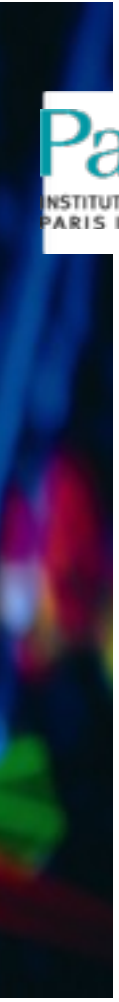


2017 ENST 0038

EDITE ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

Télécom ParisTech

Spécialité “ Informatique et Réseaux ”

présentée et soutenue publiquement par

Zeineb Zhioua

le 8 Septembre 2017

Specification and Automatic Verification of Security Guidelines for Program Certification

Directeur de thèse : Professeur **Yves Roudier**

Jury

M. Denis Caromel, Professeur, UNS	Président
M. Thomas JENSEN, Docteur, IRISA	Rapporteur
M. Brahim HAMID, Maître de Conférences-HDR, Université Toulouse Jean Jaurès	Rapporteur
M. Renaud PACALET, Directeur d'Etudes Telecom ParisTech	Examinateur
Mme Tamara REZK, Chercheur INRIA	Examinateur
Mme. Rabea AMEUR-BOULIFA, Enseignant-chercheur Telecom ParisTech	Invitée
M. Michele Bezzi, Directeur de Recherche, SAP	Invité

T H È S E

Télécom ParisTech
école de l’Institut Mines Télécom – membre de ParisTech
46, rue Barrault – 75634 Paris Cedex 13 – Tél. + 33 (0)1 45 81 77 77 – www.telecom-paristech.fr

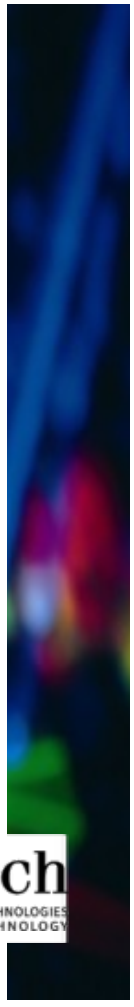
Specification and Automatic Verification of Security Guidelines for Program Certification

Abstract :

Secure software can be obtained out of two distinct processes: security by design, and security by certification. The former approach has been quite extensively formalized as it builds upon models, which are verified to ensure security properties are attained and from which software is then derived manually or automatically. In contrast, the latter approach has always been quite informal in both specifying security best practices and verifying that the code produced conforms to them.

In this Thesis work, we focus on the latter approach and describe how security guidelines might be captured by security experts and verified formally by developers. Our technique relies on abstracting actions in a program based on modularity, and on combining model checking together with information flow analysis. Our goal is to formalize the existing body of knowledge in security best practices using formulas in the MCL language and to conduct formal verifications of the conformance of programs with such security guidelines. We also discuss our first results in creating a methodology for the formalization of security guidelines.

Keywords : Security Guidelines, Security Best Practices, Program Dependence Graph, Program Certification, Information Flow Analysis, Model Checking, Labelled Transition Systems



TELECOM ParisTech

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

Contents

List of Figures

List of Tables

Listings

Acknowledgments

I want to take the opportunity to thank all the people who have been by my side during this PhD journey.

First of all, I want to express my deep and huge gratitude to my supervisor, Professor Yves Roudier for making the PhD work a success, for his continuous support, enriching discussions and his great mindset. Working with Professor Yves Roudier was a very enriching experience, and I have learned a lot from him.

Words cannot express how grateful I am towards Dr Rabea Ameur-Boulifa for every single thing that she has done for me. I want to thank her for her open-mindedness, devotion, and for believing in me.

I want also to thank Stuart Short, my supervisor at SAP for his great help and support. Special thanks to my SAP and EURECOM colleagues and friends for making this PhD journey very enriching and fruitful.

I owe my deepest gratitude to my parents, Mr Abdelkader and Mrs Rafia, my brothers Mhamed and AmenAllah for making me who I am today, to thank them for their continuous support, unwavering encouragement, devotion and love.

I want to thank Aymen Mouelhi, my dear husband for believing in me, for his unlimited sacrifices and continuous support, and for pushing me always to be the best.

Special thanks to my little baby Mohamed for making my life such a beautiful life, for allowing me to discover new feelings I have never felt before he came into my life.

Big thank you to Asma Nani, my best friend who has always been by my side since high school.

Publications

Journal

1. **Zeineb Zhioua**, Yves Roudier and Stuart Short: **Towards the Verification and Validation of Software Security Properties Using Static Code Analysis**. International Journal of Computer Science: Theory and Application ISSN: 2336-0984

Conferences

1. **Zeineb Zhioua**, Yves Roudier, Rabea Ameur-Boulifa, Takouwa. Kechiche and Stuart Short: **Tracking Dependent Information Flows**, ICISSP 2017, 3rd International Conference on Information Systems Security and Privacy

2. **Zeineb Zhioua**, Yves Roudier and Rabea Ameur-Boulifa: **Formal Specification of Security Guidelines for Program Certification**. TASE 2017, Eleventh International Symposium on Theoretical Aspects of Software Engineering

Workshops

1. **Zeineb Zhioua**, Stuart Short and Yves Roudier: **Static code analysis for software security verification: Problems and approaches**. IEEE 38th Annual Computer Software and Applications Conference, COMPSAC Workshops 2014, Vasteras, Sweden, July 21-25, 2014. (2014)

2. **Zeineb Zhioua**, Yves Roudier, Stuart Short and Rabea Ameur-Boulifa: **Security Guidelines: Requirements Engineering for Verifying Code Quality**. Requirement Engineering 2016 Conference- Beijing, China

3. **Zeineb Zhioua**, Yves Roudier and Rabea Ameur-Boulifa : **Formal Specification and Verification of Security Guidelines** The first International Workshop on Frontiers in Dependable Computing (FDC 2017)

PhD Consortium

1. Presentation of the PhD work in the ESORICS'15 PhD Consortium

Chapter 1

Introduction

Contents

1.1 Introduction

Building secure software is much more difficult than finding vulnerabilities and weaknesses in a program. It requires establishing the mechanisms to protect the software and avoid flaws that can be the source of serious vulnerabilities. Secure software can be obtained out of two distinct processes: security by design, and security by certification. The former approach has been quite extensively formalized as it builds upon models, which are verified to ensure security properties are attained and from which software is then derived manually or automatically. In contrast, the latter approach has always been quite informal in both specifying security best practices and verifying that the code produced conforms to them. The former technique, also called model-driven engineering for security, has been extensively developed in

the academic community, and follows a specify, model, verify, and then implement approach. It is well suited to classical software development featuring for instance an organized V-cycle, and security objectives generally correspond to security properties expected from software components or communication protocols (see for instance [155]). In contrast, the latter technique follows a build, then detect flaws approach, based on the description of security best practices and programming style and idioms. Apart from the safety-critical domain, this approach has received a much wider acceptance in the industry because security is often addressed from a developer point of view, in a programmatic style, rather than from a security architect perspective, as through a comprehensive model. A number of security guidelines is defined forming a program-oriented security policy that the developer has to adhere to. This model is also adapted to very different styles of software engineering including agile methods. Unfortunately, this technique lacks automation and formality in the way best practices are specified and verified.

In our work, we focus on the latter approach and describe how security guidelines might be captured by security experts and verified formally by developers. Our technique relies on abstracting actions in a program based on modularity, and on combining model checking together with control and information flow analysis. Our goal is to formalize the existing body of knowledge in security best practices using formulas in the MCL language and to conduct formal verification of the conformance of programs with such security guidelines.

1.2 Context and Motivation

The last years have witnessed the emergence of Cloud platforms, offering cost-effective, scalable, highly available and shared services for enterprises and individual end-users. This led to the large-scale development of Service Marketplaces, offering new opportunities to service providers to distribute their applications in a centralized way, and reach an important number of users consuming the services from heterogeneous target platforms. These platforms have raised new security concerns due to their distributed nature. The marketplace operator, the service provider, as well as the end-user all have specific security requirements with regards to applications deployed in this setting. The distributed and hybrid nature of software marketplaces introduced security concerns that different research projects have tried to tackle. Amongst them, the European Funded Project OPTET (OPerational Trustworthiness Enabling Technologies) that constitutes the main context of my PhD work. In the OPTET project, trustworthiness was addressed from facets other than security, and considers in addition objective elements that are the service security features, that then can be evaluated using attributes and metrics. In order to achieve this goal, systems should first provide their security features; those elements can come from certification processes and third-party security assessments or from the disclosure of some internal details of a service by its provider. The service features are then evaluated against the user requirements that are expressed in terms of attributes. Among other contributions, the OPTET project designed and developed

a Trustworthy Marketplace, to expose secure services, together with their machine-readable descriptions of functionalities and security features.

1.3 Security Requirements and Security Guidelines in the Practice of Software Development LifeCycle

Big companies usually follow a Secure Software Development LifeCycle (SSDLC) in order to help reduce the software exposure to vulnerabilities. Plenty of SSDLC variants exist and are implemented by organizations with slight differences, still with the objective of improving the security of software.

In practice, companies create review processes and conduct audit sessions [48] comprising a wide range of experiments to verify the compliance with general and security related requirements. However, manual reviews can be time-consuming and costly to the companies in terms of resources, and they can fall short in detecting requirements violations.

Since security is not only an IT problem but also a business problem, companies increasingly give due consideration to security through the adoption of security awareness trainings for developers as well as for managers. Technical security trainings help equipping developers with an awareness regarding the risks as well as the good programming practices that help reduce risks and associated costs and also mitigate the serious flaws that software might be subject to.

Secure programming approaches such as the use of static code analysis tools can also be applied in the different phases of the software engineering process. Applying static code analysis or manual code inspection allows to detect vulnerabilities early in the coding phase. The latter can be error-prone and much more time-consuming than automatic code review, especially when the entire source code has to be analyzed, or when the code to inspect is large [25].

In the SSDLC, huge efforts are invested rather late in the software engineering process, such as applying penetration testing only when the software is close to be delivered to the customer. However, fixing bugs in the late stages of the development lifecyle is much more costly to the company, and it is almost as expensive as fixing bugs when the software is already delivered to the customers, as reported by the *Systems Sciences Institute* at IBM :*"the cost to fix an error found after product release was four to five times as much as one uncovered during design, and up to 100 times more than one identified in the maintenance phase"* (Figure 1.2).

In a study published in 2002, the National Institute of Standard Technology (NIST) [124] reported that the cost of fixing one bug found in the production stage of software is 15 hours compared to five hours of effort if the same bug were found in the coding stage.

We are not claiming that effort should be put only in the development phase, but we pinpoint that this invested effort can be rendered much smaller in the verification and validation phases or the testing phases.

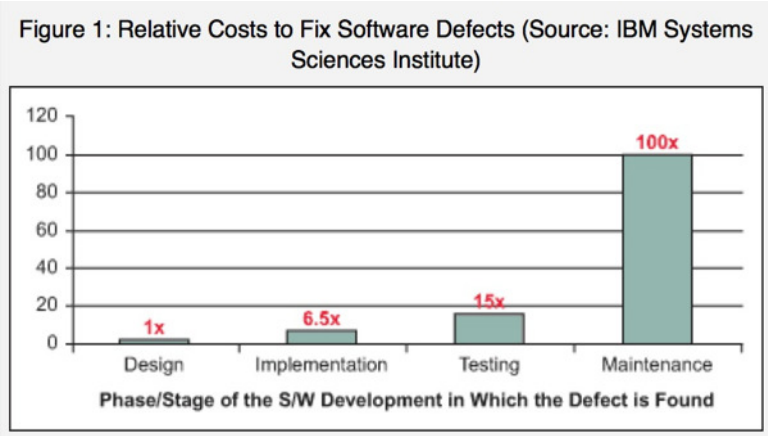


Figure 1.1: Relative cost to fix defects

Table 1-5. Relative Costs to Repair Defects when Found at Different Stages of the Life-Cycle

Life Cycle Stage	Baziuk (1995) Study Costs to Repair when Found	Boehm (1976) Study Costs to Repair when Found[a]
Requirements	1X[b]	0.2Y
Design		0.5Y
Coding		1.2Y
Unit Testing		
Integration Testing		
System Testing	90X	5Y
Installation Testing	90X-440X	15Y
Acceptance Testing	440X	
Operation and Maintenance	470X-880X[c]	

[a]Assuming cost of repair during requirements is approximately equivalent to cost of repair during analysis in the Boehm (1976) study.

[b]Assuming cost to repair during requirements is approximately equivalent to cost of an HW line card return in Baziuk (1995) study.

[c]Possibly as high as 2,900X if an engineering change order is required.

Figure 1.2: Cost to fix defects according to NIST

Hence, starting the security activities early on in the SSDLC and helping the developers throughout the security verification is important. For instance, checking the compliance of software under development with security guidelines as early as possible would represent a large progress, especially in agile software development methodologies.

Software engineering processes have grown exponentially in past years in terms of code bases complexity and development team size working on the same project. As a consequence, running tests would not be sufficient to detect flaws before the software release, and is rather costly if it requires the end of the development process to be carried out. Hence, we can derive multiple factors that justify the adoption of static code analysis and its deployment on the programmers'side. First, running static code analysis on pieces of software does not require the end of the development process, and can be run on a non necessarily complete product. Another advantage of applying static code analysis is that it does not require to run or compile the code, and can be run by the developer when development activities are ongoing. Moreover, locating the source of the detected security flaw can be much faster and more precise.

Using existing static code analysis tools requires a very large expertise by developers to interpret and fix the discovered security issues. However, developers should not be security experts, and what is really needed is to provide guidance and help to them when using static code analysis tools. There is a strong need to adapt the static code analysis tool in a way that is easy to use by and for developers, for instance through the continuous integration in the development environments (IDEs).

1.4 Use Case Scenario

In order to get a clear understanding of the problems we address in this thesis, we propose to go through a real-world scenario, where Alex, a software developer in an IT company, wants to check if his software is compliant with the security guidelines of both the project and the company. **Brian** is a security expert working in the same company as **Alex**. The security guidelines are maintained in a repository, and are in an informal representation format: textual descriptions and explanations in natural language. This induces difficulties for the developers, including Alex, to interpret and to correctly apply the guidelines without the help of security experts, such as Brian, who should in this case intervene in the development phase, in the verification phase, and possibly in the correction phase.

We aim at limiting the intervention of Brian in the different phases of the development lifecyle, and provide the necessary means to help Alex in building secure software with respect to the project and to the company's guidelines. In order to fulfill this, we suggest that Brian can carry out a formal specification of the security guidelines. This operation consists in extracting the key-concepts from the guidelines textual description, and in writing formulas in a mathematics-based formalism. The built formulas are opaque to Alex, and he does not have to deal with them directly.

During the development phase, Alex is more focused on the functional requirements of his application. We aim through this thesis work to help Alex address the security issues in the development phase, without waiting for the development to be complete to start fixing issues. Alex proceeds as follows: he selects the guidelines that his code should meet, and launches the analysis. The choice of the guidelines is imposed by the project and the organization requirements. For some cases, Alex needs to enter manual annotations on the program, such as the data corresponding to "password", or to "credit card number". The outcome of this analysis indicates whether the guideline is met or violated. In case of a violation of a security guideline, Alex should be presented with a trace explaining what caused the problem in his code.

Alex can then check the source of the violation, and can also refer to the guidelines specifications that can be seen as templates or recommendations to make the needed corrections.

Code 1.1: Sample code for the logging of user credentials

```
BufferedReader reader = new BufferedReader
        (new InputStreamReader(System.in));
   System.out.println("User name : ");
   System.out.println("Password : ");

   // input
   user.setUsername(reader.readLine());
    user.setPassword(reader.readLine());

   // copy
   String xx = user.getPassword();
   // hash
   MessageDigest hash = MessageDigest.getInstance("MD5");
   byte[] bytes = user.getPassword().getBytes("UTF-8");
   byte[] hash_password = hash.digest(bytes);

   user.setPassword(hash_password.toString());

   // log
   logMessage = "user name = " + user.getUsername() +
        ", password = " + user.getPassword() + xx;
   logger.log(Level.INFO, logMessage);
```

1.5 Thesis Contribution: A Framework for the Formal Specification and Verification of Security Guidelines

We propose in the context of this thesis a framework to help to bridge the gap between the informal presentation of security guidelines, and their automatic verification at code level

from the programmer point-of-view. This PhD work resulted in the design of a tool-chain and its almost complete implementation covering the end-to-end verification of compliance to security guidelines on the code level. The security guidelines that we consider relate to good and bad programming practices, instead of the traditional CIA (Confidentiality, Integrity and Availability) properties. We aim at verifying the adherence of the software with those security guidelines, and our goal is not to prove the correctness of the program. We are aware that the adoption of formal methods in industry is not well accepted. This is mainly due to the required investment in terms of training and effort that should be carried out. In addition, the Return On Investment can barely be justified if the software in question is not critical (avionics, medical, etc.). Formal methods can bring significant added-value in finding subtle yet serious security violations that ad-hoc approaches fail to detect.

1.5.1 Formal Specification of Security Guidelines

Considering the direction of our work, we first needed to formalize security guidelines using a formal language. A formal specification is typically a mathematical-based description of guidelines, using mathematical logic. We have first performed a deep analysis on security guidelines that we have gathered from different sources including OWASP [121], the CERT Coding Standard [39], the Juliet Test Case [112] and reports about the process carried out by the Apple AppStore [15] [14]. This analysis led to a classification of the guidelines with respect to the type of the analysis that should be performed in order to verify the guideline (see section 2.3). Upon surveying security guidelines, we noticed that there are ambiguities in interpreting the guidelines and a lack of precision in their description. As a result, we defined how to formalize the security guidelines, which constitutes an important contribution in this thesis work. We have adopted the Model Checking Language (MCL) for its ability to handle not only method calls but also value passing. The main objective behind this formalization is to strip away ambiguities regarding the guidelines informal description, and to capture subtle details such as implicit dependencies that can be a source of covert channels that can be difficult, if not impossible, to capture by traditional analysis means. This formalization prepares the ground for the automatic formal verification.

1.5.2 Security-Augmented Static Program Representation

Concerning the program model to consider as a basis for the program representation, we have chosen the Program Dependence Graph (PDG) (see section 6), that we have augmented with accurate details allowing to capture the implicit dependencies that may occur between program instructions. From the Augmented Program Dependence Graph, we automatically generate the pLTS (parameterized Labelled Transition System), which is a Labelled Transition System augmented with both explicit and implicit data dependence details that we extract and represent on the Augmented Program Dependence Graph.

1.5.3 Formal Verification of Security Guidelines for Program Security Certification

We translate the security Augmented PDG into a parametrized Labelled Transition System (pLTS). We use this pLTS, a more common model for model checking, for the automatic verification of the compliance of the program to the set of selected security guidelines. The automatic formal verification consists in comparing the behavior specified in the security guideline (MCL formula of the guideline) with the behavior permitted by the program model (pLTS). The outcome of the verification phase indicates whether the guideline is met or not, and the violation traces are returned. One major benefit of applying formal methods in this context is their capability of proving the absence of an undesired behavior, contrary to industrial ad-hoc best practices analysis tools that mostly prove the presence of such non permitted behavior. The verification of the program compliance to selected security guidelines results in a "light certificate" that allows to keep track of the respected good programming practices on that program. It is important to note that we do not aim at proving the correctness of the program, but rather to prove its compliance with application security programming best practices and recommendations.

1.5.4 Security Knowledge Base

The Security Knowledge Base (see section 7.5) is a very important component in our toolchain. The Security Knowledge Base contains a repository of formalized security guidelines with the aim of managing the good and bad programming practices. Apart from the security guidelines, the Security Knowledge Base contains a wide range of Java APIs, including security-related methods. Those APIs are decorated along with their signature and description and mapped to the (source, sink) information flow annotations, and their security level that indicates their level of confidentiality/integrity (high=secret/private, low=public). This contribution allowed us to enrich the guidelines dictionary and the labels that can be used to build the MCL formulas upon. It also serves as well to facilitate the automatic annotations and the new dependencies detection on the graph model (Augmented PDG). This repository can be used by both the security expert(s) and the programmers, as it allows to centralize security-related knowledge shared by the different security experts who formalize security guidelines. This can be a useful tool to bring security awareness and education to developers to gain a better understanding of the good programming practices, and to improve the code quality with respect to security.

1.6 Contributions to the OPTET Project

Working as a PhD student within SAP and EURECOM, and contributing to the European Funded Project OPTET offered a rich and fertile environment to direct this PhD work. The OPTET consortium is made up of fifteen participants (THALES, SAP, FORTH, IBM, etc.)

from eight countries. Exchanges, discussions and collaboration with the different OPTET partners brought new perspectives and ideas which could only be of benefit to the flourishing and success of the project. My contributions to the OPTET project ranged from research, to design and prototyping activities. In order to set the scene, I have first worked on the state-of-the art on software marketplaces: cloud and mobile marketplaces. The main objective of this first activity was to understand further the security mechanisms that are established by each of the marketplaces in order to ensure the security of its infrastructure as well as of the end-users. The second objective was to identify where major innovation can be performed and to propose a marketplace model that is able to fill the identified gaps. This state-of-the art on the main stakeholders involved and the different interactions between them is part of the Deliverable D5.1.1 [136]. The state-of-art on the marketplaces covered another aspect, which is the delivery and deployment of software or applications on the end-user's target platform. This constitutes another contribution that I have worked on, and is part of the Deliverable D5.3. The focus was mainly to have a deep understanding on how the different marketplaces ensure security in the delivery and deployment operations. In the Deliverable D5.1.2, my contributions covered the identification of the functional requirements of the marketplace model through the consideration of different personas covering different scenarios. For each of the personas, we have identified the involved components and how those components should communicate in order to achieve specific tasks.

I have also been part of the marketplace model development team, and contributed to different prototyping activities. For instance, I have implemented an algorithm for matching the user's security requirements with the application certificate. In the OPTET marketplace, available applications expose their functional description and their security features in the form of a machine-readable certificate similarly to the certification scheme proposed in the European Projects *Advanced Security Service cERTificate for SOA* (ASSERT4SOA) [13] [12]. The search of applications in the marketplace is then guided by the security requirements that the end-user expresses in the form of attributes (confidentiality, integrity, response time, etc.) and their desired values together with the implemented security mechanism. My contribution consisted in matching those security requirements with the available applications machine-readable certificates. The result is a set of applications ordered according to their compliance score. My contribution to OPTET covered also the preparation of documentation on the implemented APIs together with the Test Plan document. The details on my contributions to OPTET are presented in Appendix A.

1.7 Thesis Outline

The work presented in this thesis first presents a framework for the formal specification and the automatic verification of security guidelines.

- **Chapter 2** presents a survey on the security guidelines gathered from different sources, with the objective of pinpointing their main issues. In this Chapter, we explain how we

classified the security guidelines in different categories with respect to their type. The contributions of this Chapter have been published in [160].

- **Chapter 3** discusses different certification schemes with the objective of pinpointing their shortcomings in terms of security assessment. We stress in this Chapter that current certification schemes cannot be adopted to certify the compliance of the being developed software with security guidelines.

- **Chapter 4** presents an overview of the proposed approach that we discuss in a high level. We discuss the different steps of our approach, and explain each one of them succinctly. The contributions of this Chapter have been published in [158] [156].

- **Chapter 5** presents the methodology that we propose to formally specify the security guidelines from the security expert point-of-view. We share different difficulties that we have faced when formalizing the guidelines, and explain how we succeeded in solving them. The contributions of this Chapter have been accepted in TASE Conference (Paper accepted but not published yet)

- **Chapter 6** presents the program model construction approach that we come up with to represent the program behavior. We explain how we built a Program Dependence Graph that we have augmented with subtle and implicit details. Then, we explain how we generated a Labelled Transition System from this augmented PDG. We discuss in details the graph construction, as well the formal verification that we carried out in order to automatically verify the adherence of the program to the security guidelines.

- **Chapter 7** presents in details the prototyping activities that we have carried out in order to build the full tool chain. We present also the different tools that we integrated, as well as the different modules that we have implemented.

- **Chapter 8** discusses some limitations of our framework, such as state space explosion or scalability. We discuss in this Chapter possible directions that can help to solve those limitations.

- **Chapter 9** discusses possible directions of our work. We share different ideas that we aimed at covering, but due to time constraints, we could not achieve them. We conclude the Thesis report and share different perspectives.

Chapter 2

Security Guidelines for Programs

Contents

2.1 Introduction

Organizations and companies define non-functional security requirements to be applied by software developers, and those requirements are generally abstract and high-level. Security requirements such as confidentiality and integrity are abstract, and their application requires defining explicit guidelines to be followed in order to fulfill the requirements. Security guidelines describe bad as well as good programming practices that can provide guidance and support to the developer in ensuring the quality of his developed software with respect to the security aspect, and hence, to reduce the program exposure to vulnerabilities when delivered and running on the customer platform (on premise or in the cloud). Bad programming practices define the negative code patterns to be avoided, and that can lead to exploitable

vulnerabilities, while good programming practices represent the recommended code patterns to be applied on the code.

The definition of security guidelines stands in another context: software marketplaces. Software Marketplaces use different approaches in order to prevent malicious applications from being advertised in their official stores. For instance, Apple App Store [14] verifies if submitted applications adhere to the Apple App Store review guidelines [14], that basically serve as a guide gathering recommendations to be followed by developers before they submit their applications. Submitted applications generally go through a verification of their compliance with the marketplace security requirements. On the one hand, it is important for a service provider to be aware of domain-specific requirements as failure to be accepted in the marketplace and subsequent application rewrites may be costly, time-consuming, and affect the organization's reputation. On the other hand, the end-user wants to be reassured that their demands are respected in terms of security and privacy. However, this process is neither always transparent nor understandable to the service provider as well as to the end-user. From service providers/developers perspective, the understanding and interpretation of guidelines is not trivial, as there is no formalization that exposes the necessary program instructions for each guideline, or that explains how to apply them correctly in the software.

Another programming practices guide we can consider for instance is the CERT Oracle Coding Standard for Java [39] [102]; for each guideline, the authors provide a detailed textual explanation. For most, there are also provided examples of compliant and non-compliant sample codes in addition to the description.

The Juliet Test Suite [112] is created by the National Security Agency (NSA), and proposes a set of test cases covering multiple programming languages including Java. The provided test cases are arranged into categories with respect to the flow type; control or data, and every test case targets one specific vulnerability or weakness, referring to known entries in the Common Weakness Enumeration dictionary (CWE) [1].

In order to have a clear understanding on the security guidelines, we conducted a deep analysis on the different official sources proposing rules and examples of good/bad programming practices. The surveyed sources include OWASP[121], Oracle [116], CERT, NSA [112], NIST[113] and Apple[14], and will be discussed in details in the next section.

The main objective behind this survey is to identify the main issues with the guidelines presentation to developers, and pinpoint the main obstacles to their automatic verification on the software. The second objective is to come up with a classification of the security guidelines with respect to the type of the verification that should be applied in order to validate their adherence on the software.

Before proceeding to the classification, effort was undertaken to develop the criteria serving as basis to our categorization. One trivial criterion is the control flow, specifying the sequence

[1] https://cwe.mitre.org/index.html

of operations that should be performed for a software to be compliant. Another criterion consists in considering data and how it should be processed. Different guidelines require some data typing rules, specifying the data types that should be used in a given context. The fourth criterion we have considered is the semantic aspect. This criterion means that there is a need to study the meaning of linguistic expressions for the evaluation of the guideline's key words. The interpretation of semantic guidelines is not straightforward, and requires expertise to analyze, apply and verify them.

The main objective behind this classification is to prepare the ground for expressing these guidelines in a formal language and to reason about their satisfiability. Control, data and typing guidelines can be translated automatically into formal language. However, for those involving the semantic aspect, the formalization cannot be carried out automatically and should be led by a security expert who extracts the linguistic meaning and relate it to the guideline context.

2.2 Survey on Security Guidelines Sources

In this section, we present the survey that we have conducted on the different sources of security guidelines in order to fulfill different goals. First, to identify the main difficulties that may arise when interpreting the guidelines'textual description. Second, to pinpoint another issue: guidelines are provided by different sources, and there is no common presentation model that allows a homogeneous interpretation.
For concrete understanding, we consider some real-world guidelines [2] as introduced in different sources such as CERT [39], Apple App Store Developer Guide [14], Juliet Test Cases[112] and OWASP[119]. Each guideline is denoted by a unique code attributed by the issuing organization. The different points are discussed in details in the different sections of this chapter.

2.2.1 OWASP Security Guidelines

The OWASP Foundation [121] introduces different sets of guidelines and rules to be applied in order to protect sensitive information. The Secure Coding Practices guide[121] is a set of good programming practices that are presented in a checklist format arranged into classes, like Database Security, Communication Security, etc. The listed programming practices are general, in a sense that they are abstract and are not tied to a specific programming language. The same source, OWASP [119] provides different sets of rules arranged with respect to the specific entity that they seek to protect (password, encryption key) and to the security mechanisms to be implemented (protocols, data storage, etc.). The guides proposed by OWASP include the **Cryptographic Storage Cheat sheet** that provides a set of guidelines to be applied in order to protect data at rest; for each rule, OWASP provides recommendations

[2] https://www.securecoding.cert.org/

related for instance to the cryptographic protocol to be applied, to the minimum key length to be used, etc.

Table 2.1: OWASP security guidelines: categories and details

Category	Description	Positive patterns	Negative patterns	Total number	Programming Language	Last update
Password Storage Cheat Sheet	X	NA	NA	4		28/08/2016
Key Management Cheat Sheet	X	NA	NA	Number		04/06/2016
.NET Security Cheat Sheet[3]	X	X	X	Number		19/03/2017
.Web Service Security Cheat Sheet [4]	X	NA	NA	Number		04/08/2015
SAML Security Cheat Sheet [5]	X	NA	NA	Number		04/09/2015
PHP Security Cheat Sheet[6]	X	X	X	Number	PHP	10/08/2016

However, the guidelines are presented in an informal style, and their interpretation and implementation require security expertise. Let us consider this rule from the OWASP **Cryptographic Storage Cheat sheet**: "Store unencrypted keys away from the encrypted data" [7] In the description of this guideline, OWASP explains the encountered risks when the unencrypted key is stored in the same location as the encrypted data.

2.2.2 CERT Security Guidelines

CERT[6] is a division of the cybersecurity research and development Software Engineering Institute (SEI) [144]. It aims at improving the security of the cyber environment through providing innovative research and development solutions. The main mission is to reduce the vulnerabilities exposure in a way allowing the mitigation of the detected flaws during development and testing. The CERT Coding Standards have been adopted by corporate companies such as Oracle and Cisco. The CERT division proposes a set of security guidelines arranged first by the programming language to which they apply (Java, C, C++, etc.). Those guidelines are classified into categories with respect to the verification type and to the assets to be protected (Input Validation, Password Security, etc.). For each guideline, the authors provide a detailed textual description explaining the guideline, and the risks that can be encountered when the guideline is not correctly applied. For most, there are also provided examples of compliant and non-compliant sample codes referring to the positive and negative patterns. Furthermore, the authors maintain for each of the guidelines a risk assessment providing the probability of the violation occurrence, as well as its severity and the entailed remediation cost. Giving the fact that every programming language has its specificities, CERT provides a set of security guidelines categories that differ from one programming language to

[7]https://www.owasp.org/index.php/Cryptographic_Storage
_Cheat_Sheet#Rule___Store_unencrypted
_keys_away_from_the_encrypted_data

another, like for example the memory management[8] that is proper to the C language.

Table 2.2: CERT security guidelines for Java: categories and details

Category	Description	Positive patterns	Negative patterns	Total number	Last update
Input Validation and Data Sanitization	X	X	X	18	26/02/2017
Declaration and Initialization	X	X	X	3	26/02/2017
Expression	X	X	X	8	14/03/2017
Numeric Types and Operations	X	X	X	15	19/03/2017
Characters and Strings	X	X	X	5	26/02/2017
Object Orientation	X	X	X	14	26/02/2017
Methods	X	X	X	14	26/02/2017
Exceptional Behavior	X	X	X	10	26/02/2017
Visibility and Atomicity	X	X	X	6	26/02/2017
Locking	X	X	X	12	26/02/2017
Thread APIs	X	X	X	6	26/02/2017
Thread-Safety Miscellaneous	X	X	X	4	26/02/2017
Input Output	X	X	X	17	26/02/2017
Serialization	X	X	X	14	26/02/2017
Platform Security	X	X	X	8	26/02/2017
Runtime Environment	X	X	X	7	26/02/2017
Java Native Interface	X	X	X	5	26/02/2017
Miscellaneous	X	X	X	12	26/02/2017
Android	X	X	X	27	04/07/2015

2.2.3 NIST Security Guidelines

National Institute of Standards and Technology (NIST) provides different sets of rules and standards that serve as guidance to the cost-effective security for information systems. In [113], there are provided cryptographic keys management recommendations. The guide focuses on recommendations on how to properly use cryptographic mechanisms, with particular focus on the encryption keys management, including due consideration to their strength, their secure generation, storage, distribution, use and destruction. This guide is divided into three main parts with respect to the target audience and to the technical level. The first part gathers recommendations on basic key management, with a focus on the technical implementation aspects. Part 2 [9] on the other hand, is focused on simplifying the concepts of establishing cryptographic keys management within an organization. This part contains recommendations for system owners and managers. The main objective of Part 3 [10] is to cover implementation and usage of specific rules. Each rule description includes indications on the recommended algorithms and the key lengths. It includes also recommendations on using in a security-effective manner the existing key management processes.
NIST provides general rules in a sense that they do not reply to a specific need or consider

[8] https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=437
[9] http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-57p2.pdf
[10] http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57Pt3r1.pdf

a specific programming language, such is the case for CERT. NIST rules provide recommendations that, if applied correctly, would guarantee strong security with respect to the applied mechanisms.

2.3 Classification of Security Guidelines

As a first step, we will organize the guidelines into groups with respect to the type of verification they consider. For example, a category of guidelines can be more focused on validating data provided as input. We can also consider another category for secret security, that is data whose integrity violation can result in a loss of confidentiality of sensitive information. Sensitive information security is another category we define, and consists at regrouping the guidelines for the sensitive information, such as user's private information. We have also depicted categories that deal with specific methods invocation and execution environment security. For each guideline, we provide explanation, comments and observations.

2.3.1 Input validation

This category gathers the guidelines that perform the validation of data provided as input. Validation consists in ensuring that input data belong to the expected input domain.

IDS01-J:*Normalize strings before validating them* [36] This guideline recommends that input strings should be normalized before being validated. Normalization is according to this guideline a crucial operation as the same string can have multiple representations depending on the used Unicode. However, neither the normalization nor the validation are easy to interpret and to apply. Does the validation mean sanitization?

IDS03-J:*Do not log unsanitized user input* [30] This guideline describes the recommended behavior when dealing with user input that should be logged. Provided user input should be sanitized before being logged, however, sanitization can be performed through multiple manners, such as the elimination of suspicious characters. In the guideline description, the implementation of the sanitization operation is not specified.

IDS06-J:*Exclude unsanitized user input from format strings* [33]

The guideline recommends not to include untrusted data in a format string, as this may result in information leakage. One ambiguity in the guideline description is the notion of **untrusted data**. From a developer perspective, the operation of "'excluding"' untrusted data is not trivial; is it only about not including it in format string? Hence, this guideline can be formulated as follows: sanitize user input before including it in format string.

IDS07-J:*Sanitize untrusted data passed to the Runtime.exec() method* [38]

This guidelines proposes a way to reduce the program exposure to command and argument injection vulnerabilities, through the sanitization of untrusted data included in a specific Java method *Runtime.exec()* that allows to run an external program. It is pointed out that suspicious arguments passed to this method may expose the program to the command injection attack. Despite the detailed description, applying the guideline is not trivial, due to the unclarity of the untrusted data notion, even though there is an attempt to clarify it and put in the context of *trust boundary*.

IDS08-J:*Sanitize untrusted data included in a regular expression* [37]

In this guideline, it is recommended to sanitize data passed to a regex in order to prevent malicious attacks such as regex injection, information leak or DOS attacks. If unsanitized input is included in a regex, this might modify the original regex that will be changed and will no longer perform the original desired verification. The developer is faced with the ambiguous notion of *untrusted data*. What would determine if the data is trusted or not? Does this assume that all the data coming from external sources (user input, consumption of a web service, etc.) is untrusted?

124683:*CWE 129: Improper Validation of Array Index* [111]

The test case 124683 references a weakness in the CWE (Common Weakness Enumeration database) that points out the risk of not validating the array index[11]. It is recommended to check that the array index is within the correct range of values for the array, but it is not specified how to properly perform this verification.

