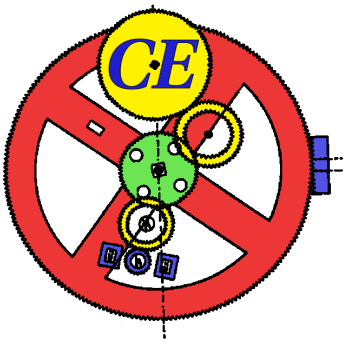

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2013

MSc THESIS

Automated Code Review for Fault Injection

Alexandru Ionut Diaconescu

CE-MS-2013-12

Abstract

The software quality relies, among others, on security. In the smartcards domain assumed throughout this paper, the focus is on security. Smartcards are embedded systems that contain sensitive information. Code review is one of the most efficient evaluation techniques applied to smartcards. It is used for the examination of the smartcard source code in order to find security weak points. Since the code review is expensive, time-consuming and error-prone when done manually, we developed an application that is performing this process automatically. According to our knowledge, this automation of the code review process for smartcards is an innovative approach. In this study, we use our application to identify the smartcard software vulnerabilities in order to further exploit them using fault injection. Fault injection is one of the most common attacks against smartcards. The developed application was evaluated and validated based on a test-suite composed of smartcard programs. The test-suite is composed of 12 smartcard programs that are based on defensive programming patterns against fault injection attacks. We were able to identify 15 vulnerability types in the test-suite. The success rate of the identified vulnerabilities from the test programs varies between 30% and 100%. As a result, we believe that the application will be a significant factor in evaluating the smartcard software.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Automated Code Review for Fault Injection

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Alexandru Ionut Diaconescu
born in Bucharest, Romania

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Automated Code Review for Fault Injection

by Alexandru Ionut Diaconescu

Abstract

The software quality relies, among others, on security. In the smartcards domain assumed throughout this paper, the focus is on security. Smartcards are embedded systems that contain sensitive information. Code review is one of the most efficient evaluation techniques applied to smartcards. It is used for the examination of the smartcard source code in order to find security weak points. Since the code review is expensive, time-consuming and error-prone when done manually, we developed an application that is performing this process automatically. According to our knowledge, this automation of the code review process for smartcards is an innovative approach. In this study, we use our application to identify the smartcard software vulnerabilities in order to further exploit them using fault injection. Fault injection is one of the most common attacks against smartcards. The developed application was evaluated and validated based on a test-suite composed of smartcard programs. The test-suite is composed of 12 smartcard programs that are based on defensive programming patterns against fault injection attacks. We were able to identify 15 vulnerability types in the test-suite. The success rate of the identified vulnerabilities from the test programs varies between 30% and 100%. As a result, we believe that the application will be a significant factor in evaluating the smartcard software.

Laboratory : Computer Engineering
Codenumber : CE-MS-2013-12

Committee Members :

Advisor:

Chairperson: Koen Bertels, CE, TU Delft

Member: Arjan van Genderen, CE, TU Delft

Member: Jos Weber, WMC, TU Delft

Member: Razvan Nane, CE, TU Delft

Member: Cees-Bart Breunesse, Riscure BV

Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my deep gratitude to Professor Koen Bertels and to my supervisor Razvan Nane from the Delft University of Technology for their guidance and useful critiques of this report. I am grateful for all the feedback given by Koen Bertels and Razvan Nane during the whole project period.

I would like to express my very great appreciation to my research coordinators Cees-Bart Breunesse and Eloi Sanfelix from the Riscure BV security test laboratory for their guidance, advice and assistance during the development of the automated code review application for fault injection. I am grateful of all the information I received during my Internship at Riscure, that helped me to successfully implement the application for this project and that provided me the theoretical support for this work. My grateful thanks are also extended to Ileana Buhan, for her advice and support.

I would also like to thank to the other members of the thesis committee: Arjan van Genderen and Jos Weber, for their time and effort to review this work.

Last but not least, I would like to thank my family and friends for their support and encouragement throughout my study.

Alexandru Ionut Diaconescu
Delft, The Netherlands
October 18, 2013

1 Smartcard vulnerabilities

In this chapter we give an overview of smartcard vulnerabilities. We present the engineering domain of the project and define the problem addressed in this work. Based on the problem definition, the scope and the objectives of the project are formulated. Subsequently, an introduction on smartcard vulnerabilities is given. The chapter ends with a section describing the smartcard standards, as well as the testing and certification requirements for evaluating smartcard programs. We define a penetration testing methodology for our application developed in this work.

1.1 Overview of Smartcard Vulnerabilities

1.1.1 Introduction

The software quality relies, among others, on security [5]. In the smartcards domain, the focus is mainly on security. Smartcards are embedded systems which in general contain sensitive information. These devices are involved in security sensitive operations. For smartcards, as well as for other sensitive embedded systems, security practices are integrated in the software development life cycle. The popularity of smartcards has increased in the last years due to the need to replace the magnetic-stripe cards which were not secure, allowing for the smartcard technology to evolve. Because of confidentiality reasons, the access control is on-card with the smartcard having computing capability.

The domains where smartcards are applied include banking, wireless market (SIM cards), electronic tickets, identity cards, health cards etc. Banking is the domain where smartcards are widely used under the format of debit cards, credit cards or cash cards. Therefore, considering this utilization of smartcards, security has to be enforced for these devices.

Pioneer development guidelines were elaborated in the smartcard industry. These can be found under the format of programming patterns and principles (see Section 2.4.). The development guidelines are specific to smartcards and intend to minimize the threats in smartcard applications. However, in order to develop a methodology for smartcard defense which is in compliance with the development guidelines, first we have to investigate how to attack the card. The offensive actions against smartards analyzed in this thesis work are represented by code reviews, which are one of the most common and efficient attacks [25].

The identification of smartcard vulnerabilities using **source code reviews** represents the goal of this thesis work. The identified vulnerabilities can be further exploited by the actual fault injection process. The **fault injection** process leads to the introduction of faults in the device running the software, resulting in unprivileged execution of critical code or recovery of sensitive data. Knowing the security vulnerabilities, rec-

ommendations can be made for smartcard software developers in order to mitigate the threats. Smartcards are the target devices used for the evaluation of the developed application. The approach can be expandable to other sensitive embedded devices, such as payment terminals or smartphones.

More industry and research areas interfere within this work. The source code review research (from this work) is focused on the identification of **smartcard vulnerabilities**, being easily expandable to other embedded systems. The motivation is that various smartcard security mechanisms are used on multiple embedded systems like mobile phones, set-top boxes, printers, payment terminals or medical equipment. The main reason for the focus on smartcards is that these devices are the perfect target to test the security of embedded systems, due the nature of smartcards which are involved in security sensitive activities. The code review application developed for this project permits **automated** smartcard vulnerabilities identification. Another related IT domain is represented by the Embedded Systems security. More concretely, we are interested in the subdomain of side channel attacks testing on Embedded Systems. The fault injection attacks are a type of side channel attacks, which are explained in *Section 1.1.5* [39]. Even if the thesis work can be extended to a larger domain of embedded devices, this project focuses on smartcards. This narrowing of the domain targeting only a specific device class will give better analysis results.

In literature, there is little guidance to smartcard developers on side channel attacks. The developers implement defense measures at the hardware, operating system and application levels [39]. In this project, we exploit the vulnerabilities from the application level of the smartcard. The code review used in this thesis work focuses on identifying the application level countermeasures which will lead to find vulnerabilities that can be further exploited using fault injection attacks.

An **application** was developed for this thesis work, with the goal to identify the smartcard vulnerabilities at the application level. Since there is no benchmark [8] for smartcard software, *Riscure* provided a test suite. We assume that the test suite covers the main part of the smartcard vulnerability types [39], which are presented in *Chapter 4*. We mention that in the smartcard industry there is **no benchmark**, referring to the fact that no benchmark is public or accepted. As mentioned in [8], benchmarks exist only in the R&D departments of smartcard manufacturers and in some smartcard users organization.

1.1.2 Problem definition

In this thesis work, we will investigate, implement and test an automated code review application used to find smartcard vulnerabilities. The problem definition is represented by the necessity of implementing an **automated** code review application for the fault injection process. This is going to replace the manual code analysis for vulnerabilities, increasing the efficiency with respect to **time** and **workload**. The automated code review is used to find the weak points in the smartcard program. Currently, the code review is done manually at Riscure and is therefore a time consuming process.

A source code review application is used to identify vulnerabilities at the application layer of the smartcard program, weaknesses that can be exploited by the fault injection

process. The inserted faults in the running software may result in unprivileged execution of sensitive code or in unwanted data leakage.

There are several possible effects of the fault injection process, e.g. skipping instructions or changing registers / memory values. By observing the effects, some protection solutions against fault injections can be proposed. For example, the code path leading to critical code should be protected with redundant checks, with the aim to verify the conditions under which the execution of the critical code is allowed. The redundant checking can be done while verifying the correctness of the cryptographic functions' results. The cryptographic functions might be integrity algorithms, authorization algorithms, encryption / decryption algorithms or confidential operations which are using sensitive data. An alternative to redundant checking can be the execution of the inverse operation on the data, followed by the verification of the results.

Source code reviews are used to identify the **checks** which set the conditions of critical code execution. The recognition of these checks leads to the identification of the smartcard vulnerabilities. The vulnerabilities can be exploited by fault injection attacks that can manipulate the sensitive data. The **sensitive** data can be exploited by unauthorized attacks (e.g. the money balance from a bank smartcard) or the data which is used by the security mechanisms (example: masterkeys). By doing the code review in an automated way, the time and workload for the security analysts is minimized. In addition, the analysis will be less exposed to **human faults**.

Smartcard vulnerabilities are not dependent on language specifics. Therefore, the smartcard source code should be analyzed in a **common representation**. One of the goals defined in *Section 1.1.4.* is to have support for the most common smartcard programming languages, in order to **translate** them into the common representation. The smartcard vulnerabilities will be identified by enabling code review on the **common representation**. A requirement of the thesis work is to find a framework in which the application can function on multiple smartcard programming languages. In the thesis work, we use LLVM to get the common representation of the source code languages.

LLVM is one of the most popular open-source compiler frameworks. LLVM is a good choice for the thesis project's application since it poses front-ends for many source code languages. The LLVM intermediate code is suitable for fault injection analysis. The alternative was to use compilers to transform C, C++ or Java Card source code into different bitcode representations, such as .NET. A front-end needs to be used to transform the smartcard programming languages into the intermediate code. An API which will analyze the resulted bitcode needs also to be designed. After analyzing the possibilities, we have chosen to use LLVM for several reasons. First, it is open-source and well documented. Second, it poses a variety of front-ends to transform multiple source code languages into common bitcode. Third, it has intermediate code representation. Fourth, it is a complete framework and provides extensive support for tool development. Using LLVM, the developed system is able to identify the weak points on the smartcard program which can be used in the fault injection process.

The defined problem has its solution in the development of an application which enables automated code review of smartcard programs. The smartcard program can be

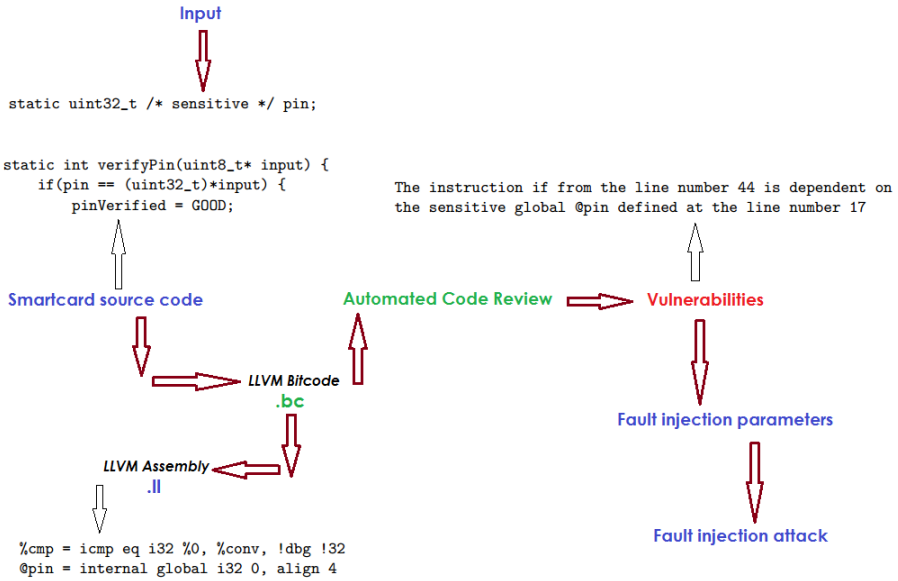

Figure 1.1: Workflow of the fault injection attack process

written in a variety of smartcard specific languages. The application is implemented on top of the LLVM compiler framework. The workflow of our application can be see in *Figure 1.1*.

The whole process will be fully described in *Chapter 2*. As we can observe in *Figure 1.1*, the smartcard source code is translated into the LLVM common bitcode representation. Our application is running the code review on the bitcode, resulting the smartcard vulnerabilities. These weaknesses are further exploited using fault injection. Additionally, the LLVM bitcode can be translated into a human-readable assembly format.

I developed the application for this project at Riscure BV. **Riscure** is an international security test laboratory, that is specialized in evaluating and testing the security of smartcards and embedded devices designed to operate securely. Riscure was founded in 2001 by Marc Witteman. In this project, we used a test-suite that is based on the Riscure guidelines which provides programming patterns to avoid fault injection attacks. Riscure provides its customers detailed feedback on security strengths and weaknesses of their products.

The Riscure personnel which will use our application are called security analysts. They play the role of an attacker who tries to break the security of the target smartcard. Based on the results, recommandations are made to increase the security of the smartcard.

The thesis title introduces two concepts. The automated code review concept is presented in *Section 2.1*. It represents an automated analysis of the source code. Using the analysis done by our application, the security weaknesses of the smartcard

software application are discovered. These weaknesses are called vulnerabilities. The fault injection concept is described in *Section 1.1.5.2*. It represents the physical technique used to exploit the found vulnerabilities.

1.1.3 Scope

The scope of the thesis work is to provide automation of smartcard vulnerabilities recognition for fault injection. This requires the usage of automated code review targeted on the smartcard source code. The code review is used to identify the weak points on the smartcard software. The code review is currently done manually at Riscure, but this approach is time consuming, costly and might be error-prone. Hence, we need to automate the code review to eliminate the impediments resulted from manual analysis [15]. The scope is used to define the objectives.

1.1.4 Objectives

The objectives are formulated from the scope of the thesis work. Smartcard vulnerabilities are not dependent on language specifics. Based on this observation, we analyze the smartcard source code in a common representation. Hence, **a goal** of the thesis work is to provide support for the most common smartcard source code languages. These are C, C++, Embedded C, Java Card. Under this assumption, it is preferable to abstract from the source code into an intermediate representation. The code review is performed on the common intermediate representation. Therefore, after transforming the source code into intermediate representation, this representation under the format of bitcode is analyzed for detecting critical code and protection measures. The code review is done by performing control flow and data flow analysis.

Another goal of this work is the thesis project evaluation. A set of categorized tests is used to check the accuracy of our application. The results are represented and explained into the evaluation chapter.

The objectives of this thesis work can be categorized into the following:

- transformation of the source code to a common intermediate representation that can be further analyzed
- analyze automatically the intermediate representation for vulnerabilities. This is done in the form of automated source code review for vulnerabilities recognition
- evaluate the results from the point of view of the efficiency regarding the vulnerabilities recognition. The evaluation is based on a relevant test-suite

The enumerated objectives are derived from the problem definition from *Section 1.2.*. The necessity of implementing an automated code review application for fault injection is illustrated by the objectives to transform the smartcard code into an intermediate language and to review the code in an automated manner. As mentioned in *Section 1.2.*, LLVM is used to obtain the intermediate code.

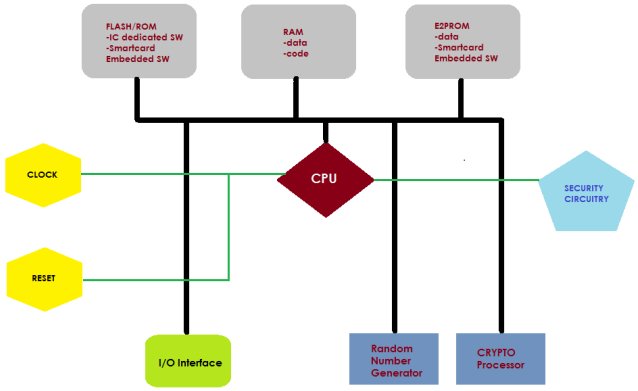

Figure 1.2: Typical scheme of a smartcard

1.1.5 Smartcard vulnerabilities

The work presented in this thesis is based on the recognition of the smartcard vulnerabilities which can be further exploited by fault injection attacks. The vulnerabilities are smartcard software weaknesses which are identified by the application developed for this thesis work.

A typical smartcard scheme can be seen in *Figure 1.2*. We present a typical smartcard integrated circuit [10].

The master unit in a smartcard is the central process unit, which is coordonating the other units and is making computations. Three types of memories are typically found on the smartcard. The basic sofware (the smartcard embedded software and the integrated circuit software) is kept in the Flash (or ROM) memory. A part of the embedded software and data are kept in the E2PROM memory module. The data and the code are stored in the RAM memory. The CPU is linked to circuits that give the clock and the reset signal to the processing unit. External signals are received via the I/O interface. The cryptographic algorithms from the smartcard needs two external circuits: the random number generator and the Crypto specialized processor. These circuits are needed for the computations of complex cryptographic operations. The attacks on smartcards can be categorized as follows [38]:

- Logical attacks: exploits vulnerabilities in software
- Physical attacks: analysis or modification of hardware
- Side channel attacks: analysis or modification of device behavior, by using physical phenomena

The application used in this thesis work identifies the vulnerabilities which can be used in fault injection attacks. The fault injection attacks are a type of side channel attacks,

which are presented below.

1.1.5.1 Side channel attacks

Side channel attacks treats security aspects like:

- Integrity: attacks to change the behavior of the smartcard program
- Confidentiality: attacks that can reveal sensitive data during program execution
- Authorization: attacks by changing the behavior of the smartcard program can lead to unauthorized access to sensitive data

Smartcard developers use secure programming methods in their efforts to defend against side channel attacks. These methods consist in sensitive operations that have to be identified by our application before using side channel attacks. In *Chapter 2*, we will present a set of secure programming patterns for critical devices. The patterns represent a base for the test-suite used in the evaluation of our application. [39]

Side channel threats differ from logical threats, the smartcard developers implementing defensive measures throughout the code. In the case of logical threats, the defensive measures focus mostly on input validation and output control.

Side channel attacks are well-known in the smartcard community and less outside of it [39]. The principle of a side channel attack is to abuse an unintended communication channel, hence the name of this kind of attack. The candidates for this kind of attacks can be embedded systems like mobile phones, smart cards, access control tokens etc., as mentioned previously in this chapter. The most common abused side channels are [39]:

- Time: the time to complete a specific operation
- Power: the power that is used by the embedded system or the power which is available
- Electro-magnetic radiation: EM radiation produced by the device

An example of an attack by abusing the time channel was reported in 2003 [6], on SSL over a network. There are two side channel attacks used to abuse the previous mentioned channels:

- Data leakage
- Fault injection

In the case of data leakage, the channel is passively listened for sensitive data leaks. In the case of fault injection, faults are inserted into the smartcard program to actively change the behavior of the device. In this thesis work, we focus on fault injection vulnerabilities.

1.1.5.2 Fault injection attacks

As mentioned above, fault injection is a type of side-channel attacks. Basically, using fault injection, an attacker aims to change the flow of the smartcard program or to change a critical value. This techniques allows to load unauthorized firmware, to skip a digital signature verification or to jump over the increase of a security counter [39]. An example of a device that performs fault injection attack is the Pay-TV unlooper, used for TV piracy. There are multiple ways of injecting faults. These are [39]:

- Power glitches: disturbing the power supply to the processor, wrong values can be read from memory
- Optical glitches: a laser can force elementary circuits to switch, involving specific change of data or behavior
- Power interruptions: interrupt the power supply to the processor, trying to stop the processor to take countermeasures against a detected attack
- Clock manipulations: imposing a short clock cycle can lead to values misinterpretations read from memory
- Differential Fault Analysis: a cryptographic key might be revealed by comparing the results of several runs of the cryptographic algorithm with injected faults and without injected faults

A form of testing used along with fault injection techniques is stress testing. It is used to check the stability of a system in testing conditions beyond the system's normal operational capacity. The robustness, availability and error handling of the software system outside the normal operational parameters can be then determined and thus this form of testing represents a way of finding vulnerabilities. Based on the results provided by our application, the security analysts can use fault injection along with stress testing, trying to exploit the found vulnerabilities. The vulnerabilities are mostly related to the **security checks** from the smartcard software. The security checks contain **conditional branches**, used to verify if the smartcard functions as expected. Therefore, a specific type of code coverage which is interesting for the thesis work is the branch coverage. One goal of our application is to identify the conditional branch coverage which is part of security checks. Basically, the branch coverage is a metric of the number of branches that are executed during experimentation. A large branch coverage can be achieved with stress testing by taking into account the conditions from security checks. This coverage can be improved by taking into account all the possible execution paths of the smartcard program, as seen in the *Section 2.3.6.*.

Fault injection types There are two types of fault injection: hardware implemented fault injection (HWIFI) and software implemented fault injection (SWIFI). The SWIFI

techniques can be compile-time injections or runtime injections. After inserting a fault into a system, the fault is propagating by following a defined cycle. When a fault executes, an error within the system can appear, illustrated by an invalid state. This phenomenon can propagate through the system. The observed error state is cataloged as failure. This aspect is important in dependability. [21] If a security check from the smartcard software is detecting a failure in a critical code block, the functionality of the smartcard can be disabled. Therefore, our application should detect the security checks which involve such smartcard disabling operations.

The first type of fault injection is the compile-time injection. The compile-time injection technique consists in modifying the source code to inject simulated faults. A type of compile-time injection is the mutation testing. This test checks if the tested software is correct and covers all the requirements of the target implementation. If the system's behavior reflected by the output is not affected, than the mutated code was not executed (the security analyst did not insert fault injection on any already executed path during the experiment) or the testing software have not located the fault that was injected (because the security analyst inserted fault injection in order to skip the security checks which aim to find the already inserted faults). A special type of mutation testing is fuzzing, in which the communication data between interfaces are mutated, having the scope of identifying the failures in data processing. The mutation can be detected by the smartcard security checks if two conditions are met: first, having the same test input, different states for the mutant and for the original program should appear; second, the output values should be checked by the test program. These kind of security checks are related to the Riscure patterns presented in *Chapter 2*. There are multiple operators used for mutation by fault injection, like statement deletion, replace operators with other operators, replaces variables with variables of the same type. More sophisticated operators of class-level exist, for object-oriented languages [23], for concurrent constructions [7] or for complex objects like containers [2]. An extension of mutation test is represented by the fault injection techniques which add code instead of just modifying it. One of these techniques is based on perturbation functions to add noise [4] to the existing values to perturb them into other values. The security checks used in our test-suite and the fault injection attacks on them are presented in *Chapter 4*.

The second type of fault injection is the runtime injection. The vulnerabilities found by our application are represented by the smartcard security checks that will be bypassed using runtime fault injection. The runtime injection technique is used for fault injection into the running smartcard software system. This is done by a software trigger injector. The trigger injectors can be time based or interrupt based. There are different techniques used for fault insertion. One of them is the network level fault injection, handling the corruption of network packets at the network interface. This might be used to obtain the sensitive data sent and received by the smartcard. Another technique is the corruption of memory space, more precisely RAM, registers, I/O map. The syscall interposition technique deals with the faults propagation from the operating system kernel interfaces to the executing systems software, by injecting fault into the operating system calls. Lastly, the most important technique is to skip the instructions from the security checks which can lead to the smartcard disabling. Some of these software security testing related

techniques are not applicable to smartcards. We focus in our project on smartcards. Therefore, we are concerned on the hardware fault injection, mostly using voltage/clock manipulation, laser or EM injection. These are presented in relevant examples in *Chapter 4*.

1.2 Testing and Certification

1.2.1 Smartcard standards

1.2.1.1 Overview of smartcard standards

This section gives an overview of the smartcard standards. The standards are not related to the developed application for this thesis work, but are coming to complete the overview of the IT domain of this project.

The growing presence of smartcards into everyday life demands for smartcard standardization. Smartcards can be used to implement different types of systems: health insurance cards, telephone cards, bank cards etc. Since on the market a large variety of smartcards and smartcard manufacturers exists, the need to generate application-independent standards has appeared. The standards allow multifunctional smartcards to be developed and the standards should be applicable to all smartcard types. Smartcards represent only the top of the iceberg of more complex systems. These networked systems are behind the card terminal, responsible for specific services. Therefore, smartcard standards should be in concordance with all the entities that are forming the complex system.[30]

Since the terms **standard** and **specification** can produce confusion while used, Rankl and Effing [30] provide a definition of *standards*. A standard is defined as a document produced by consensus and adopted by organizations, that defines rules or guidelines for activities or activities results in order to achieve optimum regulation in a given context. On the other hand, a specification refers to the explicit set of requirements to be satisfied by the smartcard. The set of specifications provides a complete description of the behavior of a system to be developed.

1.2.1.2 International standardization of smartcards

The international standards for smartcards are developed under ISO/IEC and under CEN (European Committee for Standardization). The ISO/IEC standards define the basic standards for smartcards. ISO is the International Organization for Standardization and IEC stands for the International Electrotechnical Commission. *Figure 1.3* presents the structure of ISO and IEC standards. In the figure, we observe two committees that deal with the smartcard standards: ISO TC68/SC6 for the financial area smartcards and ISO/IEC JTC1/SC17 for the general application smartcards. The CEN organization complements the ISO activities and produce application-specific standards.

The organization of the smartcard standards under ISO and IEC can be seen in

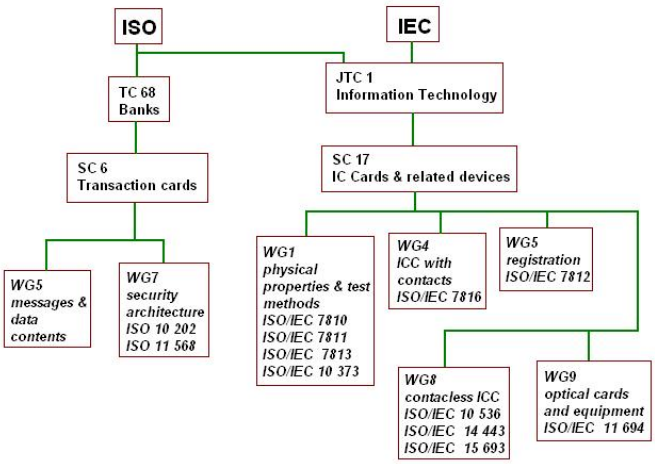

Figure 1.3: Organization of international smartcard standards [30]

Figure 1.3. A short presentation of the smartcard standards (**not only** the ones from the figure under ISO/IEC) is given in the following paragraphs.

ISO 7816 standards ISO 7816 standards deal with identification cards that are integrated circuit cards with contacts. The standards are published by ISO and cover multiple smartcard aspects: physical and contact characteristics, signals and communication protocols, interindustry commands and data elements, application characteristics. [11]

GSM standards ETSI (European Telecommunications Standards Institute) published standards that cover the smartcards for cellular phone systems. GSM is a specification for international mobile telephone systems. First, the GSM standards for mobile phone smartcards (GSM 11.11, GSM 11.14) handle the specifications of SIM (the smartcard used for mobile phones) regarding the application toolkit for the SIM-mobile equipment interface. Second, the GSM standard for mobile phone smartcards (GSM 03.48) deals with the security mechanism for the SIM application toolkit. Third, the GSM 03.19 is the standard for the SIM API of the Java Card platform. [11]

EMV specifications Europay, MasterCard and Visa define EMV, a set of specifications for smartcards. The EMV specifications are based on the ISO 7816 standards presented above. The latest version of specifications is EMV96 3.1.1., published in 1998 and aimed for the financial industry. The specifications cover the smartcard general specifications, terminal specifications and application specifications. [11]

Open Platform specifications The Open Platform defines specifications for the development and operation of multiple-application smart card systems. It contains smart-

card and terminal specifications. The smartcard specifications define requirements to implement an Open Platform card. The terminal specifications define the terminal architecture and the terminals compatibility with ISO and EMV. Initially developed by Visa, the specifications were translated to GlobalPlatform, which is promoting a global infrastructure for smartcard implementation for multiple industries. [11]

OpenCard Framework The OpenCard Framework is developed by OpenCard consortium, initially being owned by IBM. OpenCard is a framework that gives a standard interface for smartcard readers and smartcard applications. Its model includes smartcard terminal vendors, smartcard issuers, smartcard operating system providers, platform providers, having the objective to minimize the dependence on each of these parties. [11]

PC/SC The PC/SC specifications are defined by the PC/SC workgroup, describing a general purpose architecture for using smartcards on PC systems. The host-side smartcard applications are built on top of several service providers and a resource manager. The smartcard functionality is made available through an API. [11]

The code review is intended to find vulnerabilities for smartcards that use the ISO 7816 basic standard, the EMV and the Open Platform specifications. The thesis work is aimed to analyze the card at the application layer, so the hardware or the operating system specifications are not relevant. The security mechanisms introduced by these specifications are identified by the application of this thesis work. The smartcard security measurements are identified as smartcard vulnerabilities and these cases are illustrated into the *Evaluation* chapter of this thesis. The standards and specifications section is not related to our application, but they complete the overview of the smartcard domain.

1.2.2 Testing and Certification requirements

The testing and certification requirements are directed towards the smartcard target programs. The smartcard programs have to respect several standards and specifications defined in the previous section. In contrast, the application developed for this thesis work is an ethical hacking toolchain, aimed to provide assistance to the security analysts to discover vulnerabilities of embedded systems. From the point of view of Riscure, the developed application does not have certification requirements. However, the application must comply with the LLVM development requirements (being developed under LLVM) and it can be considered part of a penetration testing methodology (using the application, the security analysts make experiments to test the security of the target smartcard). For our application, we use an own-defined methodology inspired in the *CC certification process*. The CC represents the *Common criteria for information tehnology security evaluation*. The certificates include the CEM, which is the *Common methodology for information tehnology security evaluation*. [1]

1.2.2.1 Penetration Testing methodology

The aim of the application from this thesis work is to find vulnerabilities in the smartcard program source code, being part in a penetration testing methodology (that includes also the actual fault injection attack on the smartcard). A penetration testing methodology is described in [10]. The methodology can be used to define the attack steps and penetration strategies for our developed application.

This section describes a general penetration testing methodology that we used in the project. The methodology is own-defined, being inspired in the CC certification process. Our methodology is presented below.

We focus on the first step and on the second step of the methodology, more exactly on the definition of a set of potential vulnerabilities and on the identification of the vulnerabilities. This is done by the application of the *Riscure patterns* in our test-suite which is used to evaluate our application. The guidance on the attack methods is given by the accredited laboratories, such as *Riscure*. The chosen method is based on glitching the sensitive conditions from the smartcard program by a fault injection device. The fault injection device glitches on the smartcard, being able to change the flow of the program or modify sensitive data. A **glitch** is a short-lived fault in a system inserted by the attacker/security analyst using fault injection. Another example of guidance is the *AFSCM NFC cardlet development guideline* [14].

The *AFSCM* guidance is mentioned only for reference, being an alternative to the Riscure patterns. Our test-suite is based on the Riscure patterns, that cover the common smartcard vulnerabilities [39].

The Riscure patterns and the AFSCM rules are guidances for the test-suite used to evaluate our application. In contrast, we need a guidance for our application which is used to attack the smartcard. **First**, we present our own-defined **penetration testing methodology**. **Second**, we present the guidance that is based on the CC certificates (which an **overview** of the general methodology) and is it called *APSC JIL Application of Attack Potential to Smartcards*, provided by the National Schemes.

Penetration methodology for our application Our penetration testing methodology for the application satisfies the scope defined in *Section 1.1.2.* and the objectives defined in *Section 1.1.3.* First, the security analyst has to set a scope for the attack (e.g. increase the balance on the smartcard). Second, since the smartcard vulnerablities are not dependent on the language specifics, the smartcard target program should be compiled into a common intermediate represention. The result is a bitcode file which can be used for code review. Third, the attacker should define the points in the program between which it will perform code review. The code review is made at compile-time. Therefore, the fourth step is to identify all the possible execution paths of the target program. For each execution path, the application should identify the smartcard vulnerabilities (the fifth step). The last step is to insert fault injection to exploit the found

vulnerabilities, with the purpose to avoid the security mechanisms and to reach the goal.

Overview of the general methodology The *APSC* guidance [9] is based on the evaluation of the International Security Certification Initiative (*ISCI*) and the JIL Hardware Attacks Subgroup (*JHAS*).

The CC certification process is based on the CC certificates [9] and it describes the steps followed when performing an attack. The CC certificates make a distinction between the cost of identify vulnerabilities for an attack and the cost to exploit the vulnerabilities. In this thesis work, we perform the vulnerabilities identification phase.

The attack vulnerabilities identification and exploitation are inspired from the CC certification process. The *CC certificates* are mapped to factors like elapsed time, expertise, knowledge of the target program, access to the target program, equipment needed to carry out the attack, as well as evaluation samples with known secrets [9]. From these factors, the corresponding requirements are derived, that characterize the vulnerabilities recognition for our application.

The vulnerabilities identification phase is relevant to this thesis work. A set of instructions to trigger the specific equipment is needed for the exploitation phase. The exploitation phase is the next step from the methodology after the code review is done by our application. The set of instructions can contain timing information required for a perturbation attack, so the vulnerability can be directly exploited. In the case of our work, the smartcard evaluator is not spending time on looking for sensitive conditional branches into the code, work that can be error-prone.

The knowledge of the target program is an important requirement mentioned in the *CC Certificates*. It is expected that the knowledge needed to generate the attacking scripts (set of instructions) is passed from the identification phase [9].

