- Debug Information for SEND_MESSAGE_ACTIVITY_SM ----------------------------------
   =====> vulnerableFunc has :
    Total tested 49
    Negatives    7
    Positives    42
   =====> Log.d Leak has :
    Total tested 49
    Negatives    35
    Positives    14
   =====> log.t API_26 has :
    Total tested 7
    Negatives    1
    Positives    6
-------------------------------------------------------------------------------------------------------------
---------------------------------- Debug Information for ABSTRACT_SM ----------------------------------
   =====> vulnerableFunc has :
    Total tested 245
    Negatives    35
    Positives    210
   =====> Log.d Leak has :
    Total tested 245
    Negatives    175
    Positives    70
   =====> log.t API_26 has :
    Total tested 35
    Negatives    5
    Positives    30
-------------------------------------------------------------------------------------------------------------
```

Figure E.6: Result of the VPatChecker tool on model E.2

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<TestPath>
    <staticContext APIVersion="30"/>
    <state name="START">
        <transition name="APP_STARTED"/>
    </state>
    <state name="SEND_MESSAGE">
        <transition name="NONE">
            <contexts>
                <context origin="INTERNET_CONNECTIVITY">slow3G</context>
            </contexts>
            <situations>
                <situation origin="DEFAULT_ORIGIN">INTERNET_SLOW</situation>
            </situations>
        </transition>
        <dataflows>
            <dataflow name="internet" type="Source"/>
        </dataflows>
    </state>
    <state name="HANDLE_SLOW_INTERNET">
        <transition name="NONE"/>
        <dataflows>
            <dataflow name="log.d" type="Sink">
                <parameters>
                    <parameter origin="source">internet</parameter>
                </parameters>
            </dataflow>
        </dataflows>
    </state>
    <state name="SENDER">
        <transition name="WRONG"/>
        <dataflows>
            <dataflow name="enter_value" type="Input"/>
        </dataflows>
    </state>
    <state name="DISPLAY_WARNING">
        <transition name="BACK_BUTTON_PRESSED"/>
        <dataflows>
            <dataflow name="log.t" type="Sink">
                <parameters>
                    <parameter origin="source">internet</parameter>
                    <parameter origin="source">enter_value</parameter>
                </parameters>
            </dataflow>
        </dataflows>
    </state>
    <state name="SEND_MESSAGE">
        <transition name="SEND_MESSAGE_CLICKED"/>
        <dataflows>
            <dataflow name="internet" type="Source"/>
        </dataflows>
    </state>
    <state name="SENDER">
        <transition name="GOOD_MESSAGE"/>
        <dataflows>
            <dataflow name="enter_value" type="Input"/>
        </dataflows>
    </state>
    <state name="SHOW_ANSWER">
        <transition name="SUCCESS"/>
    </state>
    <state name="HANDLE_SUCCESS">
        <transition name="MESSAGE_PRINTED"/>
    </state>
    <state name="HANDLE">
        <transition name="END_HANDLE"/>
    </state>
    <state name="EXIT"/>
</TestPath>
```

Figure E.7: Generated exploit for vulnerability "private data to log.d" in context API version 30 from starting node ABSTRACT_SM

TRITA-EECS-EX-2023:78