2.3.2 Method declaration and invocation

This category is focused on guidelines that provide useful hints to ensure the program safety when invoking specific Java methods.

MET03-J:*Methods that perform a security check must be declared private or final* [34] This guideline recommends to declare methods that perform security checks as private or final to make sure they cannot be overridden. However, the notion of *security checks* can be subject to different interpretations; is it authentication? encryption verification? As a reader can notice, the understanding and implementation of this guideline are not intuitive.

MET53-J:*Ensure that the clone() method calls super.clone()* This guideline specifies that for a class that implements the *clone()* method, it should explicitly invoke the *super.clone()* method. Otherwise, the types of the cloned object and the original one can be different.

[11] https://cwe.mitre.org/data/definitions/129.html

MET56-J:*Do not use Object.equals() to compare cryptographic keys* [32] The guideline recommends avoiding the invocation of the method Object.equals() to compare cryptographic keys. The motivation behind this guideline is that *Object.equals()* method is not applicable to composite objects, such as cryptographic keys, and may return false even when the keys have exactly the same value.

EXP02-J:*Do not use the Object.equals() method to compare two arrays* [31]

This guideline specifies the proper method to be invoked for the comparison of the Java Array type.

2.3.3 Secret security

We express through this category the need to ensure the integrity of specific data whose violation can have negative impact on confidentiality of dependent data. For example, the violation of password integrity can violate authentication, same for the violation of encryption keys secrecy, this will result in violating confidentiality of encrypted data, and the encryption operation can no longer be reliable for ensuring confidentiality.

MSC62-J:*Store passwords using a hash function* [40]

This guideline specifies the technique to be used in order to ensure the secrecy of passwords and limit their exposure. The hash operation applied on passwords which are the sensitive information in this guideline, will result in an undecryptable data that can propagate to a public output without having any risk on its secrecy. From this perspective, the guideline somehow indicates a technique for declassifying the security level of sensitive information such as passwords. However, the identification of a specific variable or data as password requires semantic inference rules and a rich knowledge base.

Store unencrypted keys away from the encrypted data [119] This guideline recommends not to store encrypted data together with the encryption key, as this operation can result in a compromise for both the sensitive data and the encryption keys. However, encryption keys can be declared as byte arrays with insignificant names, which makes their identification as secret and sensitive data very difficult.

Correctly applying this guideline would provide a strong protection mechanism against this attack scenario: an attacker can get access to the encryption server or client, and can retrieve the encrypted data with the encryption key. Fetching those two elements allows the deciphering of the encrypted sensitive information. This reminds the well known Heart-Bleed[12] [52] attack that occurred couple of years ago (April 2014), and that allowed to read the memory, steal users credentials directly from the systems protected by the vulnerable

[12]http://heartbleed.com/

version of OpenSSL. This example emphasizes the critical attacks that can be performed if the guideline is not respected.

2.3.4 Sensitive information security

In this category, we collected the guidelines that deal with sensitive data, that include user's private information, passwords, credit card numbers, etc.

MSC03-J:*Never hard code sensitive information* [35]

In this guideline, it is recommended not to expose sensitive information on plain text at the code level. This guideline is not about the behavior of the program, but it is more about the code, meaning class files. The motivation behind it is the risk of exposure of sensitive data if an intruder gets access to the class files or decompile the byte code, then he can easily discover sensitive data.

IDS15-J:*Do not allow sensitive information to leak outside a trust boundary* [29]
This guideline is considered as a stub, in a sense that it is generic, and can be instantiated through different mechanisms, like for instance preventing catching an exception that exposes sensitive data, or forbidding the storage of sensitive data on external storage, etc. The notion of trust boundary can lead to misinterpretation, hence to improper implementation. From a developer perspective, it is tough to identify all the sensitive information in his program, the complexity of this operation increases with the complexity and the size of the program.

5.1.2(i) *Apps cannot use or transmit someone's personal data without first obtaining their permission and providing access to information about how and where the data will be used* [14]

This guideline explains that applications requiring access to user's data should explicitly request the access to private data and should also inform the user about where the data will be processed and to which end. From a developer point of view, the identification of private information cannot be easily carried out. This operation requires an advanced semantic knowledge base covering the main naming conventions of personal information. The developer is faced with another lack of clarity concerning the access to data; the access is not explicitly defined: is it read, write, modify, delete? There is a lack of precision about the purpose of the access, and also about how to inform the user about where and how the data will be processed. On top of that, the operation of granting or revoking a requested permission occurs at install time or even run-time for some specific permissions, hence, verifying whether the application was granted the permission or not cannot be checked statically. The problems described are also faced by the tester or the reviewer of the application.

5.1.2(ii) ***If your app doesn't include significant account-based features, let people use it without a log-in. Apps may not require users to enter personal information to function, except when directly relevant to the core functionality of the app or required by law..*** [14]

The guideline explains that applications requesting access to private information will be rejected. We are faced with the some ambiguities discussed for the previous guideline.

2.3.5 Execution environment security

This category is focused on guidelines that aim at ensuring the security of the execution environment. They are neither control flow nor data flow guidelines.

OBJ10-J:*Do not use public static nonfinal fields* This guideline recommends not to declare nonfinal fields as public static, for the risk of exposure to improper change of their values when there is a concurrent access to their content, especially in the presence of multiple threads, where the concurrent access to nonfinal public static fields can result in an inconsistent modification of the content.

By arranging the security guidelines into categories with respect to the flow aspect; control or data or some combination thereof, we will be able to have concrete understanding on the analysis type that should be performed on the program in order to verify the adherence or not to the given guideline. We have also considered some other categories that are neither control nor data, but more about the data typing or semantic aspect of the guideline.

The classification of the guidelines is presented in Table 2.3.5.

2.4 Discussion

Security guidelines are mainly meant to simplify the developer job in enhancing the software quality from security perspective. However, if we take a closer look at the guidelines presented in the previous section, we will notice that their understanding and implementation are not trivial to the developer. Considering only the events or actions from pure control flow angle is not sufficient for guidelines that are highly dependent on the information propagation through a program, such is the case for the guidelines IDS03-J - IDS15-J. Hence, considering information flow and integrating them into the formal specification of security guidelines becomes imperative.

When interpreting the guidelines, the developer is faced with one other problem related to the semantic interpretation of keywords such as the notion of sensitive data, trusted/untrusted data, trust boundary, security check, improper validation, etc. From this perspective, the developer should be able to identify the sensitive variables or data in his developed software, the problem gets even harder when it is the tester/code reviewer who has to perform this

ID	Security guideline	Validation approach			
		Control	**Data**	**Typing**	**Semantic**
	Input validation				
IDS01-J	Normalize strings before validating them	X	X		
IDS03-J	Do not log unsanitized user input	X	X		
IDS06-J	Exclude unsanitized user input from format strings	X	X	X	
IDS07-J	Sanitize untrusted data passed to the Runtime.exec() method	X	X		
IDS08-J	Sanitize untrusted data included in a regular expression	X	X		
124683	CWE 129: Improper Validation of Array Index	X	X		
	Method declaration and invocation				
MET03-J	Methods that perform a security check must be declared private or final	X		X	X
MET53-J	Ensure that the clone() method calls super.clone()	X			
MET56-J	Do not use Object.equals() to compare cryptographic keys	X		X	X
EXP02-J	Do not use the Object.equals() method to compare two arrays	X		X	
	Secret security				
MSC62-J	Store passwords using a hash function	X	X	X	X
	Store unencrypted keys away from the encrypted data	X	X		X
	Sensitive information security				
MSC03-J	Never hard code sensitive information	X	X		X
IDS15-J	Do not allow sensitive information to leak outside a trust boundary	X	X	X	X
5.1.2(i)	Apps cannot transmit data about a user without obtaining the user's prior permission and providing the user with access to information about how and where the data will be used	X	X		X
5.1.2(ii)	Apps that require users to share personal information, such as email address and date of birth, in order to function will be rejected	X	X		X
	Execution environment security				
OBJ10-J	Do not use public static nonfinal fields			X	

Table 2.3: Security guidelines classified into categories

identification on a developed program. It is also the case for software marketplaces when performing the approval process. In the guidelines that we have gathered from the different sources presented above, we notice the redundancy of the notion of trusted or untrusted data, which is ambiguous to interpret, to understand and to apply on the source code level. In the CERT source, the authors provide a glossary to explain further the different keywords and terms used in the guidelines. For instance, "*untrusted data*" is defined as Data originating from outside of a trust boundary, which leads to uncertainty with respect to the identification and determination of the notion of trust boundary; does it refer to the system input/output? It brings to the table another issue consisting in determining the perimeter of a program. How to determine the system boundaries in the context of a distributed application?

The validation operation is used in multiple guidelines, for example in IDS01-J and 124683. As one can notice, the validation of array index is not the same validation operation from program instructions point of view. Another element caught our attention; the guideline or the family of guidelines that recommends to encrypt or hash the sensitive information. This operation always known as declassification is widely known in the information flow vocabulary [133]. Declassification operation can be seen as controlled release of sensitive information using given mechanisms and techniques such as encryption, hashing, obfuscation, etc. Some rules do not have the same permissibility level, in a sense if they are applied at the same time, this might lead to ambiguities. For instance, the guidelines IDS06-J and IDS03- J; the first rule forbids user input from format string, while the second requires to sanitize user input. This brings to the table the necessity of considering the relationships and dependencies between guidelines. It is important to note that all the specified categories except *Secret Security* are focused on data confidentiality, and somehow miss the integrity which is according to Biba [23] the dual of confidentiality. Integrity unlike confidentiality, can be violated without any interaction with components external to the system. This is the main major behind considering Secret security category as an integral part of our classification. The different sources of security guidelines suffer from different issues, such as:

- The description is informal, which can be subject to misinterpretation
- The description is abstract, which can lead to ambiguities
- Guidelines are contextual
- Application guidelines differ from System guidelines

Upon surveying the security guidelines from the different sources, we noticed a lack of precision and a total absence of automation. Let's take a closer look at some rules that NIST provides:

Ensure that CAs are capable of issuing multiple certificates to users, and for all such certificates, asserting the key usage extension, including the Extended Key Usage extension

Users should generate their own key pairs for digital signatures and authentication.

Users may generate their own key pairs for key establishment, or the key establishment key pairs may be imported from a trusted source. As one can notice, the way the recommendations are presented is very vague and abstract, and there is no precision on the mechanisms or the recommended flow of actions that should take place in order to meet the rule.

We have also noticed the following; the security guideline "Do not log unsanitized user input" from the CERT coding standard appears to be decomposed into 2 guidelines in the OWASP source: "Log injection" and "Input validation". This brought to the table another dimension upon which we can propose enhancements on the guidelines; compositionality and composability of security guidelines.

We want to pinpoint another key element that gathered our attention; there is a huge effort invested in order to build and maintain the catalogs, but no attempt was undertaken to instrument their automatic verification on the code level. OWASP provides sets of security guidelines that should be met by developers, but does not provide the means to ensure their correct implementation. We aim at covering this gap through the formal specification of security guidelines and their formal verification using formal proofs.

2.5 Summary

In this chapter, we proposed a first attempt to classify the security guidelines gathered from different sources into categories. We pointed out another key issue with the guidelines, consisting in considering only *closed* systems, and how they fall short in treating distributed systems and the guidelines complexity they induce. The complexity can be perceived from the difficulty of defining the system boundaries for distributed and 3-tier systems. We pointed out the problems that arise when interpreting and implementing the guidelines: they lack precision and might be subject to misinterpretation by developers. We discuss in the next chapters how we worked towards closing the gap between the informal aspect of the guidelines, and their implementation and application through a formalism that allows to strip away ambiguities. We express these guidelines in a formal language and reason about their satisfiability. Concerning those related to the first three criteria, that is control, data and typing, they should be translated automatically into formal language. However, for those involving the semantic criterion, the formalization should be led by an expert aware of the software security domain, as semantic aspect is highly tied to the context.

Chapter 3

Security Assurance by Certification

Contents

3.1 Introduction

In this chapter, we go through different certification schemes, covering standard and software marketplaces schemes. The main objective behind the survey is to have a deep understanding on the main issues related to current certification schemes, and pinpoint their shortcomings in terms of asserting the security features of the software. Second, to pinpoint the issues related to the certification processes that are not always transparent, neither are the mechanisms that are used in order to issue the certification.

3.2 Open Certifcation Schemes and Security Standards

Security Certification is a common practice that software vendors refer to in order to have the guarantees that their software provides specific security assurance. This practice is carried out by the software vendors, but involves a third-party that is trusted by both the software vendor and the software user. This third-party is known as the Certification Authority (CA). We distinguish between basically two main categories of certification schemes: product-based and process-based security ceritifcation schemes [55]. The first category is only concerned with the technical aspect such as the technical design, the implementation and consider the IT architectures of the evaluated product. The TCSEC, which is a product certification scheme, is considered to be very much time-consuming and to be costly, in addition to the fact that it focuses mainly on the evaluation of the operating systems, which cannot scale to evaluation of the current IT environments complexity.

3.2.1 Common Criteria Certification Scheme

Common Criteria [1], formally known as *Common Criteria for Information Technology Security Evaluation* is a certification scheme gathering a set of guidelines for the security verification of information products. In this certification scheme, the security requirements are developed as part of the Protection Profile that represents a set of security requirements for specific Targets of Evaluation (TOE). More than 2180 products have been certified through the Common Criteria scheme [2]. This certification scheme proposed a wide set of assurance levels EALs (Evaluation Assurance Level) ranging from EAL1 to EAL7. EALs reflect the rigour of the evaluation that was carried out over the product. In other words, a product that is certified with EAL7 does not mean that the product has a higher security, but means instead that the product has undergone more tests. The Common Criteria evaluation process is more focused on the technical aspects, invloving the development lifecycle, the architecture, etc., and do not look at the inner details of the implemented program code. The evaluation rigour reflected by the EALs does not provide assurance about the strength of the implemented security mechanisms within the product, such is the case for FIPS-140 [27]. In addition, the choice of what should be evaluated (TOE: Target Of Evaluation) and for which configuration, are done by the software vendor, which assumes that this certification scheme considers the software vendor requirements, and not the software consumer's.

3.2.2 CSPN Certification Scheme

Certification Sécurité Premier Niveau (CSPN) [47] [*First Level Security Certification* in english] is a certification scheme created by the French certification body ANSSI (Agence nationale de la Sécurité des Systèmes d'Information) in 2008. CSPN evaluation aims at verifying

[1] http://www.commoncriteriaportal.org/

[2] https://www.commoncriteriaportal.org/products/stats/

the adherence of the software to its security specification mainly through black-box testing.

This CSPN certification process takes as average time 2 months, and can in some cases include interactions with the software developers. This would allow the developers to address security issues in their software following the CSPN analysis. The evaluation process might be very costly (around 30k€).

3.2.3 Statement on Auditing Standards 70 Certification Scheme

Statement on Auditing Standards 70 or for short SAS 70, was developed by the American Institute of Certified Public Accountants (AICPA). SAS 70 is a certification scheme that provides a set of security guidelines that should be checked by a human independent auditor on a product; the auditor, who is very often the customer's auditor, issues his or her opinion about the implemented processes and mecanisms within an organisation. According to Gartner Inc.[91], SAS 70 cannot be considered as a certification scheme for being part of an overall compliance process for financial reporting, in addition to its expensive auditing process. The same source reported also that SAS 70 is a generic guideline for the preparation, procedure and format of an auditing report, and not a security or privacy certification.

3.2.4 GlobalPlatform Certification Scheme

We present in this section a specific certification scheme that is intended for mobile devices that are being considered as execution environments that manipulate sensitive, confidential and personal user information, hence, they need to be protected against any comprise. The gap this certification scheme aims at covering is that the protection of such sensitive information cannot be insured by the mobile applications alone. The GlobalPlatform TEE Certification Process is based on a set of security requirements, known as the TEE Security Requirements. "*A TEE is a secure area that resides in the main processor of a mobile device and ensures that sensitive data is stored, processed and protected in a secure environment*" [3]. The evaluation methodology that is carried out can take upto 3 months, and combines automatic Black Box and White Box testing techniques that aim at identifying the vulnerabilities. In this scheme, the Target of Evaluation (TOE) is the execution environment, and comprises any hardware or software that insures the security functionalities. The TOE comprises also the secure usage recommendations after delivery of the trusted execution environment (TEE). The main security goals consist in insuring in addition to the traditional security properties: CIA (confidentiality, integrity and availability), secure execution, secure memory usage, secure storage through the usage of cryptographic APIs and other security mechanisms. However, this certification scheme does not provide guarantees about the absence of covert channels in the applications that are running on the TEE, or if malicious applications that when running

[3] https://www.globalplatform.org/mediapressview.asp?id=1154

in the same TEE manages to launch generic attacks that will not be captured due to their implicit nature.

3.3 Software Marketplace Certifcation Schemes

We present in this section a review on software marketplaces, ranging from cloud to mobile marketplaces. The main objective behind this study is to identify the main issues with the existing marketplaces with respect to security validation of submitted applications. The focus in this review is the security guidelines validation process, known as the **vetting process** or the **approval process**. In this scenario, the main stakeholders are the service/application developer/provider, and the marketplace operator. The former develops and submits his or her application to the marketplace operator, who will, depending on the marketplace strategy, perform a validation on the submitted software. If we compare this process to the generic certification scheme, we would note the following; marketplace operators play also the role of Certification Authorities (CA), and have control over the execution environment where services and mobile applications will be running, which is not the case for the open certification schemes. Vetting process, also known as approval process, is a process through which applications that are submitted to software marketplaces get accepted or rejected based on several criteria, depending on the marketplace policies. We can also claim that the vetting process carried out by each of the marketplace operators can be regarded as a lightweight and custom certification scheme.

3.3.1 Apple App Store

Apple App Store is the official Apple store for iOS (iPhones and iPads) applications. It was launched in July 2008 and was first integrated with iTunes music download service. Apple [54] introduced the vetting process, and defined it as a review on submitted applications to verify whether or not they conform to Apple AppStore review guidelines [15]. Apple AppStore adopts a preventive approach that consists in the vetting process and code signing mechanisms to prevent malicious applications from being advertised in its official store. Code signing mechanism consists in ensuring that only approved and signed executable by Apple are allowed to run on iOS devices. For a software developer/provider to submit his application, he needs first to enroll to the Apple's Developer Program [1]. Once the submitted application passes the vetting process, it is signed and published by Apple in its official store. However, Apple does not provide documentation or details on the verifications that they perform on applications, except in their reply to the FCC [15] questionnaire where they divulged details such as: submitted application is tested for vulnerabilities such as bugs, instability on the iOS platform, the use of unauthorized protocols and the invocation of private APIs, etc. The application is also reviewed in the purpose of preventing privacy issues, as well as, children exposure to inappropriate content. The vetting process aims also at verifying that the application works properly. To achieve this, the submitted application is tested automatically and manually by

more than one reviewer. The approach adopted by Apple is rather preventive, and aims at preventing that malicious applications find their way to the official store, and consequently to the users devices. If the application does not pass the vetting process, a note is sent to the developer explaining the rejection reasons.

Despite the rigourness and the complexity of this vetting process, malicious applications violating Apple App Store guidelines found their way to the official store upon approval [147] [81].

Researches from Georgia Institute of Technology [147] managed to get their malicious application pass the Apple AppStore vetting process. The researchers proceeded as follows: they have first submitted their "benign" application called "Jekyll", and have hidden the malicious behavior. Once the application passes the vetting process, and is installed on the user's device, it can be remotely exploitable to introduce the malicious behavior that was not even present during the vetting process. The Jekyll application contains code fragments, that when assembled together dynamically, introduce a new control flow that can steal user's private information, post tweets, etc.

Apple provides a set of public and private frameworks that contains implementations of its system interfaces. Public frameworks can be used by third-party applications, while private APIs, that can be subject to updates, are for use by Apple developers only for policy or security reasons. If a private API changes, then the third party application that uses it can crash. During the vetting process, if an application is proved to invoke private APIs would result in the immediate rejection from the Apple AppStore.

3.3.2 Google Play

Google Play (originally the Android market) is the official Android marketplace where Android applications are made public to users. Google Play offers to the users different functionalities such as browsing, downloading, and installing applications on their Android devices. In addition to mobile applications, Google Play offers movies, music and books that can be purchased. Google Play provides policies documentation [4]to Android developers who want to submit their applications to the official store. The polices comprise specific privacy policies, specifying the user's personal and sensitive data handling. The approach adopted by Google Play regarding the applications approval is rather reactive, in the sense that submitted applications, contrary to Apple App Store, do not undergo an approval process, but malicious applications are removed from the official store and from the connected devices upon user complaints.

[4]https://play.google.com/intl/en-GB_ALL/about/privacy-security/personal-sensitive/

3.3.3 Amazon AWS Marketplace

AWS Marketplace is a cloud marketplace created by Amazon. Its main stakeholders are the software provider and the software consumer. The former interacts with the AWS Marketplace through the submission of the software and making it public to the end-users. The AWS Marketplace allows those end-users to purchase and deploy services matching their needs in terms of desired functionnalities. There are two ways that this can be achieved:

- By delivering the cloud software deployed on a dedicated virtual machine, an Amazon virtual machine (Amazon Machine Image AMI) and so the customer gets exclusive use of a virtual machine running the cloud software
- Or as a Software as a Service (SaaS) running in a virtual machine on Amazon, so that the customer purchases access to software made available by a service provider

  Amazon Machine Image is "an encrypted machine image of a specific computer running an operating system that is configured in a specific way and that can also contain a set of applications and services for accomplishing a specific purpose" [5] Submitted AMIs undergo virus scans before being added to the AWS official store. In addition, the software information are reviewed with the objective of checking their compliance with AWS Marketplace guidelines.

3.3.4 Google Apps Marketplace

Google Apps Marketplace (or the new G-suite Marketplace) is a cloud services marketplace, that is widely used by more than 40 million users as reported by Google. Google Apps permits to an organization to acquire a private space (called "domain") with a number of Google services (e.g. Gmail for emails, Google Drive for document exchange and so on). This instance of Google services is reserved for the organization. The Google Apps Marketplace allows Google App customers to purchase third-party services for their domains, so to enrich the amount of available functionalities. Third-party services have to integrate with a number of specific Google API (for authentication and Single-Sign On for instance).

3.3.5 Security Requirements

Software Marketplaces use different approaches in order to prevent malicious applications from being advertised in their official stores.

Google Play, for instance, does not perform an approval process on the uploaded applications, but follows a reactive approach that removes or disables identified malicious

[5] https://aws.amazon.com/marketplace/help/201231340

applications from the marketplace, as well as from the connected devices having installed the malicious application. The malware identification is also based upon reports provided by users (crowd-sourced vetting). Google Play has chosen to let the users grant or not explicit permissions that the application requires, and can limit the application access to their devices and data; on the other hand, users when choosing not to grant required permissions to the application to install, they do compromise the usability to the detriment of the security. Application sandboxing mechanism that aims at isolating applications, was put in place in order to reduce the negative impact when a malicious application is installed. Unlike Google Play, Symbian adopts another approach consisting in requiring that applications that need to access core OS files or modify system configurations, to be submitted to the Symbian approval program "Symbian Signed". This program allows developers to test on a live device and sign their applications, and get them certified for distribution. Symbian allows users to check the validity of a certificate on an online server. Digitally signing applications is a more common security feature for the software marketplaces, allowing them to make sure, based upon signature verification, that the application was not modified and that it emanates from the intended author. Each marketplace implements the code signing technique according to its requirements. Google Play adopts a self-signing technique, where the developers generate their own signing key, and sign their applications without the involvement of Google. The utility of the self-signing approach is to verify that the application updates (such as permission restriction) originates from the same application developer. Apple App Store digitally signs applications that pass the rigorous vetting process performed when application providers upload their applications. Apple imposes that only digitally signed applications run on iOS devices. Symbian follows another approach that gives users control over their devices. Even though Nokia Store performs code signing over applications that pass the vetting process (Symbian Sign Program), the platform allows the users to configure and accept applications (whether signed or unsigned) to run on their devices. For applications that require additional privileges, Symbian wants them to be self-signed. Code signing approach is used by Blackberry, in order to track private API calls. For developers who need to use the restricted APIs, they have to purchase a signing key from RIM, that generates a unique key for each developer. This gives control over the applications and the certificates attributed.

3.3.6 Discussion

Upon the survey that we carried out over the software marketplaces, we can conclude the following; marketplaces do not disclose details on the security assessments they perform, neither on the vetting process results for each of the evaluated applications. For instance, Apple App Store might re-evaluate applications that have already passed

the vetting process, and that are published in its official store. They provide a detailed explanation on the reported issue [6], however, it can be misleading to the application developer/provider to understand the source of the violation and how to fix it. According to McDaniel and Enck [57, 56], marketplaces do not have an identifiable point for departure to begin application analysis. In the same article, the authors claim that analysis tools fall short to discover what an application can do once installed on the target platform. In other words, it is hard to predict the application behavior at runtime. In the context of a software marketplace, a huge number of applications are submitted monthly, and applying sophisticated analysis on each submitted application is very difficult, even impossible. However, the overall security of the marketplace can be reached, when insuring some trustworthiness facets, such as digitally signing applications, services certification in the marketplace, and performing dynamic analysis such as *taint tracking* on submitted applications as part of the vetting process.

3.4 Summary

One might argue about using Certification schemes such as *Common Criteria* [3] to evaluate security aspects of software, instead of applying the approach that we propose. We can provide those elements that might help to give clear view on this specific point: Certification process is expensive and lengthy, and might take upto 8 to 12 months for software to pass the evaluation. High assurance level can be even more expensive and long (1-2 years). Since software is subject to frequent updates, certification is not the most relevant process that can be used to evaluate the security of developed software. This process will be run once, and there is no guarantee after several updates that what was certified, will still meet the requirements. Certification process reports indicate the conditions for which the certified target meets the criteria that it is certified for. Those conditions appear in the certification reports as assumptions, and those strong assumptions do not take into account the actual deployment environment where the application will be running. Moreover, produced certificates require an important amount of human effort to interpret their results that are written in natural language and are intended to human operators in the first place. In addition, consumers of certified software are advised to make sure that their operating environment is consistent with the evaluated configuration, and to give due consideration to the observations and recommendations in the certification report. Besides, the details of the security assessment (what was audited, what was ignored, what security validation mechanisms have been used, etc.) performed by the certification authorities or the marketplaces are never divulgated. As pointed out in [92], understanding the results of the certification process is not trivial for consumers, and there are no means to verify whether the

[6] https://forums.developer.apple.com/thread/73640

certification claims meet their requirements. We have also identified another facet; the state-of-the art on security mechanisms evolve and get updated rapidly, and we can think of the case where products implementing MD5 or SHA1 being certified, another question we raise is do those products still find their places among security certified products? Again, certification, similarly to traditional software security mechanisms, does not really give guarantees about the behaviour of the software. Digitally signing the software might provide certainty about the origin of the code, but not its behaviour [19].

Chapter 4

Towards an End-to-End Automatic Verification of Security Guidelines

Contents

4.1 Introduction

The security guidelines are designed in order to help developers build secure software. As we stressed in the previous chapter, guidelines are informal, and present ambiguities that can lead to misinterpretation by developers. We pointed also another key element that highlights the fact that there is a huge effort invested to establish and maintain the guidelines catalogs, but, to the best of our knowledge, no automatic tool ensuring the verification of those guidelines on the code level was proposed. One might argue about using Certification schemes such as *Common Criteria* [3] to evaluate security aspects of software, instead of applying the approach that we propose and that we discuss in

details in this chapter. As we pointed out in the previous chapter, Certification process is expensive and lengthy. In addition, it does not provide guarantees about how the software behaves in the production environment. The point we are trying to make is, our approach that will discussed in details in this chapter, can be regarded as a complement to certification, as we operate on the **code level**, and the verification that we carry out is **automatic** and **cost-effective**. In addition, we aim at bringing security awareness to developers and help them give due consideration to security compliance during the development phase.

4.2 Approach for the Automatic Verification of Security Guidelines

In this section, we will go through the details of the approach that we propose with the objective of filling the gap between the informal description of security guidelines, and their automatic verification on the code level to provide precise and comprehensive feedback to the developer. We started first by doing attempts to extract security properties from the code level, but we found out that we needed to have a reference against which we can compare the extracted program parts. This brought the idea of performing a deep survey on the guidelines that we gathered from different sources, as explained in chapter 2. The positive (resp. negative) security guidelines serve to express the desired (resp. undesired) program behavior. However, we operate on the code level, meaning that we do not monitor the program execution. We need then to approximate the program behavior but still from a static point of view. This requires that we transform the program into a formal model that allows to exploit its properties, and approximate its behavior. In addition, the program model that we construct should be able to represent the whole flow of information in order to be able to reason about how data propagates, and capture possible information leakage. This induces the need to choose a formalism to represent the information flows that can be checked over this program model. Having covered the aforementioned aspects, we proceed to verify whether the program meets the specified guidelines or not. As we aim to decrease considerably the heavy load on the developer, we propose an automatic formal verification of the guidelines. The purpose is to verify that security guidelines are met, and not to prove that the program is correct. To the best of our knowledge, no prior work has filled in this gap between the informal description of security guidelines and their automatic formal verification on the code level.

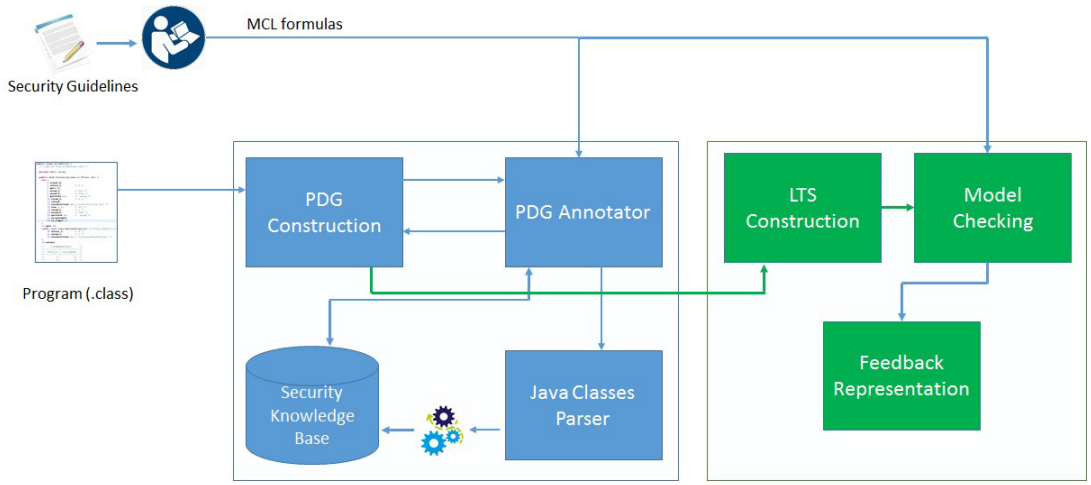


Figure 4.1: Approach for the formal specification and automatic verification of security guidelines

4.2.1 Formal specification of security guidelines

This step constitutes a fundamental part of this thesis work, and the main work it achieves is filling in the gap between the informal description of security guidelines, and their formalization in a mathematics-based formalism. As the guidelines express desired as well as undesired program behavior, we deemed apropriate to adopt temporal formalism. We make the strong assumption that the **security expert(s)**, Brian, formally specifies the security guidelines by extracting the key elements from the informal description, and builds upon them the formulas based on the chosen formalism. We provide help and guidance to Brian through the formalization of the guidelines by means of the patterns that we make available to the security experts. The security guidelines will be modeled in the form of sequence of atomic propositions or statements representing the behavior of the program. The built formulas can be supported by standard model checking tools to perform the automatic verification. The outcome of this phase is generic security guidelines that can be instantiated on different programs. For instance, the action label *save*, which defines the operation of saving a given data in a specific location, can be instantiated on save_to_DB, save_to_file, save_to_array, etc, depending on the invoked instruction and its parameters. This instantiation operation is also handled in our Security Knowledge Base 7.5. We carried out the effort of automating the formal specification of the security guidelines through an approach that we established. The automation will be discussed in further details in 5.

4.2.2 Construction of the Program Model

Considering that the guidelines are formalized in a mathematics-based formalism, they need to be checked over a program model that supports the chosen formalism. As pointed out above, the program model should be able to represent the whole flow of information, and approximates the program behavior. As the starting point of this step is the program source and byte codes, we construct a program model that captures the different dependencies that might occur between the program instructions. Dependencies mean direct and transitive data dependencies, that is explicit assignements of sensitive information to public variables, or implicit that may occur in branching conditions and influence the control flow in a program. The notion of implicit information flow is also defined as being a larger class; covert channels [23]. We studied different program representation models [161], including the Control Flow Graph (CFG), Data Flow Graph (DFG), Abstract Syntax Tree (AST), etc. Upon this survey, we found that the Program Dependence Graph is the most appropriate abstraction model that represents both the control and data dependencies, and fits our needs in terms of approximating the information flows within the program. We are aware that PDG is an over-approximation of the program behavior, and this can lead to **false alarms**, hence to the imprecision of the analysis. In addition, PDG is **not formal**, which constitutes another obstacle regarding the use of PDG as a support to perform the formal verification.

In order to increase the analysis precision and reduce the false alarms , we have first transformed the standard PDG into another PDG that we named "Augmented PDG". The term "augmented" comes from the effort that we carried out to detect implicit dependencies on the PDG, and further details that are not captured in the first place by the PDG. We augment the PDG through the Information Flow Analysis to capture the explicit as well as the implicit dependencies that can be source of covert channels, and may constitute source of sensitive information leakage. To address the second issue, we stress that the augmented PDG is used as an intermediate program representation. We then generate from this augmented PDG a formal model, a Labeled Transition System structure representation that is supported by model checking tools.

4.2.3 Formal Verification

Since we aim at providing an automatic verification of the guidelines on the program, we deemed necessary to adopt model checking as formal verification approach. This operation exploits the formalized guidelines and formal program model, and consists in proving the program correctness with respect to the specified guidelines. Formal verification is intended in the first place to check the correctness of safety-critical systems. And at this level, we claim that the formal verification that we make use of in this thesis

work is generic, in the sense that it can cover different use cases, ranging from safety to security.

Model checking [73] basically explores the state space that describes the possible program behaviors. As pointed out in the previous section, the state space refers to the formal program model that we generate automatically from the augmented PDG. A model-checking tool accepts a system model specified in a given formalism, and the system properties that should be verified by the system. The tool then outputs a boolean answer indicating whether or not the property was met. An output *yes* indicates if the given model satisfies the specification, otherwise, a counterexample is generated. The counterexample provides a detailed explanation on the violation that occurred. A thorough examination on the counter example allows to pinpoint the source of the violation in the program model, and possible indication on how to fix it.

In the first case, the verification output indicates that the guideline is valid upon the exploration of the whole flow of information in the program. The second case can be advanced further, meaning that the verification can provide more details to the developer Alex (or the tester) about circumstances under which the security guideline was not met, and counterexamples are produced. Here we stress the feedback that should be precise and easy to understand by Alex the developer.

4.2.4 Security Knowledge Base

Security Knowledge Base is a centralized repository gathering the labels of the formulas mapped to APIs, instructions, libraries or programs. This helps the automatic detection of labels on the system model. We built **Security Knowledge Base** using a Java classes parser [157] that operates as follows: for the different Java classes used in the program to analyze, we launch the parsing of this given class (html code, javadoc), and we extract all the relevant details, such as the description, the attributes, the constructors, the methods signatures and their parameters. Then, we did the effort of performing a semi-automatic semantic analysis to detect key elements, such as the keyword *secure*, *key*, *print*, *input*, etc. This operation is of a paramount importance, as it allows to map the key words used to build the formulas, to the possible Java language instructions (methods invocations, constructors invocations, specific data types declarations, etc.). For example, the Java API *KeyGenerator.generateKey()* is mapped to the label *isKey*. This label is also mapped to the traditional information flow annotation **high** level **source**.

4.3 Impact of Information Flow Analysis on Security Guidelines Verification

Controlling how information flows through out a program is of paramount importance when dealing with information security. From a historic perspective, access control mechanisms [22], as their name indicates, are used to verify the access rights at the point of access. Then, they grant or deny the access to the asset over which the mechanism is set. Similarly, encryption mechanisms, which may ensure the secrecy of data, do not provide the means to track or to control the information flow within a program. Access control mechanisms, similarly to encryption, can't provide assurance about where and how the data will propagate, where it will be stored, or where it will be sent or processed. This entails the need for controlling information flow using static code analysis. This same idea is emphasized by Andrei Sabelfeld, and Andrew C. Myers [131], who deem necessary to analyze how the information flows through the program. According to the authors, a system is deemed to be secure regarding the property confidentiality, if the system as a whole ensures this property.