Another requirement derived from the factors mentioned above is linked to the access to the target program. However, this requirement is made for the exploitation phase. This requirement is needed due the fact that an attack requires more than one sample of the device where the target program resides. Multiple samples are required because the attack success can have some probability and also the devices might be destroyed while the attack (for example, many glitches in a fault injection attack may make a smartcard not functional).

The place of our application in the methodology and a **description** of the application can be seen in *Chapter 2*. *Chapter 3* presents a **detailed description** of the developed application.

In the next paragraph, there is presented a short taxonomy of penetration attacks. We mention when a type of an attack is applicable in our work.

Penetration attacks This section also includes the main categories of penetration attacks. The attacks which are relevant for the developed application are mentioned together with the applicability of them in this thesis work.

In [9], there is given a large range of attack method examples. We briefly present them next.

The physical attacks can enable access to secret data, disconnect IC security features or even force internal signals. Other type of physical attacks can overcome sensors and filters, in order to change the program flow, induce failures in operations, change the operating mode.

The perturbation attacks are attacks that can use the output from the application developed in this thesis work. The fault injection are belonging to the perturbation attacks domain. The introduced faults can be used to recover keys or plaintext. The attack is directed against cryptographic operations or can be used to change the results of authentication checks. Several examples are [9]:

- Read operation. A read value can be changed when arrive to destination (not actually changed in memory).
- Random number generation. The attack can force all the random number generators output to be all 1s.
- Program flow modification. Our application is mainly aimed for this vulnerability. Possible effects are: skip an conditional branch instruction, replace an instruction with a dummy one, invert a test, generate an unauthorized jump, create calculation errors.

The Differential Fault Analysis (*DFA*) attack is another type of attack. It can have as input the results of the developed application for this thesis work. Secret keys can be retrieved by comparing a calculation without an error and calculations that do have an error [9]. There are two ways to conduct this attack: invasive and non-invasive. The non-invasive way consists in power glitching or by applying lasers on the target device. Using DFA the DES, 3DES and RSA keys can be revealed, by changing the cryptographic outputs.

A non-invasive retrieving of secrets can be the *Simple Power Analysis* or the *Differential Power Analysis*. These types of attacks aim to exploit data leakage obtained from variations in the power consumption [9] of the target devices. Riscure provides *Leakage Patterns* besides the *Fault Injection Patters* (presented in *Section 2.4.*), but it is not the scope of this thesis work to treat them.

Other multiple attacks can threaten the security of embedded devices. The EMA attack represent a different type of attack, directed to measure the electromagnetic emissions from an IC during operation. Other attacks aim to enter the test mode of the target device, in order to exploit the test features. A type of logical attack executes malicious applications formed from illegal sequences of byte-code instructions oriented against Java Card target programs. Buffer overflow or stack overflow attacks are done using malicious applications. [9]

A type of attacks are directed against the smartcard software. This type of attacks are logical attacks that need a number of different steps to discover vulnerabilities. The steps are [9]: observe messages, command searches, edit commands, direct attacks on cryptographic operations, man-in-the-middle attacks, replay attacks, access control and bypass authentication. The last two steps can be performed using the developed application for the thesis work. Using the application presented in this thesis, the security

operations (including the access control and authentication) that are protected by security checks can be identified. These can be further bypassed by inserting fault injection.

1.2.2.2 Other requirements

The application is developed over LLVM, hence the application can submit to LLVM requirements. In addition, several *Common Criteria* requirements are discussed.

LLVM has a policy for the tools development. The policy applies only to the tools that are added to the LLVM repository and have an open-source character. However, this is not the case for our application, which is not open-source. The requirements requires to have code reviews (manual or automated) of the developed tool, test cases for every feature, minimum quality specified in the *LLVM Coding Standards*. The LLVM Coding Standards refer more to the code that will be used in the LLVM source tree, which is not the case for our application. If the tools are not intended to the LLVM source tree, no coding standards are regarded as absolute requirements [35]. The main idea of the LLVM Coding Standards is that if an existing piece of code is extended, enhanced or bug fixed, the same style as the existing one should be used so the source code is uniform and intelligible [35].

The test-suite developed for the evaluation of our application is based on the Riscure patterns. The test-suite presented in *Chapter 4* is formed from test programs (real or experimental) and not from in-use smartcard programs (since they are proprietary). As a future work, the test-suite can submit to the testing and certification requirements for smartcard programs, if the suite will contain in-use smartcard programs. The smartcard code has to follow the ISO/IEC 15408 international standard known as the *Common Criteria* for Information Technology Security Evaluation or *CC*. The Common Criteria (CC) is a framework used for specification of a system products functional and assurance requirements. The provided assurance is related to the specification, implementation and evaluation of the product [1].

The Common Criteria organization published a supporting document guidance for smartcard evaluation [10]. The purpose of the document is to define smartcard (and similar embedded systems) evaluation terminology and to describe it.

There are more software cores on the smartcard, as follows [10]:

- IC dedicated software: needed for testing purposes, hardware utilization services, support software
- Basic software: required for generic functions of the smartcard, like the operating system, general routines and interpreters
- Application software: dedicated to applications. It is contained in the Embedded Software

- Embedded software: handles generic functions on the smartcard (contains the Basic software), but also embedded software dedicated to applications (contains the Application Software)

The application developed in this thesis work is aimed to find the vulnerabilities of the *Application part* of the Embedded software. The scope of the fault injection process is to attack the application layer of the smartcard program, which is found in the Embedded software core.

As a conclusion regarding the certification and testing requirements, these requirements are mostly oriented towards hardware and towards procedures to follow, not being detailed regarding the implementation of a code review application that identifies software vulnerabilities. We can consider the requirements presented above in the case of the test-suite. Guidelines might be used for the secure programming of test-suite, like the ones provided by Riscure and AFSCM [14]. The AFSCM was introduced in *Section 1.2.2.1.* On the other hand, we are interested into requirements for the application used to find smartcard vulnerabilities. Briefly defined procedures exist in this case, but constraints or rules are not defined. The reason is the nature of attacks, which are used to harden the security of the smartcards based on experiments. In the future, these kind of attacks against smartcards may lead to the development of a benchmark test-suite and even to the fully standardization of the attack methodology. The requirements to implement a penetration application are given by experimental attacks. These requirements can be used to describe the procedures to develop a more secure smartcard software.

1.3 Organization

This thesis is structured as follows: the first Chapter presents an overview of smartcard vulnerabilities, together with the testing and certification requirements of the developed application for this thesis work. In the second Chapter, the experimental setup is described, including an introduction to code review, to the LLVM compiler framework and a brief presentation at abstract level of the application developed for this thesis work. The Riscure patterns are also presented, which were used in the generation of the test suite, presented in the *Evaluation* chapter. The *Evaluation* chapter is the fourth chapter, that presents the vulnerability types from the test suite programs, together with the results and validation. *Chapter 5* concludes the work presented in this document, including the future work.

1.4 Summary

We present in this chapter an overview of smartcard vulnerabilities. The domain of this project and the key concepts were introduced. The problem definition is represented by the necessity of implementing an automated code review application for the fault injection process, that gives the scope of this thesis. The objectives of this thesis work can be summarized as: the transformation of the source code to a common intermediate representation; analyze automatically the intermediate representation for fault injection vulnerabilities; evaluate the results regarding the vulnerabilities recognition. Additionally, we gave an overview of the smartcard standards and of the testing and certification requirements for smartcards. Particularly, our application is an ethical hacking toolchain, aimed to provide assistance to the security analysts to discover vulnerabilities of smartcards.

2 Experimental Setup

The *Experimental Setup* chapter describes the theoretical background of the developed code review application and the setup needed for its implementation and evaluation. A short introduction to the code review process is done in the beginning of the chapter, which is followed by a presentation of the LLVM compiler framework upon which our application was developped. The central section of *Chapter2* is *Code Review Application for Fault Injection* which introduces our application, presenting the theoretical aspects, its scope and functionality. The application is presented at an abstract level. This chapter answers questions such as "What is the developed application?", "Why it is needed?" and "How it is implemented?". *Chapter3* should be checked for the detailed description of the application used to detect smartcard vulnerabilities.

Chapter2 presents the theoretical aspects (fault injection vulnerabilities) that were used to generate the test-suite from the *Evaluation* chapter. The test-suite is based on a smartcard development guideline provided by Riscure, that is presented in *Section 2.4.*

2.1 Code review

Code review is a process for the examination of source code. Its applicability is to find errors and weak points in the code. Its results can be used for code fixing and improvement. The reviews can be performed in many forms [22]. The most common vulnerabilities found by code reviews are given by security checks, buffer overflows, memory leaks, race conditions, string exploits. The goal is to correct them and to improve the software security.

Using code review, security weaknesses in the source code can be identified. Many security problems are caused by bugs. For example, misusing string functions can lead to the raise of buffer overflow vulnerabilities. In the case of this thesis work, we concentrate of the security vulnerabilities in the smartcard source code. Our code review application identifies the **security checks** from the smartcard, which can be further exploited by fault injection.

Automated code review The code review for identifying vulnerabilities can be done automatically. In literature [24] it is reported that only 17.6% of the embedded software engineers use automated tools (those tools are not related to our problem definition; the main domain for the code review tools is the detection of implementation bugs, such as format string exploits, memory leaks or buffer overflows) for code review and it is expected an increasing of 23.7% till 2014. The process of automated code review contains source code verification based on a set of rules.

Our automated code review application is a static analyzer of the target smartcard source code. It finds security vulnerabilities for fault injection at compile-time. Based on the results, the fault injection exploits the identified vulnerabilities at run-time.

2.2 Infrastructure Setup

The system on which the application is developed is Ubuntu 12.04, a recent stable release. The main tool that is needed for the automated code review is LLVM.

As mentioned in *Chapter 1*, the application will make automated code review on the smartcard target programs from the test-suite. LLVM provides the needed tools for code review and support for a multitude of programming languages. The idea is to transform any smartcard target program source code into an intermediate representation and to perform code review on it.

2.2.1 LLVM

LLVM is a compiler framework written in **C++**. Because our application extends the functionality of LLVM, the application is written in C++ as well. Basically, LLVM framework is made up from compiler and toolchain technologies. LLVM name comes from *low-level virtual machine* [35], but nowadays the name has little to do with virtual machines. Since its evolution is as an umbrella for a variety of projects, its scope was not limited to the creation of virtual machines. Nevertheless, LLVM can still be used to build virtual machines. LLVM provides flexibility and reusability, being one of the most developed compiler frameworks.

Using LLVM, the developed system is able to identify the operations that can be used in the fault injection process. The identified operations are based on security sensitive data which is marked by the analysts into the smartcard source code. This approach is chosen due the fact that the interfaces might be proprietary. If they are proprietary and no documentation about the APIs used is available, then the sensitive data has to be input by the analyst. This is the case for our application.

The application of this thesis work is based on more tools which LLVM provides. The most important one is the LLVM Core.

The LLVM Core [35] is a collection of libraries, providing compiler optimizers and code generation support. The libraries are built on the LLVM intermediate representation, called LLVM IR. Particularly for our work, the code review is made on the LLVM IR in order to find fault injection vulnerabilities.

2.2.1.1 Clang

The platform to obtain the LLVM IR (LLVM bitcode) from the target program is given by Clang. Clang is a LLVM C/C++/Objective-C front-end compiler. Its compilers are claimed to be much faster than GCC [33]. The LLVM IR obtained from C/Objective-C

smartcard target programs is given by Clang. Clang++ is the Clang version to deal with C++ target programs.

2.2.1.2 VMKit

In contrast to the C/C++ programs compiled with Clang, several smartcard programs may be written in Java or .NET. Actually, Java Card is one of the most popular smartcard programming languages. In order to obtain the LLVM IR from Java or .NET target programs, we use the VMKit tool. This tool is an implementation of the Java and .NET virtual machines for LLVM framework.

2.2.1.3 KLEE

It is not rare that the input provided by the analysts to be incomplete. In order to improve the input needed by the application from the thesis work, sensitivity has to be propagated. The most common way is to make taint analysis to detect the sensitivity propagation. One question is why **tainted analysis** is needed. By using the application without taint analysis support, only the security checks that are based on the already marked sensitive variables are detected. However, some sensitive conditions may be based on variables that are not initially considered sensitive by the analysts, but their values are influenced by sensitive variables. Therefore, in the case of a fault injection attack, if a sensitive condition based on a tainted variable is not bypassed, the attack may fail (if the condition statement is executed).

Currently, we implemented our **taint analysis** algorithm inside the thesis application for the purpose to find the tainted variables that participate in unidentified sensitive conditions. The algorithm is explained in *Section 3.3.*. In order to make the taint analysis more efficient, we aim to use symbolic execution (as a future work). The **symbolic execution** is a means of analyzing a program to determine the possible execution paths and to emulate the execution of the program. Note that the tainted variables can be inherited between multiple execution paths, so they are needed to be identified before the application from this thesis work is applied, improving the input of sensitive variables.

KLEE is a LLVM tool that implements the needed symbolic virtual machine to support symbolic execution. For the symbolic execution, KLEE uses a theorem prover to evaluate all the dynamic paths through a program (until a user-defined threshold value) [35]. As a future work, we aim to use KLEE to evaluate the dynamic paths in order to identify the tainted variables that participate in unidentified security checks.

Some LLVM-based projects provide support for other programming languages. The most known is Dragonegg [35], allowing to compile Ada and Fortran. Other projects offer support to compile Ruby, Python, Haskell, D, PHP, Pure, Lua in the LLVM IR [35]. However, the scope of this thesis work is not to test these languages, since they might be rarely used for smartcard implementation.

2.3 Code Review Application for Fault Injection

This section provides an overview of the developed application. The application is described at an abstract level in this chapter. The functionality of each tool found in our application is briefly described based on a target program example. For a more detailed description of the application, we refer the reader to *Chapter3*.

2.3.1 Pass

The code review application developed for this thesis work is a toolchain. The application contains **three** chained tools. The three tools are used to analyze and transform the LLVM IR bitcode of the smartcard target program. The tools providing such functionality are called **Passes** in the LLVM technology.

A LLVM pass is used for transformations and optimizations required by the compiler, performing code review and being a structuring technique for compiler code [35]. All the three passes of the application are subclasses of the LLVM Pass class. The LLVM Pass class gives the necessary methods to develop a LLVM pass. Our passes override virtual methods inherited from the Pass class in order to implement the required functionality.

There are 3 steps to perform for applying a LLVM pass. The first step is to set up the build environment. The source code is compiled using a *Makefile* script, that links together all the compiled files from a *workspace folder* into a shared object (*.so*). This type of object can be dynamically loaded by the LLVM tool called **opt**. The second step adds the basic code required for a pass like: the inclusion of specific headers (for example *llvm/Pass.h*), the usage of the namespace *llvm*, the proper pass declaration (for example *struct MyClass : public ModulePass*), pass identifier definition, the main method (for example *virtual bool runOnModule(Module &M)*) which overrides the inherited abstract virtual method, pass initialization using the pass *ID* address and the pass register. The third step is to run the pass using the *opt* tool, loading the created shared object. Particularly, the bitcode file (LLVM IR) is run using the opt tool. The opt tool is run together with the command line option *(–time-passes)* that gives information about the execution time of the pass. In order to see the target program as one unit, the application uses the *ModulePass* class for all its three composing tools.

2.3.2 Target program graph

This section presents the structure of a smartcard target program from our test-suite. Based on it, the functionality of each tool from our application is explained.

The structure of the target program can be seen as a **Call Graph**. A Call Graph is a graph whose vertices are the functions of the program and the edges are given by the relations between functions. Basically, a Call Graph is a directed graph indicating the connections between subroutines of a program. An edge (*a*,*b*) represents that *Function a calls Function b*. A trivial example of a Call Graph of a target program can be seen in *Figure 2.1*.

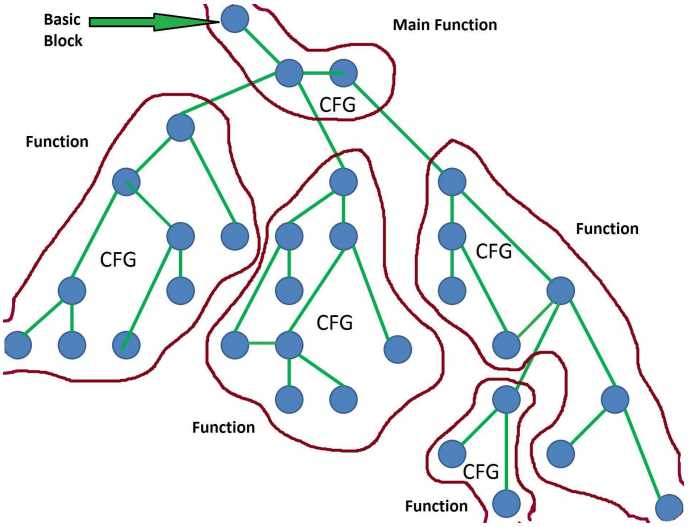

Figure 2.1: Call Graph example

Futhermore, each function of the Call Graph (CG) can be seen as a Control Flow Graph (CFG). The call graph is formed from more control flow graphs. A CFG gives the paths that can be traversed during one execution of a program. The vertices are represented by basic blocks, while the edges are jumps in the control flow. *Figure 2.2* shows a snippet code from a target program used for the evaluation of the application.

A more detailed presentation of the smartcard target programs is given in the third chapter, together with a C++ trivial example on which the application is applied. Based on the target program example, the functionality of each of the three passes is illustrated. Actual examples of target programs are shown in the *Evaluation* chapter.

2.3.3 Toolchain

The application developed for this thesis work is a toolchain. The toolchain fulfils the scope defined in *Section 1.1.3.*, namely the automation of the fault injection vulnerability recognition process. Since the vulnerability identification done by the security analysts was primarily manually, we needed to develop an automated code review application. In order to cover all the conditions based on sensitive variables, our application has to identify the dependencies between the sensitive conditional branches (the conditional branches which are part of security checks) and the original sensitive variables. The dependencies discovered by the toolchain shows:

- How many sensitive conditional braches there are in the target program
- The location of the sensitive conditional braches
- The original sensitive variables on which the conditional branches are dependent

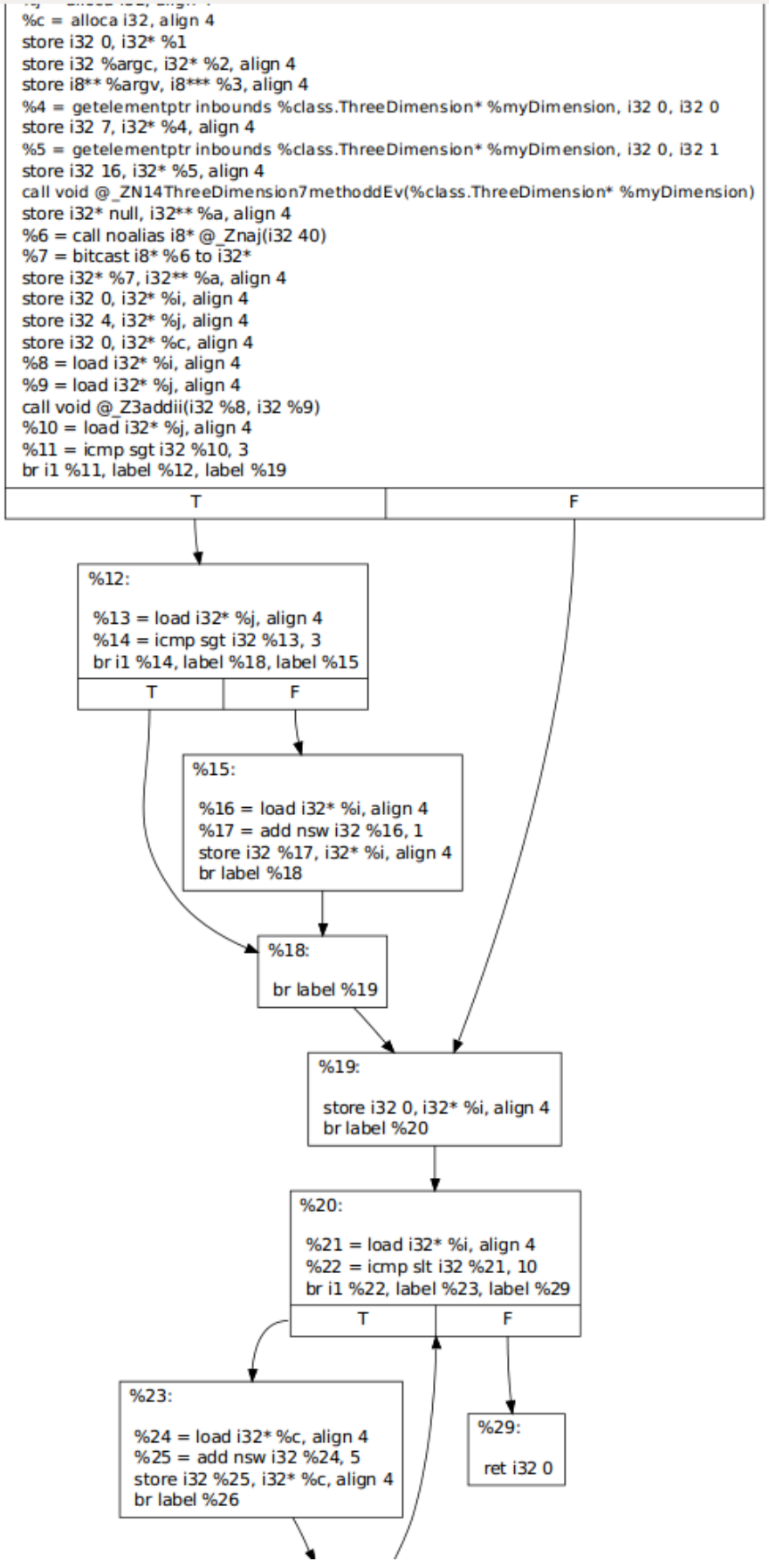

Figure 2.2: Control Flow Graph example. The figure represents a part of the CFG of the *main* function. Inside the basic blocks, we can see the instructions in LLVM assembly language format.

- The location where the original sensitive variables are declared

Why the dependencies between the original sensitive variables and the conditional branches are needed? Inside the LLVM intermediate representation, every time a variable is used inside a basic block it has a different assigned name. For example, when a global variable is allocated, it is named *@a*. When it is loaded inside a basic block, the loaded value is assigned to a local variable named *%6*, the following computations being done using the local variable. Different alias names may be assigned to the variable, depending on conversion or transformation operations.

The LLVM variable identifiers have two types: global and local. The global identifiers (for example functions or global variables) begin with the @ character, while the local identifiers (register names, local variables) begin with the % character [35]. The original sensitive variable is annotated as *sensitive* by the security analysts, but the local variables that are aliases for the original variable are not. It would be inefficient to propagate the *sensitiveness* among all the target bitcode. Therefore, we need to get the original variable which is annotated as *sensitive* by the security analysts or which is automatically marked as sensitive based on taint analysis (that is explained in the *Section 3.3.*).

The identified dependencies show in how many places and where to insert faults (fault injection) into the smartcard program in order to avoid the security checks. The dependencies are identified for **each possible execution path** of the program. Only identifying all the security checks in the program is **not relevant**, since we are interested only on one execution path per test. When the smartcard program executes, it follows only one possible execution path in a given context. Also, it is **not effective**, since the number of places to insert fault injection may be very large. By inserting faults at the sensitive checks on a possible execution path, malicious actions could be done by bypassing the conditional branches. For example, on a specific execution path, the balance from a travel smartcard may be increased by the attacker/security analyst.

The sensitive conditions implemented on the smartcard program are structures which include conditional branches dependent on sensitive variables. They represent smartcard vulnerabilities which can be exploited using fault injection. As we will see in the *Evaluation* chapter, these sensitive checks come in variants. We can see in *Figure 2.3* the variants of the security checks, composed of conditional branches and the *return BAD* statement (in the first 3 cases) that can disable the functionality of the smartcard.

2.3.4 Passchain

As mentioned in the first section of this subchapter, a pass is a LLVM tool that does analysis and transformation on the LLVM IR. Basically, the developed toolchain can be seen as a **passchain** in terms of LLVM terminology, every tool being represented by a pass. The passchain denoting the vulnerability recognition application is shown in *Figure 2.4*.

As we can see from the figure, the security analysts makes annotations into the source code of the target program. The smartcard target program is compiled into the LLVM intermediate representation: the LLVM bitcode (the three **.bc** files from the figure). The bitcode files serve as inputs for the three tools contained in our application: the *Annotation Pass*, the *Paths Pass* and the *Analysis Pass*. The output of the application

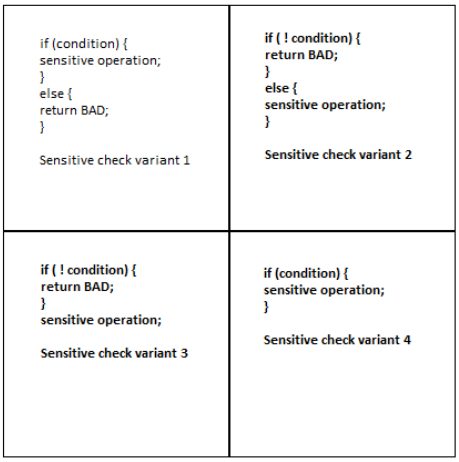

Figure 2.3: Variants of Security Checks

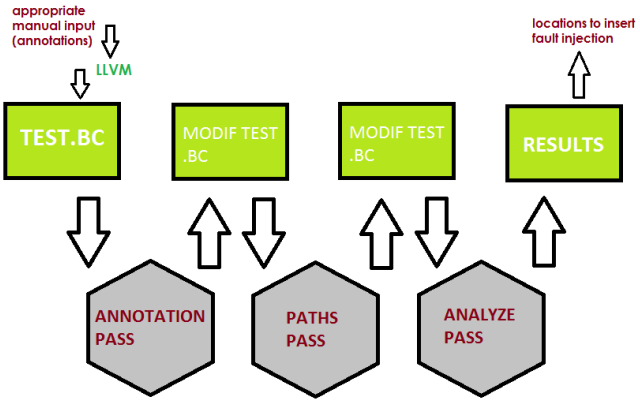

Figure 2.4: The developed Application

gives the results. Based on the smartcard vulnerability results, the security analysts determine the locations where to insert fault injection in the target smartcard code.

The security analysts mark the sensitive variables inside the source code by using annotations. The **annotation** is a medatada added to the variable. The annotations used by the security analysts are introduced in *Section 3.3*.

The passchain needs appropriate input from the security analysts. This input consists in annotating the variables that are assumed to be sensitive. The annotation can be

done partially due the fact that some variables are identified to be sensitive by the taint analysis which is integrated into our application. The next step is to compile the target program using the appropriate LLVM-based compiler (for example Clang++ for C++ target programs). The result will be a bitcode file having the extension *.bc*. The first bitcode file is transformed by the first tool of the toolchain which is called the *Annotation Pass*. The resulted bitcode file is then analyzed and transformed by the *Paths Pass*. The third produced bitcode file is analyzed by the main *Analysis Pass* which gives the final results that will indicate the locations where to insert faults into the target smartcard program.

All the three passes will be presented in detail in *Chapter 3* and briefly described in the following paragraphs.

The target program can be seen as a large graph, as previously described in *Section 2.3.2.*

2.3.5 Annotation Pass

The functionality of the *Annotation Pass* is to indicate:

- the original sensitive variables marked by the security analysts inside the source code (example from *Figure 2.6*: sensitive global variables like *pin* or *balance* and sensitive locals like a variables *a* or like an element of an array *t[10]*)
- the start and the end points between which the analysis will be applied

The end point of the analysis represents the state of the program in which the goal is achieved by bypassing the security checks. The end point is a leaf in the program graph, that leads to the success state from the point of view of the attacker/security analyst. In the majority [39] of the cases, the other leaves are *bad* states, meaning that security procedures like *faultDetect()* or *faultResponse()* are triggered (indicated by *BAD* in the following picture). These procedures are smartcard security mechanisms that can disable the smartcard functionality.

The **goals** of the *Annotation Pass* are:

- to define target program borders for the code review
- to indicate the sensitive variables in the bitcode file

2.3.6 Paths pass

This pass is used to traverse the CFGs (control flow graphs) that form the CG (call graph), with the purpose to identify the possible execution paths. *Figure 2.8* shows the two possible execution paths between the *Start* point and the *End* (*Succes*) point in the target program.

The goals of the *Paths Pass* are:

- Consider only the fragment of the target code that the security analyst is interested in. The fragment is chosen depending on the purpose of the attack. For example, if the security analyst wants to withdraw money from the bank smartcard, the *end*

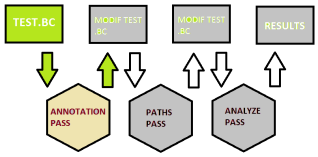

Figure 2.5:

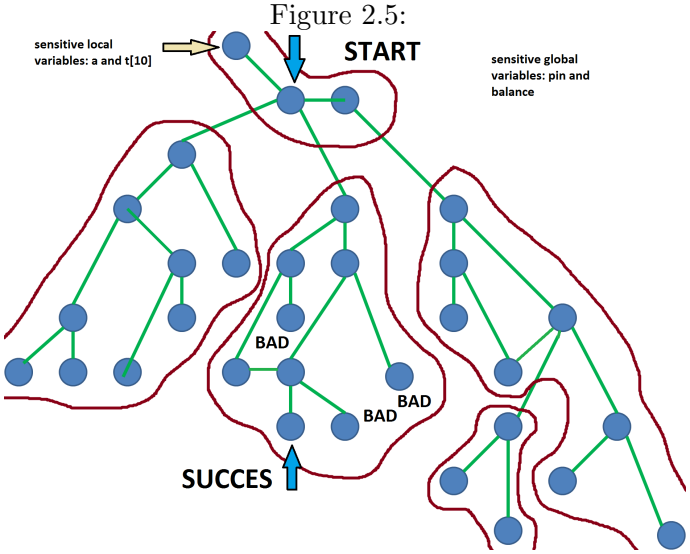

Figure 2.6: Annotation Pass example. On the top part of the figure, we can see the *Annotation Pass* highlighted in the Passchain

point of the fragment should be set after the withdraw operation in the program flow

- On the fragment of the target code (from above), get all possible execution paths on which our application will individually apply the code review

2.3.7 Analysis pass

The main functionality of the *Analysis Pass* is to identify which security checks are dependent on sensitive variables. The scope of this pass is to determine in how many places and where to insert fault injection. *Figure 2.10* shows the places where fault injection has to be inserted on the two possible execution paths previously identified.

We can see that by inserting fault injection on the particular conditional branches (from the indicated basic blocks in *Figure 2.10*), we can bypass their corresponding security checks. Hence, we can reach the *Success* state and not any *Bad* state without knowing the secrets (values of the sensitive variables) used in the conditional branches. This is done by identifying the dependencies between the conditional branches and the sensitive variables. These dependencies can be:

- Direct dependencies

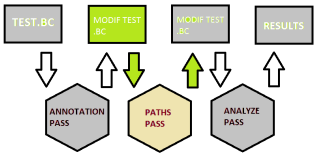

Figure 2.7:

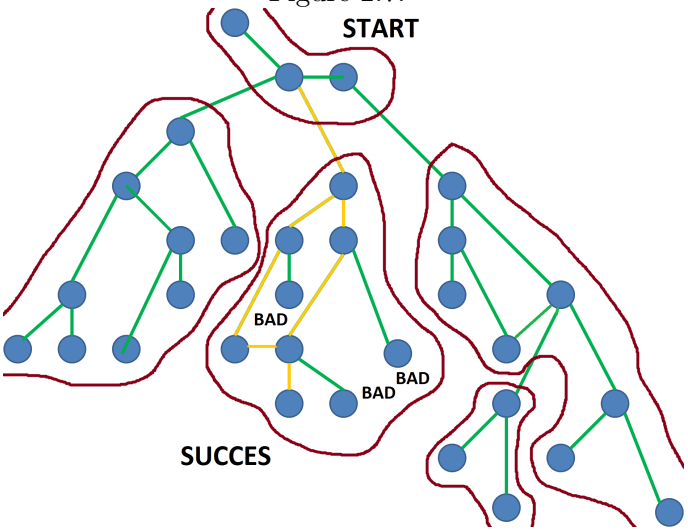

Figure 2.8: Paths Pass example. On the top part of the figure, we can see the *Paths Pass* highlighted in our Application

- Inter-procedural dependencies
- Tainted dependencies
- Indirect dependencies
- Alias dependencies

Examples with all the dependencies are given in the next chapter. Briefly described, a direct dependency is given by a pair of (*conditionalBranch, sensitiveVariable*), where the alias of the original sensitive variable is used directly by the conditional branch.

In an inter-procedural dependence, the conditional branch calls a function that could call another function or the same function recursively. The calls can be multiple, so the search for sensitive variables is done considering a *cascade of function calls*. The *Analysis Pass* identifies all the sensitive variables that are called by a specific conditional branch in other functions (procedures).

The tainted dependencies are another type of dependencies identified by the *Analysis Pass*. The concept behind taint analysis is that the security checks (more exactly their conditional branches) can be dependent also on variables that are not initially considered sensitive (like *flags*), but are influenced by sensitive variables. Therefore, security

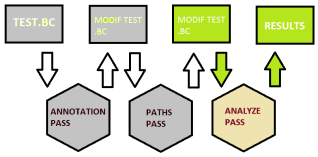

Figure 2.9:

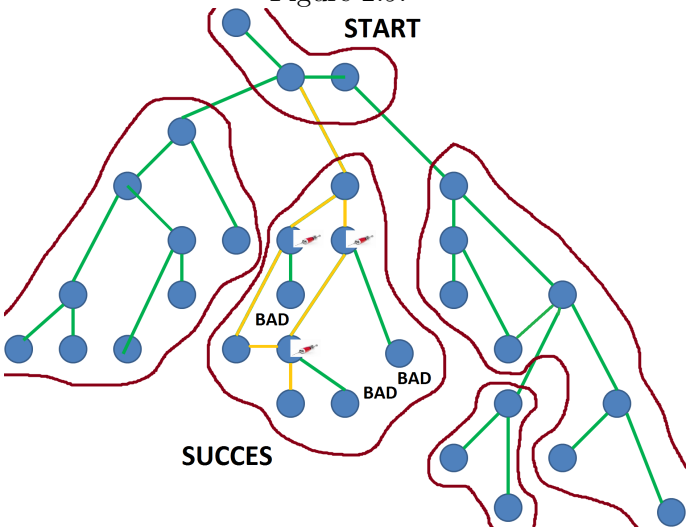

Figure 2.10: Analysis Pass example. On the top part of the figure, we can see the *Analysis Pass* highlighted in our Passchain

mechanisms on smartcards contain conditional branches that are dependent on variables that are not original sensitive variables.