Information flow analysis falls in one of the categories; static (such as language-based analysis) and dynamic (like code instrumentation and tainting).

Dynamic analysis' strength lies in the possibility to take action in the presence of dynamic inputs. This type of analysis generally considers the program to analyze as an atomic artifact. It resorts to different approaches in order to analyze the run-time behavior of the program, often without any access to the source code. Dynamic analysis reports failures at the instant they occur, and provides details allowing the developer/tester to make the required corrections. However, it is quite difficult to trace the detected vulnerability back and pinpoint the exact location in the code where it occurred. This approach requires a sufficient number of test cases and is quite time-consuming, yet it cannot ensure an automatic verification or a complete coverage of the test cases space of the analyzed program.

In comparison, static analysis in all its forms ensures a complete coverage of the program branches [123] used APIs, program dependencies, or configuration files explored. Static analysis refers to different methodologies, including model checking and model provers, to verify the execution paths of a program without actually executing it [41]. Unlike manual review, which relies on the tedious examination of sequences of the concrete or symbolic execution of a program, static code analyzers can capture comprehensive and accurate models of the software, like for instance an abstract representation of all the execution paths, which test-case execution falls short to cover.

Language-based techniques, unlike traditional software security mechanisms, operates at the code level and analyze the semantics of the program and its behaviour. Information

Flow Control (IFC) falls in the language-based approaches that aim at discovering security leaks at the code level. One might argue about using Proof-Carrying Code (PCC) [108] [109] to perform a formal verification of the desired properties on the code. In PCC, the service consumer defines the safety policies to be verified over the software, and they cover a wide range of properties, ranging from functional, to safety and security-related properties. The code producer, on the other hand, produces the formal proofs that prove the adherence of the system to the desired properties. The European project MOBIUS [19] provides a proof-carrying code architecture to secure Java-enabled mobile devices. However, the preparation of proofs is expensive. Denning and Denning [50] were the first to observe that static code analysis can be used to control information flow within a program while increasing the analysis precision and reducing run-time overhead.
In our framework, we make use of the Information Flow Analysis for many purposes, mainly the detection of implicit and subtle dependencies that can be source of covert channels and sensitive information leakage. From a security perspective, this might have serious damages on the security of the infrastructure as well as on users sensitive data.

4.4 Summary

In this chapter, we presented on a high level the approach that we propose with the objective of filling the gap for the verification of the security guidelines. We pinpoint different issues with the security guidelines that are present in different sources, but no verification means is provided to the developers to make sure that their being developed software adheres to those guidelines. Security guidelines are meant mainly for developers, but the way they are presented presents ambiguities, and this might lead to misinterpretation. Formalizing the guidelines would help stripping away ambiguities, and preparing the ground for the formal verification. This will be discussed in details in the next chapter. We stressed on the need of performing model checking as verification approach. This allows to have an automatic verification, hence to reduce the intervention of a human operator, whether the developer or the security expert to lead this verification.

Chapter 5

Formal Specification of Security Guidelines

Contents

5.1 Introduction

Transforming the informal description of security guidelines into exploitable mathematical formulas is an important part in this thesis work. We stressed in Chapter 2 the main issues that we have identified with respect to security guidelines, and the difficulties that can occur when they are interpreted and implemented by software developers, and verified afterwards by testers or code auditors. In addition, security guideline presentations differ from one source to another, hence, there is a need for a centralized and uniform approach to represent them. In this Chapter, we present the methodology that we provide to the security expert in order to help him build the guideline formulas. We

focus on the extensibility as well as on the automation that we can bring to this phase to strip away the difficulties which the security expert can face when building the guidelines formulas, especially those that involve dependent data flows. We explain in details the flow of operations, and pinpoint the effort we have undertaken to automate the formalization of security guidelines. For a clear understanding of the specification flow, we consider concrete examples of guidelines that Brian, the security expert, formalizes, starting from their informal description.

The main idea behind this Chapter is to transform the security guidelines written in natural language, into exploitable mathematical formulas. The advantage of this formalization is to strip away the ambiguities of natural language, and to increase the precision. This phase produces a formal specification of the security guidelines.

5.2 Formalism for Model Checking

Since the guidelines that we want to specify reflect the desired program behavior, we can conclude that what is needed is a formalism that allows to describe the execution of the program. Those guidelines are generally abstracted away in formal models as verification problems are undecidable for infinite systems.

Adopting Temporal Logics [105] seems a wise direction as they are suited to specify with rigorous semantics the desired properties in the form of sequences of atomic propositions or statements representing the behavior of the system. In the literature, different temporal logics have been studied and proposed for the specification of program properties. "Temporal" refers to the sequencing and chronological order of states, and not to the notion of physical time.

Different Temporal Logic formalisms exist and are widely adopted, such as LTL and CTL, with their different variants. The syntax of linear temporal logic (LTL) [125] is defined inductively using the standard boolean operators, and the temporal operators X(next), F (eventually), G (always OR globally), and U (strong until) by the following abstract syntax equation:

$$\varphi := p \mid \neg p \mid p \wedge q \mid \mathbf{X}p \mid \mathbf{G}p \mid \mathbf{F}p \mid p\,\mathbf{U}\,q$$

For example, let us consider a simple property and write the corresponding LTL formula. Let us take the known example of a microwave oven, and one of its properties: "*No heat while door is open*". This property is a safety property, hence should begin with the operator G (always). It also states that if the microwave's door is open, it cannot heat. This can be expressed in this LTL formula:

$$\mathbf{G}(Heat \rightarrow Close)$$

The predicates that we made use of in this property are: Heat and Close, and they refer to actions. Those actions, as specified in the formula, should occur under specific conditions. For instance, if the microwave is heating (Heat), it implies that its door is closed (Close).

Now if we consider another common example: "*No writes on a file after it is closed*". This property states that once a file is closed, we cannot write on it. In other words, we need to constrain the operation of writing on a file: we prohibit it if the file is closed, and allow it once it is open. The different actions (labels) that we can use to build the LTL formula are: *Write*, *Open* and *Close*, and refer respectively to the actions of writing on the file, opening the file and closing it. However, those actions operate on a parameter which is the *file*. If we write the LTL formula similarly to the previous microwave example, we would be missing the parameterized characteristic of actions. "*File*" is not an action, but rather a semantic data type that is attached to a specific resource. This cannot be expressed using the syntax of LTL: the actions *Open*, *Write* and *Close* are performed over this *file* and LTL does not describe actions parameterized with data.

$$\mathbf{G}(Close \rightarrow (\neg Write \mathbf{U}\ Open))$$

This formalism is insufficient, as it does not handle value passing. It allows reasoning about the progress of computations and actions, but not about the data that is being manipulated.

We worked towards closing this gap, and propose an extension to LTL following the same approach as [5], but we were faced with: first the difficulty of amending the LTL formalism and second, the absence of a model checker that could accept this extended formalism. As the implementation of such a tool would be too much time consuming, and would necessarily require an advanced level in formal methods and techniques, we have chosen to deepen our research and found another formalism that would meet our needs in terms of parametric labels and value passing.

Different prior works have proposed extensions to existing logics such as ETL. ETL[152] was proposed by Vardi and Wolper, and is an extension of LTL with temporal connectives. SALT [20] is another extension of LTL enriched with regular expressions. HyperLTL [45] [127] allows the specification of properties over multiple execution traces at the same time. SecLTL [51] extends LTL to express information flow properties for security. Despite the expressiveness of those extensions with respect to the original temporal logics, they are still difficult to apply by the average developer due to their complicated syntax and semantics, which require advanced level in formal methods.

The Process Specification Language (PSL) [134] is a formalism designed to specify exchanges between processes. This specification language allows to define parameterised properties and can handle value passing. However, it is specific to the hardware domain.

Since the security guidelines in our focus involve data passing in addition to execution statements that are data parameterized, we need a powerful formalism that is suitable to cover those aspects and that is enriched with value passing mechanisms.
We present in the next section the Model Checking (MCL) logic.

5.3 Model Checking Language

In this section, we explain in more details the Model Checking Language (MCL) that we adopted for the formal specification of security guidelines. The formalism that we have used is very intuitive in the sense that buidling formulas is close to the natural processing of texts. We adopt the MCL logic [103] that addresses this crucial matter of representing and handling data. It allows reasoning naturally about systems described in value-passing process algebras such as LOTOS.

MCL (Model Checking Language) is an extension of the alternation-free regular μ-calculus with facilities for manipulating data in a manner consistent with their usage in the system definition. The MCL formulas are logical formulas built over regular expressions using boolean operators, modality operators (the necessity operator denoted by ([ ]) and the possibility operator denoted by ($\langle\ \rangle$) and the maximal fixed point operator (denoted by μ).

By using data, one can specify state machines with an infinite action alphabet, in which quantification over data can be used, and propositions and variables may have data parameters. MCL is a powerful language that combines value passing mechanisms together with regular expressions, and constructs similar to programming languages

Most of the guidelines we studied can be expressed as usual safety properties that we encode in the MCL formalism by $[\phi]\texttt{false}$ formula, stating the absence of bad execution sequences characterized by regular formulas ϕ or as basic liveness properties, encoded by $\langle\phi\rangle\texttt{true}$ and stating the existence of good execution sequences characterized by ϕ. Regular formulas ϕ are built over action formulas and the standard regular expression operators namely concatenation (.), choice (|), and transitive-reflexive closure ($*$). Action formulas are built over action patterns and boolean connectors. Action patterns are of two different kinds: action patterns for matching values denoted by ($\{\texttt{A}\,!\texttt{e}_1 \ldots !\texttt{e}_n\}$), or action patterns for extracting and storing values denoted by ($\{\texttt{A}\,?\texttt{x}_1 : \texttt{T}_1 \ldots ?\texttt{x}_n : \texttt{T}_n\}$), where $\texttt{A}$ is the action name, $\texttt{e}_i$ are expressions (data variables or functions), $\texttt{x}_i$ are data variables and $\texttt{T}_i$ are types, namely `Int`, `Bool`, `String`. The `true` constant is used to match a value of any action formula.

Table 5.1: Syntax of MCL

Expressions:	e ::= x|f(e1, ..., en)
Action formulas:	a ::= c!e1...!en|c?x1 : T1...?xn : Tn|¬a|a1 ∨ a2
State formulas:	φ ::= e | ¬φ | φ1 ∨φ2 | ⟨α⟩φ | exists x1:T1, ..., xn:Tn.φ | Y (e1, ..., en) | μY (x1 :T1 :=e1 , ..., xn :Tn :=en).φ | let x1 :T1 :=e1 , ..., xn :Tn :=en in φ end let | i f φ 1 t h e n φ ′1 elsif φ2 then φ′2 ... else φ′n end if | case e is p1 →φ1 |...|pn →φn end case}

Syntax of the Model Checking Language Now if we go back to the property "*No writes on a file after it is closed*" and formalize it with MCL. The parameterized action patterns that we define and use to build our formula are:

- {File ?file : String} designates the action pattern for storing the file path "*file*" of type String
- {Write !file} designates the action pattern of writing on the file *file*
- {Close !file} designates the action pattern of closing the file *file*

```
[true*.{File ?file:String}.(not({Close !msg}))*.{Write !msg}] false
```

This formula specifies that if the file *file* is not closed, and that the action of writing on this file occurred, then this formula is false. This means that if we capture this behavior in our program, then a violation should be detected. The "(*true**)" indicates any action. The formula can be read as follows: after any action occurring, if there exists a File *file* (*File ?file:String*), and that this file is not closed (*not(Close !msg)*), followed at a some point in time by a write on this file (*Write !msg*), then a violation should be raised. Note that this MCL formula is built upon regular expressions operators (., *), and note also that they are familiar to software developers.

5.4 Construction of the Formal Specification of Security Guidelines in Model Checking Language

Having presented the formalism that we adopt to specify the security guidelines, we examine its applicability on a number of guidelines that we have considered. Before proceeding to the formal specification, we explain the methodology that we provide to the security expert to build the MCL mathematical formulas from the guidelines descriptions.

The formalization flow is the following:

- Extraction of the keywords (action patterns) and the key-concepts from the guideline textual description
- Construction of the basic MCL formula
- Construction of the MCL formula involving explicit and implicit dependencies

Each step as well as the transition from one step to another are depicted in detail in the next sections.

The proposed formalization in the MCL language presents the key concepts required by each of the guidelines. The challenge is to define those concepts and actions and have a unified yet extensible set of keywords that can be used by the security expert(s) to formalize the guidelines. We tried to reduce the expert intervention by introducing the Security Knowledge Base (See Section 7.5) which is a central repository containing the different labels used for composing the guidelines, together with their description and semantic meaning.

5.4.1 Extracting the Key-concepts

First, Brian, the security expert, is faced with the guideline written in natural language, and needs to identify the key elements at the heart of the guideline (sensitive data, security mechanism, etc.) before proceeding to the specification. Brian refers to the Security Knowledge Base (SKB) (see Section 7.5), and more specifically to the security guideline patterns and the labels that are used to build the formulas; they can be used to guide Brian when building the formulas based on the existing patterns. This operation will result in a basic formula; the term *basic* means that the formula involves the most intuitive and simplistic pattern that considers only the data assets to be protected. The basic formula does not involve the explicit or the implicit dependencies that data might have. For instance, if the data was assigned to a variable

(implicitly or explicitly), this data dependency will not be captured in the basic formula.

Let us take a closer look at some of the guidelines that we have surveyed, and explain how Brian proceeds to build them. We have considered the three guidelines IDS03-J:***Do not log unsanitized user input*** [30], MSC62-J:***Store passwords using a hash function*** [40] and OWASP: ***Store unencrypted keys away from the encrypted data*** [119]. These three guidelines illustrate several concepts, such as the notion of single or multiple parameters, patterns, etc.
Let us now look at the keywords and the concepts that Brian can extract from the textual description, and provide explanation for each of them. It is important to mention the following; considering only the textual title or the name of the guidelines is not sufficient, we need also to look at the detailed textual explanations that provide very useful details, such as the cryptographic algorithms that should be used, or the hashing method to be applied in a reliable manner. For example, in the guideline MSC62-J:***Store passwords using a hash function*** [40], the hashing should be implemented through a robust and reliable algorithm, and those details appear within the detailed description of the guideline. CERT provides the following: *Avoid defective functions such as the Message-Digest Algorithm (MD5). Hash functions such as Secure Hash Algorithm (SHA)-1 and SHA-2 are maintained by the National Security Agency and are currently considered safe.* However, SHA-1 is no longer a reliable hashing algorithm, it got recently broken [1] [2]

From this perspective, if we consider that any password hashing operation as safe, even using a non-reliable algorithm, then this might lead to a violation of the password's integrity. This brought further reflection on the dictionary enrichment considering a multitude of elements such as: the hashing/encryption algorithm to use in what context, the key length, etc. To give another example, the usage of DES for symmetric encryption purposes in indicated in our **SKB** as prohibited, as well as the hashing through MD5. Those aspects brought interesting elements that we discuss in Section 6.4.2.

IDS03-J:*Do not log unsanitized user input* [30] The keywords that Brian extracts from this guideline textual description are:

- Log
- Sanitize
- User input

[1] https://www.quora.com/Cryptography-Why-are-MD5-and-SHA1-called-broken-algorithms
[2] https://www.theverge.com/2017/2/23/14712118/google-sha1-collision-broken-web-encryption-shattered

Logging is the operation of registering, recording and saving actions in a specific format. *Sanitization* is the operation of eliminating the suspicious characters from a given data to avoid log injection attacks. *UserInput* is the characterization of data provided as input.

MSC62-J:*Store passwords using a hash function* [40] Similarly to the previous guideline, Brian identifies the main concepts involved:

- Store
- Password
- Hash

Storage is the process of persisting and saving data into a memory. *Password* designates the secret word used to authenticate a user to prove he or she has the right to access. *Hashing* is the action of transforming a text string into a fixed-length digest that cannot be reversed. The digest is generated by a formula that makes it very hard to produce the same digest for another text.

OWASP: Store unencrypted keys away from the encrypted data [119]

- Encryption key
- Store
- Encrypt

Encryption key denotes the key that is used to cipher (sensitive) data. Storage is the process of persisting and saving data in a memory. *Encryption* is defined as *Data originating from outside of a trust boundary*, or data originating from untrusted sources.

5.4.2 Building the Basic Formulas

Once the key-concepts are retrieved from the guideline textual description, Brian might refer to the Security Knowledge Base to check whether the extracted words have a matching entry. He might for instance find synonyms or semantically equivalent words (labels) that can use to build the formula. Having selected the key concepts, Brian proceeds to building the more natural and intuitive MCL formula.

Since establishing a single dictionary is the key to building the formulas, we identified the need to have a central repository containing the guidelines relevant details, such as the patterns and the labels used to build the guidelines formulas. Brian can access this central repository (Security Knowledge Base 7.5) and can use it as a reference

model. When formalizing the guidelines, Brian extracts the key concepts from the textual description, and at this level, he consults the dictionary to see if the identified concepts have already been used in other guidelines patterns, and to check if there exists a pattern that he can make use of to build the formula. If not, Brian introduces the new labels, and attributes semantics reflecting the meaning of those new labels. We need to stress that our central repository (Security Knowledge Base in section 7.5) can be extended to cover a wide range of security concepts.

IDS03-J:*Do not log unsanitized user input* [30] This guideline specifies that user information should not be logged unless it is sanitized. In other words, the guideline involves a certain precedence between the actions *Log* and *Sanitize* that are parameterized with the *input data*. Input data is not an action (like *log* and *store*), but a semantic data type that designates data provided by the user.

The MCL formula of this guideline can be written very intuitively as follows:

```
[true*.{userInput ?msg:String}.(not ({sanitize !msg}))*. {log !msg}] false
```

where *msg* is the userInput variable of type String.

The formula means that we might have any action $true*$ occurring before the declaration of a user input ($userInput?msg : String$). This action pattern specifies that the parameter *msg* of type String is the user input $userInput$. If this *msg* is not sanitized ($not(sanitize!msg)$) before being logged ($log!msg$) then this property is false. In other words, if this pattern is detected on the program under analysis, then a violation should be raised.

MSC62-J:*Store passwords using a hash function* [40] For instance, the guideline MSC62-J stating : *"Store passwords using a hash function"*, will be encoded directly by the following MCL formula:

```
[true*.{setPassword ?msg:String}.(not ({hash !msg}))*. {log !msg}] false
```

This basic formula specifies that for each possible value of *msg* (the identifier of a password as determined from the action setPassword), the action *store !msg* denoting the storage of the corresponding *msg* stored cannot be reached before *hash !msg* action. It is important to note that *msg* denotes the password data, in other words, the value of the password variable. The undesired behavior consists in storing the password data *msg* in plain text without being hashed beforehand.

The formula expresses a bad sequence of actions that we want to prevent. It tries to match sequences that begin with zero or more of any action (denoted `true*`) followed

by a password creation action ({`setPassword`?`msg:String`}) followed by any action, and end with a storage action ({`store`!`msg`}). This logic also enables to express liveness properties by using the fixed point formula.

OWASP: Store unencrypted keys away from the encrypted data [119]

```
[true*.{create_key ?key:String}.true*.
{save !key ?loc:String}
.true*.{encrypt ?data:String !key}.true*.
{save !data !loc}.true*)] false
```

This formula presents four actions: the action *{create_key ?key:String}* denoting that the encryption key *key* (of type *String*) is created, the actions *{save !key ?loc:String}*, *{save !data ?loc:String}*, *{encrypt ?data:String !key}* denoting respectively the storage of the corresponding *key* in location *loc*, the storage of the corresponding *data* in location *loc*, the encryption of *data* using *key*, and the particular action *true* denoting any arbitrary action. Note that actions involving data variables are enclosed in braces ({ }).

5.4.2.1 Patterns for the Formal Specification of Security Guidelines

When formalizing the security guidelines described in the previous section, we have noticed that there are recurring patterns. For instance, the notion of *precedence* arises in different guidelines, and constrains the order of actions. For example, in the guideline IDS03-J:***Do not log unsanitized user input***, the *Sanitization* operation on the user input should occur before the *Logging* action. Similarly, the guideline MSC62-J:***Store passwords using a hash function*** requires the *Hashing* of the password to occur before its *Logging*. Another property pattern that we have captured is the *absence* notion, meaning that an action should never occur when another action is executed. For example, in the guideline MSC03-J:***Never hard code sensitive information***: the hard coding should never occur.

Since we aim at leveraging the workload and bringing assistance and guidance to the security expert, we decided to further simplify the formalization of the security guidelines and propose property patterns. Thanks to these, the security expert will hopefully not need to have full knowledge about the formal language expressiveness nor to master its idioms. He would eventually prefer to have more guidance on how to use the language features. This would improve the transformation of the security guidelines from natural language to formulas.

Similarly to Dywer's work [53], we propose a catalog of patterns for security guidelines. These patterns identified in [103] will allow programmers who are not necessarily experts in formal language to read a formal specification. As proposed in [103], we encode guidelines as a collection of property patterns intended to simplify the specification activity. These patterns belong to particular classes: *Absence*, *Existence*, and *Universality* and occur in various scopes: *Before*, *After*, and *Between*. Those classes reflect specific requirements such as the absence of an action, precedence of actions, response of an action that is triggered by another action, etc. The proposed patterns were adopted by the Bandera Specification Language [46].

The most important of these patterns [103] are the following presented in Table 5.2.

Table 5.2: Patterns in MCL language

Pattern	Scope	Formula
Absence (α_1 is `false`)	Globally	$[\texttt{true}*.\alpha_1]\texttt{false}$
	Before α_2	$[(\neg\alpha_2)*.\alpha_1.\texttt{true}*.\alpha_2]\texttt{false}$
	After α_2	$[(\neg\alpha_2)*.\alpha_2.\texttt{true}*.\alpha_1]\texttt{false}$
	Between α_2 and α_3	$[\texttt{true}*.\alpha_2.(\neg\alpha_3)*.\alpha_1.\texttt{true}*.\alpha_3]\texttt{false}$
	Between α_2 until α_3	$[\texttt{true}*.\alpha_2.(\neg\alpha_3)*.\alpha_1]\texttt{false}$
Existence (α_1 becomes `true`)	Globally	$\mu\texttt{Y}.\langle\texttt{true}\rangle\texttt{true} \wedge [\neg\alpha_1]\texttt{Y}$
	Before α_2	$[(\neg\alpha_1)*.\alpha_2]\texttt{false}$
	After α_2	$[(\neg\alpha_2)*.\alpha_2]\mu\texttt{Y}.\langle\texttt{true}\rangle\texttt{true} \wedge [\neg\alpha_1]\texttt{Y}$
	Between α_2 and α_3	$[\texttt{true}*.\alpha_2.(\neg\alpha_1).\alpha_3]\texttt{false}$
	After α_2 until α_3	$[\texttt{true}*.\alpha_2]([(\neg\alpha_1).\alpha_3]\texttt{false} \wedge \mu\texttt{Y}.\langle\texttt{true}\rangle\texttt{true} \wedge [\neg\alpha_1]\texttt{Y}$
Universality (α_1 is `true`)	Globally	$[\texttt{true}*.\alpha_2]\texttt{false}$
	Before α_2	$[(\neg\alpha_2)*.\neg(\alpha_1 \vee \alpha_2).(\neg\alpha_2)*.\texttt{true}*.\alpha_2]\texttt{false}$
	After α_2	$[(\neg\alpha_2)*.\alpha_2.\texttt{true}*.\neg\alpha_1]\texttt{false}$
	Between α_2 and α_3	$[\texttt{true}*.\alpha_2.(\neg\alpha_3)*.\neg(\alpha_1 \vee \alpha_3).\texttt{true}*.\alpha_3]\texttt{false}$
	After α_2 until α_3	$[\texttt{true}*.\alpha_2.(\neg\alpha_3)*.\neg(\alpha_1 \vee \alpha_3)]\texttt{false}$

The provided patterns can also help security experts to identify similar concepts in the guidelines they formalize. For example, the first Group **Absence** gathers patterns that

have as a central principle the *absence* of a given action. This group allows to test for the absence of an action on all the paths. We can test the absence of a specific action *before* the occurrence of another action, or *after* another action, or test that the action never occurs *between* two actions.

Brian might also compose the same guideline formula using different terms that are semantically equivalent. Let us take the example of the guideline "*Store unencrypted keys away from the encrypted data*" that Brian has previously formulated using those keywords: *create_key*, *save* and *encrypt*. Amongst the keywords contained in the dictionary that we provide to Brian, the *create_key* is semantically equivalent to *isKey*, the word *save* is equivalent to *store*, and so on and so forth. Hence, the used keywords can be replaced with their equivalent as long as they do not alter the semantics of the guideline.

In order to cover this specific need, we created an *equivalence matrix* representation that contains the possible *one-to-many* semantic equivalence between the keywords that might be used to compose the guidelines formulas.

We showed in this section how to build the basic formulas from the textual description and using the patterns. However, the basic formulas do not allow to capture subtle and implicit dependencies. For example, if the user input in the guideline IDS03-J:***Do not log unsanitized user input*** was assigned to another variable, and then this variable was logged before being sanitized, the basic pattern will not capture this violation. Hence, we need to extend the basic formulas in order to capture the notion of explicit as well as implicit dependencies. This will be discussed in the next Section.

5.4.3 Security-Related Data Dependencies

Building the basic formula can be made easy to security experts and to developers as well, through the patterns that we provide and the rich dictionary for the labels and the security guidelines that can be used. Writing the MCL formulas does not require an advanced level in formal methods, or to master the idioms of the language, as we have shown in the previous section.

In the sample code below, the user password is provided as an input through the invocation of the **BufferedReader.readLine()** method:
$user.setPassword(reader.readLine())$;
Then, this password is logged through the call $logger.log(Level.INFO, logMessage)$;.

Code 5.1: Sample code for the logging of user credentials without data dependencies

```
BufferedReader reader = new BufferedReader
```

Table 5.3: Labels equivalence matrix

Label / Label	userInput	log	store	encrypt	hash	isKey	createKey	isSensitive	save	validate	invoke
userInput	X							X			
log		X	X								
store		X	X						X		
encrypt				X							
hash					X						
isKey						X	X	X			
createKey						X	X	X			
isSensitive								X			
save			X						X		
validate										X	
invoke											X

```
      (new InputStreamReader(System.in));
  System.out.println("User name : ");
  System.out.println("Password : ");

  // input
  user.setUsername(reader.readLine());
   user.setPassword(reader.readLine());

  // log
  logMessage = "user name = " + user.getUsername() +
       ", password = " + user.getPassword();
  logger.log(Level.INFO, logMessage);
```

Let us go back to the guideline MSC62-J:***Store passwords using a hash function*** [40] that we have specified in this MCL formula:

```
[true*.{setPassword ?msg:String}.(not ({hash !msg}))*. {log !msg}] false
```

The formal verification of this formula will return "false" indicating that the guideline is violated. The detected violation comes from the absence of the hashing action that should have occurred before the logging of the password data.

Now let us consider another sample code presenting a data dependency of the password that can constitute an implicit or even explicit data leakage.

Code 5.2: Sample code for the logging of user credentials with data dependencies

```
BufferedReader reader = new BufferedReader
      (new InputStreamReader(System.in));
  System.out.println("User name : ");
  System.out.println("Password : ");

  // input
  user.setUsername(reader.readLine());
   user.setPassword(reader.readLine());

  // copy password
  String x = user.getPassword();

  // hash
  MessageDigest hash = MessageDigest.getInstance("MD5");
  byte[] bytes_password = user.getPassword().getBytes("UTF-8");
  byte[] hash_password = hash.digest(bytes_password);

  // log
```

```
logMessage = "user name = " + user.getUsername() +
      ", password = " + user.getPassword() + x;
logger.log(Level.INFO, logMessage);
```

Similarly to the previous sample code, the user password is provided as an input. This password data is then hashed $byte[]hash_password = hash.digest(bytes)$. In this instruction, $bytes[]$ is the conversion of the password from String type to bytes array. Then, the hashed password is *logged logger.log(Level.INFO, logMessage)*. At first glance, we might claim that the guideline is verified, however, first the assignment of the password to the String variable x followed by its logging $logMessage = "user\ name = " + user.getUsername() + ", password = " + user.getPassword() + x$ constitutes a violation of the guideline. The point we are trying to make is that focusing only on the program behavior from actions execution is not sufficient. We also need to capture the data dependencies, whether explicit or implicit, as they can be the source of sensitive information leakage, and can hence cause serious damages to the overall security of the program and of the user's private information.

In this respect, we have deemed necessary to introduce a specific label that encodes the notion of (explicit and implicit) dependencies; *depend input_data output_data*. We mean by dependency any explicit or implicit data assignment, or any relationship that some data can have with other variables. This label should be added in the MCL formula, as it will allow to capture this notion of dependency that may occur between data, and that, from a security perspective, may leak sensitive information.

In order to capture the password leakage in the sample code above, we amend our MCL formula as follows:

```
[true∗.{setPassword ?msg:String}.true∗.(log !msg |
   {depend ?msg1:String !msg}.true∗.{log !msg1})]false
```

In this new formula, we capture the explicit as well as implicit dependencies of the password through {`depend ?msg1:String !msg`}. This formula specifies that for each possible value of `msg` (the identifier of a password), the logging action with this password as it stands without hashing, cannot be reached. Moreover, this is not possible for all the variables which depend on this password. It is important to note that we do the distinction between the security-related dependencies that we compute, and the actions that we define in our formulas. For instance, the password digest $log!msg$, which is an explicit password dependence, is computed as being the result of the hashing operation. As one can notice, we use deliberately `log` action instead of store to highlight the notion of semantically equivalent labels in our Security Knowledge Base (SKB). This specific notion is represented in a simplified form in Table 5.4.2.1.

As we aim to simplify Brian's task in formalizing the guidelines, we automated the

generation of the guidelines involving the explicit as well as implicit dependencies. This will be discussed in detail in the next section, and we will go through the construction of the MCL formulas extended by the explicit and implicit dependencies.

5.4.4 Computer-Assisted Extended Formulas

In the previous step, we showed how basic formulas can be built using the extracted key concepts from the guidelines. The first step seems very intuitive and follows the natural way of processing textual descriptions. However, the way those formulas are built does not represent the dependencies that data can have. In this section, we first show how to build the extended MCL formulas, then we explain how we can bring automation to the operation, and generate from the basic formulas, the extended ones through the new label that we have introduced: *depend input_data output_data*

Going back to the basic formulas that Brian built, we pinpoint the missing checks, and we show how we can automatically generate the formulas representing the different combinations of dependencies that may occur between data that we aim at protecting.

IDS03-J:*Do not log unsanitized user input* [30] In the basic guideline, we have specified the desired behavior, that is, user input should be sanitized before being logged. Now if the user input data was assigned to another variable, then the basic formula will not capture this dependence.

```
[true*.{userInput ?msg:String}.true*.
(not ({sanitize !msg}))*.{log !msg}
|
{depend ?msg1:String !msg}.true*.
((not ({sanitize !msg1}))*.{log !msg1})] false
```

In this guideline, we have only one data parameter which is the *userInput*. That is the reason we have only one pattern *depend*: $depend?msg1 : String!msg$

As the reader may notice, the first alternation of the formula is exactly the same as the basic formula that we have presented in Section 5.4.2. In the second alternation, we invoke the *depend* operator and again, we apply the basic formula on the $msg1$ which is the parameter that depends on msg, which is the $userInput$ parameter. It is true that through this extended formula, the explicit as well as implicit dependencies that the user input ($userInput$) might have are captured. However, if we had to verify this formula on a program presenting interleaving actions or concurrent threads, we

might eventually not detect the guideline violations. We discuss those specific issues in Sections 5.5 and 8.4.2.

MSC62-J:*Store passwords using a hash function* [40] In this guideline, we need to capture the dependencies of the password data. We discussed the details with a concrete sample code in the previous section. When writing the MCL formula, we found that the consideration of the hashing operation might be removed from the formula, because at the end of the day, we are trying to prevent the logging of the password data. In other words, we need to prohibit any explicit or implicit flow from the password in plain text, and the logging. The extended formula can be written as follows:

```
[true*.\{setPassword ?msg: String}.true*.
   ({log !msg}  | {depend ?msg1:String !msg }.true*.{log,!msg1})] false
```

Similarly to the previous guideline, we apply the basic formula in the first alternation, and we introduce the *depend* operator on the password parameter *msg* in the second alternation of the formula. We apply the basic formula on $msg1$ in the second alternation, hence, we capture the explicit as well as implicit dependencies of the password data.

OWASP: Store unencrypted keys away from the encrypted data [119] This guideline involves different data parameters: encryption key, encrypted data and storage location. Hence, we need to capture in the extended formula the different dependencies and their possible combinations. In order to represent eventual dependencies that encryption key *key* can have, we build the action pattern *depend !key ?key1:String* denoting the dependency of *key* with *key1*. Same for the encrypted data that is the outcome of the encryption with the key *key*: *encrypt ?data:String !key*. The storage location is a particular dependency, as it requires extra to capture it, as it is not necessarily represented in the program sources, but in the execution environment. The file location dependency is discussed in detail in Section 6.4.3 and in [156].

```
[true*.{create_key ?key:String}.true*.(({save !key ?loc1:String}.true*.
{encrypt ?data:String !key}.true*.{save !data ?loc2:String}.true*.
{depend !loc1 !loc2}
   |
{depend !key ?key1:String}.{save !key1 ?loc1:String}.true*.{encrypt ?data:String
!key1}.true*.{save !data ?loc2:String}.true*.{depend !loc1 !loc2})
|
({encrypt ?data:String !key}.true*.{save !key?loc1:String}.true*.
  {save !data ?loc2:String}.true*.{depend !loc1 !loc2}
   | {depend !key ?key1:String}.{encrypt ?data:String !key1}.true*.
```

```
  {save !key1 ?loc1:String}.true*.{save !data ?loc2:String}.true*.
{depend !loc1 !loc2})] false
```

Working on the formalization of this guideline brought different discussions regarding several aspects, such as the semantics of the *depend* operator that we have introduced. For instance, $depend!key?key1 : String$ reflects the notion of dependency in terms of explicit and implicit assignment of the encryption key key to $key1$. This infers the encryption key key information flow tracking. However, the $depend!loc1!loc2$ has rather an equality meaning, and this induces an evaluation of the parameter values $loc1$ and $loc2$. Hence, there is a need to consider the different types of the *depend* operator, that is highly dependent on the parameters data types. For instance, comparing $loc1$ and $loc2$ supposes that both have the same data type. We discuss related issues in Section 6.6.

We are totally aware that the consideration of the dependent information flows in the specification of the guideline might not be trivial to the security expert when formalizing the guideline. In order to cover this specific issue, and also to increase the automation of the guidelines specification, we are currently implementing a script that computes the possible combinations of the different parameters dependencies in the guideline basic formula. We strive through this methodology to increase the automation of the security guidelines specification while reducing the human intervention.

We gather the basic formulas of several guidelines in the table 5.4 below. We proceeded following the same approach that we have discussed in Section 5.4. More details on how we created those formulas can be found in [159] to appear in proceedings of the Eleventh International Symposium on Theoretical Aspects of Software Engineering TASE 2017.

5.5 Validation of the Formalization

The formal specification of the security guidelines is one crucial and fundamental operation in our framework. We stressed the fact that we do rely on the expertise of a security expert to carry out this formal specification. Nevertheless, we tried to reduce the overhead on the security expert and propose formally proven and validated patterns that he can refer to when formalizing the security guidelines. These patterns identified in [103] will allow programmers who are not experts in the MCL formal language to read and write formal specifications.

At this level, we cannot provide proofs on the correctness of the formalization as we do not have a model reference against which we can check the correctness. However,

Table 5.4: Summary of CERT and OWASP Security Guidelines Formalized in Model Checking Language

Code	Guidelines and corresponding MCL formulas
IDS01-J	*Normalize strings before validating them* [true∗.(¬({normalize ?msg:String}))∗.{validate !msg}]false
IDS03-J	*Do not log unsanitized user input* [true∗.{userInput ?msg:String}.(¬({sanitize !msg}))∗.{log !msg}]false
OWASP	*Store unencrypted keys away from the encrypted data* $[true * .create_key?key : String.true * .(save!key?loc : String.true * .$ $encrypt?data : String!key.true * .save!data!loc.true*)]false$
IDS07-J	*Sanitize untrusted data passed to the Runtime.exec() method* [true∗.{isUntrusted ?msg:String}.(¬({sanitize !msg}))∗.{invokemethod ″Runtime″ ″exec″ !msg}]false
IDS08-J	*Sanitize untrusted data included in a regular expression* [true∗.{isUntrusted ?msg:String}.(¬({sanitize !msg}))∗.{regex !msg}]false
CWE 129	*Improper Validation of Array Index* [true∗.{setPassword ?msg:String}.(¬({hash !msg}))∗.{log !msg}]false
MSC03-J	*Never hard code sensitive information* [true∗.{isSensitive ?msg:String}.(¬({obfuscate !msg})]false
MET53-J	*Ensure that the clone() method calls super.call()* [true∗.{invokemethod ″clone″}]μY.⟨true⟩true ∧ [¬{invokemethod ″super.clone″}]Y
MET56-J	*Do not use Object.equals() to compare cryptographic keys* [true∗.{isKey ?key1:String)}.true∗.{isKey ?key2:String}. {call ″Object.equals″ !key1 !key2)}|{call ″Object.equals″ !key2 !key1}]false
MSC62-J	*Store passwords using a hash function* [true∗.{setPassword ?msg:String}.(¬({hash !msg}))∗.{store !msg}]false
EXP02-J	*Do not use the Object.equals() method to compare two arrays* [true∗.{isArray ?ar1:String}.(isArray ?ar2 : String).{invokemethod″Object″ ″equals″ !ar1 !ar2}]false
OBJ10-J	*Do not use public static non final fields* [true∗.{isPublic ?data:String}.(isStatic !data).(¬{isFinal !data})]false

we can make use of the compliant and non-complaint codes provided for the different guidelines in the CERT Coding Standard [39] for instance as a means to validate the established formulas. It is true that this cannot ensure a complete detection of all the subtle details that we can infer from the guideline description, but this would help the security expert have a deeper understanding of the guideline and a wider vision on concrete sample codes. The security expert can then amend the MCL formula with respect to the new aspects that can arise when validating the formula on the positive and negative patterns that are provided as examples in the security guidelines catalogs. Nevertheless, validating the specification with a community of security experts may be a way forward to ensure the correctness of the proposed formulas.

Action order inversion

Let us now go back to the formalization we have proposed for the guideline MSC62-J: *Store passwords using a hash function* :
[`true`$*$.{`setPassword` ?`msg`:`String`}.(¬({`hash` !`msg`}))$*$.{`store` !`msg`}]`false` As the reader can notice, the order of actions in the MCL formula follows the actions flow in the textual description of the guideline (password, hash, store), and so is the case for the different guidelines that we have provided in Table 5.4. The proposed formalization can be verified over the sample code, and allows to capture the eventual violations. However, if we had inverted the order of the statements in the code in a way to first store a String variable and then to assign it to a password data as shown in the code below:

Code 5.3: Sample code for the logging of user credentials with instructions in inverted order

```
// input
BufferedReader reader = new BufferedReader(new
    InputStreamReader(System.in));
System.out.println("Password : ");
logMessage = reader.readLine();
data = logMessage;
// log
logger.log(Level.INFO, logMessage);

// password
user.setPassword(data);

MessageDigest hash = MessageDigest.getInstance("MD5");
byte[] bytes = user.getPassword().getBytes("UTF-8");
byte[] hash_password = hash.digest(bytes);

user.setPassword(hash_password.toString());
// log
logMessage = "user pwd = " + user.getPassword();
logger.log(Level.INFO, logMessage);
```

Neither the basic, nor the extended formula will capture the "*inverted*" flow presented in this code. The formal verification for the two formulas will return *true* denoting that the program does not violate the guideline, which is not the case. Fortunately, MCL is a powerful language that is very close to programming languages. MCL can handle quantifiers over data combined with value passing mechanisms. Now if we add the quantifier *forall* over the *msg* data denoting the password, our new MCL formula will be expressed as follows:

```
forall msg:String.(
```

```
    <true*.{log !msg}.true*> true
    implies
    [true*.{setPassword !msg}.true*] false
)
```

Through this new formula, we can capture all the dependencies of the password data, whether implicit, explicit or even upstream dependencies that can occur before the explicit declaration or assignment of the password data, as we showed above.

5.6 Related Work

The specification of security guidelines has also been addressed in the literature; some authors propose approaches that are formal but do not operate on the source code level. Hence, their output is not tied to the program code and cannot provide precise guidance to the developer. In their technical report [5], Aderhold et al. provide formalizations of secure coding guidelines with the objective of providing precise reference points. The authors make use of the LTL formalism to specify the guidelines; however, LTL leverages events and actions to model security policies, and puts more focus on actions rather than data. The mapping between the labels of LTL formulas and the program instructions is performed by the developer, which is an overhead to him.

The formal specification and verification of correctness (liveness and safety) properties have been widely discussed in the literature. AWS (Amazon Web Services) [110], that makes use of the TLA+ specification (and verification) language to specify the desired correctness properties, and the program to analyze in different abstraction models. TLA+ is mainly designed to specify concurrent and distributed systems, and does not operate on the code level.

The JIF [107] language implements type-checking that makes use of the Decentralized Label Model (DLM) [106]; it infers label annotations and allows to define a set of rules to be followed by programs to prevent information leakage. Labels are sets of security policies that constrain the flow of information within a program. JIF programs are type-checked at compile-time, which ensures type-safety as well as that rules are applied. The JIF language is expressive and allows static enforcement of policies. It defines a powerful access control that allows to statically check code privileges. However, the labels, which define policies for the use of data, apply only to a single data value, and are not checked at run-time. The JIF language also treats only sequential programs, and does not capture timing channels.

The System and Software Engineering Pattern Metamodel (SEPM) [74] is a framework for the specification of security (and dependability) patterns. This work adopts the model-driven engineering (MDE), and provides correctness proofs on the specification patterns.

In contrast, Schneider [135] carries out a rather dynamic approach for the verification of a class of security policies known as EM (Execution Monitoring). The author represents safety properties as *security automaton*. The automata serve as a basis to terminate the program once the security policy is violated. In the same line of work, Bauer et al. [21] proposed an extension of Schneider's security automata, and defined edit and suppress automata that enforce security policies through the modification of the security automata, hence to instrument programs. In the same line of work, Erlingsson et al.[58] define the Security Automata SFI Implementation (SASI) which is a tool to enforce security policies dynamically. Huisman and Tamalet [89] propose a translation from security automata to the annotations on the code level, and notably that serve to present pre- and post-conditions associated to the methods.

GraphMatch [151] [150],is a code analysis tool/prototype for security policy violation detection. GraphMatch considers examples of security patterns, that cover both positive and negative security properties, referring to good and bad programming practices. For instance, the authors introduce the double free negative pattern for the C programming language, and that prohibits the invocation of the *free()* method after another *free()* of the same memory location. The proposed patterns are close to the program instructions, however they consider the method calls and not the value passing or data handling, which might miss potential information leaks.

ConSpec [8] is an automata based formal language for the specification of security policies. Both the program to analyze and the security requirements are encoded in the ConSpec language, which is inspired by the work of Schneider [135]. Their approach consist in monitoring the program execution, which is a dynamic approach that can eventually miss potential paths that violate the security policies.

Dimitrova et al. [51] proposed an approach that integrates the dynamic analysis into the monitoring of information flow properties. The authors proposed an extension to LTL called *SecLTL* taking into account the information flow properties such as non-interference and declassification. *SecLTL* was then used as a specification for model checking. On top of the traditional LTL, the authors proposed the modal operator **Hide** through which information flow properties are specified. This operator aims at precising under which condition a certain secret variable can become observable. The

authors assume that secret data is provided as input, however, sensitive data can originate from other sources such as reading from a database. In addition, there might be the case where multiple sensitive data are provided as input, the monitoring of multiple sets of traces is then required, which can turn to be too expensive, and may lead to loss of precision.

Nisansala Prasanthi Yatapanage [153] makes use of behavior trees that produce formal specification out of informal requirements. In this work, slicing technique is adopted to reduce the state space as the behavior trees suffer from state explosion problem specially when it comes to modeling large programs. However, this approach is focused only on the functional requirements and does not verify the security properties.

The Aoraï plugin [142] provides the means to automatically annotate C programs with LTL formulas that translate required properties. The tool provides the proofs that the C program behavior can be described by an automaton. The mapping between states and code instructions is made based on the transition properties that keep track on the pre- and post- conditions of the methods invocation; those conditions refer to the set of authorized states respectively before and after the method call. Aoraï then generates a new C program based on the automatic annotations on the operations. If using the generated program can be validated, then it is proven that the program satisfies the property. The tool is only focused on the control dependencies between method calls, and the analysis is not extended to the data level.

LEGALEASE language [154] allows to specifiy constraints on how user data can be handled, through the clauses ACCEPT and DENY. This language is mainly meant to encode privacy policies.

5.7 Summary

The selected guidelines cover a broad spectrum of secure coding domains, which shows that formalization for such guidelines is feasible, and captures an important part of the informal security requirements. Specifications are powerful when they are re-used, and we have considered this as a basis to reflect on the usability and the extensibility of the specified security guidelines. We are aware that most people are more comfortable with natural language rather than mathematical notations. We are also aware that the specification of the guidelines in MCL language or any other formalism might require a considerable amount of time and energy to first understand the formalism and to gain a practical experience in building and composing formulas in the chosen formalism. Finding the most adequate formalism required a thorough and deep research and attempts

to adapt the formalism to our specific needs.We want to emphasize the strengths of the MCL language that is at the same time powerful and very intuitive. We presented in this chapter the flow of operations for the formal specification of the guidelines, while stressing on the main difficulties that we have encountered.

Building the formulas can be carried out in a rather natural way, as we did for the different examples of guidelines we provided. In addition, we tried to make the operation of building the formulas easier for the security expert, and even for developers who do not have to be security experts or savvy in formal methods. We tried to bring more automation to the different phases of the formalization, first to reduce the human intervention and increase the precision, and secondly, to allow the security experts to focus their attention on the guideline logic and subtle details.
Among the different guidelines that we have surveyed, a number of them involve quantitative information flow theories, and this is also another limitation of the formal specification we cover in our framework. For instance, the guideline ***Limit quantity of data encrypted with one key***[120] recommends the use a new encryption key when the amount of encrypted data goes beyond a certain threshold. It is obvious to the reader than our specification falls short in covering this guideline for the dynamic aspect it involves, and that can't be captured statically.

We strongly believe that the effort invested in formalizing guidelines will allow much deeper verifications to be carried out at the end of the day.

Chapter 6

Program Model Construction and Model Checking

Contents

6.1 Introduction

Security is a crucial property that program behavior should meet. Standard security mechanisms, such as access control or encryption, fall short in covering one fundamental problem: tracking information throughout the program and ensuring that sensitive information has not been leaked. One might argue about using run-time monitoring to fulfill this end. This practice reasons about a single execution at a time, and since information flow policies require in general monitoring all the possible execution paths, dynamic information flow policies cannot address information flow tracking. As we aim at providing clear and precise feedback to the developer, we need to operate on the code level and carry out static code analysis with the objective of verifying the guidelines. We deemed necessary to adopt static information flow analysis. Static code analysis operates on an representation of the program to analyze, which is basically an abstraction model that exploits and represents the properties of this program. We studied different program representation models in order to identify the most suitable program representation that allows to represent the information flows. Upon this survey [161], we have chosen the Program Dependence Graph (PDG) as the abstraction that we adopt in this thesis work to represent the program to analyze. PDG is suitable to fulfill our goals in terms of information flow tracking, as it represents all the feasible paths and is an over-approximation of the program behavior.

Since our main objective is to automatically verify the adherence of programs to formalized security guidelines, we need to model check the guidelines MCL formulas over the program model. However, the PDG is not formal, and do not consist a basis for the formal verification through model checking. Thus, we need to construct from the augmented PDG a model that is accepted by a model checking tool, and that can be verified automatically through model checking techniques.

Formally, the PDG is a directed graph whose nodes correspond to program statements (variable declarations, assignments, control predicates, methods invocations, etc.) and whose edges model dependencies in the program. There is one specific node that denotes the entry method (main): *entry node*. For Java programs, the *entry node* matches the main method. Nodes in the PDG are connected with several kinds of edges; control or data corresponding respectively to control and data dependencies. Both types are computes using respectively control-flow and data-flow analysis.
We deliberately use the term Program Dependence Graph PDG to denote the System Dependence Graph. PDG represent the local intraprocedural control and data dependencies in a method, and SDG is the extended version that computed the global interprocedural dependencies.

6.2 Methodology for Constructing the Labelled Transition System from the Program Sources

The starting key element for this step is the standard PDG that we generate from the Java program bytecode using the JOANA tool [72]. In this PDG, control and (explicit/implicit) data dependencies are captured, which constitutes a strong basis to perform a precise analysis. However, we tested JOANA on different sample codes presenting implicit violations, but the tool failed to capture some of them. We deemed necessary to perform deep information flow analysis on the program to capture implicit dependencies, such as Java Reflection. We carried out the effort of enhancing the PDG and augment it with the captured relevant details that reflect implicit and non trivial dependencies.

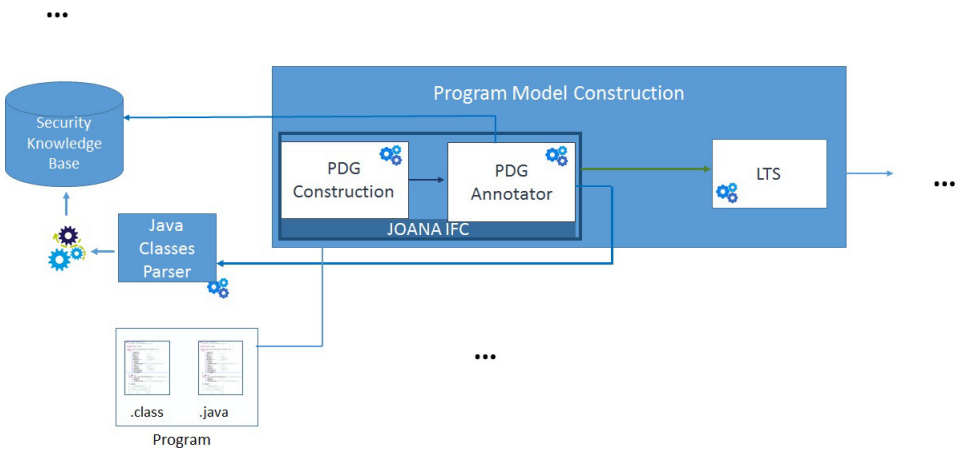


Figure 6.1: Methodology for the model construction: From program sources to the Augmented Program Dependence Graph, to the Labelled Transition System

We depict in Figure 6.1 the program model construction flow that we have adopted to generate the Labelled Transition System (LTS) from the PDG, that we generate from the program sources.

– Augmented Program Dependence Graph: this component first builds the program dependence graph (PDG) from the Java bytecode (.class) using the JOANA IFC tool [72]. We have chosen the Program Dependence Graph (PDG) as the abstraction model for its ability to represent both control and (explicit/implicit) data dependencies. The generated PDG is then annotated by the **PDG Annotator** with specific annotations (labels in the MCL formulas). The **PDG Annotator** retrieves the nodes details (method signature) from the PDG, and fetches from the Security Knowledge Base the matching label if it exists [156]. We run the

information flow analysis using the JOANA IFC, that is formally proven [72] in order to capture the explicit and the implicit dependencies that may occur between the program variables. The operation results in a new PDG that we name the **Augmented PDG**.

- LTS Construction: this component translates automatically the **Augmented PDG** into a parameterized Labelled Transition System (pLTS) that is accepted by model checking tools. The annotations on the PDG nodes are transformed into labels on the transitions in the pLTS.
- Java Classes Parser: This component that we have developed [157] takes as input the URL of the Java class official documentation [115] [114], and parses the HTML code (Javadoc) in order to extract all the relevant details: the class name, the inheritance, the description, the attributes, the constructor(s), the methods signatures, their return type and their parameters. This component populates the Security Knowledge Base with the extracted information.

6.3 Program Dependence Graph

We use the term "*Program Dependence Graph*" to denote the representation of the program as a standard PDG. We make the distinction between the PDG and the augmented PDG that we have generated from the standard one. We share the reasons that prompted us to augment the PDG and depict in details how we generated the Augmented PDG in Section 6.4.

6.3.1 Presentation

Program Dependence Graph is an intermediate representation model of programs, and it was first introduced by Kuck [95] in 1981 as a basis for optimization purposes. Different variations have been discussed ever since, depending on the intended application [60, 10, 137, 117], but they do all share the same common feature; representation of both control and data dependencies on the same graph. PDG (Program Dependence Graph) is a language-independent representation of program, taking into consideration both control and data relationships. Those two kinds of dependencies are translated into edges on the PDG. A control dependence edge a →b means that the execution of b is controlled by the evaluation of a (typically branching conditions). A data dependence edge a →b means that a variable is defined or assigned in statement a and used in statement b without being reassigned elsewhere.

PDGs are an appropriate model that represents the program's behavior according to [84], and they have the ability to represent inter- and intra- dependencies in a program.

PDGs were originally proposed for performing program slicing; this technique was originally introduced by Mark Weiser [148] [149], and serves to retrieve all the statements that influence (directly or indirectly) a certain variable in a program [117] based on iterative data flow analysis. When debugging programs, developers use slices, as they follow where a certain variable is manipulated. This technique proved its usefulness in debugging, optimizing and verifying programs [86] [149] [148] and [104].

SDG (System Dependence Graph) [138] on the other hand, is an inter-procedural dependence graph representation, and is an extension to the PDG. SDGs were first proposed by the authors of "Interprocedural Slicing Using Dependence Graph" [86] and consist now a basis to perform deep analysis [4] [149] [148] and program optimization, redundent code detection [93], as they have the strenght of representing the program semantics [77].

PDGs have the ability to precisely represent the information flow in a program, and constitute a strong basis to perform a precise information flow analysis [80]. Information flow control aims at guaranteeing confidentilaity and intgrity of critical data, and more precisely, verifying that sensitive information do not flow to or cannot be influenced by public output. PDGs have different properties that help increse the information flow analysis precision; they are *flow-sensitive*, *context-sensitive* and *object-sensitive* [80]. Being *flow-sensitive* is the ability of considering the order of statements in the program. The *context-sensitivity* is perceived from the fact that if the same method is invoked multiple times, then each call site will be represented by a separate node, and the analysis wall be carried out on each node separately. In other words, the methods calling context is considered, and this increases precision. Prior researches have proved that context- and flow- sensitivities reduce considerably the amount of false alarms [78] [82]. The *object-sensitivity*, on the other hand, consists in making the distinction between different objects of the same class. The presented sensitivities help increase the precision of the information flow analysis that will be carried out on this specific graph. However, the more precise the analysis is, the less scalable it is. PDG abstracts away irrelevant details, such as independent and non-interacting program statements, that represent the infeasible paths, and is a powerful tool that allows to perform deep information flow analysis on programs[76]. Paths in the PDG correspond to feasible information flows in the application, in other words, if there in no path from a statement a to a statement b, it is guaranteed that there is no information flow from a to b.

6.3.2 Combining Program Dependence Graph and Information Flow Analysis for Security

We made use of the JOANA IFC tool as a fundamental component that we integrated in different modules in our framework. First, we use the JOANA IFC tool to con-

struct the PDG that has the ability of representing the possible flows of information in the program. It is also able to detect the absence of probabilistic as well as possibilistic leaks for full Java. On top of the properties that PDG has, the JOANA tool contains further optimizations [70] on the PDG construction that brings enhancements on the analysis precision, thus, reducing false alarms. Those enhancements include the *exception-analysis* that consist in linking any instruction to potential exceptions. *Points-to-information* captured through *points-to analysis* that allows to check if a variable A can point to another variable B in some execution of the program. Points-to-analysis is also known as the context-insensitive analysis, and consists in assuming that the program statements can be executed in any order, not necessarily following the control flow of the program [128]. The JOANA IFC tool extends further the object-sensitivity, and adds the object *field-sensitivity* [68]. It consists in extending the analysis to the attributes level for Object Oriented Programs; object, which is an instance of a class, is not considered as an atomic entity, and the accessible fields are modeled as an object graph. JOANA IFC is also timing sensitive. This means that only PDG paths can indeed be realized by the scheduler. The implemented algorithms increase the analysis precision while reducing the time and memory-consumption up-to 90% as discussed in [68].

SDG construction takes into account the additional parameter nodes that arise from method side-effects. Once the new nodes are constructed (*formal-out* and *formal-in*), JOANA IFC computes data dependencies for those new nodes. Then, are computed the summary edges [129] [87]. Summary-edges represent the transitive dependencies between the output parameter node (*formal-out*) and the input parameter node (*formal-in*). We provide a concrete example on the introduction of this concept in the next Section.

6.3.3 Example of a Program Dependence Graph

In this section, we go through few examples illustrating the construction of PDG. We consider in the first example a simple code presenting direct and indirect information leakage. The second example is a bit more complex as it contains embedded implicit data dependencies. The two sample codes are provided with an explanation on their logics together with their PDGs.

In the first sample code [69], there is a secret variable *secret* that we want to protect and insure its confidentiality and integrity. The instruction *print(secret)* presents an explicit leak as it prints directly the value of *secret*. In the branching condition, we evaluate if *secret* is even. If the condition is evaluated to true, the message *"secret is even"* is

displayed. One might say this print message does not divulge anything about secret except that it is even, but this constitutes an implicit leakage. If the print statement is executed, we can then infer that *secret* is even. The last print statement *"Hello World"* does not reveal anything about *secret*.

Code 6.1: Sample code illustrating implicit and explicit dependencies

```
public static void main(String[] args) {
   int secret = input();
   print(secret);
   if (secret % 2 == 0) {
      print("secret is even");
   }
   print("Hello World");
}
```

The PDG representation of this sample code is illustrated in Figure 6.2. The statements are represented in separate nodes, and we can distinguish the entry node matching the *main()* method (node 1). Control dependencies are represented as dashed edges, while data dependencies are represented as solid edges. Control dependencies reflect the notion of direct *dominance* [9] in *Graph Theory*. A node n is said to post-dominate a node m, if the execution of n is controlled by the execution of m. In the sample code, the invocation of main controls the execution of the statements: *int secret = input()*, *print (secret)*, *if (secret % 2 == 0)* and *print("Hello World")*. Note that there is no control flow between *main* and *print("secret is even")* as the execution of this statement depends on the evaluation in the if clause (*if (secret % 2 == 0)*). And as we previously mentioned, control dependencies reflect the direct dominance relationships between statements in the program.

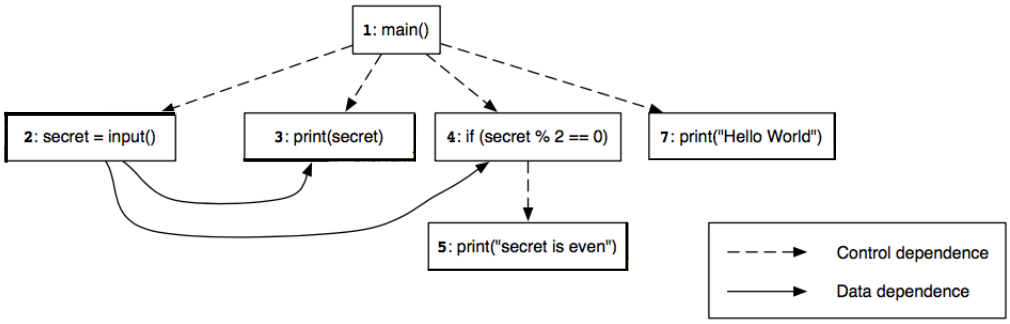


Figure 6.2: Program Dependence Graph of the sample code 6.1

The second sample code, taken from [68] is more advanced and covers different aspects that we highlighted in the previous section. Those aspects include the context and

field sensitivities, as well as the notion of additional parameter nodes (*formal-in* and *formal-out*):

Code 6.2: Sample code illustrating the interprocedural dependencies computation

```
1   class B {
2    A a;

3    public B() {
4     a = new A();
5    }

6    static void modify(B b) {
7     b.a.i = 42;
8    }

9    static int indirectMod(B x, B y) {
10      modify(x);
11      return y.a.i;
12   }

13   public static void main(String argv[]) {
14      indirectMod(new B(), new B());
15   }
  }
```

Figure 6.2 illustrates the PDG of the sample code 6.2. As the reader can notice, the entry node holds the label *indirectMod*: it refers to the method that is invoked in *main*. The method *indirectMod* takes as input parameters two objects of Class B: x and y. In the implementation of this method, we notice the invocation of another method named *modify*. This method changes the attribute value i of the object b passed as parameter. Hence, the method *indirectMod* indirectly changes the attribute value of the object x, and return the same attribute's value of object y. At a first glance, the two instructions *modify(x)* and *return y.a.i* seem independent, as they operate on two different objects. The advantage of applying points-to-analysis is to precisely distinguish the two objects parameters passed to the *indirectMod* method, and also distinguish their attributes a. The method *modify* changes the attribute value *x.a.i*. It needs first to read the object attribute a of x, then retrieves i and changes its value. Applying a precise points-to-analysis will create two input parameters for the method *modify* : the object b (node 20), and its attribute a (node 21). Note that nodes 20 and 21 are tagged as *in* (formal-in) indicating that they are input parameter nodes. The output parameter nodes, in the other hand, track not only modified fields but also all the fields representing the followed

access path to access the field i of the attribute a (node 24). The nodes on this path arise from the method *modify* side-effects. The actual input and output parameters (b and i) are represented respectively on the nodes 15 and 19. In the method *indirectMod*, *modify* is invoked with the input parameter x. Similarly to the previously described nodes computations principle, the fields of the input parameter of *modify* are connected to the fields of x used in *modify* invocation. For method *indirectMod*, formal-in nodes are 6 and 7. Formal-out nodes are 12, 13 and 14. Object x's side effects are then propagated to the main method. If an imprecise-points-to analysis was applied, we would see dependencies between the *return y.a.i* statement (nodes 3) and the input parameter x (node 6), as this analysis cannot distinguish the a attributes of different objects, but considers them as the same attribute.

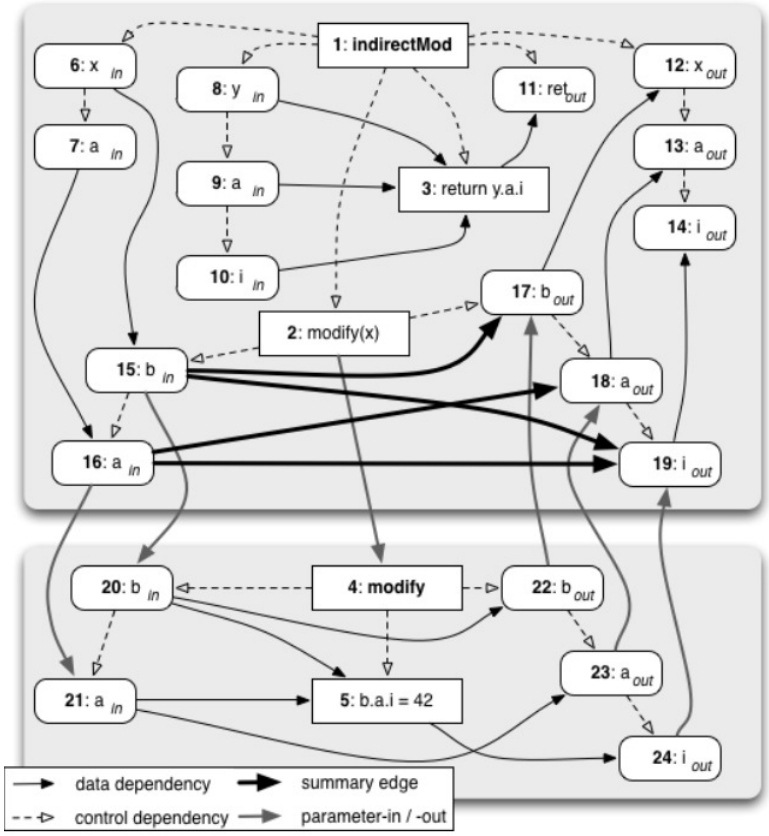


Figure 6.3: Program Dependence Graph of the sample code 6.2

6.3.4 Explicit Information Flow Analysis

PDG structure constitutes a strong basis to perform deep information flow analysis. The main property against which the program is analyzed in the noninterference. Informally, noninterference states that public output (low) should not be influenced by

variation of secret data (high). This property is widely known as the *Low Deterministic Security* [66] [26] [140]. As the reader may have noticed, noninterference supposes first a classification of information: *HIGH* for secret or confidential data, and *LOW* for public data. *HIGH* and *LOW* denote the security or the confidentiality level of data. Now let us go back to the sample code of Figure 6.1. We have previously represented the corresponding PDG that we use again in Figure 6.4. Note that this PDG is exactly the same as the previous one, except it is annotated with the *high* and *low* annotations. The node representing the *secret* data is annotated *high*, and the three *print* instructions are annotated *low*. In the information flow vocabulary, an information flows from a *source* and reaches a *sink*. Having this in mind, the annotations on this PDG would be secret: *high source*, and the three prints *low sink*. Noninterference property states that high level data should not interfere with low level output, otherwise, illegal flow is detected. Now let us run the JOANA IFC information flow analysis on this sample code after applying the annotations. Before running the analysis, the developer needs to annotate the PDG with the annotations that we have specified. However, identifying secret data and public output requires a deep knowledge on the program logics. Even if the developer is able to classify the program instructions, the manual annotations can be an overhead, and the complexity of this operation increases with the size of the program and its complexity. We deemed necessary to bring automation to the annotations application on the PDG in order to leverage the workload on the developer or the code reviewer (see Section 7.2).

Since the analysis operates on a single couple (source, sink) at a time, we need to consider the three possible combinations:

- (secret, print(secret)): 2 security violations detected: illegal paths referring to the control and the data dependency edges
- (secret, print("secret is even")): 3 security violations detected: illegal paths referring to the dependency edges and the method call side-effects
- (secret, print("Hello World")): 12 security violations detected: illegal paths referring to the transitive dependency edges and the method call side-effects

As the reader can notice, the PDG structure is an over-approximation of the program behavior, and this explains the amount of false positives that we have obtained when running the analysis by the JOANA IFC tool. The first *print* consists an explicit information leak, as *secret* was printed in plain text. It is true that the second print does not make public the *secret* data, but it is not executed unless *secret* is even. From an attacker perspective, observing the execution of the second *print* reduces the exploration space, and increases the attack risk. The third *print*, however, is not an information leak, but still, we obtained 12 security violations. This infers the need of

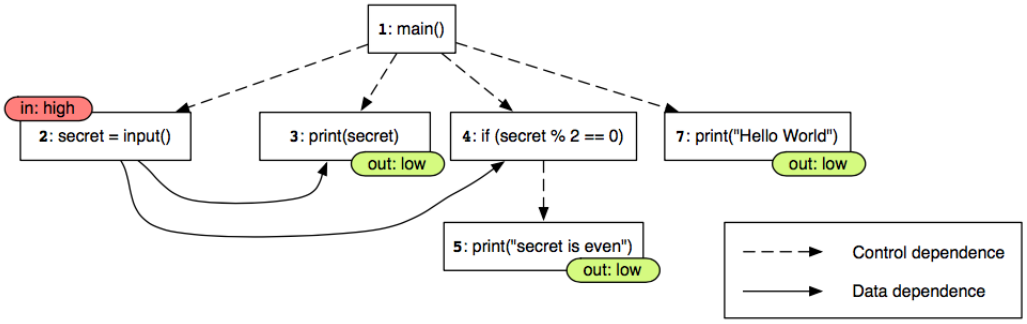


Figure 6.4: Annotated Program Dependence Graph of the sample code 6.1[69]

capturing dependencies to increase the analysis precision and reduce the amount of false alarms. This will be discussed in detail in the next Section.

6.4 Introducing New Dependencies on the Program Dependence Graph

We presented in the previous section the basic concepts related to the Program Dependence Graph structure, and showed how PDG is built from the program source. Now that we have presented the basic graph structure in our framework, let's explain how it is used for analysis purposes. As we have mentioned in the Chapter 4, we rely in the strength of the PDG structure first to represent all the information flows in the program, and to check for the noninterference property. [141] proved a theorem relating PDG with classical non-interference.

Before going through the details on the new dependencies that we have defined and captured on the PDG, let's analyze few sample codes presenting implicit and explicit violations of security guidelines.

Code 6.3: Sample code violating the security guidelines MSC62-J [40]

```
// input
BufferedReader reader = new BufferedReader
                (new InputStreamReader(System.in));
System.out.println("User name : ");
System.out.println("Password : ");
user.setUsername(reader.readLine());
user.setPassword(reader.readLine());
// log
logMessage = "user name = " + user.getPassword();
logger.log(Level.INFO, logMessage);
```

6.4.1 Customized Annotations

The JOANA tool proposes two kinds of annotations specifying the source (*SOURCE*) and the target (*SINK*) in addition to the *DECLASS* annotation. When annotating the respective source and sink on the PDG, the user (or developer) needs also to provide the security level: *High* or *Low*. High means that the annotated variable is confidential, and should be kept secret. Low, on the other hand, means that the variable or the statement can be public. The annotation declassification (*DECLASS*) allows to reduce the security level of the annotated node. Declassification is defined as the controlled release of information.

We made modifications on the source code of the JOANA tool, and added customized annotations referring to the abstract labels upon which the guidelines MCL formulas are built. The labels are translated into annotations, such as *hash*, *userInput*, *Password*, *encrypt*, *save*, etc. in addition to the predefined annotations *SOURCE* and *SINK*.

As a second step, the automatic detection of the labels on the PDG is performed. The component **PDG Annotator** (see Section 7.4) and here we refer to the Security Knowledge Base that already contains the concrete possible mappings between known APIs, methods, methods parameters mapped to the abstract labels of the security guidelines specification. We provide in Appendix D sample data from the Security Knowledge Base.

For a developer or a tester who is not aware about the semantics of methods performing security operations, detecting possible sources (resp. sinks) and their respective sinks (resp. sources) appears tedious.

6.4.2 Encryption Key Dependence

The declassification [133] operation is the controlled release of sensitive information using specific mechanisms and techniques such as encryption, hashing, obfuscation, etc. We mean by parameterized declassification, the decrease of a data's security level under conditions on another data, such is the case for encryption using an encryption key: declassification holds if the encryption key remains secret, otherwise, the encrypted data should be kept secure, and its security level remains *HIGH*. We refer to the first case as "safe" release. Sabelfeld et al. [132] discuss the different dimensions for the declassification. We introduced the annotation *DECLASS(param)* that refers to the declassification with respect to the parameter *param*: if *param* and all its dependencies are kept secret, then the declassification operation is valid, and the security level of the secret data to encrypt is declassified from *HIGH* to *LOW*, otherwise, the declassification is obsolete and the security level remains *HIGH*. It is important to note the correlation

between encrypted data and the encryption key; this relationship is translated into a dependence in the generated Program Dependence Graph.

6.4.3 File Location Dependency Detection

We have introduced this new dependencie that consists an example from a bigger set: storage location dependencie. In this section, we will go through a sample code that brought the idea of considering this dependencie. We might be faced with situations where high level and low level data share the same resource, and data can indirectly flow from high to low. This is known as storage covert channels [145].

Below we provide a sample code that presents an implicit violation of the guideline from the OWASP Cryptographic Storage Cheat Sheet [119]:"Store unencrypted keys away from the encrypted data" [1] explaining the encountered risks when the encryption key is stored in the same location as the encrypted data. If we want to verify if the code below meets this guideline or not, then we have first to annotate the sources and the sinks, and run the analysis. We need first to highlight several elements in the codes below; the data key k and *encrypted_cc* are stored respectively in file **keys.txt** and **encrypted_cards.txt**. One may conclude that the guideline is met, as key k and *encrypted_cc* are stored in separate files. However, the two files are located in the same file system, which constitutes a violation of the guideline. On the generated PDG, we annotate the node corresponding to the credit card number attribute as a high level source, and the node representing the invocation of the method "save_to_file" (save_to_file(data,"C://Users//XXXX//src//secGuidelines//keys.txt");) as low level sink. We launch the JOANA IFC analysis and no violations were detected, which is a false negative, meaning that there is a violation that should have been detected. The reason why JOANA IFC missed the violation is that it did not compute and represent the edge between the node credit card number and the method call "save_to_file" having as parameter the encryption key (save_to_file(k.toString(),"C://Users//XXXX//
src//secGuidelines//keys.txt");). A very important aspect that clearly needs to be considered, is when there are dependent information flows that should be treated with due consideration. *Dependent* means that the analysis to be performed should take into account the relationship between the two storage locations (encryption key and encrypted data in this example). As we stressed above, file location storage is a subset from the bigger set storage location. The notion of location that can be expressed in different manners such as file location, insert in database, add to an array, etc. We worked towards closing this gap through the arrangement of labels into classes, and

[1] https://www.owasp.org/index.php/Cryptographic_Storage
_Cheat_Sheet#Rule___Store_unencrypted
_keys_away_from_the_encrypted_data

have the notion of synonyms or words that express the same action (Labels equivalence matrix presented in Section 5.4.2.1) . For example, the set **save** contains labels such as *save_to_file*, *save_to_database*, *save_to_array*, etc.