The indirect dependencies are given by a pair of (*conditionalBranch, sensitiveVariable*), having one term of the conditional branch guard a function which has one of its arguments a sensitive variable.

The alias dependencies are illustrated by a pair of (*conditionalBranch, sensitiveVariable*), having one term of the conditional branch guard an alias of the sensitive variable. All the dependency types are illustrated with examples in *Chapter 4*.

To summarize, the *Analysis Pass* is the third tool of our application that is doing data flow analysis and also control flow analysis on the target program, in order to discover vulnerabilities under the form of dependencies between the conditional branches (from the encountered security checks) and the sensitive variables. The dependencies are illustrated in detail in *Chapter 4*.

2.4 The Riscure Smartcard Development Guideline

The smartcard target programs are protected by security checks which are encountered in the program flow. These security checks have to be identified by our application, in

order to bypass them by using fault injection.

In this section we present smartcard development guidelines, provided by Riscure. A part of the test suite from the *Evaluation* chapter is based on these guidelines, which are structured in the form of patterns.

2.4.1 Riscure patterns

The Riscure patterns are a set of security principles, guides, recommendations for the smartcard software development engineers. The Riscure patterns are used only in the test suite generation, presented in *Chapter 4*. The scope of the patterns is to protect the data and the program flow in the smartcard code. The smartcard software developers can use these patterns to protect confidential data such as cryptographic keys while hiding the sensitive decisions. The patterns serve only for the smartcard application layer. Additionally, the smartcard engineers have to take into account the protection for hardware and the operating system. Our application is aiming to identify the fault injection vulnerabilities, even if the extra security measures given by the patterns are used.

The smartcard developers implement mechanisms to secure sensitive operations as withdrawing money, modifying the balance, displaying personal information about the owner etc. Our application is needed to identify these security mechanisms. After the vulnerable spots from the smartcard program are found, fault injection is used to exploit them. Using fault injection, the sensitive data can be manipulated and the security checks can be avoided. An example of a weak smartcard security measure is:

```
if(conditional branch) //security check based on sensitive data
...
   faultDetection(); //dangerous for the attacker/security analyst
```

Our application is identifying this vulnerability represented by the dependency between the conditional branch and the sensitive data. Based on this result, the security analyst will mark the point where to insert fault injection. As we will see in the *Chapter 4*, the marked points for fault injection are dependent on the vulnerability type.

The patterns are used in the test-suite, which is presented in the *Evaluation* chapter. The test-suite used for this thesis work is not limited to these patterns, other various security mechanisms being taken into account, as seen in *Chapter 4*.

2.4.1.1 FAULT.DOUBLECHECK

The FAULT.DOUBLECHECK pattern [39] is the main pattern that exploits the double-checking security mechanism. The context is given by the situation when a sensitive decision may be corrupted by a fault injection attack. The solution is to double check the sensitive conditions. It is recommended that the checks are complementary. The following example illustrates the situation:

```
if (sensitiveValue == 0x45FA8C12) {
```

```
 . . .
  if (~sensitiveValue != 0x45FA8C12) {
   faultDetect();
 }
}
```

If the attacker do not glitch (inserting fault injection) on both sensitive checks, the *faultDetect()* procedure will be trigger. This procedure can lead to the temporarily or permanently disablement of the functionality of the smartcard. The fail is due the fact that the complement value is not equal to 0x45FA8C12.

This pattern can be checked by the developed application in this thesis work under the form of structural analysis. The *Analysis Pass* fulfills this need. In particular for the double-check pattern, the *Analysis Pass* tool can check if an identified sensitive condition is nested in another identified sensitive condition block (the second *if* from the example is nested in the first *if* block) or if it contains a lower block with another sensitive condition (the first *if* block from the example contains the second *if* condition). However, the sensitive conditions must be dependent on the same sensitive variable(s) in order to check this pattern. This check is provided by the *Analysis Pass*.

2.4.1.2 FAULT.DETECT

Using the fault injection process, the sensitive data can be manipulated during the smartcard program execution. [39] The solution is to verify for integrity the sensitive data when used, based for example on a *checksum*. A more complex solution is to encapsulate the sensitive data into security controlled objects that have own integrity checking methods. An example is shown below :

```
int sensitData = VALUE;
int checksum = ~VALUE+0xFA;
if ((sensitData & (checksum-0xFA)) != 0x00)   fail();
```

If the *fail()* procedure is trigger, than the integrity of the *sensitData* or its protecting checksum might have been violated.

This pattern can be checked by using the *Analysis Pass* of the developed application. The checksum is a value tainted from a sensitive condition, as mentioned in *Section 2.2.1.3.*. This means that the checksum value is changed in a nested block of a condition that is evaluated based on sensitive variables (like *pin* in the case of smartcards). *The Analysis Pass* considers the tainted values from sensitive values also as being security sensitive. Hence, the condition check based on the checksum (the *if* condition from the example) will be identified as a possible place to glitch (to perform a fault injection attack). Hence, the pattern is checked by identifying the sensitive conditions that contain the checksum in their guards.

2.4.1.3 FAULT.CRYPTO

Fault injection attacks can be applied on cryptographic algorithms. The cryptographic algorithms can reveal sensitive keys while being attacked by *Differential fault analysis* (DFA), which is based on returning data of false encryptions.

The solution proposed by the FAULT.CRYPTO pattern [39] is to check the sensitive data during or after the cryptographic operation. In the following example, the ciphered data is checked before it is transmitted to the deciphering function:

```
decipher(cipheredData, key);
if (decipheredData != 0x45FC3A98) {
faultDetect();
}
```

The conditional branch verifies if the deciphered data matches the original data which has to be ciphered. If it is a match, it is most likely that the *crypto* function was not corrupted by fault injection.

This pattern can be checked by the *Analysis Pass*. The pass can use data flow analysis by tracking the encrypted data. Based on the same principle of taint analysis from the previous pattern example, the sensitive condition is identified (the *if* statement from the example).

2.4.1.4 FAULT.CONSTANT.CODING

Fault injection can manipulate sensitive data that is using trivial constant coding, like Boolean values. The solution [39] is not to use this trivial encoding and to use instead numbers that have a large hamming distance (the number of bits that differ) between them. Using this pattern, it is difficult for an attacker to change one valid value to another valid value, because the fault injection can typically flip only a bit or only set all the bits to 0 or 1. An example [39] of suitable values, with a larger data type that permits a larger hamming distance (larger than 8 in the example) is shown below:

```
static final short STATE_INIT = (short)0x5A3C;
static final short STATE_LOCKED = (short)0xC3A5;
```

This pattern can be verified if applied by structural analysis, more exactly by calculating if the hamming distance if below a threshold value. The sensitive values using the specific encoding can be checked after the *Annotation Pass* from the developed application is applied on the smartcard's target code.

2.4.1.5 FAULT.FLOW

The fault injection process can be used for the so-called code hijacking attack. The hijacking consists in unauthorized jumps to privileged code, by manipulating the program counter or the stack. A prevention measure is given by the checking of the correct flow of the program during the execution of the privileged code. More concretely, this can be done by verifying the counters utilized for the correctness of the execution path. These counters keep track of the function calls and steps.

The method is illustrated in the example accompanying this paragraph [39]. Several methods use a counter which is increased and each of them includes its own number of steps (step value) that is used to increase the counter. The security measure consists in checking if the number of taken steps are the same as the expected step value and if

the right method is called. This can be done by comparing the counter increased over a function call with the product of steps and the step value.

An example is shown below:

```
#define M1 = 7;
#define M2 = 17;
int counter;
void main( int argc, char** argv ) {
...
counter = 0;
method1( );
counter -= 4*M1;
if (counter != 0) faultDetect();
}

void method1() {
int localCounter;
...
counter += M1;
...
localCounter = counter;
method2();
counter -= 2*M2;
if (counter != localCounter) faultDetect();
...
counter += M1;
...
}

void method2() {
...
counter += M2;
...
counter += M2;
...
}
```

In the *main* function, the counter is modified by subtracting 4 increase steps of *method1* (4*M1). If the completion check of method1 fails, *faultDetect()* procedure is triggered, than can disable the smartcard functionality. In *method1()*, the counter is increased by M1, having the *localCounter* as a backup counter. After calling *method2()*, the counter is modified by subtracting 2 increase steps of *method2* (2*M2). Then, the same procedure to verify the completion of method2 is applied. The last counter increase in *method1()* is done by adding the M1 value. In *method2()*, the counter is increased two times by M2. The method identifiers M1 and M2 are chosen to be prime because it is less likely that the same counter increase will result. Because there are multiple

security checks in the nested code, it is hard for the attacker to bypass all of them. The attacker/security analyst needs to glitch all the security checks to have a successful code hijacking. There might be the case that parts of the code have no predictable flow, like the complex loops. Other patterns should be used to protect these parts of code. In order to verify if this pattern is applied, the check counters have to be identified in the code. Using the *Analysis Pass*, the basic blocks in which sensitive data is used can be identified. Then, assumptions on check counters regarding their sensitiveness have to be made. The assumptions are based on the variable incrementation which is proportional to the number of instructions in the basic block.

2.4.1.6 FAULT.LOOPCHECK

The fault injection attacks can be directed to terminate earlier a repetitive process running in a loop. This way, later security checks can be bypassed. The solution [39] consists in checking the loop completion. An example that applies the pattern is shown below:

```
for ( i=0; i<n; i++ ) {
  ...
}
if (i != n)
  faultDetect();
```

By using the *Annotation Pass*, this pattern can be checked. Either the analyst can annotate the loop counter as sensitive or this can be done by the *Annotation Pass*. We can consider this pattern a particular case of the FAULT.FLOW pattern which is presented above.

2.4.1.7 FAULT.DEFAULTFAIL

This pattern [39] is related to the parameters that can take only a limited set of values. If these parameters are used in a *switch* or in a *if-else-if* construction, it is common that the default case deals with the last possible value without any accompanying security checking. This unprotected case is vulnerable to fault injection attacks, more concrete for potential data manipulation. The solution consists in checking all possible cases of the *switch* or of the *if* constructs and using fail by default (*default* case of *switch* or the final *else* statement in an *if-else* construct). In the following example, the sensitive variables are initialized with least-privileged data. Hence, if an attacker is able to skip later assignments, he cannot escalate privileges.

```
switch (state) {
case INITIALIZE: init(); break;
case LOCK: lock(); break;
case ACTION: action(); break;
default: fail();
}
```

2.4.1.8 FAULT.BRANCH

The smartcard program may be maliciously manipulated by changing the Boolean values used for decisions. Using fault injection, an attacker can change values to 0 and 1. This pattern recommends not to use Boolean values for sensitive decisions. In spite, non-trivial values should be used. An example [39] is provided below:

```
if (sensitiveValue == 0x4FB3) {
  . . .
}
else if (sensitiveValue == 0xF43B) {
  . . .
}
else
  . . .
```

The pattern is easily checked if applied by using the *Analysis Pass*. The sensitive conditions can be identified and the constants used can be checked against a set of rules for *triviality* (example: non-Boolean values). The type of the variables in the guard can be easily checked using LLVM methods.

2.4.1.9 FAULT.RESPOND

This pattern [39] represents a guideline to provide an efficient response to a fault injection attack. The goal is to protect the device in the case of an attack. The guideline consists in two parts: monitor and respond to fault injection attacks. The monitoring can be represented by repeating checks for data changes (using the FAULT.DETECT pattern, previously presented). The frequency of faults has to be observed, since smartcards can malfunction not only because of fault injection attacks. The guideline presents also a response. The most common measure is to temporarily or permanently disable the functionality of the smartcard. An alternative is to introduce long processing delays. In the following example, the *faultDetect()* procedure detects the faults, while the *faultResponse()* procedure responds to faults.

```
#define MAX_FAULTS = 10;
int fault_counter = 0;

void main() {
if (fault_counter >= MAX_FAULTS) {
  faultResponse();
}
processing();
}

void faultDetect() { // fault injection is detected
if (fault_counter < MAX_FAULTS) {
  fault_counter++;
```

```
} else {
  faultResponse();
}
while (true);
}

void faultResponse() {
  wipeSecretsAndDeactivate();
  while (true);
}
```

The *fault_counter* variable has to be stored in a non-volatile memory. In the *main()* function, we check if the faults exceeded the maximum number of admitted faults. If the amount is not exceeded, the normal processing is executed. The *faultDetect()* method is used to detect a fault injection attack. The fault counter is incremented until the maximum number admitted. If the number of faults is in excess, the *faultResponse()* method is triggered. The endless *while(true)* loop forces the system to reboot, after a fault is detected. The security mechanism on the smartcard uses *faultResponse()* as a defensive measure. It triggers the *wipeSecretsAndDeactivate()* method which is used to disable the device. Concretely, it clear the secrets (the sensitive data, like the pin code for smartcards) and blocks the access.

The pattern should be checked if applied only in the case when the security analysts do not have available the piece of code from the example, because in that case the security mechanisms of the smartcards (as *faultResponse()* and *faultDetect()*) are not available. The fault counter(s) can be tracker by data flow analysis and they can be identified using taint analysis. Then, the *Analysis Pass* can indicate all the sensitive conditions that lead to *faultResponse()* and to *faultDetect()*.

2.4.1.10 FAULT.BYPASS

If the fault detection procedure used in the above example is not done at the same level (in the code) as the protected functionality, multiple double-checks may be useless against a type of fault injection attack that can bypass them. In order to avoid this situation, the solution is to check for faults in the same function that executes or invokes the protected functionality. In the example from below[39], two methods of protection are proposed: *weak_protection()* and *better_protection()*. An example of *weak_protection()* method can be observed below:

```
void weak_protection() {
if (!test1())
   return;
protected_process();
}
```

The *weak_protection()* method provides protected functionality after the security check. If *test1()* does not pass the condition, the access to the protected functionality is not

allowed. If the condition is passed, the *protected_process()* may begin. The *test1()* method is presented below:

```
boolean test1() {
boolean result1 = check1();
if (check1() != result1) faultDetect();
boolean result2 = check2();
if (check2() != result2) faultDetect();
return result1 && result2;
}
```

The method *test1()* uses two methods to check for fault detection: *check1()* and *check2()*. The first *if* statement double-checks the *check1()* method, triggering *faultDetect()* in case of failure. As it can be observed from the example, the *weak_protection()* method checks the access conditions by calling *test1*, that is protected against fault injection. However, the protection is not ensured, even if *test1* might be secure enough. By using fault injection, an attacker can glitch (bypass using fault injection) the *if* condition from the *weak_protection()* method, so the call to *test1()* can be skipped. There is also an alternative attack: the fault injection attack can manipulate the return value in order to access the protected functionality.

On the other hand, the *better_protection()* method uses the already presented double-check pattern inside the function that gives access to the protected functionality. The method can be observed in the following pseudocode:

```
void better_protection() {
  boolean result1 = test2();
  boolean result2 = test2();
if (result1 != result2)
  faultDetect();
if (!result1 || !result2)
  return;
protected_process();
}
```

The *better_protection()* method offers the access to the protected functionality only after the double security check. If the conditional arguments (result1 and result2) are different, a fault detection procedure is triggered. If one of the conditional arguments is NULL, then the access is denied. The called *test2()* function can be observed below:

```
boolean test2() {
boolean result1 = check1();
boolean result2 = check2();
return result1 && result2;
}
```

The *test2()* function contains the *check1()* and the *check2()* tests, with no fault detection mechanism. This is done by the *better_protection()* method. The taint analysis has to

be used to check automatically this pattern, like in the case of the FAULT.DETECT pattern. It is possible to check if the fault detection is in the same function together with sensitive instructions. Once the tainted values are known, the *Analysis Pass* can be used to identify the sensitive conditions that might be glitch by fault injection attacks.

2.4.1.11 FAULT.DELAY

The glitch of fault injection usually needs an exact hit on the vulnerable code. The proposed solution [39] uses random length delays when sensitive code is utilized. This solution is directed against the time based fault injection attacks, the prediction of the instructions which can be affected by a glitch becoming improbable. An example is provided below:

```
void delay () {
int n = rand() & 0x2AB;
for (i=0, j=n; i<n ; i++, j--){
  if ((i+j) != n)
faultDetect();
}
}
```

This function is used to wait random time and to catch the faults. The random length delays have to be identified in order to check if the pattern is applied. More exactly, the loop with random values has to be identified. After these loops are manually or automatically identified, the *Analysis Pass* can be used to check if any sensitive conditions are present before the delay loops.

2.5 Summary

This second chapter presents the theoretical background and setup of the developed application for this thesis work. The chapter presents the scope and functionality of each tool that is contained in our code review application. In this chapter, we present the application at an abstract level. For a more detailed presentation of the application, we refer the reader to the next chapter. *Chapter 2* presents the setup of the evaluation of the application. The evaluation is based on a test-suite that is dependent on the Riscure smartcard development guideline.

Application for Fault Injection Vulnerabilities Recognition

3

The *Application for Fault Injection vulnerabilities recognition* chapter gives a detailed presentation of the developed application. The application is a toolchain composed from 3 tools (LLVM passes). In the following sections, the LLVM passes are described accompanied by a target program example. The passes were briefly presented in the 2.3 section *Code Review Application for Fault Injection*.

The application is necessary for implementing an automated code review application for the fault injection process. The code review is identifying the smartcard vulnerabilities which can be exploited using fault injection. The evaluation of the application is done in *Chapter 4*. An example of a trivial **smartcard security measure** is given in *Section 2.4.1*.

In the next section, the target program example is introduced. In the following sections, all the 3 passes are described in terms of the provided functionality, local objectives for each pass, connections with other passes/external environment and implementation. Additionally, the functionality of each pass is exemplified on the target program.

3.1 Target program example

The target program example is a trivial example which is relevant for the main types of dependencies that will be described in the next sections, together with the 3 LLVM passes of our application. All the dependency types are presented in *Chapter 4*.

The target program is part of the smartcard domain for banking applications. The problem is represented by the fact that a security analyst can increase the money balance on the smartcard by bypassing the security mechanisms. The attack point is identified by using our code review application and implemented by applying fault injection. The scope is to identify the smartcard code vulnerabilities in order to insert fault injection to bypass the security checks. Therefore, our application should be able to identify the security checks represented by the conditional branches that are dependent on sensitive variables.

The code is split in functions under the form of a call graph (graph made from the functions of the target program) that can be seen in *Figure 3.1*.

However, in order to illustrate the example in a more legible way (easy to follow, logical flow of events), the code is represented under the form of a control-flow graph which is composed from the basic blocks of each function of the target program (as seen in *Figure 3.1*). As shown in *Section 2.3.2.*, the code review is based on the call graph and on the control flow graphs (one CFG for each function) of the target program. In order to get the call graph of the target program, we use the LLVM *getAnalysisUsage()* method. This particular method extracts from the *AnalysisUsage* object the needed

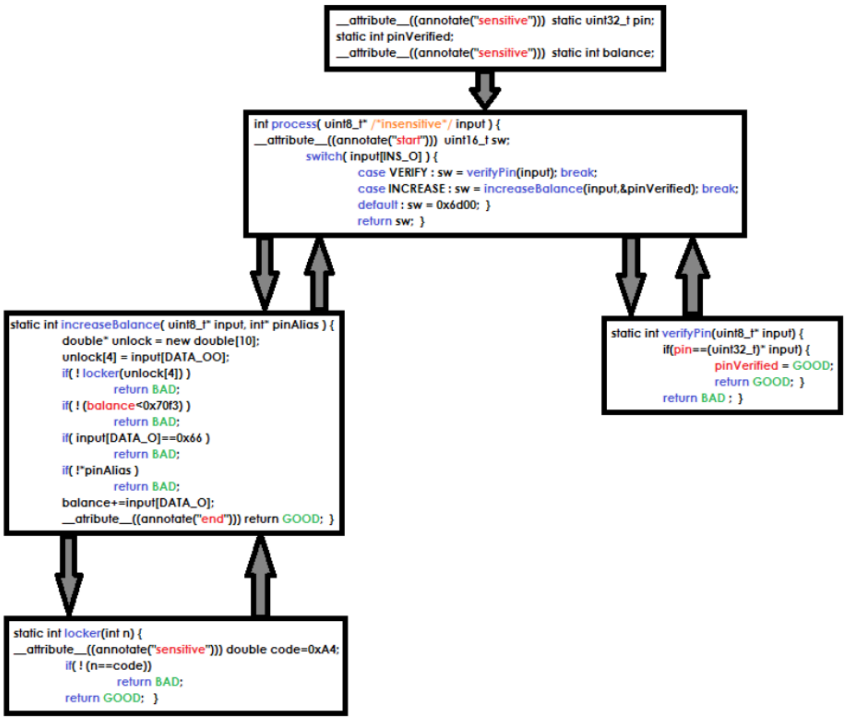

Figure 3.1: Call graph of the Target Program Example

information, more concretely the call graph nodes. From the nodes we can extract the program's functions. The reason of working with LLVM call graph nodes and not directly with functions is that the LLVM nodes provide extra information (for example: the *getCalledFunction()* method gives the callee function). The identification of the basic blocks is done by associating integer identifiers for each one of them as shown in the example below:

```
FUNCTION  main | BASIC_BLOCK  if.then16  | NR 12
```

The smartcard target program translated into LLVM IR bitcode represents a **module**. The module is composed of **functions**. Each function is composed from one or more basic blocks. A **basic block** has one entry point (no instruction from the basic block except the first one is the destination of a jump instruction anywhere in the program) and one exit point (only the last instruction of the basic block can cause the program to begin executing code in a different basic block). Each basic block contains **instructions** that might have several **operands**, which can be accesed during the code review. Therefore, the analysis is done at multiple **levels**: on the functions of the module, on the basic blocks contained into the functions, at instruction level, on the instruction operands.

The code example is written in the C++ language. The example is trivial; the purpose of this section is to illustrate how the application works on a proof-of-concept

example. Examples derived from real-world smartcard programs can be seen in the *Evaluation* chapter.

The trivial basic program includes 4 functions and the declaration of the global variables. The main function is called *process* and is used to select the path to take via a *switch* construction. The first possibility is to check the *pin*, by calling the *verifyPin* function. If the user gives the correct pin value as input, the *pinVerified* flag is set to true. We can see the example above in *Figure 3.1.*

Please note that in all the following examples, *GOOD* is defined as 1 and *BAD* is defined as 0. If the returned value is *BAD*, a fault detection routine is called which can disable the smartcard functionality with the scope to protect the smartcard from attackers, as mentioned in the first chapter.

The second option which the program flow can take is the *increaseBalance* function. After 4 *if* constructions, the program flow encounters an operation for increasing the balance. One conditional branch is calling the *locker* function, which implements a security checking mechanism. The exact functionality of our application on the target program example is shown in *Section 3.6*, since all the tools which form our application should be presented.

For an easier understanding and for a logical flow of events, the **example** is given after all the 3 passes that compose our application are presented. The example (presented in *Section 3.6.*) shows how the most common types of smartcard vulnerabilities are identified by our application. For the chosen approach, each vulnerability identification is seen as a stand-alone **event** (a result of our application). We have chosen this **approach** for a better understanding of the functionality of our application as a succession of similar events.

In contrast, in *Chapter 4* we use a **different approach**. For this approach, the functionality of each pass is presented sequentially. An example is (for the approach used in *Chapter 4*): we show all the execution paths and then we show separately the identified vulnerabilities for all the paths. **In contrast** (*for the* Chapter 3 *approach*), it is easier and more logical to present for each path (on which we have a succes target state, denoted by an annotated *end* point) which vulnerabilities are contained. Therefore, we show **separately** on each specific path where to insert fault injection in order to get to the succes target state.

The *Chapter4* provides more complicated examples which are used in real tests.

3.2 Annotation Pass

The *Annotation Pass* is the first tool of our application. As seen in the *Section 2.3.5.*, the pass receives as input the LLVM bitcode file which is compiled from the smartcard target program. The security analysts annotate in the source code a part of the sensitive variables and the points between which the code review is going to be done. The goals

of the *Annotation Pass* is to define the target program borders for the code review and to indicate the sensitive variables in the bitcode file.

The *Annotation Pass* uses the annotations marked by the security analysts and adds the correspondent information into the LLVM bitcode file. The information will be used by the following passes, the *Paths Pass* and the *Analysis Pass*. In almost all cases, the security analysts will not mark **all** the sensitive variables in the source code. Therefore, a **taint analysis** algorithm is embedded into the *Annotation Pass*, having the purpose to improve the input for our application.

The *Annotation Pass* is a module pass, meaning that it traverses the entire bitcode of the target program in order to find the annotated sensitive variables, as well as the boundaries between which the analysis is to be performed. The annotations serve as input for the *Paths Pass* and for the *Analysis Pass*. The *Annotation Pass* is the first pass of the applications toolchain, containing 3 parts:

- the search for annotations
- the taint analysis
- the metadata addition

These parts will be explained in detail in this section. The pass seeks annotations that fall into the following categories:

- annotated local sensitive variables
- annotated global sensitive variables
- the start and end points between which the analysis should be applied
- tainted variables from the annotated sensitive variables

The security sensitive variables (that can be also arrays or class members) are annotated as *sensitive* by the security analysts. Because the input given by the analysts can be incomplete, it is extended by the tainted variables provided by the taint analysis. For example, the variables are annotated in the target program by using attributes in the case of C/C++ target programs and by using built-in annotations in Java marked by @ (an example is found in *Section 4.2.8.*). For example, in a C++ target program, a variable is annotated as:

```
__attribute__((annotate(DS))) int a;
```

The annotation value (*DS* in the above example, which stands for *data sensitive*) is customizable. However, this customization should not affect the other passes from the toolchain. Therefore, the annotated values are transmitted to the other passes by adding LLVM metadata to instructions. The metadata provides coherence between the passes, by using the same string fields (for example, *start* for the starting point of the analysis) that are added to the bitcode.

In LLVM, the annotations are contained into the:

- *@llvm.global.annotations* intrinsic, which stores the annotations regarding the global variables
- *@llvm.var.annotation* intrinsic, which stores the annotations regarding the local variables
- *@llvm.ptr.annotation.p0i8* intrinsic, which stores the annotations regarding the arrays

In order to access the local variable which was annotated, the application uses the functions that are corresponding to local iterators through the module. In the case of global variables, the global iterators are used. Therefore, the *Annotation Pass* contains two similar searching loops for annotated variables.

For an annotated variable, the specific annotation intrinsic is called with a number of operators. The most important operator gives the references to the structures that contain the proper string annotations (example: *@str*):

```
@llvm.global.annotations = appending global [1 x { i8*, i8*,
 i8*, i32 }][{ i8*, i8*, i8*, i32 } { i8* bitcast (i32*
 @balance to i8*), i8* getelementptr inbounds ([3 x i8]*
 @.str, i32 0, i32 0), ..... }], section "llvm.metadata"
```

In this example, the global variable *balance* is annotated as sensitive with the string (*DS*) found in the *str* structure. We can see an example of a C/C++ annotation above. We show below the LLVM structure *@.str* that stores the *DS* string (the mark for *sensitive*).

The proper annotations are declared as constants, containing among their operands the string used to annotate the specific variable:

```
@.str = private unnamed_addr constant [4 x i8] c"DS\00",
 section "llvm.metadata"
```

As mentioned above, for each annotated variable, metadata is added to transmit the *sensitiveness* information to the other passes. The LLVM metadata can be added to instructions.

In the case of local variables, all the *alloca* instructions for adding metadata are identified. The *alloca* instructions are used to define a local variable. However, in the case of global variables there are no allocation instructions. Therefore, all the *load* instructions which correspond to global variables are identified, in order to carry the metadata. If no *load* instructions are available, the *Annotation Pass* searches for a suitable instruction to carry the metadata. The search is done by a recursive identification function, which is looking in-depth for the suitable instruction to carry the metadata. A **suitable** instruction to carry the *sensitive* metadata for a global variable is an instruction that has one operand referring the global variable and that has not any annotation metadata

attached to it (another *sensitive* metadata, *start*, *end*). For e.g., if we attach the *sensitive* metadata to an instruction that already has attached the *start* metadata, that the *start* field is replaced by the *sensitive* metadata field. References to the correspondent *load* instructions (or a suitable equivalent) are contained in the operands of the global variable.

The next step is to identify the calls to the annotation intrinsics. An intrinsic function is a function that substitutes a sequence of automatically generated instructions by the compiler. Our code review application uses the annotation intrinsics in order to find the sensitive variables, the *start* and the *end* points **marked** by the analysts inside the source code. We mentioned above the annotation intrinsics used by LLVM: *@llvm.global.annotations*, *@llvm.var.annotation* and *@llvm.ptr.annotation.p0i8*. Our initial assumption that the LLVM annotation intrinsic calls are always in the entry block of the function is wrong. Smartcard programmers can declare sensitive variables in any other basic block of the function as well. Therefore, the *Annotation Pass* analyzes all the *call* instructions from the function. The operands of a *call* instruction are the callee function arguments, followed by the function name. Hence, the called function is obtained. If it is one of the intrinsics, then the first operand of the *call* function is the annotated value. The second operand is the proper annotation. Using different casting operations, the annotated values are obtained, together with the annotation string. In the case of the global variables, the *Annotation Pass* implements synchronization between the operands of the *load* instructions (or equivalent) and the operands of the annotated variables. The synchronization was done using an array to store the positions of the *load* instruction (or equivalent) references among the global variable operands.

For each identified annotation, the correspondent metadata (*sensitive, start, end*) is added. The *Annotation Pass* gets a pointer to the LLVM context associated with the current function, that will be added of the new created metadata node. The metadata is added to the correspondent instructions like:

- *alloca* instructions for the local variables
- *alloca* instructions for the *start* and *end* points
- *load* instructions (or equivalent; example: *store*) for the global variables
- *alloca* instructions in the case of tainted variables which are locally defined
- *load* instructions (or equivalent) in the case of tainted variables which are globally defined
- *GEP* instructions in the case of arrays (local or global)
- *call* instructions in some cases of variables which are part of indirect dependencies (example: in *Section 4.2.6.*)

The metadata is set to the created metadata node. It updates or replaces the already present metadata.

Initially, the annotations were not translated into the LLVM bitcode. Hence, the LLVM compilation with debug information enabled was required.

As we mentioned in *Section 2.3.5.*, the *Annotation Pass* implements a taint analysis algorithm. Taint analysis plays an important role in providing the input for the application developed for this thesis work. The users of our application are the security analysts who try to identify the vulnerabilities on the smartcard and based on them, they attack the smartcard using fault injection. The purpose of taint analysis is to complete the set of sensitive variables that are provided to the application.

For a trivial taint analysis example, we refer the reader to *Section 3.6*: in *Figure 3.7*, the variable *pinVerified* is influenced by the *pin* sensitive global variable. By performing a taint analysis on the target program, the variable *pinVerified* will be considered also sensitive, despite not being initially marked by the security analysts.

The task of marking the sensitive variables inside the target program falls to the security analysts. Since usually the input provided is not sufficient for our application to identify all the possible vulnerabilities, a taint analysis algorithm is contained inside the *Annotation Pass*. Even if the input improvement is not the scope of this thesis, we made research in that area. In the *Future work* section it is presented a taint analysis based on symbolic execution. The reason why our taint algorithm implemented inside the *Annotation Pass* is limited is that we assume we cannot covers all the cases and this problem can be solved by the taint analysis based on symbolic execution. The limits of our implemented taint analysis can be seen in *Section 4.2.15*, *Section 4.2.5* or *Section 4.2.11*.

The taint analysis investigates how the *sensitiveness* is propagating from one sensitive variable to other variables. A level of **taint propagation** denotes the following situation: if we have a pair of variables *(a,b)* with *a* being sensitive and *a* influences the value which *b* can take, then *b* is tainted from *a* and therefore it is considered sensitive as well. A n-level of taint propagation means that: if we have a n-tuple of variables *V1, V2, ...Vn* with V1 being **annotated** by the analysts as sensitive and having the chain *V1 influences V2, V2 influences V3, ...Vn-1 influences Vn*, then V2 is tainted from V1 and therefore considered sensitive; then, V3 is tainted from the V2 which is now considered sensitive; ...Vn is tainted from Vn-1 which is considered sensitive. The implemented taint analysis algorithm for the *Annotation Pass* works for the 1st level of taint propagation. An example can be seen in *Section 4.2.10*. The embedded **taint analysis algorithm** from the *Annotation Pass* is a searching in-depth function. The taint algorithm identifies the conditional branches which are using sensitive variables in their guards. Then, the algorithm searches for the variables which will inherit the sensitiveness in all the succesor basic blocks. The variables involve in the algorithm can be locals, globals, arrays, members of a class.

However, the tainted variables from a n-level of propagation can still be identified by our application if the variables are in a specific order in the program flow. This situation can be seen in *Section 4.2.13.* Another order can make the taint algorithm to not identify all the tainted variables, as explained in *Section 4.2.13*. We did not improve the taint algorithm because we decided to use symbolic execution. The approach using

symbolic execution is described in the **Future work** section. However, an optional recursive in-depth function can be added to the *Annotation Pass* to solve the n-level taint propagation. We set a limit l on the propagation level. The function calls our taint algorithm l times, everytime the list with the sensitive variables being updated.

Still, we identified 2 situations when the *Annotation Pass* is not identifying variables that are not marked as sensitive, but are part of security checks. The **first situation** is presented in *Section 4.2.12.*, when the security analyst is not annotating as sensitive a variable which influence the values of multiple other values used in security checks, the result being a cascade of unidentified dependencies. Currently, we assume that the security analyst has the **responsability** to give a satisfactory input (necessary for the taint analysis to give the necessary results) to our application. The **second situation** is illustrated by the fact that the *Annotation Pass* does not identify other cases of *sensitiveness* propagation (which is **not** taint propagation). Examples can be see in *Section 4.2.11.*, in *Section 4.2.15.*, the alias analysing from *Section 4.2.5.* Therefore, we plan to use symbolic execution for the *sensitiveness* propagation as a future work.