Code 6.4: Unencrypted encryption key stored in the same file system as the encrypted data

```
106  public static void main(String[] args)
107        throws NoSuchAlgorithmException,
108        NoSuchProviderException,
109        FileNotFoundException {
110     int c = 123456;
111     Payment p = new Payment();
112     p.setCreditCardNumber(c);
113
114     String x = "0xe04fd020ea3a6910a2d808002b30309d";
115     byte[] y = hexStringToByteArray(x);
116     SecretKeySpec k = new SecretKeySpec(y, "AES");
117
118         // save
119
    save_to_file(k.toString(),"C:///Users//XXXX//src//secGuidelines//keys.txt");
120
121
122     // encrypted data
123     byte[] encrypted_cc = encrypt(k,
    Integer.toString(p.getCreditCardNumber()));
124
125
126     // save
127     String data = encrypted_cc.toString();
128
    save_to_file(data,"C:///Users//XXXX//src//secGuidelines//encrypted_cards.txt");
   }
```

Code 6.5: The method *save_to_file*

```
146  public static void save_to_file(String data, String file) {
147     try (PrintWriter out = new PrintWriter(
148           file)) {
149        out.print(data + GregorianCalendar.getInstance().getTime() +"\r\n");
150     } catch (FileNotFoundException e) {
151        e.printStackTrace();
152        System.out.println("file error");
153     }
154  }
```

In Figure 6.5, we represent the PDG of sample code 10.2. The *SecretKeySpec* object *k* is instantiated in the instruction *SecretKeySpec k = new SecretKeySpec(y, "AES")*. The constructor *SecretKeySpec(byte[],String)* stands out in our Security Knowledge Base as a method invocation mapped to the abstract label *create_key*. (See Appendix D, Section D.3). Hence, the variable *k* is annotated ***create_key***. The methods *encrypt* and *save_to_file* are implemented by the developer. Hence, the automatic annotations on those methods will fail, as this operation requires an advanced semantic knowledge base and a semantic analysis to be performed over the code in order to determine the method names matching encryption operation or storage. The semantic analysis is not in the scope of this thesis. We can also be faced with the case where the methods are declared with insignificant names, which makes the automatic annotations unfeasible. One might think that the method save_to_file 10.3 would be mapped to the label save_to_file , as from a semantic point of view, this is the meaning of this method name. But this is not the case. The point we are trying to make, is that we look at the Java programming language native instructions, and not at the method names implemented by the developer, as this can be misleading and not reliable. As the reader can see in the Appendix D, the Java methods are mapped to the abstract labels, that are translated into annotations on the PDG once the PDG is constructed. Now if we look at the implementation of the save_to_file method (Code 10.3), we would see that the developer invoked the Java instruction *PrintWriter.print(String)*. In our Security Knowledge Base, this method signature is mapped to the label save_to_file.
We performed advanced information flow analysis with the objective of capturing the implicit file location dependency; we captured the parameters (file names) of the specific method *PrintWriter.print(String)* invocations, and we compared their values. This comparison indicated that the two files are in the same file system. This results in the creation of a new edge of kind **DEPEND** between the nodes matching the invocations of *PrintWriter.print(String)*. The new edge is represented as red dashed edge on the augmented PDG in 6.5.

Strong edges represent the control flows, the dashed edges refer to explicit and implicit data flows. Nodes are labeled with their corresponding instructions line numbers for readability purposes.

6.4.4 Multiple Annotations on the Same Node

JOANA tool offers the possibility to annotate a node with a limited set of annotations: SOURCE, SINK or DECLASS. We added the possibility of having multiple annotations on the same node; this will appear to be useful in different cases where for example the

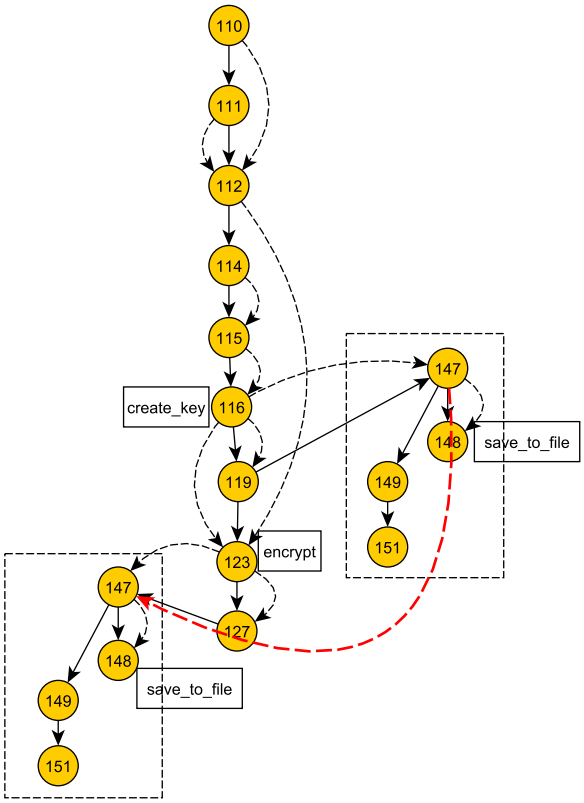


Figure 6.5: Augmented Program Dependence for the sample code 10.2

same data (the same node) is at the heart of more than one guideline.

6.4.5 Annotation Propagation

We augmented the PDG using the propagation of annotations when already annotated data are copied or concatenated. For example, if key k (annotated as *create_key*) was assigned to another variable g, then g will also be annotated as *create_key*. This will enable to provide precise feedback on data propagation to the developer, and to extend the analysis of guidelines on dependent data.

6.5 Generation of the Labelled Transition System

6.5.1 Parameterized Labelled Transition System

A parameterized Labelled Transition System (pLTS) is a labelled transition system with variables; a pLTS can have guards and assignment of variables on transitions. Variables can be manipulated, defined, or accessed in states, actions, guards, and assignments. JML [99], Z [126], B [98] allow to describe the states of the system through mathematics-based objects (machines, sets, etc.), and they describe pre- and post-conditions on the transitions between the states. Those languages deal with sequential programs and do not handle value passing for most.

An LTS is a structure consisting of states with transitions, labeled with actions between them. The states model the program states; the transitions encode the actions that a program can perform at a given state. We distinguish two types of actions: actions encoding sequential program (representing standard sequential instructions, including branching and assignment) and a call to the method (local or remote), and actions encoding the result of tracking of explicit and implicit dependencies between variables within program.

Definition 1 (pLTS)**.** *A parametrised LTS is a tuple* $pLTS \triangleq (S, s_0, L, \rightarrow)$ *where:*

- S *is a set of states.*
- $s_0 \in S$ *is the initial state.*
- L *is the set of labels of the form* $\langle \alpha,\ e_b,\ (x_j := e_j)^{j \in J} \rangle$*, where* α *is a parametrised action,* e_b *is a guard, and the variables* x_j *are assigned the expressions* e_j*. Variables in are assigned by the action, other variables can be assigned by the additional assignments.*
- $\rightarrow \subseteq S \times L \times S$ *is the transition relation.*

Informally, we interpret the behavior of a program as a set of reachable states and actions (instructions) that trigger a change of state. The states express the possible values of the program counter, they indicate whether a state is an entry point of a method (initial state), a sequence state (representing standard sequential instruction, including branching), a call to another method, a reply point to a method call, or a state which is of the method terminates. Each transition describes the execution of a given instruction, so the labels represent the instruction names.

The LTS labels can mainly be of three types: actions, data and dependencies.

- Actions: they refer mainly to all program instructions, representing standard sequential instructions, including branching and method invocations.
- Value passing: as performed analysis involves data, generated LTSs are parameterized, i.e, transitions are labeled by actions containing data values.
- Dependencies: in addition to program instructions, we added transitions that bring (implicit and explicit) data dependencies between two statements with the objective of tracking data flows. Indeed, transitions on LTS show the dependencies between the variables in the code. We label this kind of transition by *depend var1 var2* where *var1* and *var2* are two dependent variables.

6.5.2 From the Augmented PDG to the Parametrised Labelled Transition System

Since the Augmented PDG is not a formal model, we cannot model check it. We need to generate a formal model that is accepted by model checking tools from the generated Augmented PDG .

During the PDG construction, we adopt a known technique used in taint analysis, and consisting in renaming [28] the program variables. We rename each definition of a variable x to a different name and rename every use of x by the new name, to ensure that operations carried out on this specific data keep the same variable name.

We construct the pLTS of a program from its intermediate representation, the Augmented Program Dependence Graph (PDG) structure which constitutes an over-approximation of the program information flow in the program behavior. We generated the PDG using the JOANA IFC tool that has the ability to track both explicit and implicit information flows in a program.

We present in Figure 6.6 the Augmented PDG of the sample code 10.2, and in Figure 6.7 the pLTS that we have generated from this Augmented PDG. It is important to note that we make sure that the augmented PDG and the pLTS are isomorphic, in the sense that both graphs have the same number of nodes. The pLTS is similar to the augmented

PDG, except that the pLTS labels are on edges (transitions) and not on nodes. The same actions (instructions) on the PDG nodes are translated into transitions on the pLTS, which is a faithful representation of the captured dependencies translated into edges between nodes. This property is of paramount importance due to the capability it offers with respect to the faithfulness of the analysis support, and also to export the analysis results (violating traces) back to the PDG built from the source code.

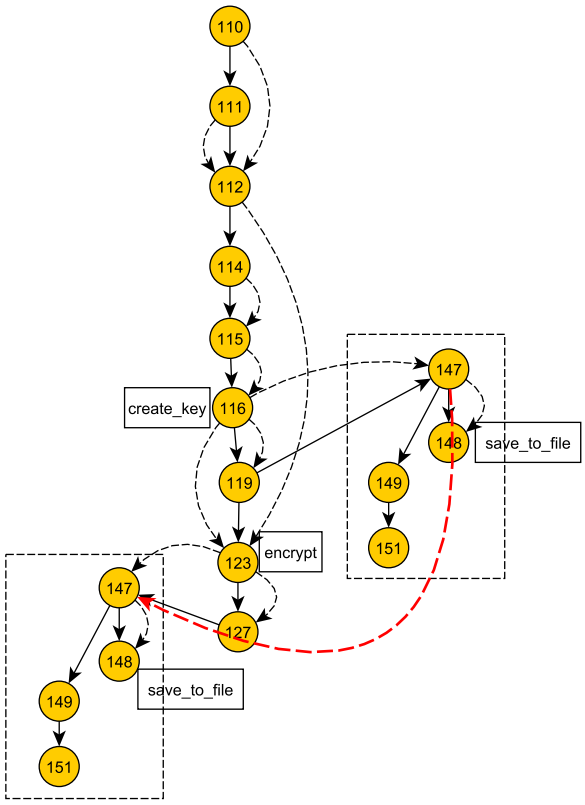


Figure 6.6: Augmented Program Dependence for the sample code 10.2

The format that we have used for the pLTS representation is the *Binary Coded Graph* (BCG). BCG is a portable file format that allows to store and represent different structures, such as Kripke structures, labelled transition systems, etc. We have chosen this specific format for different reasons. First, it is the format that is accepted by the EVALUATOR tool of the CADP toolbox that we made use of for the formal verification part, and this is discussed in details in D.1. Second, the BCG format can handle large state spaces (up to 10^{13} states and transitions). This file format is language-independent, which means that it has the ability to describe graphs that are generated from various programming languages. In addition, the representation of graphs with a huge number of states and transitions requires only few megabytes of memory. BCG is a programming language independent file format, and captures the objects that are defined in the

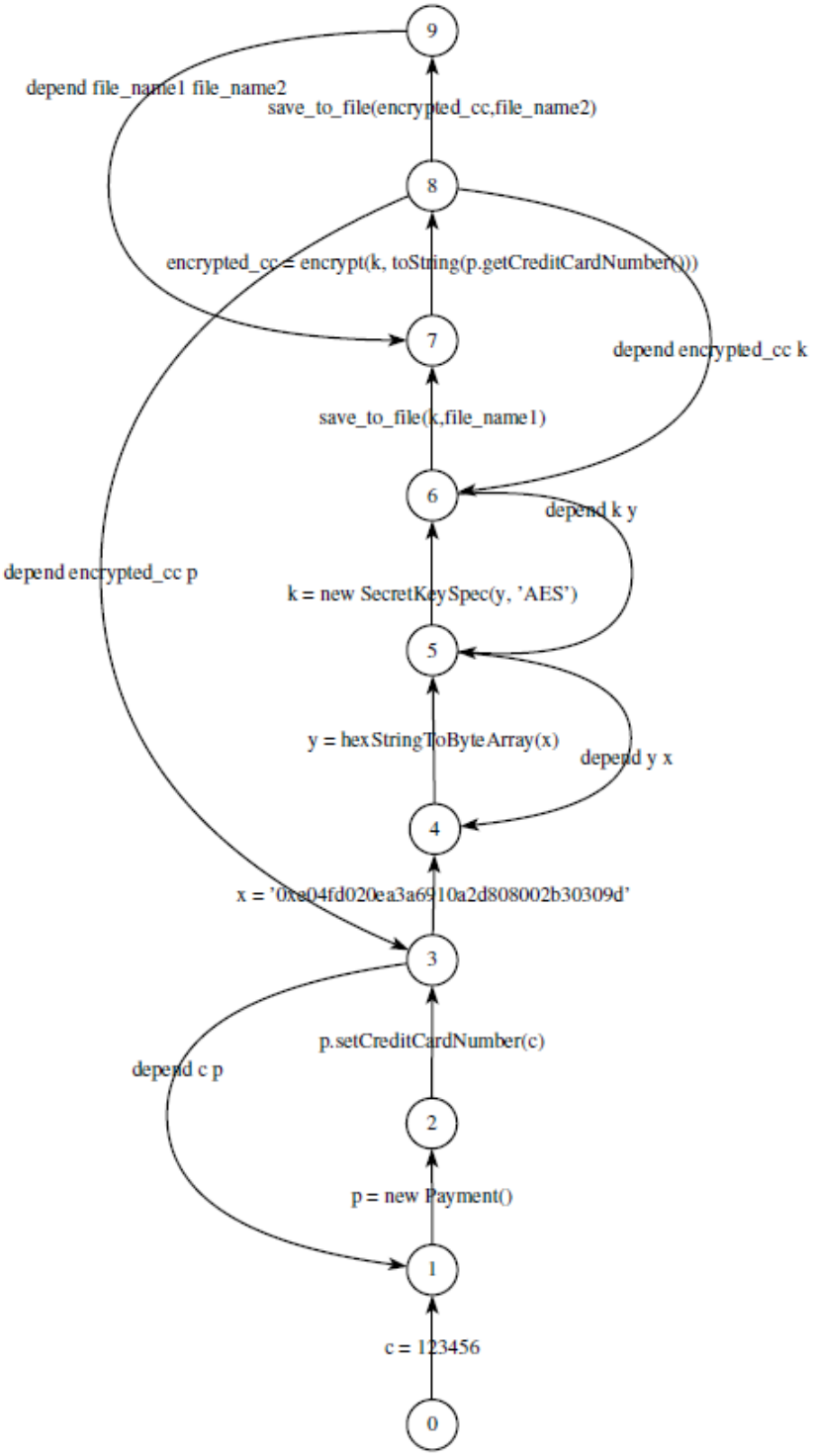


Figure 6.7: pLTS generated from the PDG of Figure 6.6

program sources (functions, variables, data types).

6.6 Model Checking

Model checking [73] is an automatic technique for verifying behavioral properties of a system model by an exhaustive enumerating of its states. The advantage of applying model checking techniques is that it is automatic, and does need the intervention of the human operator to guide the verification. In the previous section, we explained how we automatically generate the pLTS from the augmented PDG. Now that we have set the scene, and prepare the ground for the formal verification: we have on the one hand the MCL formula of the guideline, and on the other hand, the pLTS.

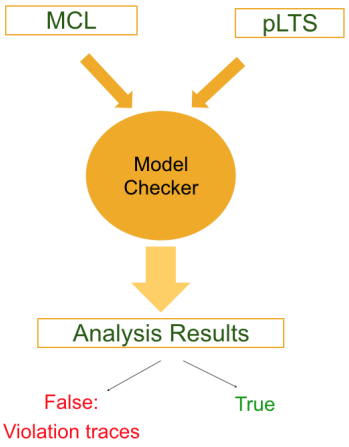


Figure 6.8: Model Checking

We can carry out the model-checking analysis directly on the pLTS, that we can simplify and reduce further. *hiding* and *renaming* are used to compute a minimized Labeled Transition System, which is an "operable" model (see Figure 6.6). First, some irrelevant actions (for the analyzed properties) are "hidden"; they are replaced by τ actions (denoted i in Figure 6.6). Second, we rename the actions by their synonyms (entry points) in the Security Knowledge Base.

We made use of the checker EVALUATOR of the CADP toolsuite [97] to verify the property *OWASP: Store unencrypted keys away from the encrypted data* that we have formalized in MCL as follows:

```
[true*.{create_key ?key:String}.true*.(({save !key ?loc1:String}.true*.
{encrypt ?data:String !key}.true*.{save !data ?loc2:String}.true*.
{depend !loc1 !loc2}
   |
```

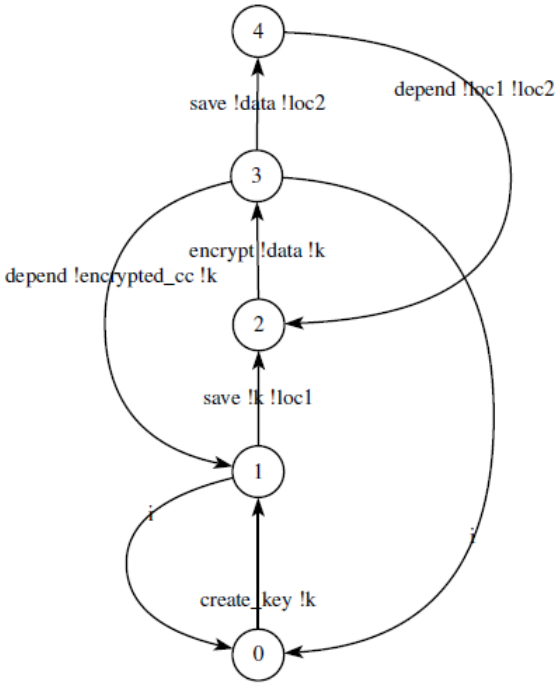


Figure 6.9: Minimized Labeled Transition System for the sample code given in 10.2

```
{depend !key ?key1:String}.{save !key1 ?loc1:String}.true*.{encrypt ?data:String
!key1}.true*.{save !data ?loc2:String}.true*.{depend !loc1 !loc2})
|
({encrypt ?data:String !key}.true*.{save !key?loc1:String}.true*.
  {save !data ?loc2:String}.true*.{depend !loc1 !loc2}
   | {depend !key ?key1:String}.{encrypt ?data:String !key1}.true*.
  {save !key1 ?loc1:String}.true*.{save !data ?loc2:String}.true*.
{depend !loc1 !loc2}))] false
```

The verification of this property on the pLTS of Figure result is false, indicating that the guideline is violated due to the implicit file location dependency indicating that the two file locations are in the same file system. In addition to a false, the model checker produces a trace illustrating the violation from the initial state, as shown in Figure 6.6.

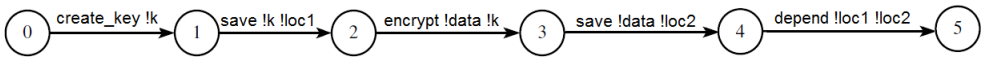


Figure 6.10: Violation trace

6.7 Related Work

Prior work in the area of information-flow security [131] has been developed during the last decades. A line of work [61] [42] adopts the Extended Static Checking, a specific technique for finding source code errors at compile-time. Eau Claire[42] framework operates as follows; it translates C program into Gurad Commands (Guarded Command Language), that are afterwards translated into verification conditions for each function of the program. The generated verification conditions serve as input to automatic theorem prover. Adopting the prototype Eau Claire is very much-time consuming, requiring annotations entered by the developer, hence it is hard to integrate in the development phase.

De Francesco et al. [49] combine abstract interpretation and model checking to check secure information flow in concurrent systems. The authors make use of the abstract interpretation to build a finite representation of the program behavior: a labelled transition system .The security properties are specified in temporal logics, and are model checked over the built LTS. Their approach consists in verifying the non-interference property, meaning that the initial values of high level (secret) variables do not influence the final values of low level (public) variables. This approach checks for the non-interference only on two program states, which might miss possible information leakage within the process itself. In addition, the adopted formalism does not support value passing, and does not reason about data propagation.

The Verification Support Environment [18] is a tool for the formal specification and verification of complex systems. The approach adopted by the authors is similar to model-driven engineering, in the sense that the formal specification results in code generation from the model.

SecureDIS [7] makes use of model checking together with theorem-proving to verify and generate the proofs. The authors adopt the Event-B method, an extension of the B-Method, to specify the system and the security policies. The authors do not make clear how the policies parameters are mapped to the system assets, and they do not extend the policy verification and enforcement on the program level. The work targets one specific system type (Data Integration System), and is more focused on access control enforcement policies, specifying the subject, the permissions and the object of the policy. However, access control mechanisms are not sufficient for the confidentiality property, as they can't provide assurance about where and how the data will propagate, where it will be stored, or where it will be sent or processed. The authors target system designers rather than developers or testers, and consider a specific category of policies

focused on data leakage only.

GraphMatch [151] [150],is a code analysis tool/prototype for security policy violation detection. GraphMatch considers examples of security properties covering both positive and negative ones, that meet good and bad programming practices. GraphMatch is more focused on control-flow security properties and mainly on the order and sequence of instructions, based on the mapping with security patterns. However, it doesn't seem to consider implicit information flows that can be the source of back-doors and secret variables leakage.

PIDGIN [11] introduces an approach similar to our work. The authors propose the use of PDGs to help developers verify security guidelines throughout the exploration of information flows in their developed software and also the specification and verification of adherence to those policies. Privacy policies are encoded in LEGALEASE language that allows to specifiy constraints on how user data can be handled, through the clauses ACCEPT and DENY [154]. The specification and verification of security properties rely on a custom PDG query language that serves to express the policies and to explore the PDG and verify satisfiability of the policies. The parameters of the queries are labels of PDG, which supposes that the developer is fully aware of the complex structure of PDGs, identify the sensitive information and the possible sinks they might leak to. For example, the authors propose a policy specifying that the guessing game program should not choose a random value that is deliberately different from the user's guess provided as input. This policy is expressed in the following query:
let input = pgm.returnsOf("'getInput"') in
let secret = pgm.returnsOf("'getRandom"') in
pgm.forwardSlice(input) ∩ pgm.backwardSlice(secret)

PIDGIN limits the verification to the paths between sinks and sources, however, there might be information leakage that occurs outside this limited search graph. The authors do not provide the proof that their specification is formally valid. In addition, it is not explained how the feedback will be presented to the developer, or how we might be guided through the correction phase.

6.8 Summary

In this chapter, we presented the methodology that we have adopted to construct the (parametric) Labelled Transition System (pLTS) from the augmented Program Dependence Graph. PDG construction follows a rather conservative approach, and presents an over-approximation of the program behavior, which can lead to false alarms. We

discussed the efforts that we have carried out in order to capture implicit and subtle dependencies between the program statements and variables through the information flow analysis that capture those implicit dependencies. We showed how the new dependencies that we have introduced and added on the PDG allowed to optimize the analysis and increase the precision. We stressed on the very importance of the augmented PDG as intermediate representation used to generate the pLTS. Someone might argue about generating the Labelled Transition System straight from the program source code (similarly to Alvis [94]). However, adopting a similar approach wouldn't capture the direct and transitive dependencies, and will definitely miss the implicit dependencies that information flow analysis captures and augment the PDG with. We used static code analysis together with model checking for the automatic verification of security guidelines. The combination of those two different approaches, which constitutes one major innovation of this work, led to the enhancement of the analysis precision.

Chapter 7

Prototyping

Contents

7.1 Introduction

In the previous chapters, we presented the approach that we propose to fill the gap in the automatic verification of the security guidelines on the code level. We shared the methodology for the specification of the security guidelines, as well as the program model construction that constitutes the basis to lead the automatic formal verification through model checking. In this chapter, we will present succinctly the different tools that we have used in order to compose the full tool-chain. We will also explain how we used static code analysis together with model checking. The combination of those different approaches for the automatic verification of security guidelines consists one major innovation of this work. The reader might probably have noticed that the different sample codes that we have presented in the form of PDG and that we have analyzed are all in Java programming language. The choice of Java was done in purpose, as we

have the intention of extending this work to cover Android applications, and validate our approach in software marketplaces (Google Play for instance). We have considered Java for many other reasons, such as being a safe programming language. The Java security architecture [67] ensures the users and the infrastructure from external programs, but falls short in protecting against flaws that occur in trusted code. Those flaws can be source of covert channels, and can infer serious damages on the system and programs assets. One other reason why we have chosen Java is for its robustness and strong data typing, and the Java Virtual Machine [100] provide different mechanisms such as automatic memory management and arrays bound checking that can mitigate bugs. On top of that, Java provides *access modifiers* (public, private and protected) that define the accessibility level of object attributes, methods and variables. Java is claimed to be a safe and secure programming language, however, it is not at secure as we can imagine. We have implemented several sample codes where we could bypass the strict memory management (see Section 8.4.4) and leaked information through threads synchronization (see Section 8.4.2).

7.2 Architecture

We describe in this section the architecture of our framework in terms of implemented components and the tools that we have used in order to accomplish the needed functions.

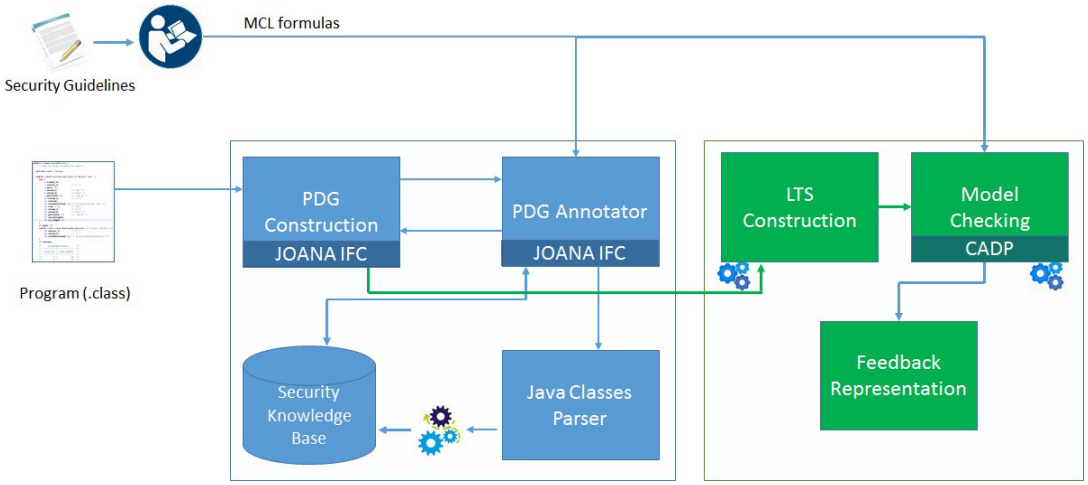


Figure 7.1: Prototype for the end-to-end verification of security guidelines

The different components of our framework are automatic, except the first step that is carried out by a human operator: the security expert. First, the security expert transforms the informal security guidelines from natural language into exploitable formulas expressed in the Model Checking Language. As the reader can notice, this operation is manual, and requires security expertise to extract the key-concepts from the textual

description of the guidelines, and then to build the formulas. Automating this operation would be an achievement, as it will help automating the end-to-end process, and bridge the gap between the informal guidelines and their systematic automatic verification from the developer side. The reasons why we did not automate this task, is that we do not have a model against which we can verify the correctness of the formulas we build. In addition, performing a deep semantic and security text analytics on the textual description of the guideline is not in the scope of our work.

Except the first step, the different components of our framework are automatic. The different components that we have implemented or adapted are:

- PDG Constructor: this component builds the standard program dependence graph from the Java bytecode (.class) which is provided as input. We herein refer to the JOANA IFC tool that we made use of for many aspects, including the construction of the PDG.
- PDG Annotator: given the standard PDG constructed in the previous step, the PDG Annotator adds annotations on the constructed PDG. The annotations are added automatically, and this was made possible through the Security Knowledge Base (Section 10.3). The PDG runs the information flow analysis on the annotated PDG in order to propagate the annotations, and to capture the explicit as well as implicit dependencies. The PDG Annotator provides as output the **Augmented PDG**.
- Java Classes Parser: with the objective of gathering the different native Java classes details, we developed a parser [1]. This component takes as input the URL of the Java class official documentation [115] [114]. The parser proceeds as follows; for the given Java class URL, it parses the HTML code and extracts all the relevant class details; the class name, the class it inherits from, the class description, the attributes, the constructor(s), the methods signatures, their return type and their parameters. This component populates the Security Knowledge Base with the extracted class details.
- Security Knowledge Base: apart from the Java classes details, this database contains the different labels that the security expert makes use of in order to build the security guidelines formulas. We carried out the effort of performing a semi-automatic and very basic semantic analysis, basically a keyword matching that aims at assigning to the different Java methods specific labels. For example, the different methods that contain in their description the word "print" are assigned the label "print". Similarly, the label "isKey" is assigned to the methods whom descriptions include the expressions "construct key" or "create key".

[1] https://github.com/zeineb/Java-classes-parser

– LTS Construction: this component translated the **Augmented PDG** into a Labelled Transition System; it retrieves the Augmented PDG details (nodes, edges, annotations, security levels, etc.) and builds the Labelled Transition System that is accepted by model checking tools.

– Model Checking: we carry out the model-checking analysis directly on the LTS that we generate from the Augmented PDG. We made use of the checker EVALUATOR of the CADP toolsuite [97] to verify the security guideline expressed in MCL. The output of this component indicates whether the guideline is met, or it is violated, and the violation traces are returned.

– Feedback Representation: this component exploits the output of the "Model Checking" to provide a precise and useful feedback to the developer to understand the source of the violation, and possibly how to fix it. This component behaves as follows; it considers the violation trace(s) returned on the LTS level. Those traces are represented on the Augmented PDG constructed from the source code, and this provides a clear feedback to the developer, explaining the source of the security guideline violation.

7.3 JOANA IFC

7.3.1 Presentation

JOANA IFC (**J**ava **O**bject-sensitive **ANA**lysis Information **F**low **C**ontrol), is a static code analysis framework that checks full Java bytecode with arbitrary threads for the property non-interference. In the information flow vocabulary, non-interference consists in guaranteeing that confidential data ($HIGH$) cannot leak to public variables (LOW). Confidential data might however be manipulated, as long as public variables cannot reveal information about the confidential data. The tool analyzes full java bytecode to check if high level data interfere with low level data. The "legal" flows are represented in the form of a security lattice that specifies a partial order on the security levels. For instance, high level can flow to high level, or low level can flow to high level. Low level can of course flow to low level. "Any flow from a statement labeled *l1* to a statement labeled *l2* is considered legal iff $l1 \leq l2$." [70].

The JOANA tool proved to be precise [78] [75] [80], and handles exceptions and inheritance. The tool deals with sequential [80] as well as concurrent programs [65] [71]. It has the strength of verifying the absence of possibilistic as well as probabilistic leaks for full Java bytecode, including exceptions, dynamic dispatch and inheritance. The JOANA tool can also handle sequential [80] and multi-threaded programs [65] [71].

7.3.2 Program Dependence Graph Construction

The JOANA IFC tool is built upon the WALA (WAtson Libraries for Analysis) that offers wide capabilities for the static analysis of Java byecode. WALA represents the program to analyze in an intermediate representation (IR)[2] in SSA-form, which is a language close to the JVM bytecode [2]. SSA (Static Single Assignment) is an intermediate representation in which every variable is assigned exactly once, and every variable should be defined before being used. This refers to the so-called **Definition-Use Chain** which is a data structure specifying for a variable, its definition D and all its uses U reachable from that definition D without any other definition in between. This step is of paramount importance as it allows to identify and track all the variables through the program sources.

JOANA operates on WALA's IR, and produce on top of this representation the Program Dependence Graph (PDG), or more precisely the SDG (System Dependence Graph). WALA has the ability of resolving dynamic dispatch, and is also able to capture potential exceptions and compute method invocations side-effects. We shared an example of the PDG construction illustrating this specific aspect in the Section6.3.3.

The JOANA IFC is a powerful tool that covers full Java bytecode, and relies on the PDG structure to perform the analysis. As mentioned previously, PDG construction adopts a rather conservative approach, and constitutes an over-approximation of the program behavior. We consider this toy sample code 7.1:

Code 7.1: Non-interference analysis raising false positives in JOANA

```
public static void main(String[] args) {
      String pwd = "abc";
      System.out.println("hello");
   }
```

For this sample code, we have annotated as HIGH level source the *pwd* variable that contains the password data, and as LOW level the display message *System.out.println("hello")*. The JOANA IFC reported 8 security violations, denoting the captured illegal information flows. As one can notice, non-interference analysis operates not only on the source and the sink, but considers the intermediate nodes, and this led to the false alarms that were captured. Through this sample code, we wanted to give a flavor on the over-approximation of the program behavior as represented in the PDG.

We will explain through this chapter the effort that we did in order to increase the

[2]http://wala.sourceforge.net/wiki/index.php/UserGuide:IR

analysis precision through the new dependencies that we have introduced.

7.4 PDG Annotator and Automatic Annotations

The component PDG Annotator is part of the JOANA IFC tool. It is invoked once the PDG is constructed, and its main functionality is to apply the annotations on the nodes. We made changes on this component in a way that it can handle the customized annotations that we have defined and associated to the traditional information flow annotations. We did also the effort of implementing a new functionality for the PDG Annotator, consisting in applying automatically the annotations on the PDG. This major change would not be possible without the Security Knowledge Base through which the PDG Annotator fetches for the method signature its corresponding annotation. Sample data from the Security Knowledge Base are provided in Appendix D.3. The Figure 7.2 illustrates the main interactions of the PDG Annotator and the Security Knowledge Base.

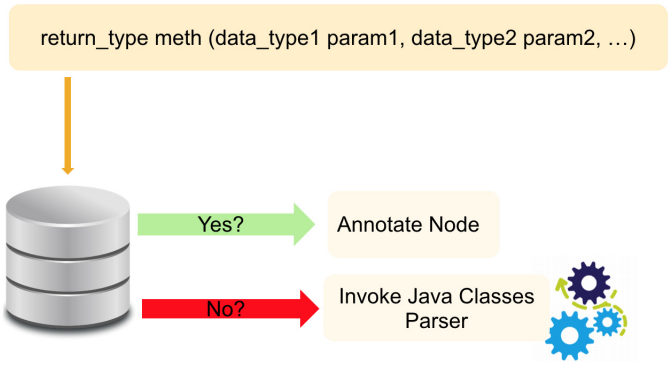


Figure 7.2: PDG Annotator

The PDG Annotator traverses the PDG, and captures for each node the program part, that refers to the bytecode instruction. The component then fetches in the SKB the corresponding label. If a match is found, the PDG Annotator applies the annotation, together with the corresponding information flow annotation (sink, source, declass) and security level (high, low). Otherwise, the Java Classes Parser is invoked to retrieve the relevant information of the class in question.

We made use of this component basically to augment the PDG with the new annotations and prepare the ground for the information flow analysis to propagate those annotations and capture the implicit dependencies that may occur between the program statements.

Note that the automatic annotations functionality of this component has several limitations that are mainly the non coverage of the semantic aspect. For instance,

the verification of the guideline "Store passwords using a hash function" requires the annotation *password*. This annotation cannot be performed automatically as the semantic analysis on the program is not in the scope of this work. At his level, the human intervention is required to enter this category of semantic-based annotations.

7.4.1 Automatic Detection of Sinks and Sources

The main objective of this operation is to help capture eventual sources for a given sink, and capture possible sinks for a given source. It appears to be useful when it is hard to detect for a secret data where it can flow using manual inspection in the code. This requires security expertise and a knowledge of the inner details of the program. For a developer or a tester who is not aware about the semantics of methods performing security operations, detecting possible sources (resp. sinks) and their respective sinks (resp. sources) appears tedious. We did a first attempt to cover this aspect that appears of paramount importance especially when the variable names are not semantically significant, or that we are not able to manually identify sensitive information. For example, we can imagine a code reviewer validating an application that is submitted to the software marketplace. This human operator knows for instance that private APIs should not be called, or that specific methods that could potentially send information to third-party advertising companies should not be called. Those methods refer to potential sinks that sensitive information should not reach. It would be interesting to have a tool that allows to annotate only the sinks, and then capture and return the potential sources. Same for sources. We can imagine that the human operator knows the sensitive information (payment information for example). It would be very helpful to annotate only the source (credit card number), and that the potential sinks are returned. We have implemented a first proof-of concept on this specific problem, and we made use of the well-known PDG techniques: slicing and chopping.