An interesting observation can be made when searching for the instructions to carry the metadata. In some cases which are not common, we can have the situation presented below:

```
On the paths:    %retval = alloca i32, align 4, !path !928
0 1 10 11 18 19 28 29 36 37 46 47 54 55 64 65 72 73 82 83
90 91 100 101 108 109 118 119 126 127 136 137 144 145 154
155 162 163 172 173 180 181 190 191 198 199 208 209

On the paths:    %retval = alloca i32, align 4, !path !928
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62
63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82
83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101
102 103 104 105 106 107 108 109 110 111 112 113 114 115 116
117 118 119 120 121 122 123 124 125 126 127 128 129 130 131
132 133 134 135 136 137 138 139 140 141 142 143 144 145 146
147 148 149 150 151 152 153 154 155 156 157 158 159 160 161
162 163 164 165 166 167 168 169 170 171 172 173 174 175 176
177 178 179 180 181 182 183 184 185 186 187 188 189 190 191
192 193 194 195 196 197 198 199 200 201 202 203 204 205 206
207 208 209 210 211 212 213 214 215
```

Apparently, an *alloca* instruction receives metadata with paths information and then the metadata is replaced with other metadata with paths information. Still, none of the two paths is lost. The reason is that the two *alloca* instructions from above are different instructions inside the LLVM bitcode. They share only the common assembly representation. Thus, when the *Analysis Pass* is searching for the instructions, they are contained in different basic blocks. In our example from above, the first *alloca* instruction is contained inside the *entry* basic block from the *_Z9RSAdecryptli* function (RSAdecrypt

in the source code), while the second instruction is contained inside the *entry* basic block from the *main* function.

For output examples, we refer the reader to *Chapter 4*.

3.3 Paths Pass

The *Paths Pass* is the second tool of our application. We can observe in *Section 2.3.6.* that the pass receives as input the LLVM bitcode file which is the output of the *Annotation Pass*.

The *Paths Pass* uses the metadata added by the *Annotation Pass* and computes all the possible execution paths between the start and the end point. The *Paths Pass* would transform the LLVM bitcode file, **marking** for each execution path the contained basic blocks. In contrast, we can see for each basic block in which paths it is contained. The metadata added by the pass is going to be used by the *Analysis Pass*, which is identifying the smartcard vulnerabilities. The **goal** of the *Paths Pass* is to select all the possible execution paths on which our application will individually apply the code review to find vulnerabilities.

The *Paths Pass* is composed of 4 parts:

- the mapping of the basic blocks from the smartcard target program into an oriented graph
- the algorithm for identifying all the paths in the oriented graph
- the generation of the set of execution paths
- the metadata addition for each basic block; the *Paths Pass* adds for each basic block the corresponding execution paths it belongs to

A structure *BBnode* is used to keep the name of a specific basic block. The basic block names are unique inside a function, but may conflict in the context of the whole module. In order to avoid any conflicts, *BBnode* keeps also the function name. A worklist is used to keep all the *BBnodes*.

The *Paths Pass* uses the LLVM method *getAnalysis<CallGraph>()* to get the call graph of the smartcard target program.

Working directly with basic blocks as nodes in the graph is very costly since the basic blocks are complex structures. Therefore, we adopted the optimal approach of mapping the basic blocks to integers. The number of the nodes is computed when traversing the graph to get the *BBnodes*.

The next step is to construct the edges of the graph. Since we took the decision to use our own graph with integers (not to work directly with LLVM basic blocks, as mentioned above), we need to build the edges between the integer nodes. Therefore, our application identifies how the LLVM basic blocks are linked inside the bitcode file

and based on this we build the edges for our graph. Our application uses two types of edges: links between functions and links between basic blocks inside functions. For the first type of edges, the *Paths Pass* identifies the connections of the call graph. Hence, pairs of (*Callee*, *Caller*) basic blocks between the functions need to be identified. For the second type of edges, the successors of every basic block are identified. Therefore, the *Paths Pass* adds connections (inside the graph with integers that the *Paths Pass* builds) between the basic blocks from the same function. Basically, the *Paths Pass* is constructing the control-flow graph for every function.

For the identification of all paths of the graph, the *Paths Pass* implements the *getAllPaths* method which is based on a *Depth-first search algorithm* (DFS). The *getAllPaths* method is presented in the pseudocode from *Figure 3.2*. The *Paths Pass* explores all the execution paths from the target program control flow and deals with the potential recursivity problems. A defined stack is used to push and pop the nodes from a work vector. A *Graph* class is defined, which contains methods like:

- verification if two nodes are connected
- edge addition to the graph
- verification if a node is a leaf
- *getAllPaths* method :

  ```
  int** getAllPaths(int, std::vector<int>&, int**, int*)
  ```

  having the arguments (in order):

  - the required node
  - the visited vector which is transmitted by reference
  - the integer matrix used to store the set of execution paths
  - the number of columns of the matrix, which is transmitted by reference

The *getAllPaths* uses a recursive algorithm. The required node (correspondent to the *end* node marked by the analysts) is marked as visited before calling the *getAllPaths* method. The *getAllPaths* method does the analysis between the *start* and the *end* points annotated by the security analysts. The instructions carrying the metadata information containing these points (received from the *Annotation Pass*) are identified in the code. The *start* and the *end* points correspond to two different basic blocks from the graph of the target program. Therefore, the *getAllPaths* method will consider only the program flow between those two basic blocks when it identifies all the possible execution paths.

We use a recursiveness level to **limit the number of recursive calls inside the graph**. In *Chapter 4*, the test-suite was tested with the level 2 of recursiveness. However, the tests on the application passed with a recursiveness level up to 4. The *getAllPaths* method uses the *visited* vector to keep the identified paths. When the end node is found,

```
current node extracted from the visited vector
for all its connected neighbors
if a neighbor is not visited
      check if required => complete the matrix with the specific path on the
                           corresponding row
      if not required => insert the neighbor in the visited vector
                  => call getAllPaths recursively
return the completed matrix
```

Figure 3.2: Pseudocode of the getAllPaths method

the path (the current *visited* vector) is stored into the result matrix. After a recursive call of *getAllPaths*, the *visited* vector and the recursiveness level are cleared.

The pseudocode presented above is implemented in C++.

The *Paths Pass* adds the paths metadata information for the *Analysis Pass*. A loop iterates over all the basic blocks contained inside the identified execution paths. For the current basic block, the *Paths Pass* searches the suitable instruction to carry the path information metadata in a similar way as the *Annotation Pass* does (described in *Section 3.3*). It is very common that a basic block is part of multiple execution paths. Each path is **mapped** to an integer, so the metadata sent to the (Analysis Pass) is reduced in size.

This mapping on paths is **different** from the mapping on basic blocks done by *getAllPaths*. Our application implements **2 mappings** involving integers. **One mapping** is used by the *getAllPaths* method that is implementing an algorithm to search for all the possible execution paths inside the target program. The algorithm analysis a graph of integers, with each integer being an **identifier for a basic block** from the bitcode file. The **second mapping** is used by the *Paths Pass* to add the metadata with path information for the *Analysis Pass*. When *Paths Pass* adds metadata, it uses an integer **identifier for every path** that is added to the metadata. The security analyst can see the paths composition as an output of the *Paths Pass*. Therefore, all the path matrix does not need to be passed to the *Analysis Pass*. It is more efficient (and less in size) to send only the paths identifiers to the *Analysis Pass* that will use them. The *Analysis Pass* needs only the paths identifiers in its processing. Therefore, using the metadata sent by the *Paths Pass*, the *Analysis Pass* can check for each basic block in which paths it is contained. Therefore, the *Analysis Pass* knows the security checks (which are formed from several instructions of the basic block) from a specific basic block in which paths they are contained. The dependencies involving the security checks are correspondent to the smartcard vulnerabilities. Therefore, we can see for **each vulnerability in which execution path it is contained**. However, as we will see in the next section, the *Analysis Pass* provides the results in **differently**, meaning that the *Analysis Pass* shows separately **for each execution path which vulnerabilities it contains**. This process will be explained in *Section 3.5.*

The composition of the paths represents the output of the *Paths Pass*. We refer the

reader to *Chapter 4* to see the examples.

3.4 Analysis Pass

The *Analysis Pass* is the third tool of our application. It is introduced in *Section 2.3.7*. The *Analysis Pass* receives as input the LLVM bitcode file which is the output of the *Paths Pass*. The output of the *Analysis Pass* gives the results of our application. Based on the results, the security analysts identify the **points** where to insert fault injection in the smartcard target source code. The scope is to exploit the founded smartcard vulnerabilities.

The *Analysis Pass* uses the metadata added by the *Annotation Pass* for identifying the sensitive variables. The *pass* identifies the dependencies between the security checks from the smartcard and the sensitive variables. It uses the metadata added by the *Paths Pass* to get the execution paths. The *Analysis Pass* identifies separately the dependencies found on each path. Based on the results, the security analysts know the places where to insert fault injection for each possible execution path.

The *Analysis Pass* is the final tool of the our toolchain. It contains the core of the application, the actual analysis part of the vulnerabilities that can be further exploited by fault injection. The *Analysis Pass* produces the results that indicate the places where to insert fault injection in order to bypass the security checks from the target program.

The *Analysis Pass* contains 2 different parts:

- main (the *runOnModule* method) that gives the final results
- dependencies identification (6 recursive methods) that does the metadata processing (the inputs from the previous passes)

All the parts of the *Analysis Pass* will be explained in detail in the following paragraphs.

3.4.1 Main method

The main method is the core of the *Analysis Pass*. This method gives the final results of our application. All the needed methods used to identify the dependencies are called from this method. Furthermore, the metadata from the other passes is handled by the *Main method* and the results are processed in the final format. The method is used to traverse the entire target bitcode. Therefore, the *Main* module method uses a loop block that iterates over every function from the target bitcode. The reason for using a module pass (instead of a function-by-function pass like *runOnFunction*) is given by the different execution paths that the target program can take. Concretely, the analysis should be applied each time only for one execution path. When the fault injection is inserted into the target program source code, only the vulnerabilities for the current execution path are taken into account. Therefore, the vulnerabilities are correspondent to each execution path received from the *Paths Pass*.

The results are stored under a defined format given by a dependency between a conditional branch instruction and a sensitive variable. A **result** of the *Analysis Pass* has the following fields:

- the conditional branch (for example, an *if* condition) contained in the security check
- the place where to insert fault injection (the line number of the instruction to be avoided from the security check; the instruction can be the conditional branch/*return BAD* statement etc.; examples are found in *Chapter 4*)
- the original sensitive variables on which the conditional branch is dependent (we are not interested in registers that are used by the guard of the conditional branch or by the aliases of the original sensitive variable, but in the original sensitive variable itself)
- the place where the original sensitive variable is declared (the line number in the source code)

The results will be distributed to each execution path. Each path received via metadata from the *Paths Pass* is memorized in an array of arrays of variable length. The results are stored in an array with elements of *result* defined-type of variable length.

The communication with the dependency identification methods is done inside a loop block, which iterates over every function of the module. A conditional branch can be dependent on more than one sensitive variable. For a particular conditional branch, the same sensitive variable can be identified multiple times. An example when this case is possible (valid for analogous examples): a conditional branch can be directly dependent on a sensitive variable and can call a function, which is using the same sensitive variable. In order to avoid the redundant results, the *Analysis Pass* uses an array of *StringRef* type elements (StrinfRef is the LLVM *string* version) that keeps track of the already identified sensitive variables.

For every function of the module, all the conditional branches are identified in order to check if they are part of a security check. An example of typical LLVM conditional branch is:

```
%cmp21 = icmp sgt i32 %call20, %5, !dbg !958
```

In the example from above, the name of the conditional branch is *%cmp21*. The type is *icmp*, which stands for Boolean comparison operation between 2 structures that do not involve floating-point operands (for example, it can be the LLVM operation correspondent to an *if* operation in the source code). *Sgt* is the condition code that represents the *signed greater than* operand. The *%call20* operand is a call to a function (whose result will be used in the *icmp* comparisson) and %5 stands for a pointer to a variable. The instruction contains debug information.

For each identified conditional branch, the correspondent dependencies are sought.

Consequently, the link with the **dependency identification** (which is presented in the next subsection) is done.

We present this paragraph in more detail because the *Analysis Pass* uses **analogous** methods (analogous to the first method that we will describe, that is *partialResLoad*). All the pointers/variables/methods on which a conditional branch is dependent are identified. This is done by getting all the *load* instructions on which the conditional branch is dependent. The *partialResLoad* method is called, which is used to identify for a given conditional branch all the *load* instructions on which it is dependent. In order to call a method of type *opChecker* (the class for our dependency recursive search methods), an object of type *opChecker* is created. The *partialResLoad* method receives as parameters: the current conditional branch instruction, a synchronization variable and an array that is keeping the partial results during recursive calls. The received results fill an array that has its elements synchronized with each conditional branch. The reason for using this synchronization mechanism is to map the corresponding set of results for each conditional branch.

The *partialResGEP* method is the analogous to *partialResLoad*. It identifies all the arrays on which a conditional branch is dependent. The difference is that the *Analysis Pass* gets all the *GEP* instructions instead of *loads*. The *Analysis Pass* marks as results only the dependecies that involve sensitive arrays.

The first dependency identification method is *resLoadAlloca*. The mechanisms used are similar to the ones used in the case of *partialResLoad*. For each *load* instruction identified by *resLoadAlloca*, the correspondent set of local variables that are marked as *sensitive* by the *Annotation Pass* are received. *Sensitive1* is the metadata used for global variables, while *sensitive* is used for local variables. The results are synchronized using the same mechanism as described in the case of *partialResLoad*. The same array is used for the results received with local variables and with global variables, for a more efficient synchronization with the current *load* instruction. Consequently, the synchronization between the *resLoadAlloca* and *resLoadGlobals* methods is implemented. The *resLoadGEP* method is analogous with the *resLoadAlloca* and *resLoadGlobals* methods. The difference is that it identifies the dependencies that involve arrays. The *Analysis Pass* considers as results only the dependecies which involve sensitive variables/arrays.

An example of the definition of a local variable is:

```
%secure_flag = alloca i32, align 4, !sensitive !929
```

The local variables in LLVM are declared using the *alloca* instruction. LLVM allocates memory on the stack of the currently executing function and it returns a pointer of the appropriate type to the program. In the example from above, the local variable *secure_flag* of type i32 was marked as *sensitive* by the *Annotation Pass*.

A similar example for a global variable is given below:

```
@unlock = global i32* null, align 4
```

As we mentioned in detail in the *Annotation Pass*, the LLVM metadata cannot be added to global variable definitions. Hence, the metadata *sensitive1* corresponding to

the global variables is added to the *load* instructions (or equivalent instructions which are suitable to carry the metadata) that reference the global variables. In the example from above, the *unlock* global variable is declared. The corresponding *load* instruction which carries the *sensitive1* metadata is:

```
%11 = load i32* @unlock, align 4, !dbg !962, !sensitive1 !963
```

Another dependency identification method is *resGlobalsFunctionsSensitive*. It works in the same manner as described in the case of *partialResLoad*. For each conditional branch, the *Analysis Pass* identifies the correspondent set of global variables that are marked as *sensitive1* by the *Annotation Pass*. It used to search in-depth for global sensitive variables which can be used in a function called by the conditional branch. However, the function that is using the sensitive variable may be called by other functions, in *cascade* (as we mentioned in *Section 2.3*). Hence, the search is made recursively, as we present in *Section 3.5.2.*. This type of dependency is an inter-procedural dependency. An example is presented in *Section 4.2.3*.

The *Analysis Pass* identifies the sensitive local variables on which the conditional branches are dependent in the same manner as in the previous paragraph. In this case, the used recursive method is *resAllocasFunctionsSensitive*. Analogous to *resGlobalsFunctionsSensitive* and *resAllocasFunctionsSensitive*, the *resGEPsFunctionsSensitive* method identifies the sensitive arrays on which the conditional branches from the security checks are dependent.

The indirect dependecies are also identified by the *Analysis Pass*. The indirect dependencies involves conditional branches that call functions that have sensitive variables/arrays as operands. The indirect dependencies are presented in *Section 4.2.6*.

The *Analysis Pass* implements a mechanism to detect the duplicated results. If the result is not a duplicate, it is marked as visited. When a result is printed: the conditional branch name of the security check is printed, together with the line number where the branch is in the code. The line number is printed using the *getLineNumber* method of the *Analysis Pass*. Then, the *Analysis Pass* prints the sensitive variable on which the conditional branch is dependent and the line number where the variable is defined. In this case, the *findDbgDeclare* method of the *Analysis Pass* is used to get the line number of the definition.

The *Analysis Pass* gets the correspondent basic block of the conditional branch and it checks on which execution paths is found, based on the input received from the *Paths Pass*. Therefore, the *Analysis Pass* prints for each execution path which results it contains.

We can encounter the case when some identified dependencies (by the *Analysis Pass*) are not printed as results by our application. The reason is that they are not contained on any execution path between the points chosen by the security analysts. Still, the *Analysis Pass* has the option to print even the depedencies which are not found on any exection path. They are printed toghether with the message *For paths, choose other start and end points*.

Next we present the resursive methods (of the *opChecker* class which is implemented by our application) used by the *Analysis Pass* to identify the dependencies between the

conditional branches and the sensitive variables. We refer the reader to *Chapter 4* for examples.

3.4.2 Dependencies Identification

The *Analysis Pass* contains 8 recursive methods that are used to identify the dependencies which involve the conditional branches which are based on sensitive variables. All the 8 methods were **mentioned** in the previous section. The sensitive variables are initially marked by the security analysts or they are tainted variables, which are considered to be sensitive due the fact that their values are influenced by sensitive variables.

An example of the **prototype** of a in-depth recursive method used by the *Analysis Pass* to identify dependencies is:

```
Instruction** opChecker::methodName(Instruction *I, int* resSize,
Instruction** localPartialRes)
```

As we can observe, the method returns a matrix of the LLVM *Instruction* type, which is used for **local** variables. In the case of **global** variables, the *GlobalValue* LLVM type is used. The class which contain the method is called *opChecker*, which is followed by the name of the method. The first argument *I* is a pointer to the instruction which is representing the conditional branch. The second argument *resSize* is an array used to synchronize the results between the conditional branches. The third argument *localPartialRes* is the matrix which stores the results that will be sent to the *Main method*.

We can see an **example** of the structure of a recursive function of the *resGlobals-FunctionsSensitive* method (described above, the method gets all the inter-procedural dependencies involving sensitive global variables) in the pseudocode from *Figure 3.3*.

The first method used is *partialResLoad*. This paragraph is described in **more detail** because there are 7 **analogous** methods. This method is used to identify all the *load* instructions on which a conditional branch is dependent on. It receives as arguments the current conditional branch, the synchronization variable and the array that keeps the partial results. The method takes all the operands of the conditional branch instruction in order to identify a reference to a *load* instruction. If found, the method stores it in the partial results array and the synchronization variable correspondent to the current conditional branch is updated. Otherwise, a recursive call of the method is searching in depth for *load* instructions. The results for the current instruction are returned to the *Main method*.

After the *load* instructions (which are operands of the conditional branches) are identified by the *partialResLoad* method, the *resLoadAlloca* method gets all the *alloca* instructions that an identified *load* is dependent on. The *alloca* instructions are used to define local variables. The *Analysis Pass* checks if the local variables are sensitive. If they are sensitive, then the found dependency is a vulnerability and mark as a result by our application. However, the current conditional branch can be dependent on sensitive global variables as well. The *resLoadGlobals* method is the analogous method

```
get the current conditional branch
foreach operand of the conditional branch
  if it is a call
    get the called function
    foreach BB from the function
      if it is a load (or equivalent)
        if it has sensitive1 metadata
          if the first metadata operand
                        is global
            insert into results
            increment synFlag //the current
            //conditional branch has one more result
    .....
  else
    aux is the current instruction
    resGlobalsFunctionsSensitive (aux,
           synFlag, localPartialResults)

return the results for the
current conditional branch
```

Figure 3.3: Pseudocode of the resGlobalsFunctionsSensitive method

used for sensitive global variables. The same array is used to store the results with local variables and with global variables, for a more efficient synchronization with the current *load* instruction. Consequently, the synchronization between the *resLoadAlloca* and *resLoadGlobals* methods is implemented.

The fourth method is *partialResGEP*. The method is similar with the *partialResLoad* method and it is used to identify all the *GEP* instructions on which a conditional branch is dependent on. Based on the identified *GEP* instructions, the *resLoadGEP* gets all the *load* instructions the *GEP* is dependent on. The *resLoadGEP* method is similar with *resLoadAlloca* and *resLoadGlobals*. For the current conditional branch, it searches the sensitive array on which the branch is dependent.

The *resGlobalsFunctionsSensitive* and the *resAllocasFunctionsSensitive* methods are used to identify the inter-procedural dependencies mentioned in *Section 3.5.1.*. In this case of dependency, the current conditional branch is dependent on a sensitive variable which is used in other function. The conditional branch calls a function, that might call another function that is using the sensitive variable. Therefore, the *resGlobalsFunctionsSensitive* and the *resAllocasFunctionsSensitive* methods search for a in-depth in a *cascade* of functions for sensitive variables. The *resAllocasFunctionsSensitive* method identifies the dependencies that involve sensitive local variables, while *resGlobalsFunctionsSensitive* identifies the dependencies that involve sensitive global variables. Analogous with *resGlobalsFunctionsSensitive* and *resAllocasFunctionsSensitive* is the *resGEPsFunctionsSensitive* method, which is used to identify the inter-procedural dependencies that involve sensitive arrays.

The *Analysis Pass* uses the method *getLineNumber* to get the line number for a

conditional branch. The conditional branch is dependent on sensitive variables. The *findDbgDeclare* method gets the line number of the definition of a local sensitive variable, while the method *findDbgGlobalDeclare* gets the line number where a global variable is defined. The methods use the LLVM structure *llvm.dbg.cu* which keeps debug information. From the LLVM structure, *Analysis Pass* gets the debug metadata nodes, that are used to characterize LLVM descriptors (DIDescriptor). The *Analysis Pass* gets the needed information from the operands of a descriptor. We are interested in the 8th operand, which gives the line number of the definition of the global variable.

3.5 Proof-of-Concept Example

This section presents the functionality of our application on a trivial target program example. For more complicated examples, we refer the reader to *Chapter4*. This example is given after each tool was presented. The **reason** for choosing this approach is given in *Section 3.2*, where the target program example is introduced. *Section 3.2* provides **explanations** of the statements from the source code. An example is the *return BAD* statement, which can disable the functionality of the smartcard.

As we can see in *Figure 2.4* from *Section 2.3.4.*, the target program is compiled into a bitcode file. The file serves as input for the *Annotation Pass*. After the *start* point, the *end* point and the sensitive variables are identified, a new bitcode file is produced. This new bitcode file serves as input for the *Paths Pass*. The output is a new bitcode file that contains the information regarding the all possible execution paths. This file is the input for the *Analysis Pass* which gives the results. Based on the results, the security analyst knows where to insert fault injection in order to exploit the vulnerabilities on the smartcard.

The target program **source code** is shown in **Figure 3.1** from *Section 3.2*. All the basic blocks are numbered as we can see in the following figures.

The global variables are declared in the starting basic block (number 1). The analysts annotated the *pin* and the *balance* as sensitive variables. Also, the local variable *code* is marked as sensitive. The global variable *pinVerified* is initialized with *BAD*.

Figure 3.4 shows the target program example under the form of a control-flow graph. The annotations in the source code can be observed in the figure. The security analysts annotated the global variables *pin* and *balance* as sensitive (basic block number 1), as well as the local variable *code* from the *locker* function (basic block number 10). The *start* point of the analysis is marked in the *process* function (which was explained in *Section 3.2*) (basic block number 2), with the purpose to make review on all the possible functions (being called from *process*), which may contain sensitive conditional branches. As previously **mentioned** in the *3.2 Target program example* section, the purpose of the attack is to increase the balance on the bank smartcard. Consequently, the *end* point is marked after the increasing operation (basic block number 18).

The *Paths Pass* identifies the possible execution paths between the annotated start and end points. In the case when the chosen option is to verify the pin, the path *2,3,4* is identified. A conditional branch which requires pin checking is encounted in the program flow (basic block number 4). The problem is how to avoid the security check without

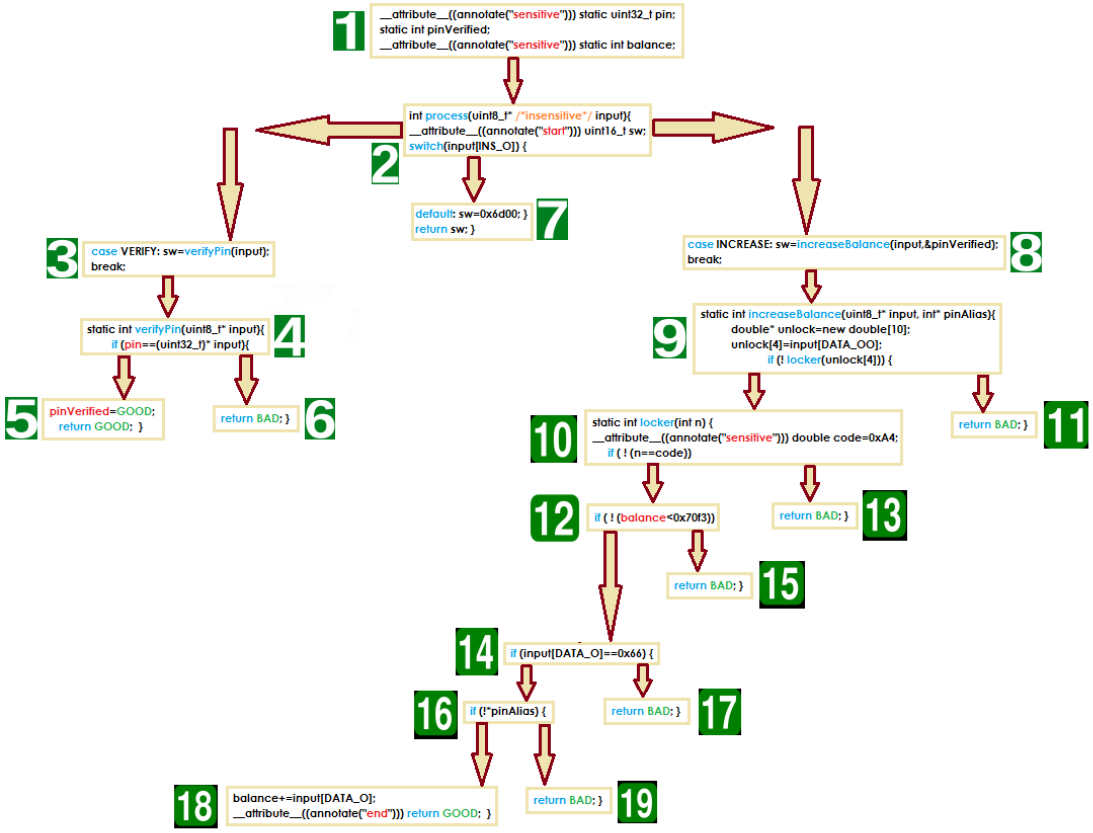

Figure 3.4: Proof-of-Concept Example

knowing the *pin* value. For a successful attack, the execution of the program should take the basic block which is returning *GOOD*. The *GOOD* value is returned to the *switch* condition from the *process* function.

The next step is to check if the encounter security check is a candidate for fault injection. The *Analysis Pass* is identifying a direct dependency between the *if* conditional branch and the sensitive global variable *pin* (shown in *basic block 4*). Based on the provided result, the security analyst has two options. One option is to skip (by inserting fault injection) the *return BAD* statement (**explained** in detail in *Section 3.2* and in the *Chapter4*) which may disable the smartcard functionality if called more than 3 times (if the pin value is wrongly inserted 3 times, the smartcard may stop function) (basic block number 6). In this case, the value of *pinVerified* is not set to *GOOD*, since the flow is taking the *return BAD* branch. The second option is to skip the conditional branch itself (basic block number 4), which leads to the execution of the left-sided basic block, as seen in the following figure. In this case, the function is returning *GOOD* (basic block number 5).

At this step, we assume that the security analyst has chosen the second option (the choice of the first/second option is not relevant, until the *Step 13*, which is discussed

below). The security check was succesfully bypassed (the execution path is *2,3,4,5*).

The program flow takes the *2,3,4,5* path, returning to the *process* function.

The *Paths Pass* has identified the paths containing the *increaseBalance* method. If the user chooses to increase the balance on the smartcard, the program flow encounters multiple security checks. In this example, we assume that the attacker wants to increase the balance on the smartcard (basic block number 18), by exploiting the smartcard vulnerabilities. The first security check is based on the *locker* authorization method (basic block number 9).

At *Step4*, the scope is to bypass the security check mentioned above (basic block number 9). The *Analysis Pass* identifies the dependency between the security check and the sensitive local variable *code*, which is found by an in-depth recursive search in the potential cascade of function calls. Again, the security analyst has two options: either skip the conditional branch which is calling the *locker* function, either enter in the *locker* function (inside the basic block number 10).

Skipping the conditional branch (first option) implies the following instruction *return BAD* to be skipped as well (basic block number 11). The second option implies to enter in the *locker* function (basic block number 10), which contains another security check. The place where to insert fault injection is shown the *basic block 11*.

In the case of the second option, the taken path is *2,8,9,10*. Then, another conditional branch part of a security check is encountered (basic block number 10).

The *Analysis Pass* identifies the dependency between the conditional branch indicated in *basic block 10* and the sensitive local variable *code*. Therefore, the security analyst inserts fault injection to skip the following *return BAD* instruction (indicated in *basic block 13*).

The security mechanisms from the *locker* function are bypassed. The *Paths Pass* identifies the path to continue, which is *2,8,9,10,12*. We can see that another conditional branch is encountered (basic block number 12).

The *Analysis Pass* identifies the same direct dependency as the first one encountered in this example. The dependency between the sensitive global variable *balance* and the conditional branch shown below is exploited as a smartcard vulnerability. Therefore, fault injection is inserted to skip the *return BAD* statement (indicated (basic block number 15).

The conditional branch based on the sensitive variable is bypassed. The program flow encounters another conditional branch (basic block number 14).

The *Analysis Pass* checks the dependency related to the encountered conditional branch. We can observe that the *if* statement is based on the user's input (basic block number 14). The *Analysis Pass* does not consider this dependency a security check, because the conditional branch is not dependent on any sensitive variable. The pass is acting correctly, since this conditional branch is not a smartcard vulnerability (from the *basic block 14*), because the security analyst can control the provided input.

The program flow advance is *2,8,9,10,12,14,16*, when the last conditional branch from our example is encountered (basic block number 16).

The *Analysis Pass* identifies the conditional branch depending on the *pinAlias* variable (basic block number 16). We can observe that the variable is an alias for the *pinVerified* global variable, which is transmitted as a parameter. At *Step3*, we mentioned that

the chosen option will affect the attack on the smartcard. We assume that the security analysts did not marked *pinAlias* as a sensitive variable. However, the *Annotation Pass* identifies the *pinVerified* global variable as being a tainted sensitive variable because its value is influenced by the sensitive global variable *pin* (if the *pin* is correctly introduced, the value of the *pinVerified* flag is changed to *GOOD*) (basic block number 5). As mentioned in *Section 4.2.5.*, the *Annotation Pass* does not found the alias as a tainted variable from *pinVerified* (which is already identified as a tainted sensitive variable). In order to identify this security check, the algorithm described in *Section 4.2.5.* should be integrated in the tainted analysis algorithm inside the *Annotation Pass*. Because this version is not generic, it can only be optionally included. The problem can be solved by making a taint analysis based on symbolic execution, as mentioned in the *Future work* section.

If the optional taint analysis code is **not** included in the *Annotation Pass*, then the succes of bypassing this conditional branch lays in the **choices** made at *Step3*. **If** the security analyst has chosen to insert fault injection to skip the *return BAD* statement, than the value of *pinVerified* is not changed to *GOOD*, so the *pinAlias* will be set to *BAD* and the control flow can end into the *return BAD* operation (correspondent to this step) (basic block number 19). However, **if** the security analyst has chosen to skip the pin verification conditional branch at *Step3*, than the value of *pinVerified* would be set to *GOOD*. Therefore, the *pinAlias* would be set to *GOOD*. We assume that the program flow is taking first the pin verification branch and after that the branch in which the balance can be increased. This assumption is sustained by the logical functionality of a smartcard.

If the security check presented at the previous steps was bypassed, the program flow is succesfully reaching the increase balance operation (as we see in *basic block 18*) (basic block number 18). Therefore, the attack using fault injection was successful and the balance could have been increased by exploiting the smartcard vulnerabilities identified by our application.

3.6 Summary

The third chapter presents in detail the application developed for this thesis. The functionality of each of the 3 tools (the *Annotation Pass*, the *Paths Pass* and the *Analysis Pass*) that form the application is illustrated by a target program example. For more complex program examples, we refer the reader to *Chapter4*. We present for each of the 3 tools their local objectives, their functionality, their implementation and the connections between them or with the external environment.

4 Evaluation

The developed application was evaluated and validated on a custom test suite. We present first this test suite which contains real smartcard test applications, but also specific experimental programs. The second section presents the obtained results, showing the efficiency of finding vulnerabilities for fault injection. The results show how the developed application was validated and gives additional comments of the obtained results.

Riscure provided a test suite, since no benchmark exists [8] for smartcards software. Other test programs were derived from the Riscure patterns. The test suite is based on the assumption that it covers the main part of the smartcard vulnerability types [39].

4.1 Test suite

The test suite is composed of target programs on which the application was tested during implementation and during the evaluation phase. There are two types of test programs:

- Riscure smartcard test programs from training targets
- Experimental programs

The test programs from training targets are smartcard applications that are provided by Riscure. They are meant to illustrate smartcard fault injection vulnerabilities. These programs are used on real devices, which are subject to physical attacks. The physical attacks used by Riscure are various, but they include also fault injection, which is the relevant attack for this work.