Slicing is a known technique that consists in simplifying programs and eliminating the parts that have no influence on the part of interest. The part of interest is called in the *slicing criterion*, and is typically defined by a pair; program location and variables of interest. The computation of the parts of the program that affect directly on transitavely the slicing criterion C result in the *program slice* with respect to the slicing criterion C. A slice S of a program is defined as an executable reduced program computed from a program P [148]. Another definition joins the same concept introduced by Weiser except that the slice is not necessarily executable, and that it is a subset of the program instructions that influence the slicing criterion C.

Backward Slicing The backward slice with respect to a variable x at a program point p is the set of paths that may influence the variable v at that point p.
BS(x) = y | y →*x

Backward slicing allows to capture all the statements that can influence directly or indirectly the variable v at the program point p.

Forward Slicing The forward slice with respect to a variable x at a program point p is the set of paths that the variable v may influence at the point p.
FS(x) = y | x →*y
We made use of the backward slicing technique to capture the sources of a given sink, and the forward slicing to capture the sinks of a given source. We were faced with several issues such as the slicing in the presence of threads or loops.

7.4.2 Information Flow Analysis

As we have mentioned previously, we make of JOANA IFC tool to fulfill several objectives. First, the construction of the PDG structure. Then, to run the information flow analysis with the objective of capturing subtle and implicit dependencies, and to propagate the customized annotations. The JOANA IFC tool introduces basically 3 types on annotations: SOURCE, SINK and DECLASS, together with their security level (HIGH, LOW). We took profit from the information flow analysis carried by JOANA IFC, that covers full Java bytecode and that is formally proven.
We need to stress a specific detail regarding a common approach that is used by compilers with respect to variables: the SSA (Static Single Assignment) that consists in representing every value in the program by a variable. The idea behind SSA is the following; each variable has only one static definition. And here we distinguish between two specific actions: *def* and *use*.
We refer to renaming, which is an extension to SSA. This operation consists in renaming each variable when it is assigned a new value. Two variants exist: static single use form and static information form. The former renames each variable at each use, and the latter renames each variable at each new assignment.

7.5 Security Knowledge Base

Security Knowledge Base (SKB) is a centralized repository of structured data gathering the labels of the formulas mapped to APIs, instructions, libraries or programs. The SKB is of paramount importance in our framework for the possibilities it offers to help the automatic detection of labels on the system model (our augmented

PDG). We designed the Security Knowledge Base in a way allowing to represent the different relationships between Java methods and the abstract labels used to compose the security guidelines formulas by the security experts. We populated the Security Knowledge Base using an open source **Java Classes Parser** [3] that we developed. The parser operates as follows; for the different Java classes used in the program to analyze, we launch programmatically the automatic parsing of this given class (html code, javadoc), and we automatically extract all the relevant details, including the class description, the attributes, the constructors, the methods signatures and their parameters. The parser crawls and queries the Oracle official APIs documentation in order to fetch the classes details, and persists them in the central repository.

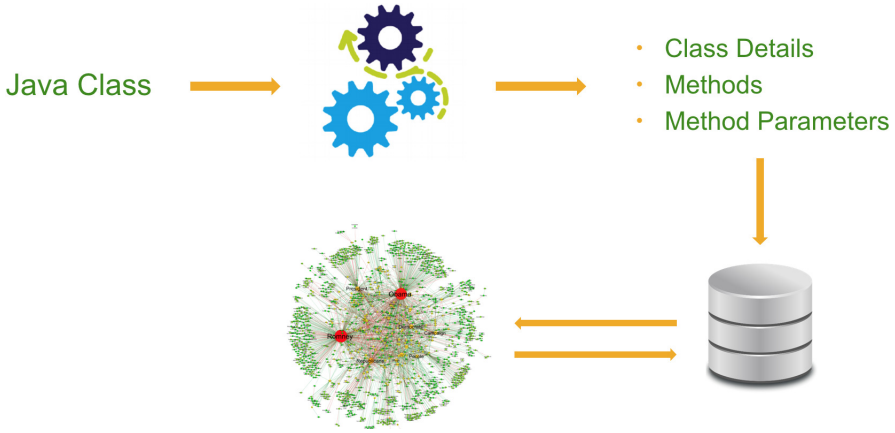


Figure 7.3: Java Classes Parser: populating the Security Knowledge Base

The operation described above populates the Security Knowledge Base. It is true that we have structured exploitable data at this level, but it needs further processing in terms of semantic enrichment, and this is another crucial work that we have taken care of covering in our thesis work. To the best of our knowledge, there is no prior work in this area that covered the semantic aspect of the Java APIs and their parameters. We performed an automatic string-based search on the Java APIs descriptions, parameters and return types, to detect specific words such as *print*, *secure*, *display*, *generate key*, *input*, etc. As the reader might have noticed, those specific words refer to the labels that we make use of to build the MCL formulas, and those labels as explained above, will be translated into annotations on the generated PDG. The reason why we have proceeded to the semi-automatic semantic analysis on the Java APIs descriptions, is to have strong basis for the automatic detection of sinks and sources on the PDG.

[3]https://github.com/zeineb/Java-classes-parser

The security expert establishes the mapping of those key words to the possible Java language instructions. For example, the method (the constructor) *SecretKeySpec(byte[] key, String algorithm)* that constructs a secret key from a byte array, will be mapped to the abstract label "create_key"; this label is described in our Security Knowledge Base as "encryption key", a sensitive information that should be kept secret. The Security Knowledge Base can also be perceived as an extensible dictionary gathering the labels with respect to their semantics and to the concepts they represent. For instance, the label "create key" can also be mapped to the method invocation *generateKey()* of the Java class **KeyGenerator** that allows to generate a secret key. In our Security Knowledge Base, we offer a wide range of labels that can be used to build the security guidelines formulas. The set of labels can be extended by the security expert if new security concepts are introduced.

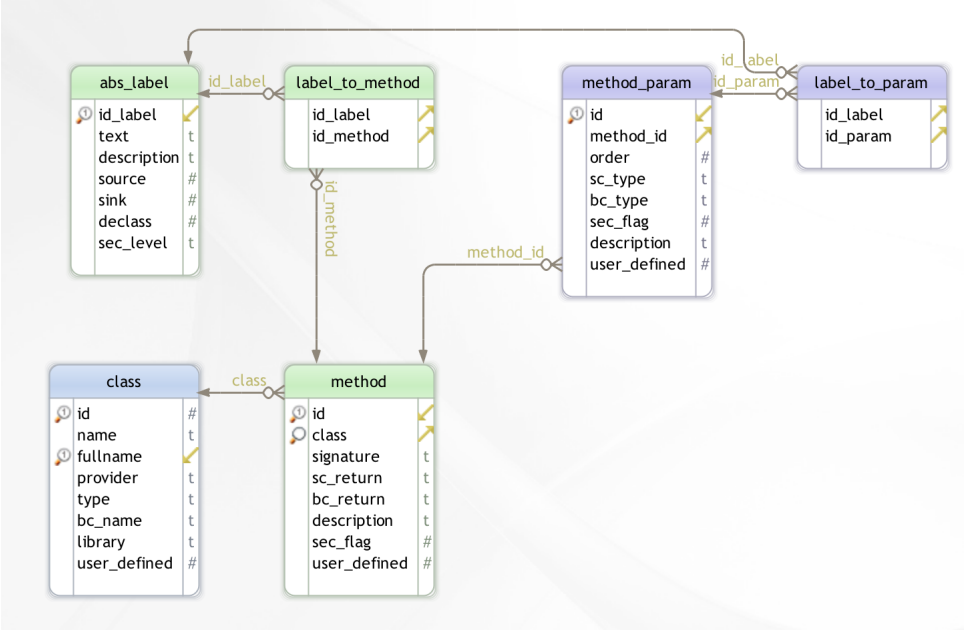


Figure 7.4: Security Knowledge Base: Graphical representation of the Java classes and methods, mapped to labels

7.6 Model Checking: CADP/Evaluator

Construction and Analysis of Distributed Processes CADP [59] [64] is a toolsuite for the design of concurrent programs. It is a convenient platform for the specification, verification and testing of distributed systems. This tool has been used in different industrial projects [4]. CADP tool has the ability of handling large state spaces

[4]http://cadp.inria.fr/case-studies/

(up to $10\hat{1}0$ states) [64] through the usage of compositional verification techniques. CADP contains different model checkers, among them the EVALUATOR model checker that we have used in our approach. The EVALUATOR 4.0 has the ability of handling MCL formulas, which adds this support to data handling.

7.7 Summary

In this chapter, we described in details how we integrated the different tools together in order to ensure the full-tool chain for the formal verification of security guidelines specified in MCL. We showed the feasibility of our approach on a sample code presenting a very subtle and implicit violation of a guideline, and we proved that through our work, we are able to capture it. This detection was made possible since we have introduced the label *depend* that is able to capture subtle dependencies, and represent them first on the Augmented PDG. We explained then how we automatically generate the pLTS from the Augmented PDG and we stressed that the two graphs are isomorphic.

For the guidelines involving the semantic aspect, such as password security rules, the automatic detection of the password data on the program cannot be performed automatically. The annotation of the password requires a deep knowledge on the program logic, and the intervention of the developer/security expert to annotate the password is required.

Chapter 8

Evaluation and Limitations of the Approach

Contents

8.1 Introduction

The main goal of this thesis is to provide a framework that allows the automatic verification of security guidelines on the code level, hence leveraging the heavy workload on the security expert and the developer when verifying the adherence of the being developed software to the security guidelines. We evaluated our approach against sample codes that we have implemented, and that present very subtle and implicit dependencies. Our approach is based on the PDG structure, which is an over-approximation of the program behavior, which can lead to an analysis imprecision and false alarms. Of notable impotance is the scalability criterion that should also be adressed. Prior work in the area of IFC can cover real world programs written in full Java [143]. The PDG that we augment with dependencies

details is constructed by the JOANA IFC tool, which suffers from scalability issues (it can scale up to 100k loc). Hence, our framework might suffer from scalability issues. But we can still think of performing improvements on the PDG construction, and enhance the scalability of the WALA tool, on which the JOANA IFC is based for the graph construction.

8.2 Security Expert Perspective: Formal Specification of Security Guidelines

Since we aim through this work to provide assistance and guidance to the security expert when formalizing the guidelines. We made available in the Security Knowledge Base (SKB) the security patterns that he can make use of to build the formulas. We have also provided a wide range of security-related Java APIs mapped to labels and their level (See Appendix D.3). From Brian, the security expert perspective, using the MCL language would not be an overhead when referring to the Security Knowledge Base (SKB) and to the validated formula patterns. We gathered in Table 5.4 some guidelines that we have formalized following the methodology that we provided to Brian.

The dictionary that we have built and made available (to security experts as well as to developers) is rich, yet extensible. For instance, if Brian is faced with new security concepts, or elements that have not yet been represented in the Security Knowledge Base, he can introduce them and provide their semantics.

Through the proposed formalization approach, we have covered an important spectrum of security guidelines. However, for guidelines involving quantitative information flow theories, our approach may not cover. For instance, the guideline ***Limit quantity of data encrypted with one key***[120] recommends the use a new encryption key when the amount of encrypted data goes beyond a certain threshold. It is obvious to the reader than our specification falls short in covering this guideline for the dynamic aspect it involves, and that can't be captured statically.

Another issue that we have identified is the semantic aspect in the guidelines description. For instance, the notion of *sensitive* information or *trusted/untrusted* data cannot be easily captured in the MCL formulas. The formalization of guidelines involving this notion requires further refinements that can be deduced from the application's context and domain.

8.3 Developer Perspective: Formal Verification of Security Guidelines

Assisting Alex the developer through the verification of security guidelines on his developed application is another main objective of this thesis work. We tried to reduce his intervention for the verification of the guidelines, and provide a precise feedback that explains the source of the violation, in a debugger-like mode.

Let us go back to the sample code that we have introduced in the Chapter Introduction.

Code 8.1: Sample code for the logging of user credentials

```
BufferedReader reader = new BufferedReader
        (new InputStreamReader(System.in));
   System.out.println("User name : ");
   System.out.println("Password : ");

   // input
   user.setUsername(reader.readLine());
    user.setPassword(reader.readLine());

   // copy
   String xx = user.getPassword();
   // hash
   MessageDigest hash = MessageDigest.getInstance("MD5");
   byte[] bytes = user.getPassword().getBytes("UTF-8");
   byte[] hash_password = hash.digest(bytes);

   user.setPassword(hash_password.toString());

   // log
   logMessage = "user name = " + user.getUsername() +
        ", password = " + user.getPassword() + xx;
   logger.log(Level.INFO, logMessage);
```

In the sample code, Alex accidentally logged the variable xx that contains the password. This violated the guideline MSC62-J:***Store passwords using a hash function*** [40], as xx has not been hashed before being logged.

In order to capture this violation, we proceed by the generation of the PDG that we augment with details we fetch from the code and the Security Knowledge Base. In fact, we reduce the overhead on Alex and we automatically annotate the node corresponding to the actions of *log* and *hash*. It is true that the code we consider is simple, but if we had a more complex program source, then the manual anno-

tations would be tedious and error-prone, as it requires a deep knowledge on the programming language semantics and idioms. The automatic annotations that we propose allows to augment the PDG with annotations, and the developer, Alex, has not to do this extra effort. We have mentioned that in this thesis work we do not perform a semantic analysis on the program, and this is one of the reasons why the intervention of Alex is required. For instance, we do not have the means to annotate the node corresponding to the *password* data automatically. Once Alex annotates the semantic data (such as credit card number, password, etc.), he can proceed to the automatic verification. In the figure below, we represent the pLTS that we have generated from the Augmented PDG. As the reader can notice, the implicit dependencies are captured and represented through the transitions.

Running the model checking on this pLTS produces the violation trace(s) that are mainly paths on the pLTS. In the Figure 8.2, we represent the violation trace as it was produced by the EVALUATOR model checker of the CADP toolsuite [97]. The feedback is precise, in the sense that it shows the exact path violating the guideline. Hence, Alex the developer can identify the source of the violation, and eventually how to fix it.

8.4 Limitations of the Proposed Approach

In this section, we pinpoint some limitations of our approach, and highlight the fact that some of them cannot be treated on the program sources. As they are more related to the execution environment.

8.4.1 State Space Explosion

State space explosion [146] [44] is another limitation that our framework might be faced to. *State space explosion* is a widely known limitation of model checking, and it occurs in programs presenting a wide range of interacting components and data structures. In these programs, the number of states can be very large, which results in an important amount of time and required memory resources. In the literature, different approaches have been proposed to solve state space explosion problem; Symbolic algorithms [43] that represent the state space symbolically (BDDs), using abstraction to reduce the size of the state space, and Partial order reduction [122] [similarly to the model checker SPIN [83]].

We can think of improving the LTS construction as follows: one the augmented PDG is constructed, we can proceed to the slicing or the chopping on this PDG to reduce its size in terms of nodes and edges. This results in a smaller PDG that the original one. This operation assumes that we will construct one PDG for every guideline, as the chopping will take into account the labels of the guideline that are

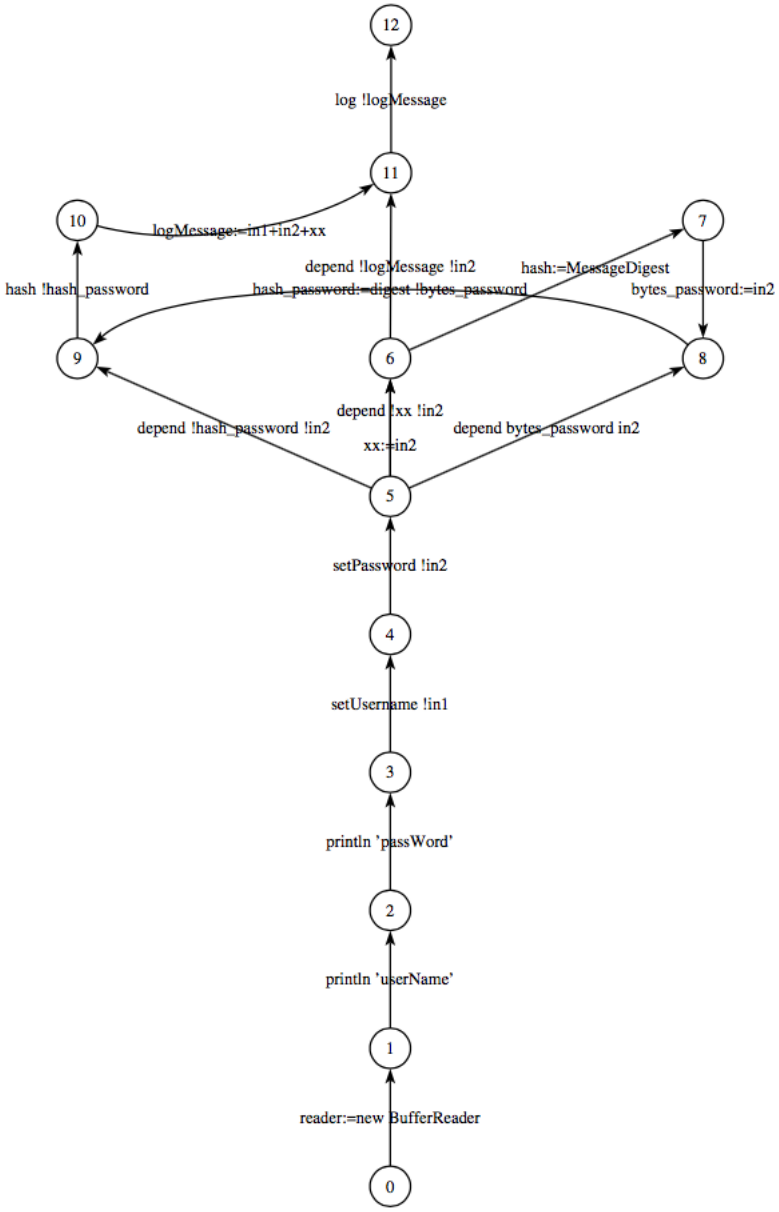


Figure 8.1: LTS for the sample code 8.1

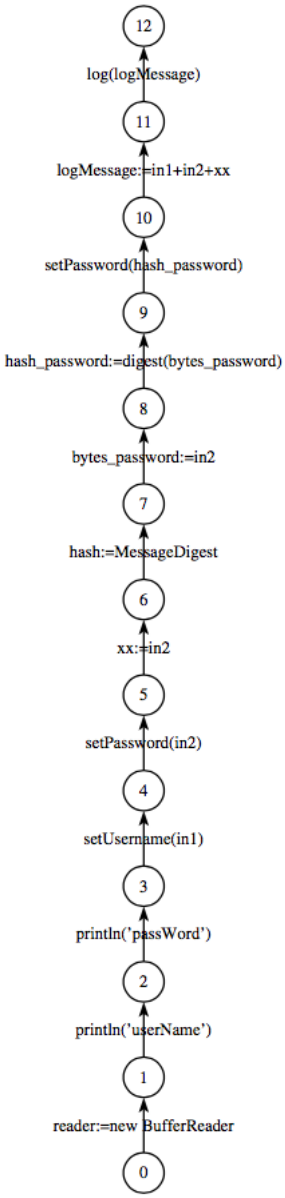


Figure 8.2: Violation trace for the sample code 8.1

translated into annotations on the augmented PDG. Then, we generate the pLTS from the reduced augmented PDG. This will come at a cost, but we can guarantee that this can be a way towards solving state explosion in our framework.

8.4.2 Threads Synchronization

In the presence of multiple threads, the control flow of the program can be altered, and the shared data can be modified in inconsistent ways. In this specific scenario, we are faced mainly with two types of information leaks: *possibilistic* and *probabilistic*. Their occurrence depends on the interleaving of concurrent threads. The occurrence of possibilistic leaks depends on the specific interleaving, while probabilistic leaks exploit the probability distribution of the order of interleaving. The latter allows non-determinism in programs and demands that the probability of any observable (public) behavior is not influenced by secret values.

In the sample code below, the logging operation is invoked in the thread *Log* (Figure 8.3), and the sanitization is called in the thread *Sanitize* (Figure 8.2). The point we are trying to make through this sample code, is that despite the threads synchronization, we managed to implicitly leak the user input data, hence to violate the security guideline IDS03-J:"*Do not log unsanitized user input*".

We explain in details the different methods that we have implemented in this specific sample code, and how the JOANA tool failed to capture them.

We proceeded as follows: we generate the PDG using the JOANA tool, without setting the configuration parameter "compute interference edges". Then, we annotate the user name (input data) as high level source, and the logging operation as low level sink. The IFC analysis produced no violations, as it failed to capture the implicit user name leakage through the threads *Log* and *Sanitize* threads.

We do a second attempt, this time with the configuration "compute interference edges" and the *precise may-happen-in parallel*. This parameter, together with the points-to-analysis, allow to compute possible inference edges in the PDG. Despite this configuration, the JOANA IFC analysis returned "no violations".

As the reader can notice, we used the methods *notifyAll()* and *wait()* that allow to send signals between threads in the same program. In this sample code, the thread *Sanitize* invokes the *sanitize* method that eliminates suspicious characters from data. Once sanitization is performed, the thread *Sanitize* notifies the thread *Log* that data was changed. The mechanism wait/notify [90] allows one thread to wait for the completion of a change by another thread.

Code 8.2: Thread *Sanitize*

```
public static class Sanitize extends Thread {

   public void run() {
      System.out.println("Sanitize");
      sanitize(userName);
      user.setUsername(userName);
      try {
         synchronized (this) {
            this.wait();
            this.notifyAll();
         }
         while (true) {

            synchronized (this) {
               this.wait();
               this.notifyAll();
            }
         }
      }

      catch (Exception e) {
      }
   }
}
```

Code 8.3: Thread *Log*

```
public static class Log extends Thread {
   Thread t;

   public Log(Thread t) {
      this.t = t;
   }

   public void run() {
      try {
         System.out.println("Log");
         log(user.getPassword());
         synchronized (t) {
            t.notifyAll();
            t.wait();
         }

         while (t.isAlive()) {
```

```
            synchronized (t) {
               t.notifyAll();
               t.wait();
            }
         }
      }

      catch (Exception e) {
      }
   }
}
```

Code 8.4: Sample code with the threads *Sanitize* and *Log*

```
public static void main(String[] args) {
     String password = "<fjhdfjhdj";
     user.setPassword(password);
     Thread t1 = new Sanitize();
     Thread t2 = new Log(t1);
     t1.start(); // Sanitize
     t2.start(); // Log
}
```

Code 8.5: Method *santize* that checks for the existence of suspicious characters the input String

```
public static void sanitize(String data) {
   Pattern pattern = Pattern.compile("[<>]+");
   Matcher matcher = pattern.matcher(data);

   if (matcher.find()) {
      data = data.replaceAll("<", " ");
   }
}
```

Code 8.6: Method *log*

```
public static void log(String data) {
   String logMessage = "user data = " + data;
   final Logger logger = Logger.getLogger("User");
   logger.log(Level.INFO, logMessage);
}
```

In our framework, we do not yet have the means to cover threads communication,

where threads exchange a given information.

8.4.3 Java Reflection Dependency

To the best of our knowledge, no prior work in the area of information flow analysis or PDG-based verification approaches has covered the Java reflection problem [140]. Java Reflection [62] has always been a tedious problem for static code analysis, as program parts invoking reflection would not be included in the graph representation of the program. This leads to an unsound analysis, as static analysis does not take into consideration Java reflection. In [101], the authors propose static reflection resolution algorithms. Their approach relies on the points-to analysis to identify to target of the reflective calls, that are then added to the call graph. Their approach falls short in covering the possible reflective calls in the program, as they do not include control and data dependencies in their analysis to capture the reflective calls dependencies.

Before going through the details of the Java reflection dependencies, let's take a look at the sample codes of Figures 8.8 and 8.7.

Code 8.7: The class **User**

```
public class User {
   private String name;
   private String password;
   private int creditCardNumber;
   public int crypto;
   private Date birthdate;

   // transferToAccount
      public void transferToAccount(float amount, String iban) {
      ...
   }
```

The class User has private attributes: the name, the password, the credit card number, the visible cryptogram and the birth date. It has a method transferToAccount that takes as parameters the amount of money to transfer, and the bank account (iban) to which the amount will be transferred.

In the code 8.8, we make use of Java reflection to do malicious operations that consist basically at changing the accessibility of private attributes and dynamically invoke the method transferToAccount without being detected. In this sample code, we have made multiple reflective calls, and we commented every instruction to

explain the operation that we are carrying out.

Code 8.8: The class **UserReflection**

```
public class UserReflection {
   public static void main(String[] args) {
// Obtain the class object if we know the name of the class
      Class user = User.class;
      try {
         // get all the constructors of the class
         Constructor[] constructors = user.getConstructors();
         System.out.println("Constructors are: " +
             Arrays.toString(constructors));

         // get constructor with specific argument
         Constructor constructor = user.getConstructor(Integer.TYPE);

         // initializing an object of the User class
         User userObj = (User) constructor.newInstance(455);

         // get all methods of the class including declared methods of
         // superclasses
         // in that case, superclass of User is the class java.lang.Object
         Method[] allmethods = user.getMethods();
         System.out.println("Methods are: " + Arrays.toString(allmethods));
         for (Method method : allmethods) {
            System.out.println("method = " + method.getName());
         }

         // get all methods declared in the class
         // but excludes inherited methods.
         Method[] declaredMethods = user.getDeclaredMethods();
         System.out.println("Declared Methods are: " +
             Arrays.toString(declaredMethods));
         for (Method dmethod : declaredMethods) {
            System.out.println("method = " + dmethod.getName());
         }

         // get method with specific name and parameters
         Method oneMethod = user.getMethod("transferToAccount", new
             Class[] { Float.TYPE, String.class });
         System.out.println("Method is: " + oneMethod);

         // call transferToAccount method with parameter float
         oneMethod.invoke(user, 500.0f, "FRPAXXCF");
```

```
        // get all the parameters of transferToAccount
        Class[] parameterTypes = oneMethod.getParameterTypes();
        System.out.println("Parameter types of transferToAccount() are: "
            + Arrays.toString(parameterTypes));

        // get the return type of computeRentalCost
        Class returnType = oneMethod.getReturnType();
        System.out.println("Return type is: " + returnType);

        // Get access to private member fields of the class
        // getDeclaredField() returns the private field
        Field privateField =
            User.class.getDeclaredField("creditCardNumber");

        String name = privateField.getName();
        System.out.println("One private Fieldname is: " + name);
        // makes this private field instance accessible
        privateField.setAccessible(true);
        // get the value of this private field
        int fieldValue = (int) privateField.get(user);
        System.out.println("fieldValue = " + fieldValue);

    } catch (NoSuchFieldException e) {
        e.printStackTrace();
    } catch (NoSuchMethodException e) {
        e.printStackTrace();
    } catch (IllegalArgumentException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (InstantiationException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    }
}
}
```

We analyzed the code above with the JOANA IFC tool (*creditCardNumer* attribute annotated as high level source, and *System.out.println("fieldValue = " + fieldValue)* as low level sink), but the analysis did not capture the violations. We started exploring different directions to cover the Java reflection, and add it as

new dependence type on our Augmented PDG. We parsed the different classes of the **java.lang.reflect** package, and included them in the Security Knowledge Base. Then, we introduced a new label named "reflect", that refers to the different reflective calls. For now, the label is generic, but can be refined further. For example, we can create other labels related mainly to the reflective calls that change the accessibility of a field, and specific labels to methods invocation with reflection. The key operation to compute the reflection dependencies is to carry out advanced information flow analysis, similarly to what we did to capture the file location dependencies (Section 6.4.3). The idea is to extend the analysis to the parameters of the reflective method calls, and compare their values with respect to the implemented class attributes and methods.
This dependency is a bit tedious to capture, and working towards solving it is very challenging. We would be working towards closing this gap and be able to represent the Java reflection dependencies accurately. This would optimize the analysis precision as it will capture very subtle and implicit dependencies.

8.4.4 Java unsafe API: sun.misc.Unsafe

Java programming language is intended to be a safe language, preventing the developers of doing unintentional mistakes due to bad memory manipulation. However, there exist different ways that may cause intentional memory mistakes, such is the case for the *sun.misc.Unsafe* API. The *Unsafe* class offers a wide range of methods (105) ranging from information (e.g. memory address size) to direct memory access methods (allocate memory, free memory, etc.). We explain in this section how we made use of this API together with the Java reflection to leak sensitive information, and get access to the secret data memory address and change its value to bypass the access control check. The class Unsafe is public, and has a private constructor. In other words, we cannot instantiate the class through the constructor invocation *Unsafe unsafe = new Unsafe()*. *Unsafe* has a private instance named *theUnsafe* that still can be invoked via the Java reflection through *Field f = Unsafe.class.getDeclaredField("theUnsafe")*.

Code 8.9: Usage of the *sun.misc.Unsafe* Java API

```
import java.lang.reflect.Field;
import sun.misc.Unsafe;

public class ErrorTesterReflectNew {
    public int secret; //<--- SOURCE

    public ErrorTesterReflectNew() {}
```

```
    public static void main(String[] args) throws Exception {
        ErrorTesterReflect test = new ErrorTesterReflect();
        try {
            test.secret = 8;
            throw new Exception();
        }
        catch (Exception e) {
            System.err.println(getInt(test,"secret")); // <--- SINK
        }
    }

    static int getInt(ErrorTesterReflect test, String name) throws
        Exception {
        Unsafe u = null; // = Unsafe.getUnsafe();
        try {
            Field f = Unsafe.class.getDeclaredField("theUnsafe");
            f.setAccessible(true);
            u= (Unsafe)f.get(null);
    } catch (Exception e) { }
        long offset =
            u.objectFieldOffset(ErrorTesterReflect.class.getDeclaredField(name));
        return u.getInt(test, offset);
    }

    static Unsafe getUnsafe() throws Exception {
        Field f = Unsafe.class.getDeclaredField("theUnsafe");
        f.setAccessible(true);
        return (Unsafe) f.get(null);
    }
}
```

We provide another sample code presenting a more concrete and dangerous memory manipulation that can lead to unlimited access to non-authorized users. We managed to get the memory location of the *is_authorized* attribute, and could change its value straight through writing in its memory location. In this sample code, we combine Java Reflection with Unsafe API, which makes the dependencies computation more complicated.

```
public class UnsafeAccessControl {
   public int is_authorized;

   public Boolean grantAccess() {
      return is_authorized == 109;
   }
```

```
public UnsafeAccessControl() {
}

public static void main(String[] args) throws Exception {
   UnsafeAccessControl accessControl = new UnsafeAccessControl();
   accessControl.grantAccess();
   System.out.println(accessControl.grantAccess()); //false
   Unsafe unsafe = getUnsafe();
   Field f = accessControl.getClass().getDeclaredField("is_authorized");
   unsafe.putInt(accessControl, unsafe.objectFieldOffset(f), 109);
   System.out.println(accessControl.grantAccess()); //true
}
```

In this sample code, we check if we have access (*accessControl.grantAccess()*). This returns *false*, as to have access, the *is_authorized* attribute should have the value 109. Then, we get access to the attribute *is_authorized* (*Field f = accessControl.getClass().getDeclaredField("is_authorized")*) and we invoke a reflective call to change its value straight in the memory location (*unsafe.putInt(accessControl, unsafe.objectFieldOffset(f), 109)*). Now the call to *accessControl.grantAccess())* returns true, as the value of the *is_authorized* attribute is set to 109.
Possible directions that we can adopt, is to add another layer of dependencies computation that covers the multitude of subtle and implicit dependencies that automatic tools fall short to cover. We have the intention of exploring further the possible directions and adding the different dependencies that we can capture on the augmented PDG. We are aware that this will adds up to the over-approximation of the program behavior, but at least, we would be able to detect potential leaks.

8.4.5 Timing and Termination Covert Channels Detection

Covert channels can be source of implicit information leakage, and can remain undetected. Lampson [96] defines three classes of channels: the first class comprises *legitimate* channels, that are the declared formal local outputs of the system. The second class involves the *storage* channels that refer to storage objects that are not in the local system, but rather in the execution environment. The third class comprises the so-called covert channels, that refer to channels that do not fall in the two previous categories. The approach that we propose in this work falls short in detecting the possible covert channels, as this requires a more fine grained level of analysis that captures not only the dependencies in the program, but also in the execution environment.

8.5 Summary

In this chapter, we shared different shortcomings that our framework can suffer from, and stressed on the fact that there are parameters that cannot be captured on the program sources, but are rather on the execution environment. We showed on sample codes presenting very subtle and implicit dependencies the limitations of the PDG construction by JOANA that fails to capture them, as the detection of such details requires an advanced analysis that can be extended on the methods parameter names (Java reflection), and on the synchronization of threads. We shared also possible directions that we can follow to solve the problems discussed in this chapter.

Chapter 9

Conclusion and Future Perspectives

Contents

9.1 Achievements

We presented a first proof-of-concept regarding the feasibility of our approach that aims at extending the guidelines verification and validation on the different phases of the software development lifecycle. We proposed a first attempt to fill the gap of the formal verification of guidelines provided in informal way through the full-too chain that we have implemented and that covers the end-to-end verification of compliance to security guidelines on the code level. The innovation of our work is the combination of existing approaches that are meant to fulfill different objectives. We did the effort of combining static code analysis with model checking. We showed how we can improve the analysis precision through the new dependencies that we have introduced, and that allowed to capture very subtle and implicit dependencies.

We did also the effort of surveying security guidelines from different sources, and classify them in categories with respect to the verification type that should be performed in order to check the adherence of the being developed software. In order

to strip away ambiguities, we adopted a formalism, MCL, to specify the guidelines and prepare the ground for their formal verification through model checking. Formal methods can add significant added-value in finding subtle yet serious and implicit security violations that traditional means fail to detect.

We stressed the difficulty encountered when the security guideline involves dependent information flows that can't be specified separately. Our framework makes use of this specification to carry out the model checking on the Labeled Transition System we built from the Program Dependence Graph that we have augmented with details such as the customized annotations and the implicit dependencies that we can capture through the Security Knowledge Base which is another achievement of our work.

The Security Knowledge Base contains a repository of formalized security guidelines and patterns giving the basis to a central treatment and management of the good and bad programming practices. In addition, the Security Knowledge Base contains a wide range of Java APIs, including security-related methods. Those APIs are decorated along with their signature, description and mapped to the traditional information flow annotations (source, sink) and their security level that indicates their level of confidentiality/integrity (high=secret/private, low=public). This repository can be used by both the security expert(s) and the programmers, as it allows to centralize security-related knowledge shared by the different security experts who formalize security guidelines. This can be a strong basis to bring security awareness and education to developers to gain a better understanding of the good programming practices, which improves the code quality with respect to security.

The verification phase output indicates whether the guideline is met, or it is violated, and the violation traces are returned. Using this output, we will be able to provide a precise and useful feedback to the developer to understand the source of the violation, and possibly how to fix it.

9.2 Perspectives

We share in this section possible directions for this thesis work. We have a broad spectrum of ideas we could not cover for the lack of time.

9.2.1 Dynamic Verification of security guidelines

Since static code analysis has several limitations in terms of precision and soundness. It generally produces false alarms or misses violations that can be left undetected and cause serious damages once the software is deployed. Moreover, static code analysis does not cover configuration and execution environment issues, as

they are not part of the source code. Performing a dynamic verification of security guidelines would be a way forward to monitor the satisfiability of the guidelines in the execution environment. We can combine dynamic monitoring with security annotations that we can propagate on the dynamic representation of the program. Prior work in the area of dynamic information flow has been proposed [130]. [16] focus on the dynamic tracking of policies for declassification. [17] combine dynamic monitoring with flow-sensitivity.

9.2.2 Scoring system for the program adherence to security guidelines

Amongst the possible future directions of our work is to establish a scoring system based on the compliance to security guidelines. The idea is to have a centralized corpus gathering the different guidelines that aims at supporting a new certification scheme that goes beyond providing binary answers, and provides a compliance score with respect to the guidelines to be verified. The approach adopted in the OPTET Project allowed to cover different limitations in the software marketplaces, and proposed innovations in terms of service discovery based on security criteria and constraints. Amongst the innovations of OPTET is to assess the security features of services and make them visible to users. Those security features are represented in the form of a certificate in a machine-readable format similarly to the certification scheme proposed in the European Projects *Advanced Security Service cERTificate for SOA* (ASSERT4SOA) [13] and *FI-WARE* projects. The proposed certification scheme supposes the intervention of a trusted third-party: the Certification Authority that leads a test-based evaluation of services security properties belonging to the well-known classification: confidentiality, integrity and availability.

9.2.3 Covering multiple programming languages

Another possible future direction that we would like to explore in the near future is to extend our work to cover different programming languages. It is true that Java is a strongly typed programming language, and is regarded as *secure* programming language. The term *secure* does not mean that developing in Java guarantees to have a secure software. It is rather explained by the fact that manual memory management in Java is very limited. In addition, for Java, like C or C++, the program statements happen synchronously except for threads for example. This is not the case for other programming languages, such as Javascript which is an interpreted programming language and has different difficulties such as callback nesting, known as *callback-hell* [63], that induce a non-linear control-flow.

We aim also at covering a wider range of security guidelines, hence to extend the Security Knowledge Base in order to capture more security concepts, and possibly, to cover different programming languages, since the PDG is language independent, and so is the LTS.

9.2.4 Feedback to the Developer

We worked close to the program source code, and closer to the developer in order to guide him through the development process and insure the quality of his code with respect to security. It is true that we did the effort of rendering a clear feedback to the developer indicating if the guideline was met or not. In the latter case, we return the violation trace(s) on the pLTS that match the paths on the PDG that in turn match the exact program instructions. Different perspectives can be considered to enhance further the verification result. We can for instance think of rendering the verification output on the Integrated Development Environment (IDE), which is even closer to the developer rather that running the analysis on another tool. This can have multiple benefits such as the security-education of the programmer.

Another direction that we can think towards is to provide also the conditions under which the guideline can be met or be violated. The work of [79] and [139] brought to our mind the idea of augmenting the edges on the PDG with precise conditions for information flow. This would increase the analysis precision, as it allows to refine further the PDG. It is true that on the PDG only feasible paths are represented, but path condition refinement will determine if such path is impossible even though it is represented on the PDG. Applying such a good approach would reduce the size of the PDG, assuming that impossible paths will be removed from the graph. Consequently, the size of the generated Labelled Transition System will also be reduced, and this would eventually be a step forward in the state-space explosion problem for our framework.

Regarding the feedback representation to the developer, we can work on enhancing the representation of the verification outcome in a clear and precise way, allowing the developer to understand the source of the raised violation. Future work includes the representation of the model checking output on the Program Dependence Graph, and on the code level in the Integrated Development Environment. We can also take profit from the model checking principle: *correct and test again.* In other words, the feedback can be interactive allowing the developer to make corrections and verify again, until he ends up with a compliant program. This can constitute another facet of the security-education for developers.

9.3 Summary

Through this PhD work, we wanted to stress on the benefits that we can take through the combination of different approaches. We showed the strength of information flow analysis on the PDG structure for the detection of subtle and implicit information flows that we could capture through the new dependencies that we have introduced. The combination of different existing techniques and tools is one major innovation of our work, and we take profit from the precision and automation of model checking, together with implicit details detection on the program sources. The specification of the security guidelines allowed to strip away ambiguities and enhance the precision of the textual description. We wanted also to find common areas for the different communities: security, information flow control, requirements engineering and formal methods. We strongly believe that this work could be advanced further to cover wide range of security guidelines, and be applied on real world software.

Chapter 10

Résumé de Thèse: Spécification et Vérification Formelles des Règles de Bonnes Pratiques pour la Certification des Programmes

10.1 Introduction

10.1.1 Introduction

Les processus d'ingénierie logicielle ont connu une croissance exponentielle au cours des dernières années en termes de complexité de code et de taille d'équipe de développement travaillant sur le même projet. En conséquence, le développement de logiciels sécurisés est devenu une tâche de plus en plus difficile, du fait qu'elle nécessite tout d'abord d'établir les mécanismes de sécurité nécessaires pour éviter les failles et les erreurs pouvant être source de vulnérabilités. Exécuter des tests ne permet pas de détecter les erreurs avant le lancement ou le déploiement du logiciel, en plus du fait qu'ils soient plus coûteux surtout s'ils doivent attendre la fin du développement pour être exécutés. Adopter l'analyse statique du code et sa mise en place du côté des programmeurs seraient une décision judicieuse pour différentes raisons. Tout d'abord, l'analyse statique ne doit pas forcément attendre la fin du cycle de développement, et peut s'exécuter sur des morceaux de logiciel, ou même sur un produit qui n'est pas nécessairement complet ou fini. Un autre avantage de l'application de l'analyse statique est qu'il ne nécessite pas d'exécuter ou de compiler le code, et peut être géré par le développeur lorsque les

activités de développement sont encore en cours. De plus, identifier et détecter avec précision la source de la violation peut être beaucoup plus rapide et plus précise. L'utilisation d'outils d'analyse statique existants nécessite une expertise très importante par les développeurs pour interpréter, comprendre et appliquer le correctif nécessaire pour remédier aux problèmes de sécurité détectés. Cependant, les développeurs ne devraient pas à la base être des experts en sécurité, et il serait donc nécessaire de leur fournir les directives et orientations nécessaires lors de l'utilisation d'outils d'analyse statique du code. En outre, ces outils sont plutôt centrés sur la détection de vulnérabilités et ne se concentrent pas sur la vérification des règles de bonnes pratiques de sécurité, étant que ces derniers sont présentés de manière informelle, ce qui rend leur interprétation et leur mise en œuvre difficiles pour les développeurs. L'objectif de cette thèse étant de réduire l'écart entre la présentation informelle des règle de bonnes pratiques de programmation sécurisée et leur vérification automatique au niveau du code de point de vue du programmeur. Les règles de bonnes pratiques de sécurité que nous considérons concernent les bonnes ainsi que les mauvaises pratiques de programmation. L'intégration de la spécification formelle et la vérification automatique des règles de codage sécurisé dans les différentes phases du cycle de développement est un autre objectif essentiel de ce travail. Nous expliquons comment ces règles pourraient être formalisées par des experts en sécurité, ensuite vérifiées formellement par les développeurs. Notre approche repose sur l'abstraction des actions dans un programme, et sur la combinaison de la vérification du modèle avec l'analyse de contrôle et de flux de données. Notre objectif est de formaliser l'ensemble des connaissances existantes dans les bonnes pratiques de sécurité à l'aide de formules dans le langage MCL, et de vérifier formellement la conformité des programmes avec ces règles de codage sécurisé, moyennant la vérification du modèle. Fournir un retour précis et clair au programmeur est un des objectifs de ce travail, et qui vise à fournir de l'aide au développeur pour lui expliquer succinctement la source de la violation, et éventuellement comment la corriger. Ce travail de thèse a abouti à la conception d'une chaîne d'outils et à sa mise en œuvre couvrant la vérification automatique et systématique de bout en bout de la conformité aux règles de codage sécurisé au niveau du code.

10.1.2 Contexte et Motivation

Les dernières années ont vu l'émergence des plateformes Cloud, offrant des services rentables, évolutifs, hautement disponibles et partagés pour les entreprises et les utilisateurs finaux individuels. Cela a conduit au développement à grande échelle des marchés de services, offrant de nouvelles opportunités aux fournisseurs

de services de distribuer leurs applications de manière centralisée, et d'atteindre un nombre important d'utilisateurs consommant les services à partir de plates-formes cibles hétérogènes. Ces plates-formes ont soulevé de nouveaux problèmes de sécurité en raison de leur nature distribuée. L'opérateur de marché, le fournisseur de services, ainsi que l'utilisateur final ont tous des exigences de sécurité spécifiques en ce qui concerne les applications déployées dans ce cadre. La nature distribuée et hybride des marchés de logiciels a introduit des problèmes de sécurité différents projets de recherche ont essayé de s'attaquer. Parmi eux, le projet européen financé OPTET (OPerational Trustworthiness Enabling Technologies) qui constitue le contexte principal de mon travail de doctorat. Dans le projet OPTET, la fiabilité a été abordée à partir de facettes autres que la sécurité, et considère en outre les éléments objectifs qui sont les fonctionnalités de sécurité du service, qui peuvent ensuite être évalués à l'aide d'attributs et de mesures. Pour atteindre cet objectif, les systèmes doivent d'abord fournir leurs fonctions de sécurité; ces éléments peuvent provenir de processus de certification et d'évaluations de sécurité par des tiers ou de la divulgation de certains détails internes d'un service par son fournisseur. Les fonctionnalités de service sont ensuite évaluées par rapport aux exigences de l'utilisateur exprimées en termes d'attributs. Entre autres contributions, le projet OPTET a conçu et développé un marché digne de confiance, afin d'exposer des services sécurisés, ainsi que des descriptions lisibles par machine de fonctionnalités et de fonctions de sécurité.

10.1.3 Les règles de Bonne Pratique et les Exigences de Sécurité

Les grandes entreprises suivent généralement un SSDLC (Secure Software Development LifeCycle) afin de réduire l'exposition des logiciels aux vulnérabilités. Beaucoup de variantes SSDLC existent et sont mises en œuvre par des organisations avec de légères différences, toujours dans le but d'améliorer la sécurité des logiciels.

En pratique, les entreprises créent des processus de révision et mènent des sessions d'audit [48] comprenant un large éventail d'expériences pour vérifier la conformité aux exigences générales et de sécurité. Cependant, les revues manuelles peuvent être longues et coûteuses pour les entreprises en termes de ressources, et elles peuvent ne pas détecter les violations des exigences.

La sécurité n'étant pas seulement un problème informatique, mais aussi un problème commercial, les entreprises accordent de plus en plus d'importance à la sécurité en adoptant des formations de sensibilisation à la sécurité à l'intention des développeurs et des gestionnaires. Les formations en sécurité technique aident à sensibiliser les développeurs aux risques ainsi qu'aux bonnes pratiques de programmation qui aident à réduire les risques et les coûts associés et à atténuer les graves

failles auxquelles les logiciels pourraient être soumis.

Des approches de programmation sécurisées telles que l'utilisation d'outils d'analyse de code statique peuvent également être appliquées dans les différentes phases du processus d'ingénierie logicielle. L'application d'une analyse de code statique ou d'une inspection manuelle de code permet de détecter les vulnérabilités au début de la phase de codage. Cette dernière peut être sujette aux erreurs et beaucoup plus longue que la révision automatique du code, en particulier lorsque tout le code source doit être analysé ou lorsque le code à inspecter est grand [25].

Dans le SSDLC, d'énormes efforts sont investis assez tardivement dans le processus d'ingénierie logicielle, comme l'application de tests de pénétration uniquement lorsque le logiciel est proche d'être livré au client. Cependant, corriger les bogues dans les derniers stades du développement est beaucoup plus coûteux pour l'entreprise, et c'est presque aussi cher que de corriger les bogues lorsque le logiciel est déjà livré aux clients, comme indiqué par le *Systems Sciences Institute* chez IBM: *"le coût de correction d'une erreur trouvée après la sortie du produit était quatre à cinq fois plus élevé que celui découvert lors de la conception, et jusqu'à 100 fois supérieur à celui identifié dans la phase de maintenance"* (Figure ref costFixBugs).

10.2 Les Règles de Bonne Pratique

10.2.1 Introduction

Les organisations et les entreprises définissent les exigences de sécurité non fonctionnelles à appliquer par les développeurs de logiciels, et ces exigences sont généralement abstraites et de haut niveau. Les exigences de sécurité telles que la confidentialité et l'intégrité sont abstraites, et leur application nécessite la définition de directives explicites à suivre pour satisfaire aux exigences. Les directives de sécurité décrivent les mauvaises et bonnes pratiques de programmation qui peuvent fournir des conseils et un soutien au développeur pour assurer la qualité de ses logiciels développés en ce qui concerne la sécurité et, par conséquent, réduire l'exposition du programme aux vulnérabilités lors de leur livraison. plate-forme client (sur site ou dans le cloud). Les mauvaises pratiques de programmation définissent les modèles de code négatif à éviter, ce qui peut conduire à des vulnérabilités exploitables, tandis que de bonnes pratiques de programmation représentent les modèles de code recommandés à appliquer sur le code.

La définition des lignes directrices de sécurité s'inscrit dans un autre contexte: les marchés de logiciels. Software Marketplaces utilise différentes approches afin d'éviter que des applications malveillantes soient annoncées dans leurs magasins officiels. Par exemple, Apple App Store [14] vérifie si les applications soumises re-

spectent les directives de révision Apple App Store [14], qui servent essentiellement de guide pour recueillir les recommandations que les développeurs doivent suivre avant de soumettre leurs applications. Les demandes soumises subissent généralement une vérification de leur conformité aux exigences de sécurité du marché. D'une part, il est important pour un fournisseur de services d'être conscient des exigences spécifiques à un domaine, car celles-ci ne sont pas acceptées sur le marché et les réécritures ultérieures d'applications peuvent être coûteuses, chronophages et affecter la réputation de l'organisation. D'un autre côté, l'utilisateur final veut être rassuré que ses exigences soient respectées en termes de sécurité et de confidentialité. Cependant, ce processus n'est ni toujours transparent ni compréhensible pour le fournisseur de services ainsi que pour l'utilisateur final. Du point de vue des fournisseurs de services / développeurs, la compréhension et l'interprétation des lignes directrices ne sont pas triviales, car aucune formalisation n'expose les instructions de programme nécessaires pour chaque directive, ou explique comment les appliquer correctement dans le logiciel.

Un autre guide de pratiques de programmation que nous pouvons considérer est par exemple la norme CERT Oracle Coding pour Java [39] [102]; Pour chaque directive, les auteurs fournissent une explication textuelle détaillée. Pour la plupart, des exemples de codes d'échantillons conformes et non conformes sont également fournis en plus de la description.

La suite de tests Juliet [112] est créée par la National Security Agency (NSA) et propose un ensemble de cas de test couvrant plusieurs langages de programmation, y compris Java. Les cas de test fournis sont classés en catégories par rapport au type de flux; contrôle ou des données, et chaque cas de test cible une vulnérabilité ou une faiblesse spécifique, en se référant aux entrées connues dans le Common Weakness Enumeration Dictionary (CWE).

Afin d'avoir une compréhension claire des consignes de sécurité, nous avons effectué une analyse approfondie des différentes sources officielles proposant des règles et des exemples de bonnes / mauvaises pratiques de programmation. Les sources étudiées comprennent OWASP [121], Oracle [116], CERT, NSA [112], NIST [113] et Apple [14], et seront discutées en détail dans le prochain section.

L'objectif principal de cette enquête est d'identifier les principaux problèmes liés à la présentation des lignes directrices aux développeurs et de cerner les principaux obstacles à leur vérification automatique sur le logiciel. Le deuxième objectif est de proposer une classification des consignes de sécurité en fonction du type de vérification à appliquer pour valider leur adhésion au logiciel.

10.2.2 Etudes de différentes Sources des Règles de Bonne Pratique

Dans cette section, nous présentons l'enquête que nous avons menée sur les différentes sources de directives de sécurité afin d'atteindre différents objectifs. D'abord, identifier les principales difficultés qui peuvent survenir lors de l'interprétation de la description textuelle des lignes directrices. Deuxièmement, pour cerner un autre problème: les lignes directrices sont fournies par différentes sources, et il n'existe pas de modèle de présentation commun permettant une interprétation homogène.Pour une compréhension concrète, nous considérons quelques lignes directrices réelles telles que présentées dans différentes sources telles que CERT [39], Guide du développeur Apple App Store [14], Juliet Test Cases [112] et OWASP [119]. Chaque ligne directrice est indiquée par un code unique attribué par l'organisation émettrice. Les différents points sont discutés en détail dans les différentes sections de ce chapitre.

10.2.2.1 OWASP

The OWASP Foundation [121] introduces different sets of guidelines and rules to be applied in order to protect sensitive information. The Secure Coding Practices guide[121] is a set of good programming practices that are presented in a checklist format arranged into classes, like Database Security, Communication Security, etc. The listed programming practices are general, in a sense that they are abstract and are not tied to a specific programming language. The same source, OWASP [119] provides different sets of rules arranged with respect to the specific entity that they seek to protect (password, encryption key) and to the security mechanisms to be implemented (protocols, data storage, etc.). The guides proposed by OWASP include the **Cryptographic Storage Cheat sheet** that provides a set of guidelines to be applied in order to protect data at rest; for each rule, OWASP provides recommendations related for instance to the cryptographic protocol to be applied, to the minimum key length to be used, etc.

10.2.2.2 CERT

CERT [6] est une division de l'Institut de génie logiciel de recherche et développement en cybersécurité (SEI) [144]. Il vise à améliorer la sécurité de l'environnement cyber en fournissant des solutions de recherche et développement innovantes. La mission principale est de réduire l'exposition aux vulnérabilités d'une manière permettant d'atténuer les défauts détectés pendant le développement et les tests. Les normes de codage CERT ont été adoptées par des entreprises telles qu'Oracle et

Cisco. La division CERT propose un ensemble de consignes de sécurité classées d'abord par le langage de programmation auquel elles s'appliquent (Java, C, C ++, etc.). Ces directives sont classées en catégories en fonction du type de vérification et des actifs à protéger (validation des entrées, sécurité par mot de passe, etc.). Pour chaque ligne directrice, les auteurs fournissent une description textuelle détaillée expliquant la ligne directrice, et les risques qui peuvent être rencontrés lorsque la ligne directrice n'est pas correctement appliquée. Pour la plupart, il y a aussi des exemples de codes d'échantillons conformes et non conformes se référant aux modèles positifs et négatifs. De plus, les auteurs maintiennent pour chacune des lignes directrices une évaluation des risques fournissant la probabilité de l'occurrence de la violation, ainsi que sa gravité et les coûts de remise en état associés. Étant donné que chaque langage de programmation a ses spécificités, le CERT fournit un ensemble de lignes directrices de sécurité qui diffèrent d'un langage de programmation à l'autre, comme par exemple la gestion de la mémoire qui est propre au langage C.

10.2.2.3 NIST

L'Institut National des Normes et de la Technologie (NIST) fournit différents ensembles de règles et de normes qui servent de guide pour la sécurité économique des systèmes d'information. Dans [113], des recommandations de gestion de clés cryptographiques sont fournies. Le guide se concentre sur des recommandations sur la façon d'utiliser correctement les mécanismes cryptographiques, en mettant l'accent sur la gestion des clés de cryptage, en tenant compte de leur force, de leur génération sécurisée, stockage, distribution, utilisation et destruction. Ce guide est divisé en trois parties principales par rapport au public cible et au niveau technique. La première partie rassemble des recommandations sur la gestion des clés de base, en mettant l'accent sur les aspects de mise en œuvre technique. D'autre part, la partie 2 est axée sur la simplification des concepts d'établissement de la gestion des clés cryptographiques au sein d'une organisation. Cette partie contient des recommandations pour les propriétaires et les gestionnaires de système. L'objectif principal de la partie 3 est de couvrir la mise en œuvre et l'utilisation de règles spécifiques. Chaque description de règle inclut des indications sur les algorithmes recommandés et les longueurs de clé. Il comprend également des recommandations sur l'utilisation efficace des processus de gestion des clés existants.
Le NIST fournit des règles générales dans le sens où elles ne répondent pas à un besoin spécifique ou ne tiennent pas compte d'un langage de programmation spécifique, comme c'est le cas pour le CERT. Les règles du NIST fournissent des recommandations qui, si elles sont appliquées correctement, garantiraient une sécurité

renforcée par rapport aux mécanismes appliqués.

10.2.3 Classification des Règles de Bonnes Pratiques

Dans un premier temps, nous allons organiser les directives en groupes en fonction du type de vérification qu'elles considèrent. Par exemple, une catégorie de lignes directrices peut être plus axée sur la validation des données fournies en entrée. Nous pouvons également envisager une autre catégorie pour la sécurité secrète, à savoir les données dont la violation d'intégrité peut entraîner une perte de confidentialité des informations sensibles. La sécurité des informations sensibles est une autre catégorie que nous définissons, et consiste à regrouper les directives pour les informations sensibles, telles que les informations privées de l'utilisateur. Nous avons également décrit des catégories qui traitent de l'invocation de méthodes spécifiques et de la sécurité de l'environnement d'exécution. Pour chaque ligne directrice, nous fournissons des explications, des commentaires et des observations.

10.2.4 Discussion

Les directives de sécurité visent principalement à simplifier le travail du développeur en améliorant la qualité du logiciel du point de vue de la sécurité. Cependant, si nous regardons de plus près les directives présentées dans la section précédente, nous remarquerons que leur compréhension et leur mise en œuvre ne sont pas triviales pour le développeur. Considérer seulement les événements ou les actions issus de l'angle de flux de contrôle pur n'est pas suffisant pour les directives qui dépendent fortement de la propagation d'information à travers un programme, comme c'est le cas pour les directives IDS03-J-IDS15-J. Par conséquent, considérer le flux d'informations et les intégrer dans la spécification formelle des directives de sécurité devient impératif.

Lors de l'interprétation des lignes directrices, le développeur est confronté à un autre problème lié à l'interprétation sémantique de mots-clés tels que la notion de données sensibles, données sécurisées / non fiables, limite de confiance, vérification de sécurité, validation incorrecte, etc. Il devrait être capable d'identifier les variables sensibles ou les données dans son logiciel développé, le problème devient encore plus difficile quand c'est le testeur / réviseur de code qui doit effectuer cette identification sur un programme développé. C'est également le cas pour les marchés de logiciels lors de l'exécution du processus d'approbation. Dans les lignes directrices que nous avons recueillies à partir des différentes sources présentées ci-dessus, nous notons la redondance de la notion de données fiables ou non fiables, ambiguë à interpréter, à comprendre et à appliquer au niveau du code source. Dans la source CERT, les auteurs fournissent un glossaire pour expliquer davantage les

différents mots-clés et termes utilisés dans les directives. Par exemple, "*données non fiables*" est défini comme des données provenant de l'extérieur d'une frontière de confiance, ce qui conduit à une incertitude quant à l'identification et à la détermination de la notion de frontière de confiance; fait-il référence à l'entrée / sortie du système? Il apporte à la table un autre problème consistant à déterminer le périmètre d'un programme. Comment déterminer les limites du système dans le contexte d'une application distribuée?

L'opération de validation est utilisée dans plusieurs directives, par exemple dans IDS01-J et 124683. Comme on peut le remarquer, la validation de l'index de tableau n'est pas la même opération de validation du point de vue des instructions du programme. Un autre élément a attiré notre attention. la ligne directrice ou la famille de directives recommandant de crypter ou de hacher les informations sensibles. Cette opération connue sous le nom de déclassification est largement connue dans le vocabulaire du flux d'information [133]. L'opération de déclassification peut être considérée comme une libération contrôlée d'informations sensibles en utilisant des mécanismes et des techniques donnés tels que le cryptage, le hachage, l'obfuscation, etc. Certaines règles n'ont pas le même niveau d'admissibilité, dans un sens si elles sont appliquées en même temps, cela pourrait conduire à des ambiguïtés. Par exemple, les directives IDS06-J et IDS03-J; la première règle interdit l'entrée de l'utilisateur à partir de la chaîne de format, tandis que la seconde exige de désinfecter l'entrée de l'utilisateur. Cela amène à la table la nécessité de considérer les relations et les dépendances entre les lignes directrices. Il est important de noter que toutes les catégories spécifiées sauf *Sécurité secrète* sont axées sur la confidentialité des données, et manquent d'une certaine façon l'intégrité qui est selon Biba [23] le duel de la confidentialité. L'intégrité contrairement à la confidentialité, peut être violée sans aucune interaction avec des composants externes au système. C'est la principale raison de considérer la catégorie Sécurité secrète comme partie intégrante de notre classification. Les différentes sources de directives de sécurité souffrent de différents problèmes, tels que:

* La description est informelle, qui peut être sujette à une mauvaise interprétation
* La description est abstraite, ce qui peut conduire à des ambiguïtés
* Les lignes directrices sont contextuelles
* Les directives d'application diffèrent des instructions du système

En examinant les directives de sécurité provenant des différentes sources, nous avons remarqué un manque de précision et une absence totale d'automatisation. Comme on peut le constater, la façon dont les recommandations sont présentées est très vague et abstraite, et il n'y a aucune précision sur les mécanismes ou les

flux d'actions recommandés qui devraient être mis en place pour respecter la règle.

Nous avons également remarqué que la directive de sécurité "Ne pas enregistrer les entrées utilisateur non nettoyées" de la norme de codage CERT semble être décomposée en deux lignes directrices dans la source OWASP: "Injection de log" et "Validation d'entrée". Cela a apporté une autre dimension sur laquelle nous pouvons proposer des améliorations sur les lignes directrices; la compositionnalité et la composabilité des directives de sécurité. Nous voulons identifier un autre élément clé qui a retenu notre attention; il y a un énorme effort investi pour construire et maintenir les catalogues, mais aucune tentative n'a été entreprise pour instrumenter leur vérification automatique au niveau du code. OWASP fournit des ensembles de directives de sécurité qui devraient être respectées par les développeurs, mais ne fournit pas les moyens d'assurer leur mise en œuvre correcte. Nous visons à combler cette lacune à travers la spécification formelle des directives de sécurité et leur vérification formelle à l'aide de preuves formelles.

10.2.5 Conclusion

Dans ce chapitre, nous avons proposé une première tentative de classification des directives de sécurité recueillies auprès de différentes sources en catégories. Nous avons souligné un autre problème majeur avec les directives, consistant à ne considérer que les systèmes *closed*, et comment ils ne parviennent pas à traiter les systèmes distribués et la complexité des directives qu'ils induisent. La complexité peut être perçue à partir de la difficulté de définir les limites du système pour les systèmes distribués et à trois niveaux. Nous avons souligné les problèmes qui surgissent lors de l'interprétation et de la mise en œuvre des lignes directrices: ils manquent de précision et peuvent être sujets à des interprétations erronées de la part des développeurs. Nous discutons dans les prochains chapitres de la manière dont nous avons travaillé pour combler le fossé entre l'aspect informel des lignes directrices, et leur mise en œuvre et leur application à travers un formalisme. cela permet de supprimer les ambiguïtés. Nous exprimons ces directives dans un langage formel et motivons leur satisfaction. En ce qui concerne les trois premiers critères, à savoir le contrôle, les données et la dactylographie, ils devraient être traduits automatiquement en langage formel. Cependant, pour ceux impliquant le critère sémantique, la formalisation doit être menée par un expert conscient du domaine de la sécurité logicielle, l'aspect sémantique étant fortement lié au contexte.

10.3 Approche pour la Vérification Automatique des Règles de Sécurité

10.3.1 Introduction

Les consignes de sécurité sont conçues pour aider les développeurs à créer des logiciels sécurisés. Comme nous l'avons souligné dans le chapitre précédent, les directives sont informelles et présentent des ambiguïtés qui peuvent conduire à une mauvaise interprétation par les développeurs. Nous soulignons également un autre élément clé qui souligne le fait qu'un énorme effort est investi pour établir et maintenir les catalogues de directives, mais, à notre connaissance, aucun outil automatique assurant la vérification de ces directives au niveau du code n'a été proposé. On pourrait argumenter sur l'utilisation de schémas de certification tels que *Common Criteria* [3] pour évaluer les aspects de sécurité des logiciels, au lieu d'appliquer l'approche que nous proposons et dont nous discutons en détail dans ce chapitre. Comme nous l'avons souligné dans le chapitre précédent, le processus de certification est coûteux et long. En outre, il ne fournit aucune garantie sur le comportement du logiciel dans l'environnement de production. Le point que nous essayons de faire est, notre approche qui sera discutée en détails dans ce chapitre, peut être considérée comme un complément à la certification, car nous opérons sur le **niveau de code**, et la vérification que nous effectuons est **automatique** et **rentable**. En outre, nous visons à sensibiliser les développeurs aux questions de sécurité et à les aider à prendre en compte la conformité à la sécurité pendant la phase de développement.

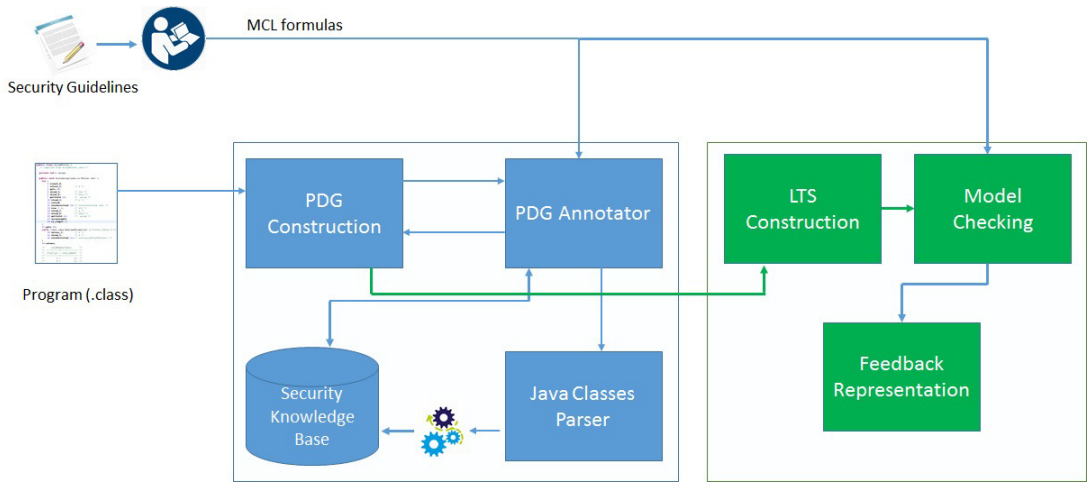


Figure 10.1: Approche pour la spécification formelle et la vérification automatique des règles de sécurité

10.3.2 Spécification Formelle des Règles de Sécurité

Cette étape constitue une partie fondamentale de ce travail de thèse, et le travail principal qu'il accomplit est de combler le fossé entre la description informelle des directives de sécurité, et leur formalisation dans un formalisme basé sur les mathématiques. Comme les directives expriment le comportement souhaité et non désiré du programme, nous avons jugé approprié d'adopter le formalisme temporel. Nous faisons l'hypothèse forte que le **expert (s) de sécurité**, Brian, spécifie formellement les directives de sécurité en extrayant les éléments clés de la description informelle, et construit sur celles-ci les formules basées sur le formalisme choisi. Nous fournissons de l'aide et des conseils à Brian par la formalisation des lignes directrices au moyen des modèles que nous mettons à la disposition des experts en sécurité. Les directives de sécurité seront modélisées sous la forme d'une séquence de propositions atomiques ou d'énoncés représentant le comportement du programme. Les formules construites peuvent être prises en charge par des outils de vérification de modèle standard pour effectuer la vérification automatique. Le résultat de cette phase est des directives de sécurité génériques qui peuvent être instanciées sur différents programmes. Par exemple, l'étiquette d'action save, qui définit l'opération de sauvegarde d'une donnée dans un emplacement spécifique, peut être instanciée sur save_to_DB, save_to_file, save_to_array, etc, en fonction de l'instruction invoquée et de ses paramètres. Cette opération d'instanciation est également gérée dans notre base de connaissances de sécurité (SKB). Nous avons fait l'effort d'automatiser la spécification formelle des règles de sécurité à travers une approche que nous avons établie.

10.3.3 Construction du Modèle de Programme

Considérant que les directives sont formalisées dans un formalisme basé sur les mathématiques, elles doivent être vérifiées sur un modèle de programme qui supporte le formalisme choisi. Comme indiqué ci-dessus, le modèle de programme devrait être capable de représenter l'ensemble du flux d'informations et de se rapprocher du comportement du programme. Comme le point de départ de cette étape est la source du programme et les codes d'octets, nous construisons un modèle de programme qui capture les différentes dépendances qui pourraient survenir entre les instructions du programme. Les dépendances désignent les dépendances de données directes et transitives, c'est-à-dire les assignations explicites d'informations sensibles à des variables publiques, ou implicites qui peuvent se produire dans des conditions de branchement et influencer le flux de contrôle dans un programme. La notion de flux d'information implicite est également définie comme étant une classe plus grande; canaux cachés [23]. Nous avons étudié différents modèles de représen-

tation de programme [161], y compris le diagramme de flux de contrôle (CFG), le diagramme de flux de données (DFG), l'arbre de syntaxe abstraite (AST), etc. modèle d'abstraction le plus approprié qui représente à la fois le contrôle et les dépendances de données, et correspond à nos besoins en termes d'approximation des flux d'information au sein du programme. Nous sommes conscients que PDG est une sur-approximation du comportement du programme, et cela peut conduire à **fausses alarmes**, d'où l'imprécision de l'analyse. De plus, PDG est **non formel**, ce qui constitue un autre obstacle concernant l'utilisation de PDG comme support pour effectuer la vérification formelle.

Afin d'augmenter la précision de l'analyse et de réduire les fausses alarmes, nous avons d'abord transformé le PDG standard en un autre PDG que nous avons appelé "PDG Augmenté". Le terme "augmenté" vient de l'effort que nous avons effectué pour détecter les dépendances implicites sur le PDG, et d'autres détails qui ne sont pas capturés en premier lieu par le PDG. Nous augmentons le PDG à travers l'Information Flow Analysis pour capturer les dépendances explicites et implicites qui peuvent être source de canaux cachés, et peuvent constituer une source de fuite d'informations sensibles. Pour aborder le second problème, nous soulignons que le PDG augmenté est utilisé comme une représentation de programme intermédiaire. Nous générons ensuite à partir de ce PDG augmenté un modèle formel, une représentation de la structure du Système de Transition Étiqueté qui est supportée par des outils de vérification de modèle.

10.3.4 Vérification Formelle

Puisque nous visons à fournir une vérification automatique des lignes directrices sur le programme, nous avons jugé nécessaire d'adopter la vérification du modèle comme approche de vérification formelle. Cette opération exploite les directives formalisées et le modèle de programme formel, et consiste à prouver l'exactitude du programme par rapport aux directives spécifiées. La vérification formelle vise en premier lieu à vérifier l'exactitude des systèmes critiques pour la sécurité. Et à ce niveau, nous affirmons que la vérification formelle que nous utilisons dans ce travail de thèse est générique, en ce sens qu'elle peut couvrir différents cas d'utilisation, allant de la sécurité à la sécurité.

La vérification du modèle [73] explore essentiellement l'espace d'état qui décrit les comportements possibles du programme. Comme indiqué dans la section précédente, l'espace d'état fait référence au modèle de programme formel que nous générons automatiquement à partir du PDG augmenté. Un outil de vérification de modèle accepte un modèle de système spécifié dans un formalisme donné et les propriétés du système qui doivent être vérifiées par le système. L'outil génère ensuite

une réponse booléenne indiquant si la propriété a été satisfaite ou non. Une sortie *yes* indique si le modèle donné satisfait à la spécification, sinon un contre-exemple est généré. Le contre-exemple fournit une explication détaillée de la violation survenue. Un examen approfondi de l'exemple de compteur permet de localiser la source de la violation dans le modèle de programme et d'indiquer éventuellement comment la réparer.

Dans le premier cas, le résultat de la vérification indique que la recommandation est valide lors de l'exploration de l'ensemble du flux d'informations dans le programme. Le deuxième cas peut être avancé, ce qui signifie que la vérification peut fournir plus de détails au développeur Alex (ou au testeur) sur les circonstances dans lesquelles la directive de sécurité n'a pas été respectée, et des contre-exemples sont produits. Ici, nous insistons sur les commentaires qui devraient être précis et faciles à comprendre par Alex le développeur.

10.3.5 Base de Connaissances Sécurité

La Base de Connaissances Sécurité ou Security Knowledge Base est un référentiel centralisé rassemblant les étiquettes des formules mappées aux API, instructions, bibliothèques ou programmes. Cela facilite la détection automatique des étiquettes sur le modèle du système. Nous avons construit Security Knowledge Base en utilisant un analyseur de classes Java [157] qui fonctionne comme suit: pour les différentes classes Java utilisées dans le programme à analyser, nous lançons l'analyse de cette classe donnée (code html, javadoc) , et nous extrayons tous les détails pertinents, tels que la description, les attributs, les constructeurs, les signatures de méthodes et leurs paramètres. Ensuite, nous avons fait l'effort d'effectuer une analyse sémantique semi-automatique pour détecter les éléments clés, tels que le mot-clé *sécurisé*, *clé*, *print*, *input*, etc. l'opération est d'une importance primordiale, car elle permet de mapper les mots clés utilisés pour construire les formules, aux instructions possibles en langage Java (invocations de méthodes, invocations de constructeurs, déclarations de types de données spécifiques, etc.). Par exemple, l'API Java *KeyGenerator.generateKey ()* est mappée sur l'étiquette *isKey*. Cette étiquette est également mappée à l'annotation traditionnelle de flux d'informations extbf high level **source**.