The test suite contains experimental programs as well. The purpose of these programs is to cover the test cases that are not present in the Riscure real smartcard test programs, but might occur in other smartcard applications. The experimental programs cover all the dependency types described in *Chapter 3*, but also cases when the vulnerabilities are not identified by our application. Furthermore, the relevant Riscure patterns are included into the test programs.

The test suite programs cover the most common smartcard programming languages, including C++, Java, C. For the evaluation of our application, the types of vulnerabilities covered are **more relevant** than the number of different programming languages in which the target programs software is written. The testing methodology follows the steps described in *Section 1.2.2*. A description of the steps is presented in the following paragraphs.

The security analysts mark the sensitive variables inside the target program. Because usually the input is not sufficient for the analysis to identify all the possible vulnerabilities, a taint analysis was attached to the *Annotation Pass*. Even if the input improvement

is not in the scope of this thesis, we briefly investigated how the input accuracy could be improved. In *Section 3.3.* and in the *Future work* section, the taint analysis based on symbolic execution is presented, which can be used efficiently to improve the input initially provided by the security analysts. The taint algorithm included into our application is not covering all the taint propagation flow, as mentioned in *Chapter3* and illustrated by several examples in the next sections.

The next step is to compile the target programs, using LLVM specific front-ends: Clang is used for the C target programs, Clang++ is used for the target programs written in C++, VMKit is used for the Java/Java Card programs. The optimization level used is *-O0*, so the front-end compiles the source code in the most straightforward way possible, performing no optimization. This is the best option to consider, since the developed application will perform analysis and transformations on the resulting LLVM bitcode, taking into consideration all the source code data (e.g. if we would set the *-Os* optimization level, Clang would perform dead code removal; smartcards use redundant security checks based on conditional branches; therefore, some equivalent conditional branches could be removed by Clang; the result leads to the fact that the *Analysis Pass* would not identify a security check because it was removed by the compiler; the security analysts want to have all the security checks identified). All the statements from the target source code are converted into the corresponding instructions without rearrangement or code redundancy removal. For example, we want to avoid the situations when the compiler considers two conditional branches redundant and eliminates one of them. Therefore, we want to analyze all the security checks in order to avoid the case when a vulnerable point to insert fault injection is not taken into account (such a security check as we already mentioned in this paragraph), even if the developed application is properly functioning.

The communication between the *passes* that form the toolchain is made through the usage of metadata. Therefore, the source level debug information should be generated.

This is done by adding the *-g* flag when compiling the target programs. The *-emit-llvm* flag is also needed, in order to be used the LLVM representation for the assembler and object files. In order to run only the preprocess, the compile and the assemble steps needed, the *-c* flag is used. The *-o* flag is used to write the output to the resulting LLVM bitcode file (for example: *Test.bc*). An optional step can give the resulting bitcode into a human-readable LLVM assembly form. For this, the *llvm-dis* disassembler is used.

Based on the bitcode generated from the target program, the *Annotation Pass* is producing a new bitcode file, which serves as input for the *Paths Pass*. Its output is a bitcode file that represents the input of the *Analysis Pass*, which produces the final results. The results can optionally be in the form of a new LLVM bitcode file.

The test suite is composed from 12 target programs. Since there is no benchmark suite available for smartcard programs [8], we tried to cover the majority of the cases that Riscure is dealing with. In the next subsection, the cases corresponding to different types of fault injection vulnerabilities are presented. The different types of vulnerabilities to

FAULT.BRANCH	1
FAULT.DETECT	2
FAULT.CONSTANT.CODING	3
FAULT.DOUBLECHECK	4
FAULT.CRYPTO	5
FAULT.BYPASS	6
FAULT.DEFAULTFAIL	7
FAULT.FLOW	8
FAULT.RESPOND	9
FAULT.LOOPCHECK	10
FAULT.DELAY	11

Table 4.1: Riscure patterns

be found by our application are corresponding to different security mechanisms included by the developers into the smartcard source code. The test target programs include real smartcard and experimental test programs, written in C, C++, Java.

In *Table 4.1.*, we present the identification number for each of the Riscure patterns. The numbers are used in *Table 4.2.* that presents the test-suite.

In *Table 4.2.*, we can see how the Riscure patterns from *Table 4.1.* are applied inside the tests from the test-suite.

The test-suite is summarized in the table. We can see on the first row the name of the test and the type of the test program. On the second row, we can see the Riscure patterns that are included into the test program.

The **scope** of the *Evaluation chapter* is to present all the cases that were encountered while we were testing our application. In the following sections, we present all the cases encountered that were related to different types of **vulnerabilities**.

Please note that the purpose of the fault injection attack is to make a specific piece of code to return *GOOD*. This means that a particular security check from the piece of code is avoided and harming actions can be taken, such as increasing the balance on the card or withdrawing money. If the piece of code is returning *BAD*, than a *faultDetect()* procedure (or a similar procedure) is triggered, disabling the functionality of the smartcard.

Please note that the *return GOOD* statement is just an example, other similar success actions (in attacking the smartcard) will be presented in the following cases.

The *return BAD*-like statements (the *return BAD* format is used in **our** test suite; the statement can have other form in smartcard programs, but the functionality is the same with *return BAD*). An example of detecting these fail statements for Java Card programs is by looking for *ISOException*. The *ISOException* is the most common way to end a process on Java Card programs. Other mechanisms are represented by the flag status codes defined in ISO7816. These statements can easily by detected by the *Analysis Pass*, but the results can lead to numerous false positives as explained in the following.

Test01	experimental smartcard test program
Patterns	1; 2; 3; 4
Test02	experimental smartcard test program
Patterns	1; 2; 3; 4
Test03	smartcard test program from training target
Patterns	1; 3; 5
Test04	smartcard test program from training target
Patterns	1; 3; 5; 6; 7
Test05	smartcard test program from training target
Patterns	1; 3; 5
Test06	smartcard test program from training target
Patterns	1; 2; 3; 5; 6
Test07	smartcard test program from training target
Patterns	1; 3; 5; 6;
Test08	smartcard test program from training target
Patterns	1; 2; 3; 4; 5; 7
Test09	smartcard test program from training target
Patterns	1; 3; 5; 7; 8, 9, 10;
Test10	smartcard test program from training target
Patterns	1; 2; 3; 4; 5; 7
Test11	smartcard test program from training target
Patterns	1; 2; 3; 4; 5; 6; 7
Test12	smartcard test program from training target
Patterns	1, 3; 5; 10; 11

Table 4.2: Evaluation tests

One example of a **false positive** is the fail operations due the abnormal behavior of the smartcard. Therefore, the smartcard does not fail only because of external attacks (such as fault injection). Example of a *return BAD* mechanism is presented in *Section 2.4.1.9.*

The purpose of the application being under evaluation in this section is to detect the place in the source code where to insert fault injection, in order to efficiently attack the card.

4.2 Types of smartcard vulnerabilities

We are interested in the types of the smartcard vulnerabilities encountered in our tests. The types of vulnerailities are more relevant than the actual test-suite, since we want to show in which cases our application works and in which cases our application does not identify the smartcard vulnerabilities. We can see the composition of the test programs in terms of vulnerability types in *Section 4.3.*

Successful	Yes
Vulnerability type	Direct dependency
Part of	Test01, Test02, Test03, Test05, Test06, Test08, Test10, Test11, Test12

Table 4.3: Case1

4.2.1 Case1

4.2.1.1 Overview

Case1 is a type of vulnerability that is found in 9 target programs, an example being the first test of our test suite. The *Test01* is a smartcard test program from a training target, written is C. The code example given in *Chapter 3* is part of *Test01*. *Table 4.3* shows that *Case1* is a *successful* case because the vulnerability is identified. We can see from *Table 4.3* in which tests this type of vulnerability appears.

The vulnerability is part of the mentioned tests, including the following piece of code:

4.2.1.2 Code snapshot

```
__attribute__((annotate("DS"))) static uint32_t /* sensitive */ pin;
...
static int verifyPin(uint8_t* input) {
    if(pin == (uint32_t)*input) {   // line 44
        pinVerified = GOOD;

        __attribute__((annotate("END"))) int qwe;
        return GOOD;
    }

    return BAD;
}
```

4.2.1.3 Description

This is one of the most common and trivial vulnerabilities. The purpose of our application is to set the *pinVerified* flag to *GOOD* and to return *GOOD*, without knowing the *pin* value.

The security analysts annotate the global variable *pin* as sensitive. Additionally, the analysts annotate the *start* point in the main *process* function (which is not shown) and the *end* point in the *verifyPin* function, with the purpose to increase the balance. Since the *verifyPin* function is called immediately after the *start point*, there are only 2 possible execution paths. However, if the *start* and the *end* points would have a larger number of basic blocks number between them (as it can be seen in the following cases), the number of possible execution paths would be greater than 200.

The *Annotation Pass* marks *pin* as *sensitive* (using metadata) inside the LLVM bitcode. The *Paths Pass* identifies the execution paths which include the specific piece

of code.

Below is a snapshot of the 2 execution paths, which include the particular vulnerability type.

```
1 2 6 7

1 2 6 8 9 2 6 7
```

The succession of the basic block from the paths is represented by integers, based on the mapping presented in *Section 3.4*. As an example, the path *1 2 6 7* is represented in the output of the developed application as:

```
BasicBlock entry Function process Nr 1 => BasicBlock sw.bb
Function process Nr 2 => BasicBlock entry Function verifyPin Nr 6
=> BasicBlock if.then Function verifyPin Nr 7
```

The *Analysis Pass* checks for dependencies between the conditional branches and the sensitive variables. Therefore, it finds the vulnerability and indicates the place where to insert fault injection. The output of the pass is:

```
The instruction if from the line number 44 is dependent on
the sensitive global @pin defined at the line number 17
```

We can see the ouput in LLVM representation:

```
The path 1 2 6 7 contains the dependencies
  %cmp = icmp eq i32 %0, %conv, !dbg !32  i32 44
  @pin = internal global i32 0, align 4  i32 17
...
```

We notice that no *sensitive* metadata field is attached to *@pin*. The metadata is attached to the correspondent *load* instruction. The detailed explanation can be found in *Section 3.3*.

Based on the output of the application, the analysts know that on a specific execution path they have to insert fault injection into the indicated point. In this vulnerability example, the indicated point is at the line 44. The conditional check based on *pin* is skipped, so the pinVerified flag is set to *GOOD*, accomplishing the objective.

4.2.2 Case2

4.2.2.1 Overview

Case2 is a type of vulnerability which is similar to the first case, but now the sensitive variable is local. The chosen example is also taken from the *Test01*, for a better consistency with the first example. *Table 4.4* shows that the case is *successful* and it shows in which tests this type of vulnerability appears.

Successful	Yes
Vulnerability type	Direct dependency
Part of	Test01, Test02, Test03, Test06, Test07, Test08, Test10, Test11

Table 4.4: Case2

The part of the code in which this vulnerability is encountered is:

4.2.2.2 Code snapshot

```
static int locker(int n) {

__attribute__((annotate("DS"))) /* sensitive */ double uL_code=0xA4;
...
   if (!(n== uL_code))
        return BAD;  // line 58

    __attribute__((annotate("END"))) int qwe;
    return GOOD;
}
```

4.2.2.3 Description

We are still in the area of trivial basic vulnerabilities. The purpose of the attack is to force the *locker* function to return the value *GOOD*. The security analysts annotate the local *uL_code* as sensitive. The *start* point is preserved in the *process* function, which calls the *increaseBalance* function, which is calling the *locker* function. The *end* point is marked inside the *locker* function.

Since the number of basic blocks between the *start* basic block and the *end* basic block (including them) is relatively small, only 15 possible execution paths are between the two annotated points (considering also the calls to other functions). The *Annotation Pass* is marking *uL_code* as being *sensitive* inside the LLVM bitcode (using metadata). The *Paths Pass* identifies the execution paths between the annotated *start* and *end* points. Below is a snapshot of only 3 execution paths from the total of 15 paths.

```
25 27 5 6 9 21 23
25 27 5 6 9 21 22 24 9 11 12 20 27 5 6 9 21 23
25 27 5 6 9 21 22 24 9 11 13 14 20 27 5 6 9 21 23
```

An example on how the mapping is done for the *25 27 5 6 9 21 22 24 9 11 12 20 27 5 6 9 21 23* path is shown below (notice that compared to the mapping from *Case1*, the mapping *integer-basic block* is different after changing the *end* point inside the test source code):

```
BasicBlock entry Function process Nr 25 => BasicBlock
sw.bb3 Function process Nr 27 => BasicBlock entry Function
increaseBalance Nr 5 => BasicBlock for.cond Function
increaseBalance Nr 6 => BasicBlock for.end Function
increaseBalance Nr 9 => BasicBlock entry Function locker Nr
21 => BasicBlock if.then Function locker Nr 22 => BasicBlock
return Function locker Nr 24 => BasicBlock for.end Function

increaseBalance Nr 9 => BasicBlock if.end Function
increaseBalance Nr 11 => BasicBlock if.then4 Function
increaseBalance Nr 12 => BasicBlock return Function
increaseBalance Nr 20 => BasicBlock sw.bb3 Function process
Nr 27 => BasicBlock entry Function increaseBalance Nr 5
=> BasicBlock for.cond Function increaseBalance Nr 6 =>
BasicBlock for.end Function increaseBalance Nr 9 => BasicBlock
entry Function locker Nr 21 => BasicBlock if.end Function
locker Nr 23
```

We notice that the *locker* function is called 2 times from the *switch* block from the main *process* function (via the intermediate function *increaseBalance*). The basic block number 23 is the one containing the *end* annotation and the *return GOOD* instruction.

The vulnerability is found by the *Analysis Pass*. The relevant line of the output is:

```
The instruction if from the line number 57 is dependent on the sensitive
local %uL_code defined at the line number 55
```

In LLVM representation, the result is represented like:

```
The path 25 27 5 6 9 21 22 24 9 11 12 20 27 5 6 9 21 23
contains the dependencies
...
%cmp = icmp eq i32 %0, %1, !dbg !35  i32 57
%code = alloca i32, align 4, !sensitive !29  i32 55
...
```

The analyst observes the vulnerability from the line 57. Since the scope is to force the *locker* function to return *GOOD*, the instruction to be skipped is the *return BAD* from line 58, being the place where to insert fault injection. The scope of the attack is to skip the security check, with the purpose to withdraw money from the card (operation which is found later in the program flow).

Therefore, by exploiting the vulnerability from the line 58, the security check from the example is bypassed.

Successful	Yes
Vulnerability type	Inter-procedural dependency
Part of	Test01, Test02, Test03, Test05, Test08, Test10, Test11, Test12

Table 4.5: Case3

4.2.3 Case3

4.2.3.1 Overview

Case3 presents a vulnerability type which is linked to *Case2*. The context is given by the fact that the function from *Case2* is called from a conditional branch inside the *increaseBalance* function. This is also a successful test of the application.

If the first two vulnerabilities were given by direct dependencies (the sensitive variables being used directly by the conditional branch expression), in this case the dependency is inter-procedural. This means that a conditional branch calls a function (which may call other functions, in *cascade*) which uses a sensitive variable on which the original conditional branch is dependent. As mentioned in *Chapter 3*, the *Analysis Pass* uses recursive depth searching methods to identify this kind of dependencies.

The vulnerability is part of the extended piece of code from *Case2*:

4.2.3.2 Code snapshot

```
static int locker(int n) {

__attribute__((annotate("DS"))) /* sensitive */ double uL_code=0xA4;
...
    if (!(n== uL_code))
        return BAD;
    return GOOD;
}

...

static int increaseBalance(uint8_t* input, int* pinAlias) {
    ...
    if(!locker(unlock[4]))
        return BAD;    //  line 69
    ...
    balance += input[DATA_O];
    __attribute__((annotate("END"))) int qwe;
    return GOOD;
```

4.2.3.3 Description

We can noticed that the *end* point in the program is now in the *increaseBalance* function, due the fact that the purpose of the attack has changed. The new purpose is to execute the instruction which increases the balance on the smartcard, by bypassing the encountered security checks.

In this case, the sensitive variable is annotated exactly like in *Case2*. Therefore *uL_code* is a local sensitive variable, marked in the LLVM bitcode by the *Annotation Pass*. The *Paths Pass* is calculating all the possible execution paths between the two points, which are 19 for this case.

Below is a snapshot of 4 execution paths:

```
25 27 5 6 9 21 22 24 9 11 13 15 17 19
25 27 5 6 9 21 22 24 9 11 13 15 17 18 20 27 5 6 9
                                    11 13 15 17 19
25 27 5 6 9 21 23 24 9 11 12 20 27 5 6 9 11 13 15 17 19
25 27 5 6 9 21 23 24 9 11 13 14 20 27 5 6 9 11 13 15 17 19
```

An example of mapping (integer - basic block) for the path *25 27 5 6 9 21 22 24 9 11 13 15 17 19* is:

```
BasicBlock entry Function process Nr 25 => BasicBlock sw.bb3
Function process Nr 27 => BasicBlock entry Function increaseBalance
Nr 5 => BasicBlock for.cond Function increaseBalance Nr 6 => BasicBlock
 for.end Function increaseBalance Nr 9 => BasicBlock entry Function
 locker Nr 21 => BasicBlock if.then Function locker Nr 22 => BasicBlock

return Function locker Nr 24 => BasicBlock for.end Function
increaseBalance Nr 9 => BasicBlock if.end Function increaseBalance
Nr 11 => BasicBlock if.end5 Function increaseBalance Nr 13 =>
BasicBlock if.end9 Function increaseBalance Nr 15 => BasicBlock
if.end15 Function increaseBalance Nr 17 => BasicBlock if.end18
Function increaseBalance Nr 19
```

The basic block 9 contains the *if* conditional branch from the example. The vulnerability is identified using the *Analysis Pass*. The output is:

```
The instruction if from the line number 68 is dependent
on the sensitive local %uL\_code defined at the line number 55
```

The output has the LLVM representation:

```
%tobool = icmp ne i32 %call, 0, !dbg !47  i32 68
%code = alloca i32, align 4, !sensitive !29  i32 55
```

Successful	Yes
Vulnerability type	Tainted Dependency
Part of	Test01, Test02, Test06, Test07, Test10, Test11

Table 4.6: Case4

Similar to *Case2*, the analyst observes the vulnerability related to the line 68 and marks the line 69 as a fault injection target. Therefore the line 69 is skipped and the correspondent *return* instruction is not executed. Furthermore, the attack is a success because the *balance +input[DATA_O]* instruction can be executed.

Please note that: We can observe from the example that the 2 conditional branches are **complementary**. This means that if we use fault injection to skip the *return BAD* statement from the callee function *locker*, then the function will return *GOOD* to the *increaseBalance* caller function. Therefore, it is not needed to insert fault injection to skip the *return BAD* statement from the *increaseBalance* function, since the *if* condition is not NULL. Therefore, we can state that the 2 *if* conditional branches are complementary. The security analysts have the task of identifying these complementary conditional branches.

It is not the purpose of our application to automatically detect the conditional branches which are complementary. The implementation of this task at compile-time (like the entire *Analysis Pass*) can be prone to errors, since we do not know in which order the pass is identifying the conditional branches and if one conditional branch is influenced by the other one for each execution path. The solution is to emulate the execution of the target program and to use symbolic execution. In conclusion, by using our application, the analyst can see which points are suitable for fault injection. The security analyst has the task of choosing which points will serve as input for the physical fault injection hardware.

4.2.4 Case4

4.2.4.1 Overview

This case presents a vulnerability given by a dependence between a conditional branch and a variable that is not initially considered sensitive. However, the variable is influenced by another sensitive variable. The variable is called *tainted* (as seen in *Section 3.3*).

This type of the vulnerability can be observed in the code snapshot from below:

4.2.4.2 Code snapshot

```
__attribute__((annotate("DS"))) static uint32_t /* sensitive */ pin;
```

```
                static int pinVerified;
....
    if(pin == (uint32_t)*input) {
        pinVerified = GOOD;
        return GOOD;
    }
....
   if(!pinVerified) {

        return BAD;  // line 71
    }
```

4.2.4.3 Description

The code is formed from three pieces: the first piece is part of the globals declaration and the other two pieces reside in different functions. The purpose of the attack is to bypass the *if(pinVerified)* conditional branch.

Case 4 is connected to *Case1*. In *Case1*, the conditional branch instruction (illustrated also here) was skipped, so the *pinVerified* value was set to GOOD. On the other hand, the *pinVerified* variable is used by a conditional branch later in the program (after the pin is checked for authentication, so *pinVerified* is set to *GOOD*).

However, *pinVerified* is not initially considered sensitive by the security analysts, so the branch might not be considered to be vulnerable. The solution is given by the *Annotation Pass*, which integrates a taint analysis algorithm that helps the *sensitiveness* to be propagated through the source code. In the authentication part of the code, we observe that the value of *pinVerified* is influenced by the sensitive global variable *pin*. Therefore, the *pinVerified* flag is tainted by the *pin* global. The *Annotation Pass* checks all the conditional branches which use sensitive variables and takes the target basic blocks (the conditional branch operands). In each target basic block, it searches for instructions that change the values of variables that are not considered sensitive. An example for such an instruction is *store*, having the value to be stored (1 for *GOOD*) as the first operand and the place to store it (*pinVerified*) as the second operand. Consequently, the *Annotation Pass* adds the *sensitive* metadata field to the tainted variable. The implementation used by the *Annotation Pass* does not take into consideration all the possible cases of taint analysis (like the particular case of alias analysis), which is described in *Case5*. In order to efficiently cover the taint analysis, a symbolic execution approach using KLEE over LLVM is proposed as future work. The KLEE tool is introduced in the *Section 2.2.1.3*.

The *Paths Pass* discovers 19 paths, like in *Case3* (since the target program is identical). Consequently, the *Paths Pass* output is analog. The conditional branch where the vulnerability exists is part of the *if.end* basic block from the *increaseBalance* function.

The *Analysis Pass* identifies the conditional branch that is dependent on the tainted *pinVerified* variable:

```
The instruction if from the line number 70 is dependent on the
```

Successful**	No
Vulnerability type	Alias Dependency
Part of	Test01, Test03, Test07, Test09

Table 4.7: Case5

```
tainted global @pinVerified defined at the line number 18
```

The equivalent LLVM representation is:

```
%tobool3 = icmp ne i32 %5, 0, !dbg !50, !path !28  i32 70
@pinVerified = internal global i32 0, align 4  i32 18
```

In contrast to the other LLVM representations from above, this conditional branch contains the metadata with the paths information provided by the *Paths Pass*. This means that in the particular basic block number 11 (the *if.end* basic block from the *increaseBalance* function), the *icmp* instruction is the first instruction that does not have already metadata added. Therefore, it is chosen to transport the paths metadata to the *Analysis Pass*. The **detailed** logic is presented in *Section 3.3*.

The analyst observes the vulnerability from the line 70 by checking the output of the *Analysis Pass*. Since the purpose is to bypass the security check, the security analyst triggers fault injection at the line 71 with the purpose to skip the *return BAD* statement.

4.2.5 Case5

4.2.5.1 Overview

Case 5 shows a vulnerability that the developed application does not detect. This case is the last example given from the *Test01* target program. It is related to the sensitivity propagation using aliases.

**This case is considered *unsuccessful* in the general case. However, a small algorithm can be optionally added to the developed application in order to solve some particular cases, like the one which is presented below. This algorithm will be presented in the *Description* section.

The following piece of code illustrates the vulnerability:

4.2.5.2 Code snapshot

```
int process(uint8_t* /* insensitive */ input) {


__attribute__((annotate("START"))) /*start*/  int asd;
    uint16_t sw;
....
```

```
//pseudocode
foreach call instruction //or similar
    forall operands
        if(operand[i] is sensitive)

        //or tainted from sensitive

            goto definition
            of call instruction

add metadata sensitive to operand[i] //from the same
                                     position if existent
```

Figure 4.1: Case 5 pseudocode solution

```
    switch(input[INS_O]) {
....
        case INCREASE:
            sw = increaseBalance(input,&pinVerified);
            break;
....
static int increaseBalance(uint8_t* input, int* pinAlias) {
....
    if(!*pinAlias) {
        return BAD;
    }
....
__attribute__((annotate("END"))) /*end*/  int qwe;
```

4.2.5.3 Description

The *start* point is indicated in the *process* function and the *end* point is in the *increaseBalance* function. The purpose of the attack is to bypass the *if(!*pinAlias)* conditional branch. As seen in *Case4*, the *pinVerified* global variable is tainted by the sensitive global variable *pin*, so is the *sensitiveness* is propagated to *pinVerified*. On the *INCREASE switch* branch from the *process* function, the *increaseBalance* is called with an argument which represents a reference to *pinVerified*.

The *increaseBalance* function uses *pinAlias* as an alias for the *pinVerified* variable. The *pinAlias* variable is used in an alias dependency. The scope is to bypass the *if* conditional branch, but the developed application does not take it into account as a vulnerability during the analysis. The reason is that our application does not support alias analyzing.

In order to solve the problem, we present two solutions:

- using a straight algorithm (for this particular case, the algorithm can be optionally added to the application). The algorithm is **shown** in *Figure 4.1*.
- using symbolic execution with KLEE (presented in *Future Work*)

The first solution (using the small algorithm) solves the particular case from this example, but it cannot be generalized.

In order to solve the lapses of the first solution, we propose the symbolic execution as a future work. The target program can be symbolically executed.

4.2.6 Case6

4.2.6.1 Overview

Case6 is a type of vulnerability which is part of a test case that simulates the *Oracle* padding function [31]. In this case, the target program is written in C++, so classes, methods and objects are used. Also, a sensitive array is used inside the conditional branch presented in this test case. Padding oracles are part of a vulnerability class that provides side channel information, usually by abusing the padding check in order to decrypt ciphertexts.

This example is taken from a test case used for a SIM (subscriber identity module) card, a special type of smartcard used for mobile phones. The flow of the relevant operations is the following: The ciphertext C is encrypted from the plaintext P using the secret key K. After several operations, the ciphertext C is decrypted using the same key K into P. Then, P is checked if valid, using the *confidentialityPadding1* method.

The context of this example is given by the following situation: Before a SIM is released on the market, it is programmed with a secret key Ki, which is 128-bit long.The key Ki should be used on special algorithms that run internally on the SIM. A copy of the key is kept at the Authentication Center of the network operator. The key is used in the authentication process in the GSM protocol, with the purpose to encrypt the voice. In the test case *Test06*, the purpose of the attacker is to change the value of Ki on the SIM card. Therefore, the attacker can perform a DOS (denial of service) attack, so the mobile client cannot authenticate to the Authentication Center for accessing the voice encryption service (since the value of Ki is different on the SIM and at the Authentication Center). Furthermore, the attacker can listen the unencrypted voice.

The key Ki is kept encrypted on the SIM and we use the 128-bit long array C for this purpose. The masterkey K is used in order to change the Ki value on the SIM. The decrypted plaintext Ki is checked if valid using a confidentiality padding function (in this test case). If the result is validated, the atttacker can undertake actions like changing the value of Ki. The attacker does not know the secret key K used to decrypt the ciphertext C. Therefore, the **scope** of the attack is to bypass the validation security check of the decrypted Ki.

The resulted padding is checked using the *confidentialityPadding1* method of the *SecCheck* class. The details of the Oracle confidentiality checking (checking if the last bytes are either 1, 22, 333 etc.) are not relevant for the test case presentation. One possible attack against the Oracle function is to bypass the conditional branch. The attack is accomplished by our application. The *Analysis Pass* highlights the *indirect dependency* type.

The *Indirect Dependency* type can be observed in the following piece of code, being

Successful	Yes
Vulnerability type	Indirect dependency
Part of	Test06, Test09, Test10, Test11

Table 4.8: Case6

explained in the following section.

4.2.6.2 Code snapshot

```
__attribute__((annotate("DS"))) unsigned char Ki[128];
.....

class SecCheck {
private:
...
  int confidentialityPadding1(unsigned char *Ki1)
  {
    int i;
    for (i=128-Ki1[127]; i<127; i++){
      if(Ki1[127]!=Ki1[i]){
        return 0;
      }
    return (Ki1[127]!=0);
    }
  }
};
.....
SecCheck padd;
...
__attribute__((annotate("START"))) int asd;
...
decrypt(C, Ki, padd.K);
...
if(!(padd.confidentialityPadding1(Ki)))
    return BAD;   // line 124
....
unsigned char* input;
read(input, 128);
....
    for(i=0;i<128;i++){
      Ki[i]=input[i];
....
__attribute__((annotate("END"))) int qwe;
```

....

4.2.6.3 Description

The *SecCheck* class has multiple security checking methods, the *confidentialityPadding1* being relevant for this test case. Inside the *main* function, the conditional branch that is calling the *confidentialityPadding1* method is between the annotated start and end points. The security analysts have annotated the array *Ki* as sensitive, because it can be used in security checks. The **scope** of the attack is to bypass the security branch invoking the *confidentialityPadding1*.

The type of dependency discovered by the application is an indirect dependency. In this type of dependency, the conditional branch is calling a function having one of the arguments a sensitive variable.

This type of dependency is different from a *direct dependency*, since the sensitive variable is not one of the condition arguments. Also, it is different from an *inter-procedural dependency*, since inside the called method no variable is considered sensitive. As mentioned in *Chapter 3*, the declarations of global variables (arrays in this case) cannot have metadata. Therefore, the *call* instruction which is the argument of the conditional branch and which is using the sensitive array *Ki* is annotated as sensitive:

```
%call = call i32 @_ZN8SecCheck17confidentialityPadding1EPh
(%class.SecCheck* %padd, i8* getelementptr inbounds ([128 x i8]
* @Ki, i32 0, i32 0)), !dbg !955, !sensitive1 !949
```

This annotation is valid because the *call* instruction is using only one variable. Therefore, we know that the specific *call* is carrying *sensitive* metadata **only** for *Ki*. Consequently, the *sensitiveness* cannot propagate to other variables, like in the case when the *sensitiveness-carrier* instruction has as arguments more variables (in this case the *call* instruction is the *sensitiveness-carrier* for *Ki*). The *Paths Pass* identifies 89 possible execution paths, the relatively small number being the consequence of the location of the start point, which is close to the conditional branch. The basic block containing the conditional branch (which the attacker wants to bypass) is *for.end* from the *main* function, while the confidentiality method is part of 6 basic blocks. The actual confidentiality check is part of the *for.body* basic block of the *_ZN8SecCheck17confidentialityPadding1EPh* function.

The *Analysis Pass* identifies the indirect dependency between the conditional branch and the sensitive array *P* by verifying if the arguments of the *call* instruction are sensitive variables. Since the declaration of *Ki* could not carry the metadata transmitted by the *Annotation Pass* (because it is global, as explained in *Section 3.3*), the *Analysis Pass* searches the instruction carrying the metadata. The *call* instruction is used to carry the sensitive metadata for the *Ki* array. The *Analysis Pass* identified the *call* instruction, takes the *Ki* operand and discovers that the operand is sensitive. This leads to the fact that the conditional branch is dependent on *Ki*.

The resulting dependency is:

Successful	No
Vulnerability type	Direct Dependency
Part of	Test07, Test12

Table 4.9: Case7

```
The instruction if from the line number 123 is dependent on the sensitive
global @Ki defined at the line number 10
```

In LLVM format:

```
%tobool = icmp ne i32 %call, 0, !dbg !957, !path !934  i32 123
@Ki = global [128 x i8] zeroinitializer, align 1  i32 10
```

The purpose is to bypass the security check which leads to *return BAD* (as mentioned in the other test cases). Therefore the point marked to insert fault injection is the line 124, which is the next code line after the identified conditional branch. Consequently, the *return BAD* statement is skipped. The attacker/security analyst changes the key *Ki* and consequently, the mobile client cannot authenticate in order to use the encrypted voice service.

4.2.7 Case7

4.2.7.1 Overview

The *Case7* type of smartcard vulnerability is based on the test case programs used for debit cards. This case is unsuccessful, since it needs a different approach of interpreting the dependencies compared to the current functionality of the *Analysis Pass*. The dependency is identified, but our application does not consider it as a vulnerability.

The focus is on the operation which encrypts using the secret key K the data buffer into an encrypted buffer. DES is the used encryption algorithm. If the security analyst would inject faults at the line where the encryption function is called, than the encryption of the buffer would be skipped. A conditional branch from the *Test12* is double checking if the encryption was accomplished, using the *desVerified* flag. However, this conditional branch (not **relevant** for *Case7*) is identified by our application as a tainted dependency and consequently not presented here, but shown in the code (being based on a flag that changes its value inside the encryption function; the concept is explained in *Case4*). The vulnerability case based on this example is presented into the *Description* section.

The situation is illustrated in the next code snapshot:

4.2.7.2 Code snapshot

```
//stored password
__attribute__((annotate("DS"))) unsigned char deskey[168];
```

```
//given password
__attribute__((annotate("DS"))) unsigned char inputkey[168];
....

int displayUserData(unsigned char *ukey,
const char* field){
....

if( ! desVerified )
        return BAD;

snameC=extract(buffer,field);

//ukey is an alias for deskey
desDecrypt(ukey, snameC, DECRYPT);

....
return sname;
}
....

int main(int argc, char* argv[]){
....

unsigned char* inputkey;
read(inputkey, 168);
....
__attribute__((annotate("START"))) int asd;
desEncrypt(inputkey, buffer, ENCRYPT);
....

displayUserData(deskey,surname);
....
__attribute__((annotate("END"))) int qwe;
....
```

4.2.7.3 Description

The goal of the attack is to get the personal data of the smartcard user in **plaintext**. In the presented example, the attackers aim for the surname field. The personal data of the user is part of the encrypted buffer.

The data is retrieved by calling the *displayUserData* method, based on a secret key and on the surname field. The attacker does not know the *deskey* used to obtain the secret data. Inside the DES encryption function, the *inputkey* data required from the user

is matched against the *deskey* secret key stored on the smartcard. The given password from the user is *inputkey*, while *deskey* is the password stored on the smartcard.

The way to achieve the goal is to insert fault at the line number 54, so the encryption requiring the correct *inputkey* is skipped.

The *Analysis Pass* does identify the dependency between the *desEncrypt* method and the sensitive array *deskey*. However, the application is currently looking for conditional branches that can lead to fault detection procedures that can disable the functionality of the smartcard. The dependencies between method calls and sensitive variables need a different approach. The application does not consider the dependency as a result that gives a point where to insert fault injection.

The dependency is illustrated below in LLVM format:

```
call void @_Z7desEncryptPhS_PKc(i8* getelementptr inbounds
([168 x i8]* @deskey, i32 0, i32 0), i8* getelementptr inbounds
([40 x i8]* @buffer, i32 0, i32 0), i32 1), !dbg !936
```

The *buffer* is encrypted using the DES algorithm. A conditional branch based on the tainted variable *desVerified* is insuring that the user data is **displayed** only if the buffer has been encrypted. This conditional branch can be bypassed as presented in *Case4*. From the encrypted buffer, there is extracted the field corresponding to the users *surname*.

Still, the field is encrypted, because our application did not detected the *desEncrypt* operation as a place where to insert fault injection. As a summary, the **solution** is: consider the DES encryption as an operation to be skipped by using fault injection, so the returned surname will be in plaintext. Also, the conditional branch from the *displayUserData* method (that is checking if the encryption was done should be bypassed) as we did in *Case4*.

The question *Why we do not consider the call to the desEncrypt instruction operation a result?* may arise. The reason is that we need to define a procedure for considering a *call* instruction (having sensitive variables as arguments) a result. False positives cases may arise if you consider all such instructions as results. Such a situation is presented in the code snapshot from above, given by the *desDecrypt* function. Assuming that the *ukey* is found to be sensitive (by *sensitiveness* propagation for aliases), than the application would consider the *desDecrypt(ukey, snameC, DECRYPT)* operation as a point to insert fault injection, because the operation is a *call* instruction using a sensitive argument. Since the decryption is not requiring any user input for the key, this operation should not be skipped, in order to have a successful attack against the confidentiality of the user data (because the decrypt operation is needed for the attack in order to have access to plaintext). Therefore, we need a procedure to differentiate between the valid results and the **false positives**.

Successful	Yes
Vulnerability type	Direct Dependency
Part of	Test08

Table 4.10: Case8

4.2.8 Case8

4.2.8.1 Overview

This case aims not to present a different type of vulnerability, but to show an example with Java source code. The case contains a direct type of dependency based on a sensitive global variable. However, we needed to make adjustments to the *Annotation Pass*. The chosen example is taken from the *Test case 8* which is part of bank smartcard programs.

The part of the code that contains the dependencies is shown below:

4.2.8.2 Code snapshot

```
class MYGL {
  static int pin;
  static int balance;
}
....
public class Main {
    private static final int SW_WARNING_STATE_CHANGED
    = 33;
    public static void main(String[] args)
    throws Exception {
        try{
        TestAnnotationParser parser =
        new TestAnnotationParser();
        parser.parse(Annotated.class);
        int creditAmount;
        creditAmount=4597;

....
        if(MYGL.pin!=buffer){

            return BAD;   // line 117
            // ISOException.throwIt(SW_WARNING_STATE_CHANGED);
        }

        MYGL.balance=creditAmount+6094;

....
```

```
        if(MYGL.balance>3006)

            return BAD1;  // line 132
            // ISOException.throwIt(SW_WRONG_P1P2);
        }

....
        catch (IOException ioe) {
        System.out.println("Warning  incorrect pin : " +
        SW_WARNING_STATE_CHANGED + ioe.getMessage());
        }


    }
}
```

4.2.8.3 Description

The standard *Annotation Pass* cannot be used in the same format for the Java target programs, so modifications were done in the code. The **reason** is that the VMKit tool (presented in *Section 2.2.1.3*) does not support yet Java source code annotations. More exactly, when the Java source code is translated into LLVM bitcode, the source code annotations are **ignored** in the generation phase. A distinction is made between the source code annotations which are not supported (needed by our application, to perform queries) and the proper internal annotations structures used in *J3* (tool from VMKit, whom implementation does **not** rely on LLVM annotations as well). In the VMKit representation, a Java objects is a set of bytes in the heap and these bytes can be moved by the garbage collector. The runtime representation of a class is another object (based on the same logic). Therefore, an annotation at Java level should be in fact an object, which is a description/metadata of the members inside the class. However, the source code annotations are not supported by VMKit.

Below we can see the custom source code Java annotation that we tried to use to mark the sensitive variables:

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.reflect.Method;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@interface Sensitive {
    String info() default "";
}
```

```
class Annotated {
    @Sensitive (info = "DS")
    public void foo(String myParam) {
        System.out.println("This is "
        + myParam);
    }
}

class TestAnnotationParser {
    public void parse(Class clazz)
    throws Exception {
        Method[] methods = clazz.getMethods();
    for (Method method : methods) {
        if (method.isAnnotationPresent
        (Sensitive.class)) {
           Sensitive test = method.getAnnotation
           (Sensitive.class);
            String info = test.info();

            if ("DS".equals(info)) {
                 System.out.println("Variable
                  is sensitive!");
                 // try to invoke the method
                 with param
                 method.invoke(
                    Annotated.class.newInstance(),
                    info);
....
public class Main {
    public static void main(String[] args)
    throws Exception {
        TestAnnotationParser parser = new
        TestAnnotationParser();
        parser.parse(Annotated.class);
....
```

The *Annotation Pass* was modified to receive the variables which are considered sensitive as input from the keyboard. The global variables *pin* and *balance* are identified in the LLVM code by looking at the *@MYGL_static* global:

For *pin*:

```
({ i32, i32 }* @MYGL_static, i32 0, i32 1)
```

For *balance*:

```
balance = ({ i32, i32 }* @MYGL_static,
i32 0, i32 0)
```

The *Annotation Pass* was also modified regarding the searching for the variable names. The LLVM bitcode from Java does not preserve the variable names like in the C/C++ case. The tool was changed to look for arrays of UTF-16 type that are used for strings. Therefore, the *Annotation Pass* identified the variables and the class by looking at the ASCII code format contained in LLVM internal constants. For example, the string *balance* is contained in:

```
@43 = internal constant { i32, [7 x i16] } { i32 7, [7 x i16]
[i16 98, i16 97, i16 108, i16 97, i16 110, i16 99, i16 101] }
```

Translating the array elements from the ASCII decimal format, the resulting string is *balance*. The *Annotation Pass* marks as sensitive the indicated global variable. In the case of the *pin* variable, the transformed instruction which carries the metadata is:

```
%95 = load i32* getelementptr inbounds ({ i32, i32 }*
@Global_static, i32 0, i32 1), align 4, !sensitive1 !6
```

A problem (which is not affecting this case example) may arise when dealing with Java-into-LLVM translation of the local variables. In our example, the local variable is *creditAmount*. Even if in our example the compiler is optimizing the code and considers the local variable as a constant (it stores *10691* directly into *balance*, which is the addition result between the creditAmount value of 4597 and the added value of 6094; also there is no *alloca* instruction), the local variable name is not preserved in any case. The reason is that the LLVM Java bitcode does not give and use the name of the local variables. In the standard Java bitcode, the names of the local variables are found in an attribute used for debugging, but currently VMKit does not use this attribute. The LLVM bitcode which *vmjc* (tool from VMKit) emits does not contain the names of the local variables.

The goal of the attack is to bypass the two security checks from the smartcard program. Both dependencies from the target program example are direct dependencies. In the case of the *pin* variable, the output of the *Analysis Pass* in LLVM format is:

```
%96 = icmp slt i32 %95, 1966  printing a <null> value
i32* getelementptr inbounds ({ i32, i32 }* @Global_static, i32 0,
i32 1)  printing a <null> value
```

As we can see, the line numbers are not printed because the specific methods from the *Analysis Pass* do not work on the Java LLVM bitcode. The analysts have to do an extra manual check to identify the corresponding conditional branches in the code (based

Successful	Yes
Vulnerability type	Inter-procedural Dependency
Part of	Test04

Table 4.11: Case9

on the LLVM output), operation that will slightly increase the workload. Therefore, one place to insert fault injection is at the line 117 to avoid the *return BAD* statement.

This statement may call the *ISOException.throwIt(SW_WARNING_STATE_CHANGED)* Java Card API operation that can temporary disable the functionality of the smartcard if the *pin* value is incorrect more than 3 times. In the case of the *balance* sensitive global variable, the place to insert fault injection is at the line 132. In this case the *return BAD1* may be used to trigger the *ISOException.throwIt(SW_WRONG_P1P2)* JAVA Card API operation that forbids the balance to exceed a certain value.

4.2.9 Case9

4.2.9.1 Overview

This case is part of a **test** target program that emulates the FAULT.BYPASS Riscure pattern, which is presented in *Section 2.4*. This type of smartcard vulnerability denotes an inter-procedural dependence (as in *Case3*), but it requires a different approach as presented in the *Description* paragraph. On the other hand, the **specific** FAULT.BYPASS examples (part of the same **test**) are presented in *Case10* and *Case11*.

We can see the inter-procedural dependency in the code snapshot from below:

4.2.9.2 Code snapshot

```
int flag11=0;
int flag12=1;
int flag21=1;
int flag22=1;

class SecCheck {
public:

  __attribute__((annotate("DS"))) const
  char K[7]="secret";

....

  int integrityPadding2(unsigned char *P2)
  {
```

```
        const char* sloc = "secret";

      flag12=0;
      if (K != sloc){
        flag21=0;  // line 45
        flag22=0;
        return BAD;   // line 47
      }

      return GOOD;
  }
};

....

int main(int argc, char* argv[]){
....

if(!(padd.integrityPadding2(P)))
    return BAD;  // line 88
....
```

4.2.9.3 Description

This case presents an inter-procedural dependency which is different from the dependency from *Case3*. The difference is given by the fact that a conditional branch is dependent on a sensitive local variable which is declared inside a class definition, not and is declared inside a method.

In order to identify the dependency from this case, the *Annotation Pass* queries the LLVM intrinsic *llvm.ptr.annotation.p0i8*. The sequence of instructions between the local variable used (*%arraydecay*) and the definition of the sensitive variables is longer, as we can see in the next snapshot.

```
%K = getelementptr inbounds %class.SecCheck* %this1, i32 0,
i32 0, !dbg !991, !sensitive !992
%0 = bitcast [7 x i8]* %K to i8*, !dbg !991
%1 = call i8* @llvm.ptr.annotation.p0i8(i8* %0, i8* getelementptr
inbounds ([3 x i8]* @.str2, i32 0, i32 0), i8* getelementptr
inbounds ([9 x i8]* @.str1, i32 0, i32 0), i32 22), !dbg !991
%2 = bitcast i8* %1 to [7 x i8]*, !dbg !991
%arraydecay = getelementptr inbounds [7 x i8]* %2, i32 0,
i32 0, !dbg !991
```

For this type of vulnerability, the *Analysis Pass* is looking for a *getelementptr* instruction for the definition of the sensitive local variable, instead of an *alloca* instruction which is more common in our test suite. We can see that the *Annotation Pass* is using a *getelementptr* instruction for carrying the *sensitive* metadata. The *Paths Pass* takes all the execution paths. A snapshot of 5 of the total of 147 possible execution paths is:

```
7 1 2 5 7 8 9 10 8 11 22 24 25
             11 13 15 16 17 19
7 1 2 5 7 8 9 10 8 11 22 24 25
                11 13 15 17 19
7 1 2 5 7 8 11 13 15 16 19
....
7 1 2 3 4 2 5 7 8 9 10 8 11 22 23
                25 11 13 15 16 19
7 1 2 3 4 2 5 7 8 9 10 8 11 22 23
             25 11 13 15 16 17 19
```

The last execution path from the snapshot is mapped as:

```
BasicBlock entry Function main Nr 7 => BasicBlock entry
Function _Z7decryptPhS_PKc Nr 1 => BasicBlock for.cond
Function _Z7decryptPhS_PKc Nr 2 => BasicBlock for.body
Function _Z7decryptPhS_PKc Nr 3 => BasicBlock for.inc Function
_Z7decryptPhS_PKc Nr 4 => BasicBlock for.cond Function
_Z7decryptPhS_PKc Nr 2 => BasicBlock for.end Function
_Z7decryptPhS_PKc Nr 5 => BasicBlock entry Function main Nr 7
=> BasicBlock for.cond Function main Nr 8 => BasicBlock for.body
Function main Nr 9 => BasicBlock for.inc Function main Nr 10 =>
BasicBlock for.cond Function main Nr 8 => BasicBlock for.end
Function main Nr 11 => BasicBlock entry Function
_ZN8SecCheck17integrityPadding2EPh Nr 22 => BasicBlock if.then
Function _ZN8SecCheck17integrityPadding2EPh Nr 23 => BasicBlock
return Function _ZN8SecCheck17integrityPadding2EPh Nr 25 => BasicBlock
for.end Function _ZN8SecCheck17integrityPadding2EPh Nr 11 =>
BasicBlock if.end Function main Nr 13 => BasicBlock if.end8 Function
main Nr 15 => BasicBlock lor.lhs.false Function main Nr 16 => BasicBlock
lor.lhs.true Function main Nr 17 =>  BasicBlock if.end13 Function
main Nr 19
```

On the execution path from above, the *Analysis Pass* identifies the inter-procedural dependency and a direct dependency (which is linked with the first dependency). Both of them are related to the sensitive key K:

The relevant inter-procedural dependence for this case is:

```
The instruction if from the line number 87 is dependent on
the sensitive local %K defined at the line number 22
```

The direct dependency present inside the *integrityPadding2* method of the *SecCheck* class is:

```
The instruction if from the line number 44 is dependent on
the sensitive local %K defined at the line number 22
```

The two dependencies in the LLVM specific format are:

```
Instruction    %cmp = icmp ne i8* %arraydecay, %3, !dbg !949
from line number i32 44  is dependent on
instruction    %K = getelementptr inbounds %class.SecCheck*
%this1, i32 0, i32 0, !dbg !991, !sensitive !992  i32 22

Instruction    %tobool = icmp ne i32 %call,
0, !dbg !960
from line number i32 87  is dependent on
instruction    %K = getelementptr inbounds %class.SecCheck*
%this1, i32 0, i32 0, !dbg !991, !sensitive !992  i32 22
```

Like in *Case3*, we observe that the 2 sensitive checks are complementary. If we skip the sensitive check from *integrityPadding2* using fault injection, then the method will return *GOOD*, so the sensitive check from the lines 87-88 will be not return *BAD* since the guard of the conditional branch is *true*. In contrast, if we insert fault injection to bypass the security check from *main*, then the method *integrityPadding2* will not be called and the security analyst does not have to insert another fault injection inside the method. However, this scenario is valid only when the *start* and the *end* points are set as for *Case9*. In this context, the points characterize the purpose of the attack.

The goal of the attack is to skip the two *return BAD* statements correspondent to the identified dependencies from above. Therefore, based on the results provided by the *Analysis Pass*, the security analysts will mark the lines 47 (the *return BAD* statement) and 88 (the *return BAD* statement) as places to insert fault injection. Next, two other cases from the same target test example can be seen.

4.2.10 Case10

4.2.10.1 Overview

This case presents a smartcard vulnerability which is part of the same target test as *Case9* and *Case11*. This case is given by the implementation of the FAULT.BYPASS Riscure pattern, which is described in the *Section 2.4.* of this thesis.

We can see the tainted dependency in the code snapshot from below:

Successful	Yes
Vulnerability type	Tainted Dependency
Part of	Test04, Test11

Table 4.12: Case10

4.2.10.2 Code snapshot

```
int flag11=0;
int flag12=1;
int flag21=1;
int flag22=1;

class SecCheck {
public:

  __attribute__((annotate("DS"))) const
  char K[7]="secret";

....

  int integrityPadding2(unsigned char *P2)
  {
        const char* sloc = "secret";

      flag12=0;
      if (K != sloc){
        flag21=0;
        flag22=0;
        return BAD;
      }

      return GOOD;
  }
};

....

int main(int argc, char* argv[]){
....

if((!flag21 || !flag22) &&
(flag21 != flag22)){

  return BAD;  // line 94
```

```
}
....
```

4.2.10.3 Description

The aim of the attacker is to bypass the conditional branch depending on the 2 global flags (*flag21 and* flag22). As we can be see in the code, the conditional branch is not initially dependent on any sensitive variable marked by the security analysts. However, the *Annotation Pass* is handling this situation. We need to insert fault injection in the situations when the pairs (*flag11-flag12* and *flag21-flag22*) have elements with different values.

For example, this situation can be achieved if (in *Case9* which is previously presented) the security analyst does not check the fact that the *return BAD* statement is not immediately after the conditional branch (at line 44) and the fault injection is triggered earlier (at the line 45 instead of 47 as seen in *Section 4.2.9.*), skipping the *flag21=0* statement. Therefore, *flag21* is not set to 0. We consider the situation when on a specific execution path the conditional branch *if (K != sloc)* is not executed (the program is executed like for *Case9*), but on another execution path (the program is executed like for *Case10*) the conditional branch *if (K != sloc)* is executed and therefore the *flag21=0* statement can be skipped (fault injected) by mistake. Consequently, the value of *flag21* will be 1 (unchanged), while the value of *flag22* will be 0 (changed).

The *Annotation Pass* detects the sensitiveness propagation from the conditional branch *if(K != sloc)* which is dependent on the sensitive variable K to the statements *flag21=0* and *flag22=0*. Therefore, the global variables *flag21* and *flag22* are automatically marked as sensitive by the *Annotation Pass*. The two variables are tainted from the sensitive variable K.

For these types of vulnerabilities, the tainted algorithm from the *Annotation Pass* identifies the sequences of *getelementptr*, *bitcast* and *call* instructions which use a local sensitive variable declared inside the *SecCheck* class. The local variables declared inside a class are different from the local variables declared inside a function. The declaration is done using a *getelementptr* instruction instead of an *alloca* instruction, as we can see below:

```
%K = getelementptr inbounds %class.SecCheck* %this1,
i32 0, i32 0, !dbg !991, !sensitive !992  i32
```

The *Annotation Pass* automatically marks the tainted variables from K:

```
%11 = load i32* @flag21, align 4, !dbg !962, !sensitive1 !963
%12 = load i32* @flag22, align 4, !dbg !962, !sensitive1 !963
```

Successful	No
Vulnerability type	Direct Dependency
Part of	Test04, Test06, Test07

Table 4.13: Case11

The execution paths are the same detected paths from *Case9*, the target program being the same. The *Analysis Pass* detects the conditional branch from the *main* function which is dependent on the tainted global variables *flag21* and *flag22*.

The conditional branch *if((!flag21 —— !flag22) && (flag21 != flag22))* is detected by our application as an instruction to be bypassed using fault injection:

```
The instruction if from the line number 93 is dependent
on the sensitive local %flag21 defined at the line number 15

The instruction if from the line number 93 is dependent
on the sensitive local %flag22 defined at the line number 16
```

The security analysts mark the *return* instruction from the line number 94 for inserting fault injection, in order to bypass the security check.

4.2.11 Case11

4.2.11.1 Overview

Case11 is part of the same target test program which contains the *Case9* and the *Case10* vulnerability types. This case is built inspired by the FAULT.BYPASS Riscure pattern. The pattern is presented described in *Section 2.4* of this report.

The situation is presented in the following snapshot:

4.2.11.2 Code snapshot

```
int flag11=0;
int flag12=1;
int flag21=1;
int flag22=1;

class SecCheck {
public:

  __attribute__((annotate("DS"))) const
  char K[7]="secret";

....
```

```
  int integrityPadding2(unsigned char *P2)
  {
        const char* sloc = "secret";

      flag12=0;
      if (K != sloc){
        flag21=0;
        flag22=0;
        return BAD;
      }

      return GOOD;
  }
};

....

int main(int argc, char* argv[]){
....

if(flag11 != flag12){

  return BAD;

}
....
```

4.2.11.3 Description

In this case, the attacker aims to bypass the conditional branch which is based on the *flag11* and *flag12* globals. The global variables are not marked by the security analysts as *sensitive*. The situation is different from *Case10*, since *flag12* is not tainted from any sensitive variable, so it will not be detected by the *Annotation Pass* and not marked as sensitive.

We can observe that *flag11* is initialized with 0, while *flag12* is initialized with 1. The smartcard target program requires that the *integrityPadding2* method is executed as a mandatory security mechanism (implies *flag12* to be set to 0). Therefore, the value of *flag12* is changed to 0. However, in the control flow of the program, the *if* conditional branch from the snapshot from above is present. The conditional branch checks if the two variables are equal. If not, the program will return *BAD*. Basically, the conditional branch implies the indirect checking of the conditional branch which is presented in *Case9*. The *Analysis Pass* does not identify the vulnerability, so the security check is not bypassed.

Successful	No
Vulnerability type	Direct and Tainted Dependencies
Part of	Test05

Table 4.14: Case12

4.2.12 Case12

4.2.12.1 Overview

Case12 is part of the *Test5* target program which implements mechanisms against data leakage [39]. This case illustrates a situation when the security analyst does not annotates as *sensitive* a variable which influence the values of multiple other values used in sensitive checks. This situation results in a cascade of unidentified dependencies.

The situation is shown in the following code:

4.2.12.2 Code snapshot

```
__attribute__((annotate("DS"))) char masterkey[256];

int parityCheck1=1; //should remain 1
int parityCheck2=1; //should remain 1
......

int checkParity ( char* key, int offset) {

char odd_parity[]= {
1, 1, 2, 2, 4, 4, 7, 7, 8, 8, 11, 11,.... 113, 113, 117, 117,
153, 153, 154, 154, 199, 199, 200, 200, .....
248, 248, 251, 251, 253, 253, 254, 254};

  int r = rand() & 255;

  for ( int i=0, j = r; i<256; i++, j = (j+1)&255 ) {

    if (key[j] != odd_parity[key[j+offset]]) {
     parityCheck1 = 0;
     return 0;
    }
  }

return 1;
}

.....
```

```
int main(int argc, char* argv[])
{

__attribute__((annotate("START"))) int asd;

char keycode[256];

....

for (int i = 0, j = (rand() & 255); i < 256; i++, j = ((j+1) & 255))
  keycode[j] = masterkey[j];

int flag=checkParity(keycode, 1);

if(!checkParity(keycode, 1))
  return BAD;

if(!flag)
  return BAD;

if(!parityCheck1){
  parityCheck2=0;
  return BAD;
}

if(!parityCheck2)
  return BAD;

....
```

4.2.12.3 Description

The code snapshot is part of *Test5* which implements security checks based on tainted variables. The purpose of this case is to show that the tainted variables of a level greater than 1 are not identified, as mentioned in *Section 3.3*. The security analysts only marked *masterkey* as a sensitive variable. Inside the *main* function, the value of the *masterkey* is safely copied into the local variable *keycode*. The copying is done by starting at a random index in the range [0,3]. The *Annotation Pass* identifies this *sensitiveness* propagation and automatically marks the *keycode* local variable as sensitive. A conditional branch successfully identified by the *Analysis Pass* (we do not explain the procedure since it is

similar to *Case2*) calls the *checkParity* function.

The output of the *Annotation Pass* shows the two marked variables:

```
%arrayidx = getelementptr inbounds [16 x i8]* @masterkey, i32 0,
i32 %1, !dbg !956, !sensitive1 !957

%keycode = alloca [16 x i8], align 1, !sensitive !938
```

The *checkParity* function uses an alias for the *keycode* local variable, similar to *Case5*. Since our application does not consider the sensitiveness propagation using aliases (it considers only for particular cases, as explained in *Case5*), the conditional branch from the *checkParity* function is not considered by the *Analysis Pass* as an instruction to be bypassed. The *key* parity is checked by using a table. Each entry of the table represents the odd parity of the index. The parity verification is made inside a loop that is selecting random numbers for the index. The loop is considering all key bytes. If the key does not respect the parity, the *checkParity* function will return 0. The function returns 1 if all bytes were found to be odd. This result will not affect the attack on the *if(!checkParity(keycode, 1))* conditional branch from the *main* function, since this sensitive condition is already identified as a place to insert fault injection, being dependent on the *keycode* sensitive variable. However, if the conditional branch from the *checkParity* function is executed, then the value of the *parityCheck1* flag is changed to 0. The security checks are safety passed if the *parityCheck1* and the *parityCheck2* flags remains 1.

The sensitiveness is succesfully propagated from the *masterkey* to the *keycode* variable which is used inside the computation. The procedure of this taint identification is described in the following paragraph. For this type of taint dependency, the *Annotation Pass* identifies all the *store* instructions which have the *source* operand (the one from which it copies the value) marked as sensitive. There are two cases: the source operand can be a global variable or a local variable. For the local variable, the tool checks directly if the variable contains the *sensitive* metadata. For the global variables, the tool searches in depth the instructions which carry the *sensitive* metadata for them: *load* instructions or *getelementptr* instructions (as explained in *Chapter3*). After the identification of the sensitive source instructions, the *Annotation Pass* marks as sensitive the *destination* instructions (the operand of the *store* instructions in which the sensitive values are copied). Again, this instruction can be local and the *sensitive* metadata is directly added, or global and in this situation that implies to find an instruction to carry the metadata.

The aim of the attacker is to bypass the security checks dependent on the 2 flags mentioned above. However, in the case when the alias sensitiveness is not propagated to the local variable *key*, the tainted variable *parityCheck1* will not be considered sensitive. Therefore, the *Analysis Pass* does not identify the *if(!parityCheck1)* condition as a place to insert fault injection. **On the other hand**, we assume that we are using a similar function as the one presented in *Case5* and the alias *key* is considered sensitive. Consequently, the *Annotation Pass* marks the *parityCheck1* as sensitive, so the *Analysis*

Successful	Yes
Vulnerability type	Tainted and Direct Dependencies
Part of	Test09

Table 4.15: Case13

Pass would identify as a result the conditional branch dependent on the *parityCheck1*. However, the *Annotation Pass* does not mark as sensitive the tainted variable *parityCheck2*, as explained in *Section 3.3*. Our application is able to determine only the first level tainted variables, so the *Annotation Pass* does not mark *parityCheck2* as sensitive. In order to solve this problem, we propose the symbolic execution based on KLEE as a future work. Our application does not identify the tainted variables in a *cascade* of taint propagation blocks.

4.2.13 Case13

4.2.13.1 Overview

Case13 aims to show that in some situations the vulnerabilities from *Case12* are identified by our application. This does not mean that the *Annotation Pass* always identifies the sensitiveness propagation in a cascade of tainted blocks, as we can see in *Case12*. However, the identification depends on the order of the sensitiveness propagation by taint analysis. This situation is described in detail in the *Description* section. This case, as well as the next presented cases - *Case14* and *Case15*, are part of *Test09* from the test suite. This case shows the usage of the *double* data type inside the conditional branches. In this case the LLVM *fcmp* instruction being used. In addition, we present the conditional branch containing a direct dependency which includes the above-mentioned tainted variables.

The relevant part of the code for *Case13* from *Test09* is shown below:

4.2.13.2 Code snapshot

```
#define BAD 0
#define GOOD 1

#define MAX_PIN_CTR 3
#define CLA 0x00
#define INS 0x01
#define P1 0x02
#define P2 0x03
#define LC 0x04
#define DATA 0x05
#define MAX_WRONG 2
#define MAX_FAULTS 12
```

```
#define METHOD1 13
#define METHOD2 17


/** Global vars **/

uint8_t buffer[35];

unsigned char deskey[8];

unsigned char ee_deskey[8];

int faultCounter=0;

int flowCounter=0;

__attribute__((annotate("DS"))) unsigned
char pin[4]={8,2,6,9};
unsigned char ee_pin[4]= {8,2,6,9};

uint8_t pin_ctr = MAX_PIN_CTR;
uint8_t auth=false;
double pinwrong=0;

int process(){
__attribute__((annotate("START"))) int asd;
switch(buffer[CLA]){
  case 0x00:  // select application, or
  //whatever starts with CLA=0x00
    return 0x08;
    break;

  case 0xA0:  // specific defined app's
    if ( 0x02 == buffer[INS] && buffer[LC]==0x04)
      set_pin();
    else if(0x04 == buffer[INS] &&
    0x00 == buffer[P1] &&
    0x00 == buffer[P2] &&
    0x08 == buffer[LC])
      do_des();
    else if ( 0x08 == buffer[INS] && buffer[LC]==0x08 )
      des_decrypt();
    else if ( 0x19 == buffer[INS] && buffer[LC]==0x04 )
      better_pin_double();
    else if ( 0x1B == buffer[INS])
```

```
      is_auth();
    else
    //Echo command
      return (DATA+buffer[LC]+1);
      break;
    default:  // illegal card usage
      return 0x00;
      break;
  }

}

int main(){

//Enable global interrupts
//sei();


while(1){
//Read Command
//readAPDU();
//Check command and act accordingly
//determine();


  int d=MAX_FAULTS;

  if(faultCounter>d)  // line 110
    return BAD;  // line 111

  method1();
  flowCounter -= 2*METHOD1;
  if(flowCounter != 0)   // line 115
    return BAD;   // line 116

  return GOOD;

}
//unreachable code...
just avoiding compiler complaints
return 0;
}


//explic din paper + ce am eu
```

```
int method1(){
  int localCounter;
  flowCounter += METHOD1;
  localCounter = flowCounter;
  method2();
  flowCounter -= 3*METHOD2;

  if(flowCounter != localCounter)   // line 134
    return BAD;  // line 135

  flowCounter += METHOD1;
}

int method2(){
  double c=MAX_WRONG;
  if(pinwrong>c){ //fcmp  // line 142
    faultCounter++;
    return 0x69;  // line 144
  }
}

 //PIN implementation with auth flag,
 // double verification using better
 //comparison routin and try counter

int better_pin_double(){
  int i,j;
  if(pin_ctr>0) {
    --pin_ctr;
    int r = rand() & 3;
    for ( i=0, j = r; i<4; i++, j = (j+1)&3 ) {
      if (pin[j] != buffer[j]+DATA) {  // line 158
        auth=false;  // line 159
        pinwrong++;
        return 0x69; // line 161
        }
      else {
.....
```

4.2.13.3 Description

The scope of the attack is to bypass all the security checks which can lead to a dangerous operation, like disabling the functionality of the smartcard. The target test program

implements the Riscure patterns *FAULT.RESPOND* and *FAULT.FLOW* introduced in *Section 2.4*. *Case13* presents the situations when the unidentified vulnerabilities presented in *Case12* are still identified.

The *FAULT.FLOW* Riscure pattern introduces extra security checks among the code which are dependent on flow counters, as we see in the code example. The flow counter flags (*flowCounter* and *localCounter*) have the purpose to check if important parts of code where executed or if the smartcard is under attack. The procedure is explained in details in *Section 2.4*. The *FAULT.RESPOND* Riscure pattern checks the number of faults and reacts to them. It was integrated in the test program *Test09*.

The *Annotation Pass* marks the sensitive variables in the LLVM bitcode and then executes the taint analysis algorithm. As we can see from the code example, the variable *pinwrong* is identified by the taint algortihm before the variables *faultCounter* and *flowCounter*. These 2 variables are tainted from *pinwrong*. In addition, *faultCounter* is identified as sensitive before the *flowCounter* variable. *flowCounter* is tainted from *faultCounter*. This situation shows a 3-level taint propagation which is identified by out application, but only because the variables are tainted in the correct order (as we explained in *Section 3.3*). As we can see, another order can make the taint algorithm to not identify all the tainted variables. We mention again that the taint algorithm was not been improved since we made the decision to use symbolic execution, which is described in the *Future work* section. An example of the output of the *Annotation Pass* involving the *pin*, the *auth* and the *pinwrong* variables is:

```
%arrayidx = getelementptr inbounds [4 x i8]* @pin, i32 0,
i32 %4, !dbg !943, !sensitive1 !945
%0 = load i8* @auth, align 1, !dbg !932, !sensitive1 !934
%0 = load double* @pinwrong, align 8, !dbg !935, !sensitive1 !936
```

We can see that the sequence of LLVM instructions in the bitcode does matter for *Case13*. The *pin* global variable is the **single** variable which is annotated as sensitive by the security analysts in the target program source code. The rest of the identified variables are tainted directly or indirectly (higher level of taintness) from *pin*.

The taint analysis algorithm of the *Annotation Pass* identifies the tainted variables similar to the description from *Case10*, by searching in depth for the variables. The first step is to identify the instruction which accesses the sensitive global variable. The local variable which loads the value of *pin* is identified after a search in depth for the instructions which have *pin* as an operand. This case of vulnerability includes a larger sequence of instructions until the right instruction which accesses the sensitive variables is found (*getelementptr*). The second step is to look for the identified global variable into the array containing the instructions which carries the *sensitive* metadata added by the *Annotation Pass*. After our application verifies that the global variable is sensitive, it identifies the succesors of the conditional branch. For every succesor (represented by a basic block), the *store* instructions are identified in order to see which *store destination*

operands are influenced by the *pin* sensitive variable (as explained for *Case12*). The identified variables are automatically marked as sensitive by the *Annotation Pass*.

The sensitiveness is propagated starting from the *if (pin[j] != buffer[j]+DATA)* conditional branch. From this branch, the *auth* and *pinwrong* variables are marked as sensitive by following the algorithm described above. As mentioned at the begining of this section, the order in which the taint analysis is done is important. After the *Annotation Pass* marks *pinwrong* as sensitive, the LLVM *fcmp* conditional branch *if(pinwrong>c)* influences the value of *faultCounter*. In a similar way, the *Annotation Pass* marks the *faultCounter* variable as sensitive. Analogously, the *Annotation Pass* marks the *flowCounter* variable as sensitive based on the *if(faultCounter>d)* conditional branch.

The *Paths Pass* identifies all the possible execution paths between the marked *start* and *end* points. A snapshot of 4 short execution paths from the total of 203 is presented below:

```
15 17 20 21 22 23 25 26 28 29
               31 32 56 58 60
15 17 20 21 22 23 25 26 28 29 31
      32 56 58 59 61 32 56 58 60
15 17 20 21 22 23 25 26 28 31 32 56
                  57 61 32 56 58 60
15 17 20 21 22 23 25 26 28 31 32 56
                           58 60
```

The third tool of our application is the *Analysis Pass*. The **relevant** part of the results for this case are shown in the following results:

```
The instruction if from the line number 158 is dependent
on the sensitive global @pin defined at the line number 54

The instruction if from the line number 110  is dependent
on the sensitive global @faultCounter defined at the line number 36

The instruction if from the line number 115  is dependent
on the sensitive global @flowCounter defined at the line number 38

The instruction if from the line number 134  is dependent
on the sensitive global @flowCounter defined at the line number 38

The instruction if from the line number 142  is dependent
on the sensitive global @pinwrong defined at the line number 61
```

Or in LLVM representation:

```
%cmp5 = icmp ne i32 %conv2, %add, !dbg !944  158  @pin = global [4 x i8]
```

```
                                                    c"\08\02\06\09", align 1  54
%cmp = icmp sgt i32 %0, %1, !dbg !937  110  @faultCounter = global i32 0,
                                                     align 4  36
%cmp1 = icmp ne i32 %3, 0, !dbg !942  115  @flowCounter = global i32 0,
                                                   align 4  38
%cmp = icmp ne i32 %3, %4, !dbg !940  134  @flowCounter = global i32 0,
                                                   align 4  38
%cmp = fcmp ogt double %0, %1, !dbg !935  142  @pinwrong = global double
                                                 0.000000e+00, align 8  61
```

As we can observe, all the conditional branches dependent on the already-annotated variable *pin* (as a direct dependency) and the tainted variables (as tainted dependencies from it) were identified. Using the results, the attacker can successful bypass of the security checks implemented by the Riscure patterns *FAULT.RESPOND* and *FAULT.FLOW*, which are explained in detail in *section 2.4*.

The reason of the appearance of the *fcmp* conditinal branch instruction is that it is dependent on a global variable of type *double*. The *fcmp* instruction deals with floating point operands, in contrast with the *icmp* instruction.

Based on our application's output, the security analysts will insert fault insert at the line 111 (the *return BAD* statement) for the *faultCounter*, at the lines 159 (the *auth=false* statement) and 161 (the *return 0x69* statement) for the *pin* global, at the line number 116 (the *return BAD* statement) and 135 (the *return BAD* statement) for the *faultCounter* flag and at the line number 144 (the *return 0x69* statement) for the *pinwrong* global variable. The *return 0x69* statement is similar with the returning *BAD* statement, which is explained in this chapter.

Case13 shows good results given by our application for a successful fault injection attack.

As we can observe, the paths calculated by the *Paths Pass* which are seen in the *Case13* do not have very long traces. The reason is the target programs structure: from a *switch* construct the user can choose an option from the smartcard. An example of one option is presented in the code from the *Case14* (part of the same test), where the security analysts annotated the *end* point inside the *is_auth* method. Therefore, the *Paths Pass* will compute all the possible execution paths only between the switch with the smartcard options and the point within the specific method. This way, our applications output will tell all the vulnerabilities of the computed execution paths, while the vulnerabilities from the rest of the code will be shows standalone.

4.2.14 Case14

4.2.14.1 Overview

Case14 is part of the same target program *Test09*. It gives an example with a *switch* conditional branch and it shows a false positive result. Also, we show an example of

Successful	Yes
Vulnerability type	Tainted Dependencies
Part of	Test04, Test08, Test09, Test10, Test11

Table 4.16: Case14

complementary results.

The relevant part of the code for the *Case14* from the *Test07* is shown below:

4.2.14.2 Code snapshot

```
 //PIN implementation with auth flag,
 // double verification using better
 //comparison routin and try counter

int better_pin_double(){
  int i,j;
  if(pin_ctr>0) {
    --pin_ctr;
    int r = rand() & 3;
    for ( i=0, j = r; i<4; i++, j = (j+1)&3 ) {
      if (pin[j] != buffer[j]+DATA) {
        auth=false;
        pinwrong++;
        return 0x69;
        }
      else {
        //Authentication complete!
        auth=true;
        pin_ctr++;
        return 0x90;
      }
    }

    switch(pin_ctr){   // line 174
      case 3: auth=true; return 0x90; break;
      default: auth=false; pinwrong++; return 0x69;
    }

  } else {
      auth=false;
      pinwrong++;
      return 0x69;
  }
```

```
}


// Retrieve whether authentication was successful or not
int is_auth(){
  if(auth==true){
    buffer[0]=0x01;
    return 0x90;
  } else if (auth == false){
  buffer[0]=0x00;
  return 0x90;
  } else { __attribute__((annotate("END"))) int qwe;
  return 0x69; } // line 196

}
```

4.2.14.3 Description

This case is part of *Test09* and is linked to *Case13* which was previously presented. The *Annotation Pass* executes the integrated taint algorithm which finds the *auth* and the *pin_ctr* flags as variables which inherit the sensitiveness analogously as described in *Case13*.

```
%0 = load i8* @auth, align 1, !dbg !932, !sensitive1 !934
%1 = load i8* @pin_ctr, align 1, !dbg !938, !sensitive1 !937
```

The execution paths are the same as in *Case13* due the fact that the start and the end points in the program are preserved. The *Analysis Pass* identifies the dependency between the tainted variable *pin_ctr* and the *switch* conditional branch from the line 174. The line of the output presenting the situation is:

```
The instruction switch from the line number 174 is dependent
on the sensitive global @pin_ctr defined at the line number 58
```

In LLVM format:

```
  switch i32 %conv13, label %sw.default [i32 3, label %sw.bb
], !dbg !957  174  @pin_ctr = global i8 3, align 1  58
```

The *pin_ctr* variable leads to a false positive result. The result is identified by our application as it should be. Hence, the analysts have the task to determine the false positive results. The result is given by the conditional branch *if(pin_ctr¿0)* from the lines number 154. This conditional branch is not a security check and this fact can been seen immediately by the analysts by observing that there are no dangerous instructions which lead to security measures on the smartcard (like the *return BAD* statement present in all the test cases). The output with the result is:

```
The instruction switch from the line number 154 is dependent
on the sensitive global @pin_ctr defined at the line number 58
```

In LLVM format:

```
%cmp = icmp sgt i32 %conv, 0, !dbg !936  i32 154
@pin_ctr = global i8 3, align 1  i32 58
```

The *pin_ctr* variable is an example on how the taint analysis from the *Annotation Pass* works. The *pin_ctr* variable is tainted from the *else* branch of the *if (pin[j] != buffer[j]+DATA)* conditional branch, more exactly when it is incremented at the line number 166. The *Analysis Pass* identifies the dependency from the line 154 as seen above.

In addition, *Case14* shows an example of complementary results. The *Analysis Pass* identifies the security check *if (pin[j] != buffer[j]+DATA)* in *Case13*. Based on this result, the security analysts insert fault injection to the next line number, in order to skip the statement which sets the *auth* flag to false. However, the *auth* flag is marked as sensitive (as mentioned above). Therefore, the *Analysis Pass* identifies the following vulnerabilities:

```
The instruction if from the line number 189 is dependent
on the sensitive global @auth defined at the line number 59

The instruction if from the line number 192 is dependent
on the sensitive global @auth defined at the line number 59
```

In LLVM format:

```
%cmp = icmp eq i32 %conv, 1, !dbg !934  189  @auth = global i8 0,
                                                      align 1  59
%cmp2 = icmp eq i32 %conv1, 0, !dbg !941  192  @auth = global i8 0,
                                                      align 1  59
```

We can observe the following situation: assuming that at a specific moment of time *auth* is set to true, the security analysts insert fault injection and skip the *auth=false* operation, described in *Case13*. Therefore, the value of the flag remains true, so the dependencies identified above (the two *if* statements from lines 189 and line 192 dependent on *auth*) will not lead to the dangerous statement *return 0x69*. Therefore, in this situation the fault injection is not necessary for these two dependecies, since only the branches related to *auth* (having the *true* value) will be executed. However, as mentioned before in this thesis, this is **not** the scope of our application. The security analysts have the task of deciding which results are complementary.

Based on the results provided by the *Analysis Pass* for *Case14*, the security analysts insert fault injection at the line 196, in order to avoid the *return 0x69* instruction.

Successful**	No
Vulnerability type	Tainted Dependency
Part of	Test09, Test12

Table 4.17: Case15

4.2.15 Case15

4.2.15.1 Overview

Test09 brings another vulnerability case. *Case15* is derived from the implementation of the Riscure pattern *LOOPCHECK*.

**This case is considered unsuccessful in the general case. We can optionally add a small algorithm to the *Annotation Pass*, which works for particular cases. The solution is presented in the *Description* section.

The part of the code from *Test09* illustrating the situation is:

4.2.15.2 Code snapshot

```
int better_pin_double(){
......
    for ( i=0, j = r; i<4; i++, j = (j+1)&3 ) {
      if (pin[j] != buffer[j]+DATA) {
        auth=false;
        pinwrong++;
        return 0x69;
        }
      else {
        .....
    }


    if(i != 4){ //important loop that must be completed
      return 0x69;
    }
```

4.2.15.3 Description

This vulnerability type comes from the implementation of the Riscure pattern *LOOPCHECK* described in *Section 2.4*. A conditional branch checks the counter of the important loop containing the security operations presented in *Case13* and *Case14*, with the **purpose** to verify if the loop has been completed for every element of the *pin* array. The attackers can skip the security operations from the loop, as seen in *Case13*. Therefore, the smartcard developers applied the Riscure pattern *LOOPCHECK* to check the loop completion. Initially, the security analysts did not annotate the loop counter as a sensitive variable. Since the loop counter does not have any direct link with any

```
//pseudocode

foreach BasicBlock.name ==for.inc

    //store all the loop counters
    foreach load -> getOperand -> store array

    //get to the branch instruction of for.inc
    if (BasicBlock.iterator == br instruction)

        //it jumps to the for condition basic block
        if (br.getOperand.name == for.cond)

            //get the branch instruction of for.cond
            if (BasicBlock.iterator == br instruction)

            //if an instruction is using an already
            //sensitive marked variable
            if (BasicBlock.iterator.getOperand is sensitive)
            add sensitive metadata to the counter from the array
```

Figure 4.2: Case 15 pseudocode solution

sensitive variable, any dependency between a conditional branch and the counter is not considered as a vulnerability.

The solution is analogous to the one presented in *Case5*, with the exception of the algorithm integrated in the *Annotation Pass* which is shown in *Figure 4.2*.

This algorithm can be embedded into the *Annotation Pass*. Therefore, the *Analysis Pass* will identify the tainted dependencies between the loop counter and the security check in the smartcard target program.

The solution is not permanently integrated into our application, since the symbolic execution is a better solution for the taint analysis, as explained in the *Future work* section.

4.3 Results and Validation

The success rate of the evaluation of our application is presented in *Figure 4.3*. For each test that is part of our test-suite, we can see the percentage of the identified vulnerabilities that can be exploited using fault injection (in blue) and the percentage of the unidentified vulnerabilities (in red). We can observe that all the vulnerabilities were identified for *Test02*, *Test08*, *Test10* and *Test11*. We explain for every test the success rate and the reason for the corresponding percentage value.

Test01 is an experimental test program, which was used to present the functionality of our application in *Section 3.6*. It implements the FAULT.BRANCH, the FAULT.DOUBLECHECK, the FAULT.DETECT and the FAULT.CONSTANT.CODING Riscure patterns. *Figure 4.3* aggregates the results and shows the success rate of 80% for *Test01*, as we can see in *Figure 4.4*. This percentage shows that a fifth of the vulnerabilities are unidentified, which are related by *Case5* (the

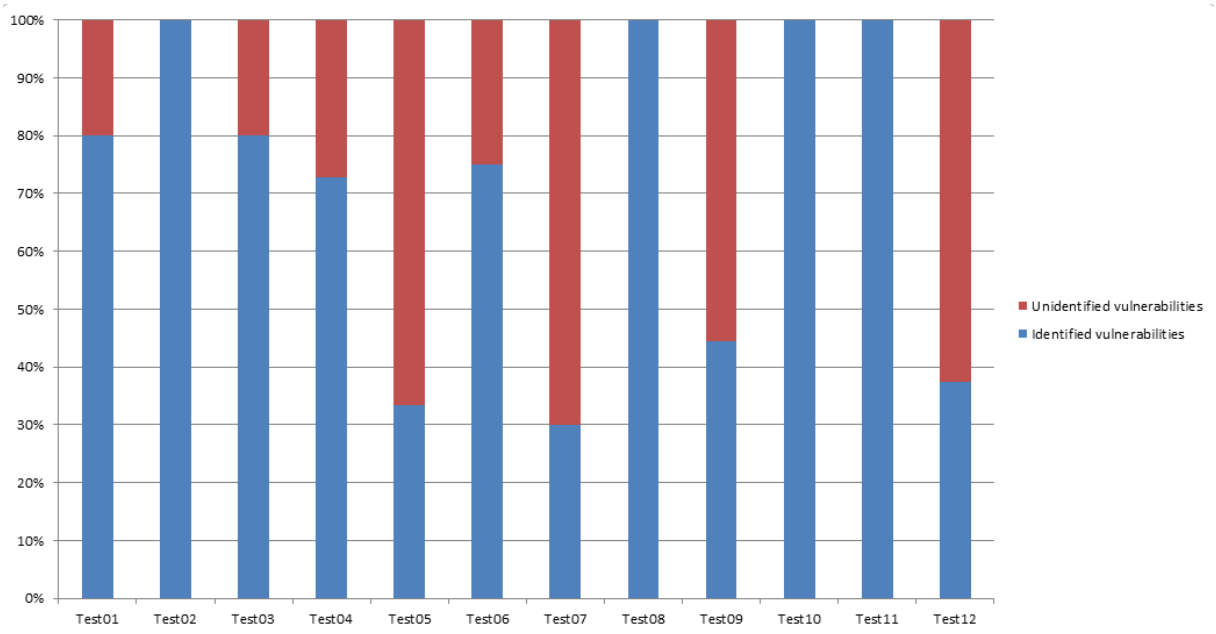

Figure 4.3: Success rate of the Results

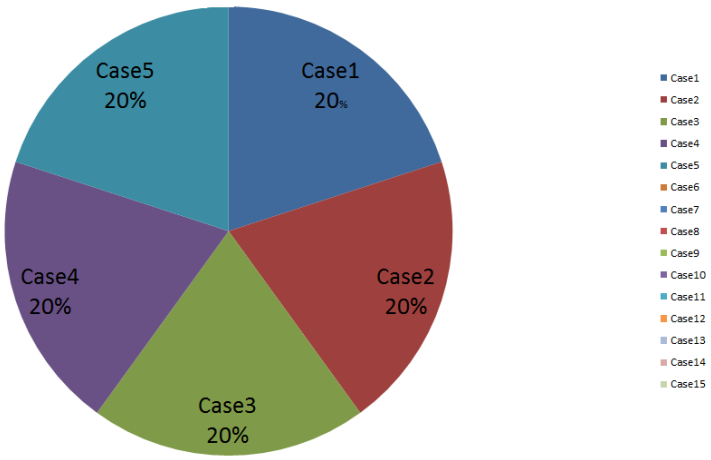

Figure 4.4: Vulnerability types of Test1

alias dependency is not identified by our application). This test program contains the most common smartcard vulnerabilities (as encountered in our test suite). As mentioned in *Section 4.2.5.*, we can increase the success rate at 100% if we include the algorithm described in *Section 4.2.5.* in the *Annotation Pass*. However, we consider unsuccessful the *Case5* vulnerability type because the described algorithm from *Section 4.2.5.* is not generic. The generic solution is presented in the *Future work* section.

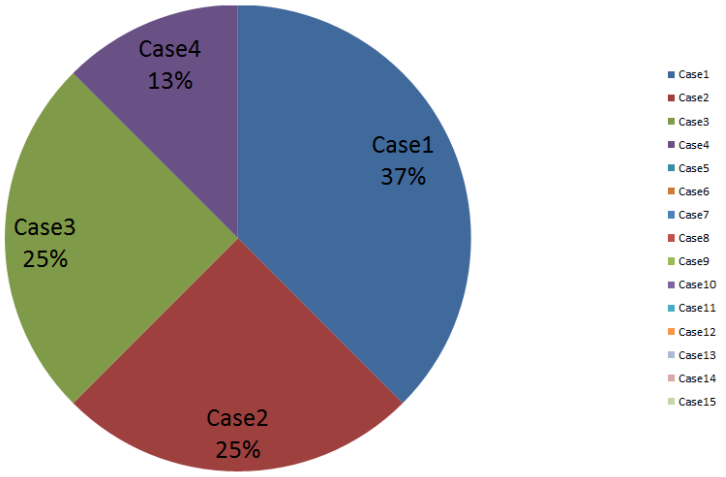

Figure 4.5: Vulnerability types of Test2

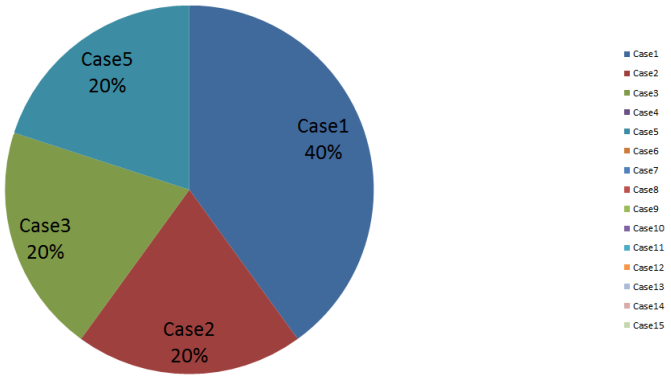

Figure 4.6: Vulnerability types of Test3

The second test of the evaluation suite has a success rate of 100%. It uses only smartcard vulnerabilities which were identified by our application, as we can see in the *Figure 4.5*. It is an experimental program and it implements the FAULT.DOUBLECHECK, the FAULT.DETECT and the FAULT.CONSTANT.CODING Riscure patterns.

We observe that only unidentified vulnerabilities of *Test03* are related to alias analyzing (as we seen in *Figure 4.6*). This vulnerability type is presented in *Section 4.2.5*. It is a test program from a Riscure training target and it implements the FAULT.BRANCH, the FAULT.CRYPTO and the FAULT.CONSTANT.CODING Riscure patterns. The success rate for this test program is high, at 80%.

Test4 is a test program from a Riscure training target based on the implementation of the FAULT.BYPASS Riscure pattern. The other included Riscure pat-

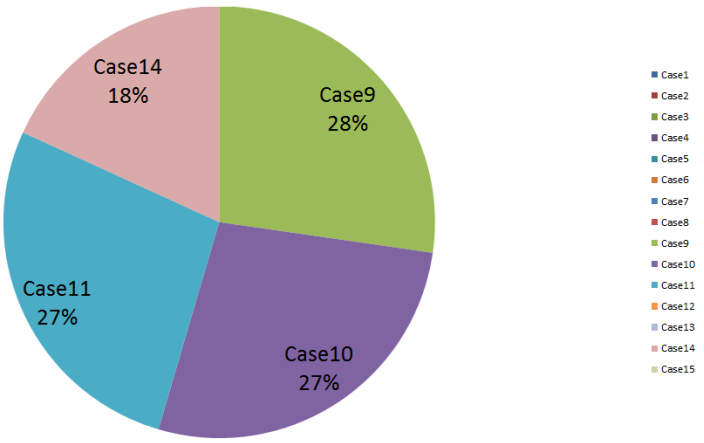

Figure 4.7: Vulnerability types of Test4

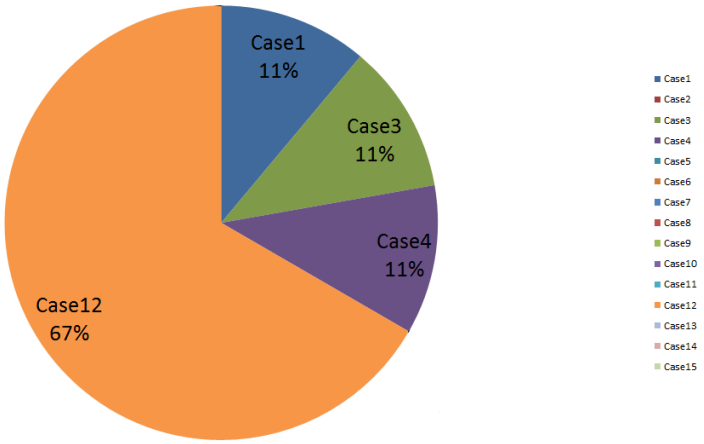

Figure 4.8: Vulnerability types of Test5

terns are FAULT.BRANCH, FAULT.DEFAULTFAIL, FAULT.CONSTANT.CODING and FAULT.CRYPTO. We can see in *Figure 4.3* that the success rate is 73%. The special type of direct dependency vulnerabilities related to *Case11* are not identified by our application (as we seen in *Figure 4.7*). The unidentified vulnerabilities are given by the side effects of inserting fault injection, as we describe in *Section 4.2.11.*

Test5 is a smartcard test program from a Riscure training target that has a success rate of 33%. The small rate of successful identified vulnerabilities is related to *Case12*, which is explained in *Section 4.2.12.* In this test program, we assumed that the security analysts did not annotated the sensitive variable which influences the values of multiple variables used in security checks among the program flow. Therefore, the *Annotation*

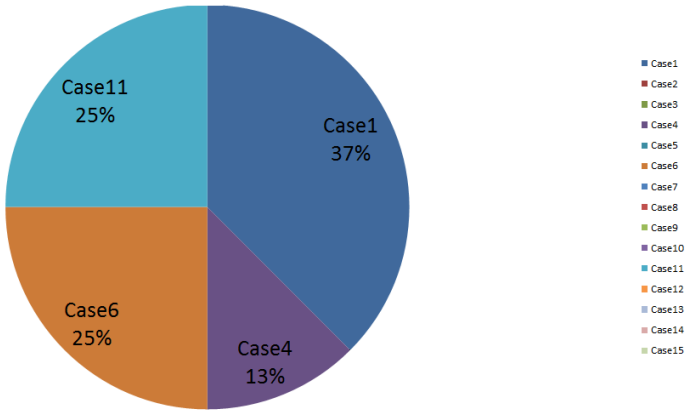

Figure 4.9: Vulnerability types of Test6

Pass does not mark the tainted variables as being sensitive, resulting in a cascade of unidentified tainted dependencies. This leads to a percentage of 67% of unidentified vulnerabilities, as we see in the *Figure 4.8*. The Riscure patterns FAULT.BRANCH, FAULT.CRYPTO and FAULT.CONSTANT.CODING are included in this test.

Test06 is a smartcard test from the public domain based on the Oracle padding function. The device target under attack is a SIM smartcard for mobile phones. This test includes the FAULT.BYPASS, FAULT.BRANCH, FAULT.DETECT, FAULT.CONSTANT.CODING and FAULT.CRYPTO Riscure patterns. The percentage of identified smartcard vulnerabilities is 75%. Like in the case of *Test4*, our application does not identify the *Case11* vulnerability type (as we seen in *Figure 4.9*). This vulnerability type is a special type of direct dependency vulnerability resulted as a side effect of inserting fault injection (previously in the program execution).

We can see in *Figure 4.10* the cases which are contained in the *Test07* test program (from a Riscure training target). The test program uses the DES encryption algorithm. The included RISCURE patterns are FAULT.BYPASS, FAULT.BRANCH, FAULT.CONSTANT.CODING and FAULT.CRYPTO.

The success rate is 30%. This low rate is a result of the presence of 3 unidentified vulnerability types. Two of them are presented in *Case5* and *Case11*. In addition, one fifth of the cases is represented by the *Case7* vulnerability type. This is an unidentified direct dependency between an encryption operation and a sensitive variable. Since the vulnerability is not a security check, our application does not consider it as a location to insert fault injection.

The purpose of the test program from a Riscure training target *Test8* is to check if our application preserves its functionality in the case of Java target test programs. The success rate is 100%. We introduced the *Case8* vulnerability type in *Section 4.2.8.*, which denotes a vulnerability found in a Java smartcard program. However, as we mentioned in *Section 4.2.8.*, this vulnerability type is different from the others only from the point of view of the programming language of the smartcard. Therefore, *Case8* can be translated

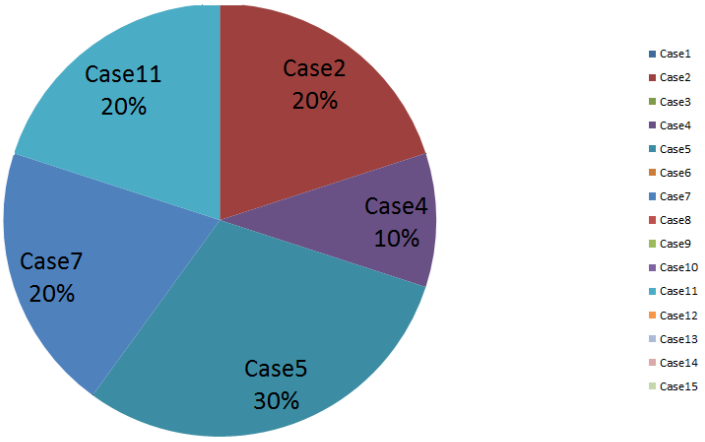

Figure 4.10: Vulnerability types of Test7

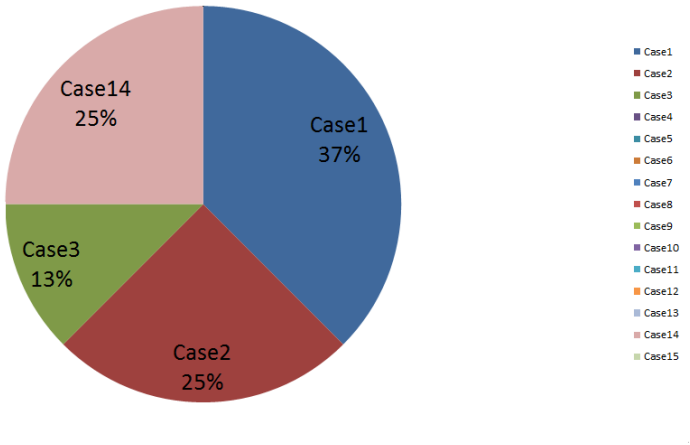

Figure 4.11: Vulnerability types of Test8

into 4 types of vulnerabilities, as we see in *Figure 4.11*.

The included RISCURE patterns are FAULT.DOUBLECHECK, FAULT.BRANCH, FAULT.DEFAULTFAIL, FAULT.DETECT, FAULT.CONSTANT.CODING and FAULT.CRYPTO.

The success rate of the test program *Test9* (from a Riscure training target) is 44%, as we can see from *Figure 4.3*. The lower success percentage is given by the *Case5* alias vulnerability and by *Case15* (as we seen in *Figure 4.12*). The usage of the *Case15* shows the presence of multiple implementations of the FAULT.LOOPCHECK Riscure pattern.

The included RISCURE patterns are FAULT.FLOW, FAULT.BRANCH, FAULT.RESPOND, FAULT.LOOPCHECK, FAULT.DEFAULTFAIL, FAULT.CONSTANT.CODING and FAULT.CRYPTO.

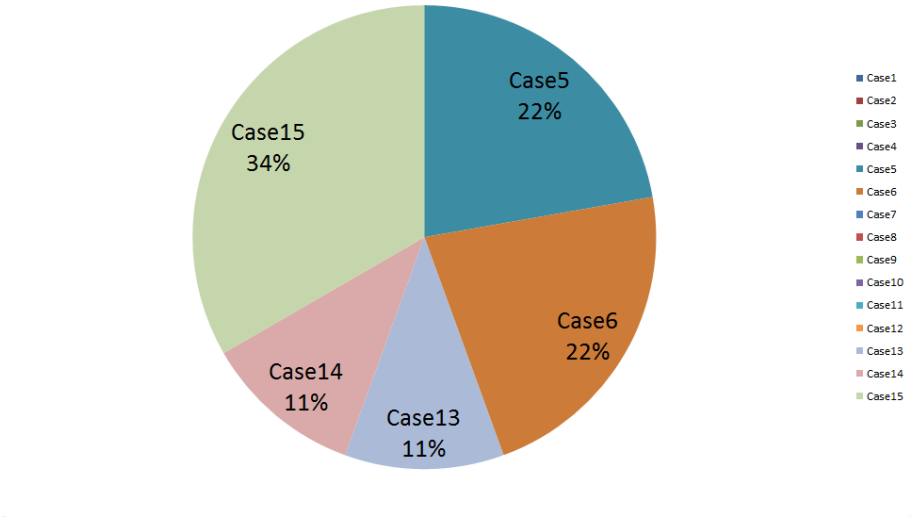

Figure 4.12: Vulnerability types of Test9

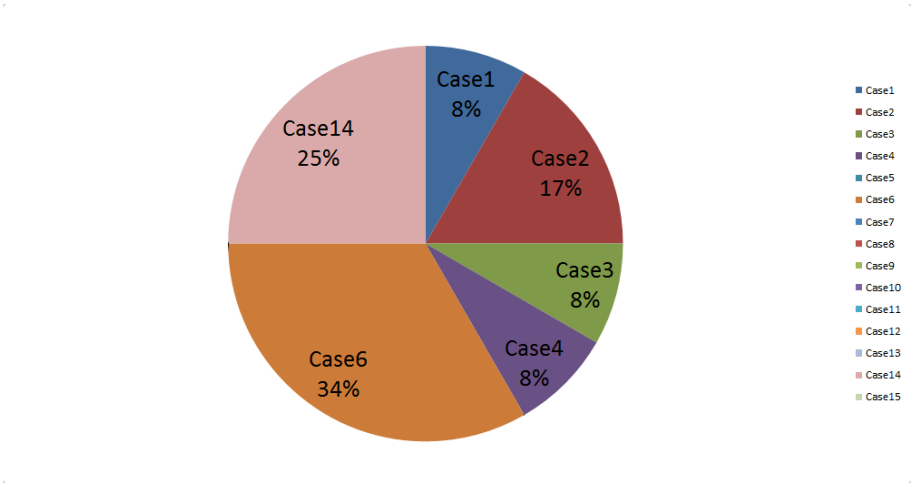

Figure 4.13: Vulnerability types of Test10

The success rate of *Test10* is 100%, our application identifying all the 6 vulnerability types shown in *Figure 4.13*. The Riscure patterns included into this test program from a Riscure training target are FAULT.BRANCH, FAULT.DOUBLECHECK, FAULT.DEFAULTFAIL, FAULT.DETECT, FAULT.CONSTANT.CODING and FAULT.CRYPTO.

The rest test target program *Test11* has a success rate of 100% in identifying the vulnerabilites which can be exploited by fault injection. The test contains 6 vulnerability types, as we see in *Figure 4.14*. The Riscure patterns used for this test case are FAULT.BYPASS, FAULT.BRANCH, FAULT.DOUBLECHECK, FAULT.DEFAULTFAIL, FAULT.DETECT, FAULT.CONSTANT.CODING and

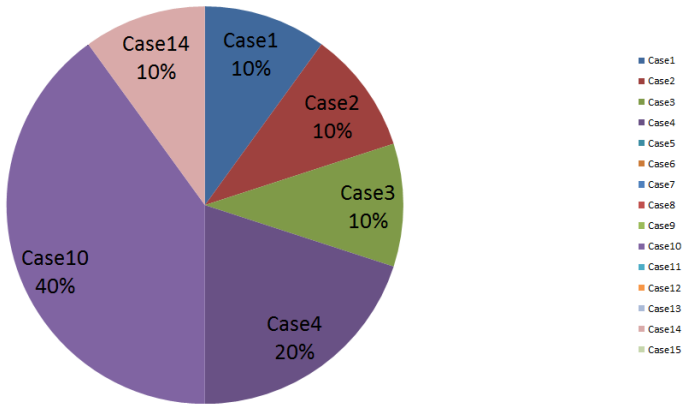

Figure 4.14: Vulnerability types of Test11

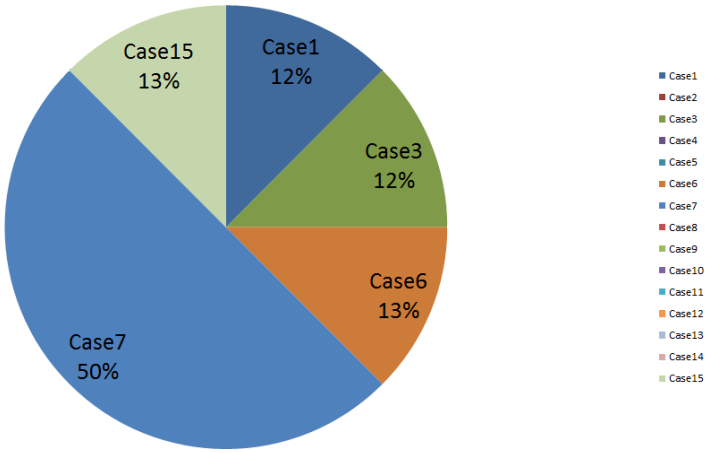

Figure 4.15: Vulnerability types of Test12

FAULT.CRYPTO.

The *Test12* test program from a Riscure training target is the most related (from our test suite) to the domain of cryptanalysis. All the other tests imply cryptographic operations, but *Test12* aims to show vulnerabilities inside the cryptographic implementations. However, the attack on cryptographic algorithms is not so relevant for our test suite, since the most **effective** way to avoid cryptographic security checks is to bypass the conditional branch invoking the cryptographic function. The succes rate is 37%. The low success rate is given by the usage of the loop counter check of *Case15* and by the *Case7* cryptographic calls which are not part of a conditional branch. The Riscure patterns used are FAULT.BRANCH, FAULT.CRYPTO, FAULT.LOOPCHECK, FAULT.DELAY, FAULT.CONSTANT.CODING. We can see the vulnerability types of

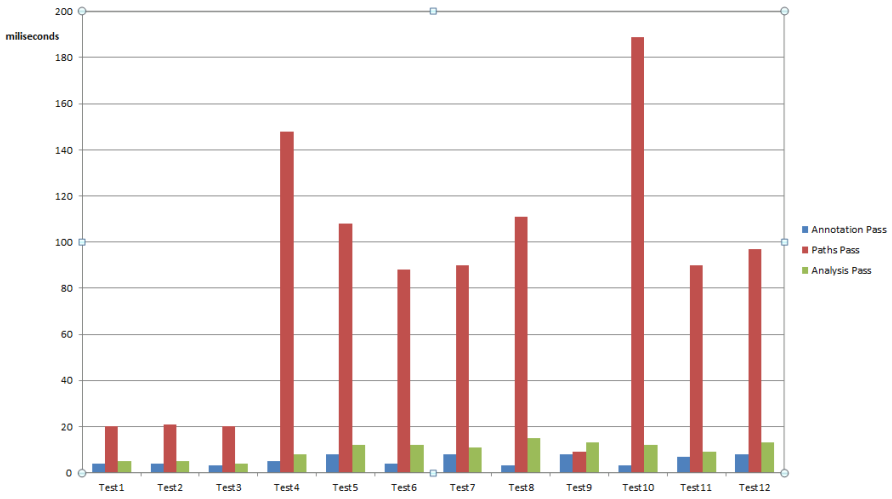

Figure 4.16: Execution times of the 3 passes

Test12 in *Figure 4.15*.

In *Figure 4.16*, the execution times of the 3 passes of our application are presented. The study is relevant in order to see which computation part is the most resource-consuming. As we can see from *Figure 4.16*, the *Paths Pass* takes the most time to execute. Based on this knowledge, the security analysts can made multiple code reviews on the target program while keeping the same *start* and *end* points **without** executing the *Paths Pass* multiple times. In this situation, the security analysts can experiment by changing the annotated sensitive variable, while preserving the same *start* and *end* points. This situation is **possible** because if the *start* and the *end* points are not modified by the analyst, then the execution paths calculated by the *Paths Pass* remain the same.

We can see in *Figure 4.16* that the execution times of the *Annotation Pass* and of the *Analysis Pass* are less than 20 miliseconds for all the tests from the test suite. The computations are not so expensive in comparisson with the processing of the *Paths Pass*. In 8 tests, the execution time of the *Paths Pass* varies between 80 and 200 miliseconds. The most expensive part of the *Paths Pass* is the *getAllPaths* method, which is used to compute all the possible execution paths between 2 nodes. However, we see that in the case of *Test 9*, the execution time is less than 20 miliseconds. The reason is that the security analysts have chosen very close the *start* and the *end* points in the **moment** when we measured the execution time. In addition, the metadata addition with the paths information is expensive. In contrast, the most expensive computation part on the *Annotation Pass* is the taint analysis algorithm, while on the *Analysis Pass* the most processing part is given by the recursive in-depth methods that search for dependencies. In conclusion, the security analysts have the task to choose carefully the *start* and the *end* points before starting the code review if the execution time is required to be minimized.

4.4 Summary

The evaluation presented in this chapter **validates** our application. The scope defined in *Section 1.1.2.* and the objectives defined in *Section 1.1.3.* are satisfied by our application. We were able to automate the code review process for finding smartcard vulnerabilities. Our test-suite based on the Riscure patterns is relevant to evaluate our application since we assume that it covers the most common smartcard vulnerabilities [39]. We used the Riscure test-suite since no benchmark exist for the smartcard software [8]. Our application successfully identified the most common smartcard vulnerability types, as we can see in *Section 4.2.* In addition, we provide solutions for the cases of unidentified vulnerabilities. Foreach type of fault injection vulnerability, we present an overview, we provide a relevant code snapshot and we give the correspondent description.

5 Conclusion

5.1 Summary

In the area of software security, *code review* is one of the most effective practices [25]. Smartcards are the chosen embedded systems for evaluation in this thesis work. Smartcards have been used in multiple applications. Some smartcards, like banking cards, present a high risk of security. In order to test the security of such devices, the developed application in this thesis work employs code review. This thesis project includes an application that allows the code review process to be performed in an automated way. The reason for developing the application is that the manual code review is error-prone and it is costly with respect to time and workload. The developed application is focused on finding security fault injection vulnerabilities in a smartcard program.

This document presents an overview of smartcard vulnerabilities in the first chapter, together with the domain of this project and the key concepts. The problem is defined as the necessity of implementing an automated code review application targeting smartcard program. The code review leads to the identification of the smartcard vulnerabilities, that can be further exploited using the fault injection attack. The objectives of this thesis work can be epitomized as: we need to transform the smartcard source code to a common intermediate representation that is going to be analyzed automatically in order to find fault injection vulnerabilities; also, we need the evaluation of the results regarding the vulnerabilities recognition. This report gives an overview of the smartcard standards and of the testing and certification requirements for smartcards.

The implemented application for this thesis work presents support for almost all smartcard programs languages. Our application is an ethical hacking toolchain, which is used by the security analysts in order to discover vulnerabilities of smartcards. Using the LLVM compiler framework, the application transforms them into a common intermediate representation. By using the implemented application, the code review is done on the common intermediate bitcode. LLVM provides a view in LLVM assembly language of the intermediate representation, if manual debugging is required.

The developed application satisfies the scope defined in *Section 1.1.2.* and the objectives defined in *Section 1.1.3.* of this thesis work. Our application provides automation of the fault injection vulnerabilities recognition. The application was needed because the code review is currently done manually and this process is costly, error-prone and time consuming. It is developed under LLVM, that provides support for a multitude of smartcard programming languages such as C, C++, Java, Embedded C etc. The programs of our test-suite are written in these languages.

The theoretical background and setup of the developed application for this thesis work are presented in the second chapter. Our code review application contains 3 tools, the *Annotation Pass*, the *Paths Pass* and the *Analysis Pass*. The scope and the function-

ality of each tool is presented in the second chapter. For a more detailed presentation of the 3 tools, we refer the reader to the third chapter. The setup of the evaluation of our application in presented in the second chapter. The evaluation is based on a test-suite that is based on the Riscure smartcard development guideline that is composed from the Riscure patterns. In the third chapter, the functionality of each of the 3 tools that form the application is illustrated by a target program example. We refer the reader to the fourth chapter for more complex smartcard test cases.

We present for each of the 3 tools their functionality, their local objectives, their implementation, together with the connections between the tools or with the external environment. The *Annotation Pass* is the tool used to search for the source code annotations introduced by the security analysts in the code. This tool implements the taint analysis, used to check the data flow analysis performed at each step of execution of a run of the smartcard program. Basically, the taint analysis is used to determine the values that are derived from the source code annotations. Additionally, the *Annotation Pass* performs metadata addition. The security analysts annotate in the smartcard source code the sensitive local variables, the sensitive global variables and the the start and end points between which the analysis should be applied. The *Paths Pass* handles the mapping of the basic blocks from the smartcard target program into an oriented control-flow graph. This tool used an algorithm for identifying all the paths in the oriented graph and it generates the set of execution paths. The *Paths Pass* adds for each basic block the corresponding execution paths it belongs to. This informations is further used by the *Analysis Pass*. The *Analysis Pass* contains a main part that produces the final results and a part that identifies the dependencies between conditional branches and sensitive variables, in order to identify the smartcard security checks. The results of the *Analysis Pass* are used by the security analysts in order to determine the places where to insert fault injection. Basically, a result contains the conditional branch contained in the security check, the place where to insert fault injection given by the line number of the instruction to be avoided, the original sensitive variables on which the conditional branch is dependent and the place where the original sensitive variable is declared. The results are distributed to each execution path.

Using the toolchain of the application, the vulnerabilities that can be exploited by fault injection are found in the smartcard source code. In addition, the application is used to evaluate the compliance of the automated code review on our test-suite. The test-suite contains the most common smartcard fault injection vulnerabilities [39] and it integrates the Riscure development guidelines under the format of programming patterns [39]. We used a test-suite based on Riscure patterns because no benchmark exist for the smartcard source code [8]. The test-suite is composed of 12 smartcard programs that are based on defensive programming patterns against fault injection attacks. We were able to identify 15 vulnerability types in our test-suite. The evaluation from *Chapter 4* shows that our application successfully identified the most common smartcard vulnerability types. The success rate of the identified vulnerabilities from the test programs varies between 30% and 100%. Nevertheless, for the unidentified vulnerabilities we provided solutions at an algorithmic level.

The evaluation validates our application, which satisfies the scope defined in *Section 1.1.2.* and the objectives defined in *Section 1.1.3.*. Using the developed application, we

were able to automate the code review process for finding smartcard vulnerabilities. The test-suite based on the Riscure patterns is relevant to evaluate our application since we assume that it covers the most common smartcard vulnerabilities [39]. Our application successfully discovers the most common smartcard vulnerability types (as seen in *Section 4.2.*). Additionally, we provide solutions for the cases of unidentified vulnerabilities. We present for every type of fault injection vulnerability an overview, a code snapshot and a detailed description. As a result of the evaluation, we believe that the application will be a significant factor in evaluating the smartcard software.

5.2 Future work

The future works consists mainly in implementing the taint analysis based on symbolic execution. The concept was introduced in *Section 2.2.1.3.* and in *Section 3.3.*. We can see in *Section 4.2.12.* and in *Section 4.2.5.* why the symbolic execution is needed as a future work. On the other hand, other solutions given as future work are described in *Section 4.2.7.* and in *Section 4.2.15.*

The *Annotation Pass* implements a taint analysis algorithm in order to improve the input with sensitive variables. Our algorithm is limited as we can see in *Section 4.2.* We did not improve the algorithm since we assume that we cannot covers all cases and this problem can be solved by taint analysis based on symbolic execution. The symbolic execution can be implemented in *KLEE*, which is a LLVM tool and which is introduced in *Section 2.2.1.3.* The developed application for the thesis project attacks the confidentiality of a smartcard data. The taint analysis can leak sensitive information used for finding variables that can be utilized in sensitive conditional branches, in spite of not being initially considered sensitive. The example from *Section 4.2.13.* shows the necessity for detecting all the taint leaks.

The tool built on top of KLEE [12] can be used to perform taint analysis based on symbolic execution. The taint analysis can be used as an input for the *Annotation Pass*, the first tool of our application. As a future work, the taint analysis based on symbolic execution is going to replace the taint analysis code currently contained in the *Annotation Pass*. As explained in *Section 3.3.* and in *Chapter 4*, the current taint analysis implementation is limited to one level of *taintness* and the functionality is provided only in the most common case of tainted variables from a conditional branch dependent on a sensitive variable. The taint analysis that we want to use as future work is an open-source KLEE patch [12], which is applied before building KLEE over LLVM. The taint analysis is made by using the KLEE symbolic execution engine, which is introduced in *Section 2.2.1.3.* The taint analysis track flows of data. The tool considers also indirect flows from the control flow (these concepts are described below). The scope is to discover the taint propagation that involves the already defined sensitive variables.

The KLEE patch [12] used together with the developed application defines the LLVM semantics for the **direct** and the **indirect** flows that arise from branch operations. The scope of our thesis work is to use a more applied taint analysis, to detect variables used in sensitive conditions inside the target program. We derive some examples from the examples given in literature [12], in order to show how the taint propagation is

observed in the LLVM IR. The tainted data is assumed not to be initially in memory, but introduced in the executing code by external sources.

The **direct flow** [12] is shown in the example from below. The taint propagation from a sensitive variable to other variables is done through an assignment. In the example, file *F1* (which is used as input file) is considered *1* in the LLVM IR, while *F2* (being used as output file) is considered *2*.

The C code:

```
int a,b;
fread(F1,&a,1);
b=a;
fwrite(F2,&b,1);

The corresponding LLVM IR (in the format of LLVM assembly):

%a1 = alloca i32
%b1 = alloca i32
call i32 @fread (1, i32 * a1, 1)
%a = load i32 * a1
store i32 %a, i32 * b1
call i32 @fwrite 0, (2, i32*%b1,1)
```

In the C code, we can see how the data that is read is leaked when written to the output. The variable *a* is considered sensitive, while *b* is initially not. The tainting is done when *a* is assigned to *b*. In the correspondent LLVM IR code (seen under LLVM assembly mode), we see that the C variables *a* and *b* correspond to different memory locations. The *a1* and *b1* are pointers that reference both variables. The data is leaked at the 5th and the 6th lines, when the *a1* value is loaded to *a* and then stored into the memory location *b1*. The taint propagation is done using the memory locations and the registers.

In the case of **indirect flows** [12], a variables holds the value of a sensitive variable even is there is no direct flow between the two variables. The variable *a* is considered sensitive, while *b* is initially not. Therefore, *a* is not directly tainting *b*. This indirect taint propagation is derived from the control flow. The original sensitive variable *a* influence the *b* variable through the *switch* conditional block. In the corresponding LLVM IR code, the mapping in done by he LLVM *switch* primitive and the tainting is done via *jumps*. The way of identifying the taint propagation is done using symbolic execution, the control flow being tracked during execution.

The C code:

```
int a,b;
fread(F1,&a,1);
switch (a) {
    case 0 :
        b=0;
    case 1 :
```

```
        b=1;
    case 2 :
        b=2;
}
fwrite(F2,b,1);

The corresponding LLVM IR (in the format of LLVM assembly):

%a1 = alloca i32
%b1 = alloca i32
call i32 @fread(1, i8 * %a1,1)
%a = load i32 * %a1, align 1
switch i32 %a, label %bb3 [
    i32 0, label %bb0
    i32 1, label %bb1
    i32 2, label %bb2 ]
bb0:
    store i32 0, i32 * %b1
    br label %bb3
bb1:
    store i32 1, i32 * %b1
    br label %bb3
bb2:
    store i32 2, i32 * %b1
    br label %bb3
bb3 :
    call i32 @fwrite(2, i32 * %b1, 1)
```

The pointer arithmetic and memory is another way of taint propagation [12]. The second *for* loop from the example from below is used to count the zeros in the array.

```
int array[100];
for (i=0; i<100; i++)
    array[i]=0;
int a,b;
fread(F1,&a,1);
array[a]=1;
for(b=0; array[b]==0; b++);
fwrite(F2,b,1);
```

In the code example, the taint propagation from b to a is done in a particular way: an array filled with zeros stores *1* into the position that is given by a [12]. The output is incremented with every traverse of the array elements. Therefore, b will keep the value of a. If a is sensitive, then the *sensitiveness* is propagated to b. The tainting propagation is based on the dependency between a and the array, on which b is dependent. Basically, the array stands as an intermediate link for taint propagation.

The taint analysis tool uses *symbolic execution* for analyzing all the execution paths of the target program, so we have to be take into consideration the semantics. Semantic rules formally describe how an LLVM machine executes [12]. These rules are needed by the taint analysis since they are used to characterize the evolution of the system from one state to another state depending on the current instruction. The semantics involves the following statements:

- The input values are considered symbolic variables
- The conditional branches are dependent on the assignments to the symbolic variables

The taint analysis is done by checking the taint propagation among variables, starting from the initial sensitive variables. The methodology used in literature [12] involves an execution trace under analysis, that complies to a set of semantic rules from the initial context. The output comes from the application of the rules in the taint analysis. The open-source KLEE patch [12] is necessary since sensitive information flows from the target program can be exploited. The taint analysis patch was mainly designed for code that uses cryptography, like the smartcard software. The KLEE patch [12] is using dynamic taint analysis that required the target program to run, while out current implemented taint algorithm is static.

The reason why the symbolic execution is appropriate for taint analysis is that the target program is dynamically explored through all its branches. But this context can lead to a large input spectrum. In order to avoid this situation, the symbolic execution uses symbolic variables as inputs. The symbolic variables are initially uninstantiated, but then are constrained during execution time. Symbolic functions are also introduced, in order to deal with the fact that the search space may become very large when trying to explore all the branches. The symbolic functions are aimed to replace the concrete functions that can be very expensive in the cost of exploration (e.g. an AES encryption function). These functions have their properties described using rules. There are also drawbacks of using the KLEE patch [12]. The tool does not function based on receiving as input the source code target program and performing analysis. A special KLEE API is defined in the patch which can be used to control the variables.

Please note that using the KLEE patch [12], we can implement the taint analysis **as defined** (our definition) in the *Chapter 3* and *Chapter 4*. However, the term *taint* used for the patch [12] is **different** from our definition. Still we can implement the *taint analysis* with the same scope. The term *taint* used for the patch [12] is explained in the following paragraphs. For the patch [12], the concept of *taint value* is defined. The taint values are used to describe the *taint levels*, which characterize variables. The taintness as defined in *Chapter 3* (**our definition**) will propagate (based on inheritance) through the executing program and will mark the encountered variables as L (low) or H (high) if tainted. In our particular case defined in *Section 1.1.2.*, the variables which are marked with H tainted from sensitive variables will be considered sensitive as well. In our case, the taint analysis discovers which variables are dependent on other variables. In the case of the KLEE patch [12], *taint* means simply a mark. A byte of memory (or llvm register) may be marked (tainted) or not marked (not tainted). These marks(taints) are

propagated through the execution of the instructions of the LLVM program. The user can use this marks (taints) to detect if a chunk of memory is influenced by any of the previously tainted variables.

In order to control the taint value of variables in the target program, KLEE methods like *klee_taint*, *klee_get_taint* or *klee_assert* should be use and the following arguments should be set:

```
(taint,buffer,size)
```

representing the taint level, target variable and the size.

Another drawback of the tool is that the API is built only for C target programs. For other programming languages, the currently implemented taint analysis from the *Annotation Pass* should be used. A solution is to develop a similar tool as the KLEE patch [12], based on symbolic execution.

A requirement of the KLEE patch [12] is to run the program under the LLVM tool *klee*, which gives the homonym command. The command is the common KLEE workflow. If there are no symbolic values marked, KLEE may be seen as a plain llvm bitcode interpreter. The KLEE API should be used directly from target C code, setting and asking for taints and reacting to that. The KLEE patch [12] will not trigger any fork, so by default it will execute only one trace. In order to execute more traces, the KLEE API usage should be mixed with symbolic values. This requires the usage of the KLEE API method *klee_make_symbolic*. Based on the new perspective of *taintness* as defined for the KLEE patch [12], we can detect if some variable depend on other variable. For this purpose, inside the C source code, a call to *klee_set_taint* and a call to *assert/klee_get_taint* methods should be added in the correct place to check for the taint. The user should also define which variables or bytes from the target program will be symbolic and which variables or bytes will be tainted. In order to check which values are tainted, we need to add explicit checks inside the target program with *klee_get_taint* and *klee_assert*. The *klee_get_taint* method expects a pointer to memory and a size and returns the union of taint values of all the bytes in the chunk of the selected memory. The taint propagation is transparent to the user. However, the user can set and query taint values using the special functions *taint_set*/*taint_get*, but the propagations are done automatically. The problem is that the user needs to access the chunk of memory where the union of taint variables is. The taints are saved in an internal KLEE structure *Cell*. The user cannot access that directly from the target program only by means of *taint_set*/*taint_get*. The user needs to set the corresponding Cell taint value to a taint argument that can be accessed. Initially, every variable is considered not to be tainted. The programmer sets the taints with *klee_set_taint* and the internal taint value of the specific chunk of memory gets the corresponding taint value. Then, the target program is run, copying the taint values from one variable to other and from one byte of memory to another (propagating the taintness). Then, the programmer can check if a chunk of memory is tainted (or which taint it has, L or H) with *klee_get_taint(buffer,size)*. In the case of local variables and arguments there are internal Cell instances associating the taint values with the actual variable value. The taint values are modified accordingly when the program progress. A trivial example is:

```
int b;
int a;
klee_set_taint(1, &a, sizeof(a));
//from now on, a is tainted with taint "1"
b = a + 1
// now b is also tainted because a was tainted
assert ( klee_get_taint(&b,sizeof(b)) == 1)
```

The user is *tainting* the memory that holds the variable *a* with *klee_set_taint(1, &a, sizeof(a))*. Then, the user set the memory holding *a* as symbolic with *klee_make_symbolic(&a, sizeof(a), ïar)*, so KLEE tests every possible trace depending on *a*. After this operation, the programmer can put an assert checking the taint of the memory holding the variable *b*, with the instruction *klee_assert(klee_get_taint(&b, sizeof(b)) == 1)*. This assert shows if *b* is tainted with taint *1*. It fails to emit an exception if the variable is not tainted. If KLEE finds any trace in which *b* is not tainted with the taint *1*, it will output an error showing which values of variable *a* reached the error condition.

Using the taint analysis implemented by the KLEE patch [12], we can see which variables are influenced by other values during the symbolic execution. The user can use the marks (taints) from the KLEE patch [12] to detect if a chunk of memory is influenced by any of the previously tainted variables. Based on the output, the *Annotation Pass*

can mark the found variables as sensitive, improving the input for our application.

AFSCM	non-profit association promoting the technical
	development of contactless mobile services [14]
annotation	in the context of source code
	annotations, it is a feature of
	debugging tools; an annotation is
	metadata attached to data [29]
API	software interface, specified in detail, that
	provides access to specific functions of a
	program [30]
attack	exploitation of a vulnerability by a threat
	agent [37]
attacker	person (or program) who attempts to perform
	a malicious action against a system [37]
authentication	verification process that the identity
	claimed by a subject is valid [37]
authorization	process that ensures that the requested
	activity or object access is possible with
	respect to the rights and privileges assigned
	to the authenticated identity [37]
availability	ability to ensure users to have timely and
	reliable access to their information assets [40]
basic block	part of the source code that has
	one entry point (no instruction from the
	basic block except the first one is
	the destination of a jump instruction
	anywhere in the program) and one exit
	point (only the last instruction of the
	basic block can cause the program to
	begin executing code in a different
	basic block) [3]
benchmark	act of running a computer program, a set of
	programs in order to assess the relative
	performance of an object, normally by
	running a number of standard tests and
	trials against it [13]
bitcode	in the context of LLVM, it represents the
	intermediate code produced from the source code
	by LLVM front-end compilers [35]
bug	mismatch between implementation and
	specification [28]
call graph	graph whose vertices are the functions
	of the program and the edges are
	given by the relations between functions;
	it represents a directed graph indicating
	the connections between subroutines
	of a program [32]
certification	it refers to the confirmation of certain
	characteristics of an object, person or
	organization [18]

Table 5.1: Glossary

ciphertext	message that has been encrypted for transmission [37]
Clang	LLVM C/C++/Objective-C front-end compiler [33]
code review	process for the examination of source code [22]; in the context of smartcards, its applicability is to ind vulnerabilities in the code; the results can be used for code or code fixing and improvement
conditional branch	an instruction that directs the computer to another part of the program based on the results of a compare; in the context of smartcards, a sensitive conditional branch is the decisional part of a security check
confidentiality	ability to prevent of unauthorized use or disclosure of information [40]
control-flow graph	graph whose vertices are represented by basic blocks of the program and the edges are jumps in the control flow [3]
countermeasure	action taken to patch a system vulnerability against an attack [37]
denial of service	attack that prevents a system from processing or responding to requests for resources [37]
dependency	in the context of smartcard code review, it is a situation in which a program statement refers to the data of a preceding statement [17]
depth-first search	algorithm for traversing or searching tree or graph data structures [16]
encrypt	process used for converting a message into ciphertext [37]
encryption	science of hiding the meaning or intent of communication data from recipients not meant to receive it [37]
exploit	instance of attack on a system as a result of the system's vulnerabilities [27]
glitch	very short voltage dropout or voltage spike [30]; short-lived fault in a system inserted by the attacker/security analyst using fault injection
GlobalPlatform	association founded by various smartcard companies to standardize technologies for multiapplication smartcards [30]

Table 5.2: Glossary

guideline	recommendation hinting the actions to do or
	avoid during software development. Guidelines
	are at an intermediary level, being more concrete
	than principles, but not more than rules [27]
integrity	ability to ensure if the information
	is complete and has not been modiied by
	unauthorized recipients [40]
intrinsic function	function that substitutes a sequence of
	automatically generated instructions
	by the compiler [35]
Java Card	multiapplication smartcard that embeds the
	Java Card operating system [30]
KLEE	LLVM tool that implements a symbolic
	virtual machine to support
	symbolic execution [34]
LLVM	is a compiler framework written in C++,
	used for compiler construction and
	code review [35]
logical attacks	malicious actions that exploits
	vulnerabilities in software [39]
metadata	description of the data in a
	source, distinct from the actual
	data [26]
methodology	the systematic, theoretical analysis of the
	methods applied to a field of study
	or the theoretical analysis of methods and
	principles associated with a branch of
	knowledge [19]
pass	in the context of LLVM, it is a
	tool used to analyze or transform
	a source code [35]
pattern	in the context of Riscure, generalization derived
	from the Riscure sets of security principles, guides,
	recommendations for the smartcard software
	development engineers [39]
penetration testing	live test of the effectiveness of security
	defenses through mimicking the actions
	of real-life attackers [20]
physical attacks	malicious actions that analyze
	or modify the hardware [39]
plaintext	message that has not been encrypted
	the system's vulnerabilities [37]
principle	abstract statement of general security
	knowledge that tends to be generic in nature [25]
Riscure	company specializing in the security analysis
	embedded devices, including smartcards
RAM	volatile memory that is used as working memory
	in smartcards [30]
ROM	non-volatile memory that is used to store
	programs and data that cannot be altered [30]

Table 5.3: Glossary

rule	more concrete version of a guideline. The
	description of the rules is done at syntax level,
	while the guidelines are described at
	semantic level [27]
security check	structures composed of sensitive
	conditions and return instructions; in
	the context of smartcards, the security
	checks aim to protect the sensitive data
	on the smartcard
security analyst	part of the security testing
	personnel which plays the role of
	an attacker
sensitive data	it refers to data whose unauthorized
	disclosure may have serious adverse effects
	on the private data from the system
side channel attacks	malicious actions that analyze
	or modify the device behavior by
	using physical phenomena [39]
SIM	GSM-specific smartcard that is used to secure
	the authenticity of the mobile system
	with respect to the network [30]
smartcard	card containing embedded integrated
	circuits [30]
software security	the engineering of software that is
	aimed to ensure the correct functionality
	of a system under malicious attacks [25]
specification	it refers to the explicit set of
	requirements to be satisfied by the system;
	it provides a complete description of the
	behavior of a system to be developed [30]
standard	document produced by consensus and adopted
	by organizations, that defines rules or
	guidelines for activities or activities results
	in order to achieve optimum regulation
	in a given context [30]
symbolic execution	means of analyzing a program
	to emulate the execution of the
	program [34]
taint analysis	data flow analysis performed
	at each step of execution of a
	single run; used to determine if values
	are derived from user input
target program	the program to be evaluated
terminal	device that provides power to the smartcard
	and enables it to exchange data [30]
test-suite	in the context of smartcards, it is a
	collection of test cases that are intended
	to be used to test the smartcard program in
	order to show if the expected behavior
	is preserved [24]

Table 5.4: Glossary

threat	potential occurrence that may cause an
	unwanted behavior of the system [37]
VMKit	LLVM Java/.NET front-end
	compiler [36]
vulnerability	result of a software defect that can be
	exploited by an attacker in order to
	undertake malicious actions [27]

Table 5.5: Glossary

AFSCM	Association Franaise du Sans Contact Mobile
API	Application Programming Interface
APSC	Attack Potential to Smartcards
CC	Common Criteria
CEN	European Committee for Standardization
CFG	Control Flow Graph
CG	Call Graph
CPU	Central Processing Unit
DFA	Differential Fault Analysis
DFS	Depth-First Search
EEPROM	Electrically Erasable Programmable Read Only Memory
EM	Electro-Magnetic
ETSI	European Telecommunications Standards Institute
GSM	Global System for Mobile Communications
HWIFI	Hardware Implemented Fault Injection
IBM	International Business Machines Corporation
IEC	International Electrotechnical Commission
IR	Intermediate Representation
ISCI	International Security Certification Initiative
ISO	International Organization for Standardization
IT	Information Technology
JHAS	JIL Hardware Attacks Subgroup
LLVM	Low Level Virtual Machine
PIN	Personal Identification Number
RAM	Random Access Memory
ROM	Read Only Memory
SIM	Subscriber Identity Module
SWIFI	Software Implemented Fault Injection
VMKit	Virtual Machine Kit

Table 5.6: Abbreviations

Bibliography

[1] ISO/IEC 15408, *Common criteria*, www.commoncriteriaportal.org/, 2013.

[2] Alexander, Bieman, Ghosh, and Ji, *Mutation of java objects*, www.inf.ufpr.br/silvia /topicos/artigos/OO3.pdf.gz, 2002.

[3] F.E. Allen, *Control flow analysis*, dl.acm.org/citation.cfm?id=808479, 1970.

[4] Machine Learning Group at the University of Waikato, *Weka project*, www.cs.waikato.ac.nz/ml/weka/, 2012.

[5] BITS, *Software assurance framework*, http://www.bits.org/publications/security/BITSSoftwareAssurance0112.pdf, 2012.

[6] Boneh and Brumley, *Timing attack on unprotected ssl implementations*, cs.ucsb.edu/ koc/docs/c36.pdf, 2005.

[7] Bradbury, Cordy, and Dingel, *Mutation operators for concurrent java (j2se 5.0)*, http://www.irisa.fr/manifestations/2006/Mutation2006/papers/14_Final_version.pdf, 2006.

[8] Cedric, *Open benchmark for java card technology*, cedric.cnam.fr /fichiers/RC885.pdf, 2010.

[9] Common Criteria Methodology [CEM], *Joint interpretation library application of attack potential to smartcards version 2.1*, http://www.ssi.gouv.fr /site_documents/JIL/JILThe_application_of_attack_potential_to_smartcards_V2-1.pdf, 2006.

[10] National Cryptologic Centre, *Supporting document guidance, smartcard evaluation, version 2.0*, www.commoncriteriaportal.org/files/supdocs/CCDB-2010-03-001.pdf, 2010.

[11] Zhiqun Chen, *Java card technology for smart cards*, http://www.oracle.com /technetwork/java/javacard/javacard-142511.html, 2004.

[12] Ricardo Corin and Felipe Manzano, *Taint analysis of security code in the klee symbolic execution engine*, www.meals-project.eu/sites/default/files/761802642012.

[13] D.Salomon and G.Motta, *Handbook of data compression*, http://books.google.com/books/about/Handbook_of_Data_Compression.html?id=LHCY4VbiFqAC&redir_esc=y, 2010.

[14] Association Franaise du Sans Contact Mobile Specifications, *Afscm. nfc cardlet development guidelines, release 2.2*, www.afscm.org, 2012.

[15] Evans and Larochelle, *Improving security using extensible lightweight static analysis*, http://dx.doi.org/10.1109/52.976940, 2002.

[16] Even and Shimon, *Graph algorithms*, http://books.google.com/books /about/Graph_Algorihms.html?id=adtQAAAAMAAJ&redir_esc=y, 1979.

[17] J.L. Hennessy and D.A. Patterson, *Computer architecture: a quantitative approach*, http://books.google.com/books/about/Computer_Architecture.html?id=gQ-fSqbLfFoC&redir_esc=y, 2011.

[18] IEEE, *Technav. ieee certification*, technav.ieee.org/tag/375/certification, 2013.

[19] S.I. Irny and A.A. Rose, *Designing a strategic information systems planning methodology for malaysian institutes of higher learning (isp- ipta), issues in information system*, iacis.org/iis/2005/Ishak_Alias.pdf, 2005.

[20] ISACA, *Glossary*, http://www.isaca.org/Pages/Glossary.aspx?tid=651&char=P, 2013.

[21] Jones and Capers, *Measuring defect potentials and defect removal efficiency*, http://www.rbcs-us.com/images/documents/Measuring-Defect-Potentials-and-Defect-Removal-Efficiency.pdf, 2010.

[22] A. Kolawa and D.Huizinga, *Automated defect prevention: Best practices in software management*, http://books.google.com/books /about/Automated_Defect_Prevention.html?id=PhnoE90CmdIC&redir_esc=y, 2007.

[23] Ma, Offutt, and Kwo, *Mujava: An automated class mutation system*, www.ist.tugraz.at/teaching/pub/Main/QS/mujava.pdf, 2004.

[24] A. Mathur, *Foundations of software testing*, http://books.google.com/books /about/Foundations_of_Software_Testing.html?id=yU-rTcurys8C&redir_esc=y, 2011.

[25] McGraw, *Automated code review tools for security*, http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4712512, 2008.

[26] McGraw and Hill, *Dictionary of scientific and technical terms*, http://books.google.com/books/about/McGraw_Hill_dictionary_of _scientific_and.html?id=9t83ABfTBvQC&redir_esc=y, 2003.

[27] G. McGraw, *Software security: Building security in. addison wesley professional*, http://books.google.com/books /about/Software_Security.html?id=HCQdypbpZXgC&redir_esc=y, 2006.

[28] A. Mller, *An introduction to analysis and verification of software*, http://cs.au.dk/ amoeller/talks/verification.pdf, 2003.

[29] Oracle, *Docs: Annotations*, http://docs.oracle.com /javase/tutorial/java/annotations/, 2010.

[30] Wolfang Rankl and Wolfang Effing, *Smart card handbook, third edition*, http://books.google.com/books/about/Smart_Card_Handbook.html?id=JBAGF0v5LqMC&redir_esc=y, 2003.

[31] Juliano Rizzo and Thai Duong, *Practical padding oracle attacks*, usenix.org/events/woot10/tech/full_papers/Rizzo.pdf, 2010.

[32] B.G. Ryder, *Constructing the call graph of a program*, http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=1702621&url=http1.pdf

[33] Open source. University of Illinois, *Clang: a c language family frontend for llvm*, clang.llvm.org/, 2013.

[34] ———, *The klee symbolic virtual machine*, klee.llvm.org/, 2013.

[35] ———, *The llvm compiler infrastructure*, http://llvm.org/, 2013.

[36] ———, *Vmkit: a substrate for virtual machines*, vmkit.llvm.org/, 2013.

[37] E. Tittel, J.M. Stewart, and M. Chapple, *Cissp: Certified information systems security professional*, http://books.google.com/books/about/CISSP_Certified_Information_Systems_Secu.html?id=r7bwQG33aTUC&redir_esc=y, 2004.

[38] Witteman, *Advances in smartcard security*, www.riscure.com/archive/ISB0707MW.pdf, 2002.

[39] ———, *Secure application programming in the presence of side channel attacks*, www.riscure.com/benzine/documents/Paper_Side_Channel_Patterns.pdf, 2012.

[40] J. Wylder, *Strategic information security*, http://books.google.com/books/about/Strategic_Information_Security.html?id=gPWoe-8MGZkC&redir_esc=y, 2003.