10.3.6 Impact de l'Analyse du flux d'Informations sur la Vérification des Règles de Sécurité

Contrôler la façon dont l'information circule à travers un programme est d'une importance primordiale lorsqu'il s'agit de la sécurité de l'information. D'un point de vue historique, les mécanismes de contrôle d'accès [22], comme leur nom l'indique,

sont utilisés pour vérifier les droits d'accès au point d'accès. Ensuite, ils accordent ou refusent l'accès à l'actif sur lequel le mécanisme est défini. De même, les mécanismes de cryptage, qui peuvent assurer la confidentialité des données, ne permettent pas de suivre ou de contrôler le flux d'informations dans un programme. Les mécanismes de contrôle d'accès, de la même manière que le chiffrement, ne permettent pas de savoir où et comment les données se propageront, où elles seront stockées ou où elles seront envoyées ou traitées. Cela implique le besoin de contrôler le flux d'informations en utilisant une analyse de code statique. Cette même idée est soulignée par Andrei Sabelfeld, et Andrew C. Myers [131], qui jugent nécessaire d'analyser comment l'information circule dans le programme. Selon les auteurs, un système est considéré comme sécurisé en ce qui concerne la confidentialité de la propriété, si le système dans son ensemble assure cette propriété.

10.3.7 Conclusion

Dans ce chapitre, nous avons présenté à un niveau élevé l'approche que nous proposons dans le but de combler le vide pour la vérification des lignes directrices de sécurité. Nous identifions des problèmes différents avec les directives de sécurité qui sont présentes dans différentes sources, mais aucun moyen de vérification n'est fourni aux développeurs pour s'assurer que les logiciels développés respectent ces directives. Les consignes de sécurité s'adressent principalement aux développeurs, mais la façon dont elles sont présentées présente des ambiguïtés, ce qui peut conduire à des interprétations erronées. Formaliser les lignes directrices aiderait à éliminer les ambiguïtés et à préparer le terrain pour la vérification formelle. Ceci sera discuté en détails dans le chapitre suivant. Nous avons insisté sur la nécessité d'effectuer une vérification de modèle en tant qu'approche de vérification. Cela permet d'avoir une vérification automatique, donc de réduire l'intervention d'un opérateur humain, que ce soit le développeur ou l'expert en sécurité pour mener cette vérification.

10.4 Spécification Formelle des Règles de Bonnes Pratiques

10.4.1 Introduction

Transformer la description informelle des directives de sécurité en formules mathématiques exploitables est un élément important de ce travail de thèse. Nous avons souligné dans le chapitre "Les Règles de Bonne Pratique" les principaux problèmes que nous avons identifiés en ce qui concerne les consignes de sécurité et les difficultés qui peuvent survenir lorsqu'elles sont interprétées et implémentées par des

développeurs logiciels et vérifiées ensuite par des testeurs ou des auditeurs de code. De plus, les présentations des directives de sécurité diffèrent d'une source à l'autre, d'où la nécessité d'une approche centralisée et uniforme pour les représenter. Dans ce chapitre, nous présentons la méthodologie que nous fournissons à l'expert en sécurité afin de l'aider à construire les formules de directives. Nous nous concentrons sur l'extensibilité ainsi que sur l'automatisation que nous pouvons apporter à cette phase pour éliminer les difficultés auxquelles l'expert en sécurité peut faire face lors de la construction des formules de directives, en particulier celles qui impliquent des flux de données dépendants. Nous expliquons en détail le flux des opérations et mettons en évidence les efforts que nous avons déployés pour automatiser la formalisation des directives de sécurité. Pour une compréhension claire du flux de spécification, nous considérons des exemples concrets de directives que Brian, l'expert en sécurité, officialise, à partir de leur description informelle.

L'idée principale de ce chapitre est de transformer les directives de sécurité écrites en langage naturel en formules mathématiques exploitables. L'avantage de cette formalisation est de supprimer les ambiguïtés du langage naturel et d'augmenter la précision. Cette phase produit une spécification formelle des consignes de sécurité.

10.4.2 Formalisme pour le Model Checking

Puisque les directives que nous voulons spécifier reflètent le comportement du programme désiré, nous pouvons conclure que ce qui est nécessaire est un formalisme qui permet de décrire l'exécution du programme. Ces directives sont généralement abstraites dans les modèles formels car les problèmes de vérification sont indécidables pour les systèmes infinis.

Adopter des Logiques Temporelles [105] semble être une sage direction car elles permettent de spécifier avec une sémantique rigoureuse les propriétés désirées sous la forme de séquences de propositions atomiques ou d'énoncés représentant le comportement du système. Dans la littérature, différentes logiques temporelles ont été étudiées et proposées pour la spécification des propriétés du programme. "Temporel" fait référence au séquençage et à l'ordre chronologique des états, et non à la notion de temps physique.

Différents formalismes Logiques temporels existent et sont largement adoptés, tels que LTL et CTL, avec leurs différentes variantes. La syntaxe de la logique temporelle linéaire (LTL) [125] est définie de manière inductive en utilisant les opérateurs booléens standard, et les opérateurs temporels X (suivant), F (finalement), G (toujours OU globalement), et U (strong until) par l'équation de syntaxe abstraite suivante:

$$\varphi := p \mid \neg p \mid p \wedge q \mid \mathbf{X}p \mid \mathbf{G}p \mid \mathbf{F}p \mid p \,\mathbf{U}\, q$$

Par exemple, considérons une propriété simple et écrivons la formule LTL correspondante. Prenons l'exemple connu d'un four à micro-ondes, et l'une de ses propriétés: "Pas de chaleur tant que la porte est ouverte". Cette propriété est une propriété de sécurité, donc devrait commencer par l'opérateur G (toujours). Il indique également que si la porte du micro-ondes est ouverte, elle ne peut pas chauffer. Cela peut être exprimé dans cette formule LTL:

$$\mathbf{G}(Heat \rightarrow Close)$$

Les prédicats que nous avons utilisés dans cette propriété sont: Heat et Close, et ils se réfèrent à des actions. Ces actions, comme spécifié dans la formule, doivent se produire dans des conditions spécifiques. Par exemple, si le micro-ondes est en train de chauffer (chaleur), cela signifie que sa porte est fermée (fermeture).

Maintenant, si nous considérons un autre exemple commun: "*No écrit sur un fichier après qu'il est fermé*". Cette propriété indique qu'une fois qu'un fichier est fermé, nous ne pouvons pas écrire dessus. En d'autres termes, nous devons contraindre l'opération d'écriture sur un fichier: nous interdisons si le fichier est fermé, et l'autorisons une fois qu'il est ouvert. Les différentes actions (étiquettes) que nous pouvons utiliser pour construire la formule LTL sont:Write, Open et Close, et se réfèrent respectivement aux actions d'écriture sur le fichier, ouvrant le fichier et le fermer. Cependant, ces actions fonctionnent sur un paramètre qui est le *fichier*. Si nous écrivons la formule LTL de manière similaire à l'exemple précédent, nous aurions manqué la caractéristique paramétrée des actions. "Fichier" n'est pas une action, mais plutôt un type de données sémantique attaché à une ressource spécifique. Ceci ne peut pas être exprimé en utilisant la syntaxe de LTL: les actions Open, Write et Close sont effectuées sur ce fichier et le LTL ne décrit pas les actions paramétrées avec des données.

$$\mathbf{G}(Close \rightarrow (\neg Write \mathbf{U}\ Open))$$

Ce formalisme est insuffisant, car il ne gère pas le passage de valeur. Il permet de raisonner sur la progression des calculs et des actions, mais pas sur les données manipulées Puisque les directives de sécurité impliquent le passage de données en plus du flux de contrôle des instructions, nous avons besoin d'un formalisme puissant qui couvre ces aspects et qui est enrichi de mécanismes de passage de valeurs.Nous présentons dans la section suivante le langage de Model Checking (MCL).

10.4.3 Langage de Model Checking (MCL)

Dans cette section, nous expliquons plus en détail le langage MCL (Model Checking Language) que nous avons adopté pour la spécification formelle des directives de sécurité. Le formalisme que nous avons utilisé est très intuitif en ce sens que les formules de construction sont proches du traitement naturel des textes. Nous adoptons la logique MCL [103] qui traite de cette question cruciale de la représentation et du traitement des données. Il permet de raisonner naturellement sur les systèmes décrits dans les algèbres de processus de passage de valeurs telles que LOTOS.

MCL (Model Checking Language) est une extension du mu-calcul régulier sans alternance avec des facilités pour manipuler les données d'une manière cohérente avec leur utilisation dans la définition du système. Les formules MCL sont des formules logiques construites sur des expressions régulières utilisant des opérateurs booléens, des opérateurs de modalité (l'opérateur de nécessité noté ([ ]) et l'opérateur de possibilité noté ( ) et l'opérateur de point fixe maximal (noté mu).

10.4.4 Construction de la Spécification Formelle des Règles de Sécurité dans le Langage de Model Checking (MCL)

Après avoir présenté le formalisme que nous adoptons pour spécifier les consignes de sécurité, nous examinons son applicabilité sur un certain nombre de lignes directrices que nous avons considérées. Avant de procéder à la spécification formelle, nous expliquons la méthodologie que nous fournissons à l'expert en sécurité pour construire les formules mathématiques MCL à partir des descriptions des lignes directrices.

Le flux de formalisation est le suivant:

* Extraction des mots-clés (patterns d'action) et des concepts-clés de la description textuelle de la ligne directrice
* Construction de la formule de base MCL
* Construction de la formule MCL impliquant des dépendances explicites et implicites

Chaque étape ainsi que la transition d'une étape à l'autre sont décrites en détail dans les sections suivantes. Le vrai challenge est de définir l'alphabet, c'est à dire trouver la bonne notation des actions La formalisation proposée dans le langage MCL présente les concepts clés requis par chacune des lignes directrices. Le défi consiste à définir ces concepts et actions et à disposer d'un ensemble de mots-clés unifié mais extensible pouvant être utilisé par les experts en sécurité pour formaliser les directives. Nous avons essayé de réduire l'intervention des experts en

introduisant la base de connaissances de sécurité (voir la section ref SKB) qui est un référentiel central contenant les différentes étiquettes utilisées pour composer les lignes directrices, ainsi que leur description et leur signification sémantique

10.4.5 Modèles pour la Spécification Formelle des Règles de Sécurité

Lors de la formalisation des consignes de sécurité décrites dans la section précédente, nous avons remarqué qu'il existe des modèles récurrents. Par exemple, la notion de *precedence* apparaît dans différentes directives et contraint l'ordre des actions. Par exemple, dans la directive IDS03-J: "Ne pas enregistrer l'entrée d'utilisateur non nettoyée", l'opération de nettoyage (Sanitization) sur l'entrée utilisateur doit avoir lieu avant l'action de stockage (Logging). De même, la directive MSC62-J:"Stocker les mots de passe à l'aide d'une fonction de hachage" nécessite que le mot de passe soit haché avant son stockage (Logging). Un autre modèle de propriété que nous avons capturé est la notion d' "absence", ce qui signifie qu'une action ne devrait jamais se produire lorsqu'une autre action est exécutée. Par exemple, dans la directive MSC03-J:"Ne jamais stocker les informations sensibles en clair": le stockage dans cette directive ne devrait jamais avoir lieu.

Puisque nous visons à apporter de l'aide et des conseils à l'expert en sécurité, nous avons décidé de simplifier davantage la formalisation des lignes directrices en matière de sécurité et de proposer des modèles de propriété. Grâce à ceux-ci, l'expert en sécurité n'aura probablement pas besoin d'avoir une connaissance complète de l'expressivité formelle du langage ni de maîtriser ses idiomes. Il préférerait éventuellement avoir plus de conseils sur la façon d'utiliser les fonctionnalités du langage. Cela améliorerait la transformation des directives de sécurité du langage naturel aux formules.

Tout comme le travail de Dywer [53], nous proposons un catalogue de modèles pour les directives de sécurité. Ces modèles identifiés dans [103] permettront aux programmeurs qui ne sont pas nécessairement des experts en langage formel de lire une spécification formelle. Comme proposé dans [103], nous codons les directives comme une collection de modèles de propriétés destinés à simplifier l'activité de spécification. Ces modèles appartiennent à des classes particulières: *Absence*, *Existence*, et *Universalité* et se produisent dans diverses étendues: *Avant*, *Après* et *Entre*. Ces classes reflètent des exigences spécifiques telles que l'absence d'action, la précédence des actions, la réponse d'une action déclenchée par une autre action, etc. Les modèles proposés ont été adoptés par Bandera Specification Language [46].

Table 10.1: Modèles pour les règles de sécurité

Pattern	Scope	Formula
Absence (α_1 is `false`)	Globally	$[\texttt{true}*.\alpha_1]\texttt{false}$
	Before α_2	$[(\neg\alpha_2)*.\alpha_1.\texttt{true}*.\alpha_2]\texttt{false}$
	After α_2	$[(\neg\alpha_2)*.\alpha_2.\texttt{true}*.\alpha_1]\texttt{false}$
	Between α_2 and α_3	$[\texttt{true}*.\alpha_2.(\neg\alpha_3)*.\alpha_1.\texttt{true}*.\alpha_3]\texttt{false}$
	Between α_2 until α_3	$[\texttt{true}*.\alpha_2.(\neg\alpha_3)*.\alpha_1]\texttt{false}$
Existence (α_1 becomes `true`)	Globally	$\mu\texttt{Y}.\langle\texttt{true}\rangle\texttt{true} \wedge [\neg\alpha_1]\texttt{Y}$
	Before α_2	$[(\neg\alpha_1)*.\alpha_2]\texttt{false}$
	After α_2	$[(\neg\alpha_2)*.\alpha_2]\mu\texttt{Y}.\langle\texttt{true}\rangle\texttt{true} \wedge [\neg\alpha_1]\texttt{Y}$
	Between α_2 and α_3	$[\texttt{true}*.\alpha_2.(\neg\alpha_1).\alpha_3]\texttt{false}$
	After α_2 until α_3	$[\texttt{true}*.\alpha_2]([(\neg\alpha_1).\alpha_3]\texttt{false} \wedge \mu\texttt{Y}.\langle\texttt{true}\rangle\texttt{true} \wedge [\neg\alpha_1]\texttt{Y}$
Universality (α_1 is `true`)	Globally	$[\texttt{true}*.\alpha_2]\texttt{false}$
	Before α_2	$[(\neg\alpha_2)*.\neg(\alpha_1 \vee \alpha_2).(\neg\alpha_2)*.\texttt{true}*.\alpha_2]\texttt{false}$
	After α_2	$[(\neg\alpha_2)*.\alpha_2.\texttt{true}*.\neg\alpha_1]\texttt{false}$
	Between α_2 and α_3	$[\texttt{true}*.\alpha_2.(\neg\alpha_3)*.\neg(\alpha_1 \vee \alpha_3).\texttt{true}*.\alpha_3]\texttt{false}$
	After α_2 until α_3	$[\texttt{true}*.\alpha_2.(\neg\alpha_3)*.\neg(\alpha_1 \vee \alpha_3)]\texttt{false}$

10.4.6 Validation de la Formalisation

La spécification formelle des directives de sécurité est une opération cruciale et fondamentale dans notre cadre. Nous avons insisté sur le fait que nous nous appuyons sur l'expertise d'un expert en sécurité pour réaliser cette spécification formelle. Néanmoins, nous avons essayé de réduire les frais généraux de l'expert en sécurité et de proposer des modèles formellement éprouvés et validés auxquels il peut se référer lors de la formalisation des consignes de sécurité. Ces modèles identifiés dans [103] permettent aux programmeurs qui ne sont pas experts dans le langage formel MCL de lire et d'écrire des spécifications formelles. A ce niveau, nous ne pouvons pas fournir de preuves sur l'exactitude de la formalisation car nous n'avons pas de référence de modèle contre laquelle nous pouvons vérifier l'exactitude. Cependant, nous pouvons utiliser les codes conformes et non-réclamés fournis pour les

différentes directives dans la norme de codage CERT [39] par exemple comme un moyen de valider les formules établies. Il est vrai que cela ne garantit pas une détection complète de tous les détails subtils que nous pouvons déduire de la description de la ligne directrice, mais cela aiderait l'expert en sécurité à mieux comprendre la ligne directrice et une vision plus large des codes d'échantillons concrets. L'expert en sécurité peut alors modifier la formule MCL en ce qui concerne les nouveaux aspects qui peuvent survenir lors de la validation de la formule sur les modèles positifs et négatifs qui sont fournis comme exemples dans les catalogues de directives de sécurité. Néanmoins, valider la spécification auprès d'une communauté d'experts en sécurité peut être un moyen d'assurer l'exactitude des formules proposées.

10.5 Construction du modèle de programme et vérification du modèle

10.5.1 Introduction

La sécurité est une propriété cruciale que le comportement du programme doit respecter. Les mécanismes de sécurité standard, tels que le contrôle d'accès ou le cryptage, ne couvrent pas un problème fondamental: le suivi des informations tout au long du programme et la garantie que les informations sensibles n'ont pas été divulguées. On pourrait discuter de l'utilisation de la surveillance de l'exécution pour atteindre cet objectif. Cette pratique raisonne sur une seule exécution à la fois, et puisque les politiques de flux d'information requièrent en général un contrôle de tous les chemins d'exécution possibles, les stratégies de flux d'informations dynamiques ne peuvent pas traiter le suivi du flux d'information. Comme nous visons à fournir un feedback clair et précis au développeur, nous devons opérer au niveau du code et effectuer une analyse de code statique dans le but de vérifier les directives. Nous avons jugé nécessaire d'adopter une analyse statique des flux d'informations. L'analyse de code statique fonctionne sur une représentation du programme à analyser, qui est fondamentalement un modèle d'abstraction qui exploite et représente les propriétés de ce programme. Nous avons étudié différents modèles de représentation de programmes afin d'identifier la représentation de programme la plus appropriée qui permette de représenter les flux d'information. Lors de cette enquête [161], nous avons choisi le programme PDG (Program Dependence Graph) comme abstraction que nous adoptons dans ce travail de thèse pour représenter le programme à analyser. PDG est adapté pour atteindre nos objectifs en termes de suivi des flux d'informations, car il représente tous les chemins possibles et est une sur-approximation du comportement du programme.

Puisque notre objectif principal est de vérifier automatiquement l'adhérence des programmes aux directives de sécurité formalisées, nous devons modéliser les direc-

tives MCL sur le modèle de programme. Cependant, le PDG n'est pas formel et ne constitue pas une base pour la vérification formelle par la vérification du modèle. Ainsi, nous devons construire à partir du PDG augmenté un modèle qui est accepté par un outil de vérification de modèle, et qui peut être vérifié automatiquement par des techniques de vérification de modèle.

Formellement, le PDG est un graphe orienté dont les nœuds correspondent à des instructions de programme (déclarations de variables, affectations, prédicats de contrôle, invocations de méthodes, etc.) et dont les arcs modélisent les dépendances dans le programme. Il y a un nœud spécifique qui indique la méthode d'entrée (principale): *node d'entrée*. Pour les programmes Java, le *node d'entrée* correspond à la méthode principale. Les nœuds dans le PDG sont connectés avec plusieurs types d'arêtes; contrôle ou données correspondant respectivement aux dépendances de contrôle et de données. Les deux types sont des calculs utilisant respectivement l'analyse du flux de contrôle et celle du flux de données. Nous utilisons délibérément le terme PDG du graphique de dépendance du programme pour désigner le graphique de dépendance du système. PDG représente le contrôle intraprocedurales et les dépendances de données dans une méthode, et SDG est la version étendue qui a calculé les dépendances inter procédurales globales.

10.5.2 Méthodologie pour la construction du système de transition étiqueté à partir des sources du programme

L'élément clé de départ pour cette étape est le PDG standard que nous générons à partir du bytecode du programme Java en utilisant l'outil JOANA [72]. Dans ce PDG, le contrôle et les dépendances de données (explicites / implicites) sont capturés, ce qui constitue une base solide pour effectuer une analyse précise. Cependant, nous avons testé JOANA sur différents exemples de codes présentant des violations implicites, mais l'outil n'a pas réussi à capturer certains d'entre eux. Nous avons jugé nécessaire d'effectuer une analyse approfondie des flux d'informations sur le programme afin de capturer les dépendances implicites, telles que Java Reflection. Nous avons fait l'effort d'améliorer le PDG et de l'augmenter avec les détails pertinents capturés qui reflètent les dépendances implicites et non triviales.

10.5.3 Graphe de Dépendances (PDG)

Nous utilisons le terme Program Dependence Graph pour désigner la représentation du programme en tant que PDG standard. Nous faisons la distinction entre le PDG et le PDG augmenté que nous avons généré à partir du PDG. Nous expliquons les raisons qui nous ont incités à augmenter le PDG et à décrire en détails comment nous avons généré le PDG augmenté.

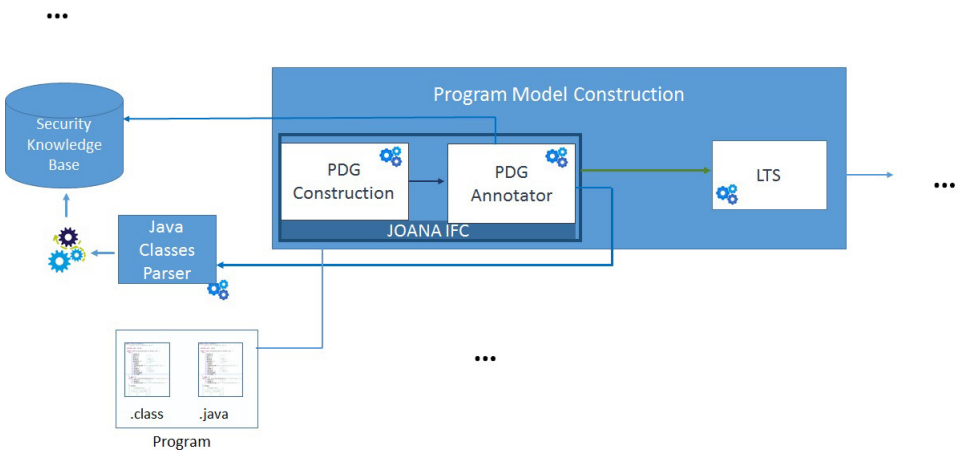


Figure 10.2: Méthodologie pour la construction du modèle de program: Des sources du programme au PDG augmenté, au Graphe de Transitions Étiqueté

Les PDG ont la capacité de représenter précisément le flux d'informations dans un programme, et constituent une base solide pour effectuer une analyse précise du flux d'informations [80]. Le contrôle du flux d'information vise à garantir la fiabilité et l'intégrité des données critiques, et plus précisément, à vérifier que les informations sensibles ne sont pas ou ne peuvent pas être influencées par les résultats publics. Les PDG ont des propriétés différentes qui aident à augmenter la précision de l'analyse du flux d'information; ils sont *sensible au flux*, *contextuel* et *sensible aux objets* [80]. Être *sensible au flux* est la capacité de considérer l'ordre des instructions dans le programme. Le *context-sensitivity* est perçu du fait que si la même méthode est invoquée plusieurs fois, alors chaque site d'appel sera représenté par un nœud séparé, et le mur d'analyse sera effectué sur chaque nœud séparément. En d'autres termes, les méthodes appelant le contexte sont considérées, ce qui augmente la précision. Des recherches antérieures ont prouvé que les sensibilités au contexte et au débit réduisent considérablement le nombre de fausses alarmes [78] [82]. Le *object-sensitivity*, d'autre part, consiste à faire la distinction entre différents objets de la même classe. Les sensibilités présentées aident à augmenter la précision de l'analyse du flux d'informations qui sera effectuée sur ce graphique spécifique. Cependant, plus l'analyse est précise, moins elle est évolutive. PDG fait abstraction des détails non pertinents, tels que les déclarations de programme indépendantes et non-interactives, qui représentent les chemins infaisables, et est un outil puissant qui permet d'effectuer une analyse approfondie des flux d'informations sur les programmes [76]. Les chemins dans le PDG correspondent à des flux d'informations réalisables dans l'application, en d'autres termes, s'il n'y a pas de chemin d'une instruction a à une instruction b, il

n'y a pas de flux d'information a à b.

10.5.4 Graphe de Dépendances Augmenté

10.5.4.1 Introduction de Nouvelles Dépendances sur le Graphe de Dépendance (PDG)

Nous avons présenté dans la section précédente les concepts de base liés à la structure du graphe de dépendance au programme, et montré comment PDG est construit à partir de la source du programme. Maintenant que nous avons présenté la structure graphique de base dans notre cadre, expliquons comment il est utilisé pour l'analyse. Comme nous l'avons mentionné, nous nous appuyons d'abord sur la force de la structure PDG pour représenter tous les flux d'information dans le programme, et pour vérifier la propriété de non-interférence. [141] a prouvé un théorème associant PDG avec la non-interférence classique. Avant d'entrer dans les détails sur les nouvelles dépendances que nous avons définies et capturées sur le PDG, analysons quelques exemples de codes présentant des violations implicites et explicites des consignes de sécurité.

Code 10.1: Code présentant une violation de la règle de sécurité MSC62-J

```
BufferedReader reader = new BufferedReader
        (new InputStreamReader(System.in));
   System.out.println("User name : ");
   System.out.println("Password : ");

   // input
   user.setUsername(reader.readLine());
    user.setPassword(reader.readLine());

   // log
   logMessage = "user name = " + user.getUsername() +
        ", password = " + user.getPassword();
   logger.log(Level.INFO, logMessage);
```

10.5.4.2 Annotations Personnalisées

L'outil JOANA propose deux types d'annotations spécifiant la source (*SOURCE*) et la cible (*SINK*) en plus de l'annotation *DECLASS*. Lors de l'annotation de la source et du puits respectifs sur le PDG, l'utilisateur (ou le développeur) doit également fournir le niveau de sécurité: *High* ou *Low*. Élevé signifie que la variable annotée est confidentielle et doit être gardée secrète. Faible, d'un autre côté,

signifie que la variable ou l'instruction peut être publique. La déclassification des annotations (*DECLASS*) permet de réduire le niveau de sécurité du noeud annoté. La déclassification est définie comme la libération contrôlée d'informations.

Nous avons effectué des modifications sur le code source de l'outil JOANA et ajouté des annotations personnalisées se référant aux étiquettes abstraites sur lesquelles sont construites les directives des formules MCL. Les étiquettes sont traduites en annotations, telles que *hash*, *userInput*, *mot de passe*, *encrypt*, *save*, etc. en plus des annotations prédéfinies *SOURCE* et *SINK*.

Dans un deuxième temps, la détection automatique des étiquettes sur le PDG est effectuée. Le composant **PDG Annotator** et nous nous référons ici à la base de connaissances de sécurité qui contient déjà les mappings possibles concrets entre les API connues, les méthodes, les paramètres de méthodes mappés aux étiquettes abstraites du spécifications de directives de sécurité. Nous fournissons dans Appendice ref appendSKB des exemples de données de la base de connaissances de sécurité. Pour un développeur ou un testeur qui n'est pas au courant de la sémantique des méthodes effectuant des opérations de sécurité, la détection des sources possibles (ou puits) et de leurs puits respectifs (sources) semble fastidieuse.

10.5.4.3 Dépendance de la Clé de Chiffrement

L'opération déclassification déclassification est la libération contrôlée d'informations sensibles à l'aide de mécanismes et de techniques spécifiques tels que le cryptage, le hachage, l'obfuscation, etc. Nous entendons par déclassification paramétrée, la diminution du niveau de sécurité des données dans des conditions sur d'autres données, comme le cryptage avec une clé de cryptage: la déclassification est maintenue si la clé de cryptage reste secrète, sinon les données cryptées doivent être sécurisées son niveau de sécurité reste *HIGH*. Nous nous référons au premier cas comme étant une libération "sûre". Sabelfeld et al. [132] discute des différentes dimensions de la déclassification. Nous avons introduit l'annotation DECLASS (param) qui fait référence à la déclassification par rapport au paramètre *param*: si *param* et toutes ses dépendances sont gardées secrètes, alors l'opération de déclassification est valide, et le niveau de sécurité des données secrètes à chiffrer est déclassifié de *HIGH* à *LOW*, sinon, la déclassification est obsolète et le niveau de sécurité reste *HIGH*. Il est important de noter la corrélation entre les données cryptées et la clé de cryptage; cette relation est traduite en une dépendance dans le graphique de dépendance de programme généré.

10.5.4.4 Annotations multiples sur le même noeud

L'outil JOANA offre la possibilité d'annoter un nœud avec un ensemble limité d'annotations: SOURCE, SINK ou DECLASS. Nous avons ajouté la possibilité d'avoir plusieurs annotations sur le même noeud; cela semblera utile dans différents cas où, par exemple, les mêmes données (le même nœud) sont au cœur de plus d'une ligne directrice.

10.5.4.5 Propagation d'annotation

Nous avons augmenté le PDG en utilisant la propagation des annotations lorsque des données déjà annotées sont copiées ou concaténées. Par exemple, si la clé k (annotée comme *create_key*) a été assignée à une autre variable g, alors g sera aussi annotée comme *create_key* .

10.5.5 Génération du système de transition étiqueté

10.5.5.1 Système de transition étiqueté paramétré

Un système de transition étiqueté paramétré (pLTS) est un système de transition étiqueté avec des variables; un pLTS peut avoir des gardes et l'assignation de variables sur les transitions. Les variables peuvent être manipulées, définies ou accédées dans des états, actions, gardes et affectations. JML [99], Z [126], B [98] permettent de décrire les états du système à travers des objets mathématiques (machines, ensembles, etc.), et ils décrire les conditions préalables et postérieures aux transitions entre les états. Ces langages traitent des programmes séquentiels et ne traitent pas la transmission de valeur pour la plupart.

Un LTS est une structure composée d'états avec des transitions, étiquetés avec des actions entre eux. Les états modélisent les états du programme; les transitions codent les actions qu'un programme peut effectuer dans un état donné. Nous distinguons deux types d'actions: les actions codant un programme séquentiel (représentant des instructions séquentielles standard, y compris la dérivation et l'assignation) et un appel à la méthode (locale ou distante), et les actions codant le résultat du suivi des dépendances explicites et implicites entre variables du programme.

Informellement, nous interprétons le comportement d'un programme comme un ensemble d'états atteignables et d'actions (instructions) qui déclenchent un changement d'état. Les états expriment les valeurs possibles du compteur de programme, ils indiquent si un état est un point d'entrée d'une méthode (état initial), un état de séquence (représentant une instruction séquentielle standard, y compris une

dérivation), un appel à une autre méthode, un point de réponse à un appel de méthode, ou un état qui est de la méthode se termine. Chaque transition décrit l'exécution d'une instruction donnée, de sorte que les étiquettes représentent les noms d'instructions. Les étiquettes LTS peuvent principalement être de trois types: actions, données et dépendances.

* Actions: elles se réfèrent principalement à toutes les instructions du programme, représentant les instructions séquentielles standard, y compris les invocations de branchements et de méthodes.
* Passage de valeur: comme l'analyse effectuée implique des données, les LTS générés sont paramétrés, c'est-à-dire que les transitions sont étiquetées par des actions contenant des valeurs de données.
* Dépendances: en plus des instructions du programme, nous avons ajouté des transitions qui apportent des dépendances de données (implicites et explicites) entre deux instructions dans le but de suivre les flux de données. En effet, les transitions sur LTS montrent les dépendances entre les variables du code. Nous étiquetons ce type de transition par it depend var1 var2 où *var1* et *var2* sont deux dépendants variables

10.5.5.2 Du PDG augmenté au système de transition étiqueté paramétré

Puisque le PDG augmenté n'est pas un modèle formel, nous ne pouvons pas le modéliser. Nous devons générer un modèle formel accepté par les outils de vérification de modèle à partir du PDG augmenté généré.

Au cours de la construction du PDG, nous adoptons une technique connue utilisée dans l'analyse de la souillure et consistant à renommer [28] les variables du programme. Nous renommons chaque définition d'une variable x en un nom différent et renommons chaque utilisation de x par le nouveau nom, pour garantir que les opérations effectuées sur ces données spécifiques conservent le même nom de variable.

Nous construisons le pLTS d'un programme à partir de sa représentation intermédiaire, la structure PDG (Augmented Program Dependence Graph) qui constitue une sur-approximation du flux d'information du programme dans le comportement du programme. Nous avons généré le PDG à l'aide de l'outil JOANA IFC qui permet de suivre les flux d'informations explicites et implicites dans un programme.

Nous présentons dans la figure "PDG augmenté" le PDG augmenté de l'exemple de code de la Figure “Chiffrement”, et dans la figure “LTS” le pLTS que nous avons généré à partir de ce PDG augmenté.

Il est important de noter que nous nous assurons que le PDG augmenté et le pLTS sont isomorphes, en ce sens que les deux graphes ont le même nombre de nœuds.

Le pLTS est similaire au PDG augmenté, sauf que les étiquettes pLTS sont sur les bords (transitions) et non sur les nœuds. Les mêmes actions (instructions) sur les nœuds PDG sont traduites en transitions sur le pLTS, qui est une représentation fidèle des dépendances capturées traduites en arêtes entre les nœuds. Cette propriété est d'une importance primordiale en raison de la capacité qu'elle offre en ce qui concerne la fidélité du support d'analyse, et aussi pour exporter les résultats d'analyse (violation des traces) vers le PDG construit à partir du code source.

Code 10.2: Chiffrement

```
106  public static void main(String[] args)
107        throws NoSuchAlgorithmException,
108        NoSuchProviderException,
109        FileNotFoundException {
110     int c = 123456;
111     Payment p = new Payment();
112     p.setCreditCardNumber(c);
113
114     String x = "0xe04fd020ea3a6910a2d808002b30309d";
115     byte[] y = hexStringToByteArray(x);
116     SecretKeySpec k = new SecretKeySpec(y, "AES");
117
118         // save
119
    save_to_file(k.toString(),"C://Users//XXXX//src//secGuidelines//keys.txt");
120
121
122     // encrypted data
123     byte[] encrypted_cc = encrypt(k,
    Integer.toString(p.getCreditCardNumber()));
124
125
126     // save
127     String data = encrypted_cc.toString();
128
    save_to_file(data,"C://Users//XXXX//src//secGuidelines//encrypted_cards.txt");
  }
```

Code 10.3: La méthode *save_to_file*

```
146  public static void save_to_file(String data, String file) {
147     try (PrintWriter out = new PrintWriter(
148           file)) {
149        out.print(data + GregorianCalendar.getInstance().getTime()
```

```
    +"\r\n");
150     } catch (FileNotFoundException e) {
151        e.printStackTrace();
152        System.out.println("file error");
153     }
154  }
```

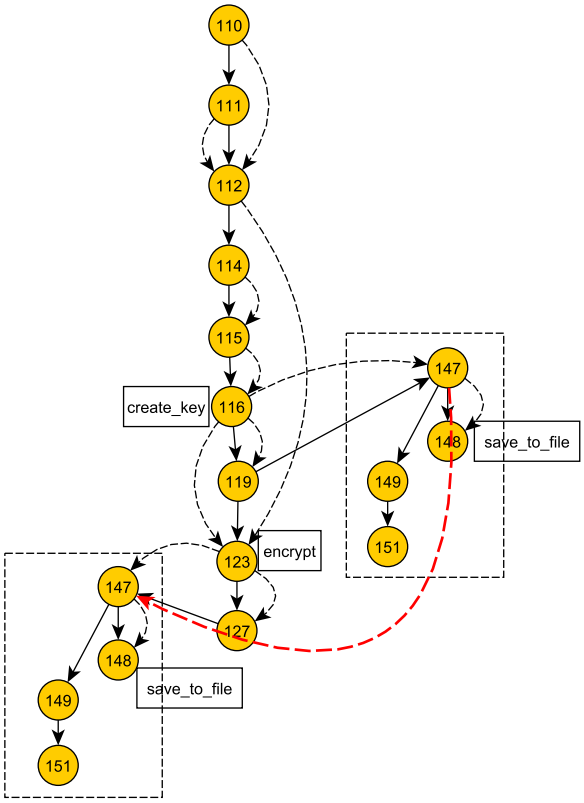


Figure 10.3: PDG augmenté

10.5.5.3 Model Checking

La vérification de modèle ou le Model Checking cite grumberg200825 est une technique automatique de vérification des propriétés comportementales d'un modèle de système par une énumération exhaustive de ses états. L'avantage de l'application des techniques de vérification de modèle est qu'il est automatique et nécessite l'intervention de l'opérateur humain pour guider la vérification. Dans la section précédente, nous avons expliqué comment générer automatiquement le pLTS à partir du PDG augmenté. Maintenant que nous avons préparé le terrain et préparé le terrain pour la vérification formelle: nous avons d'une part la formule MCL de la

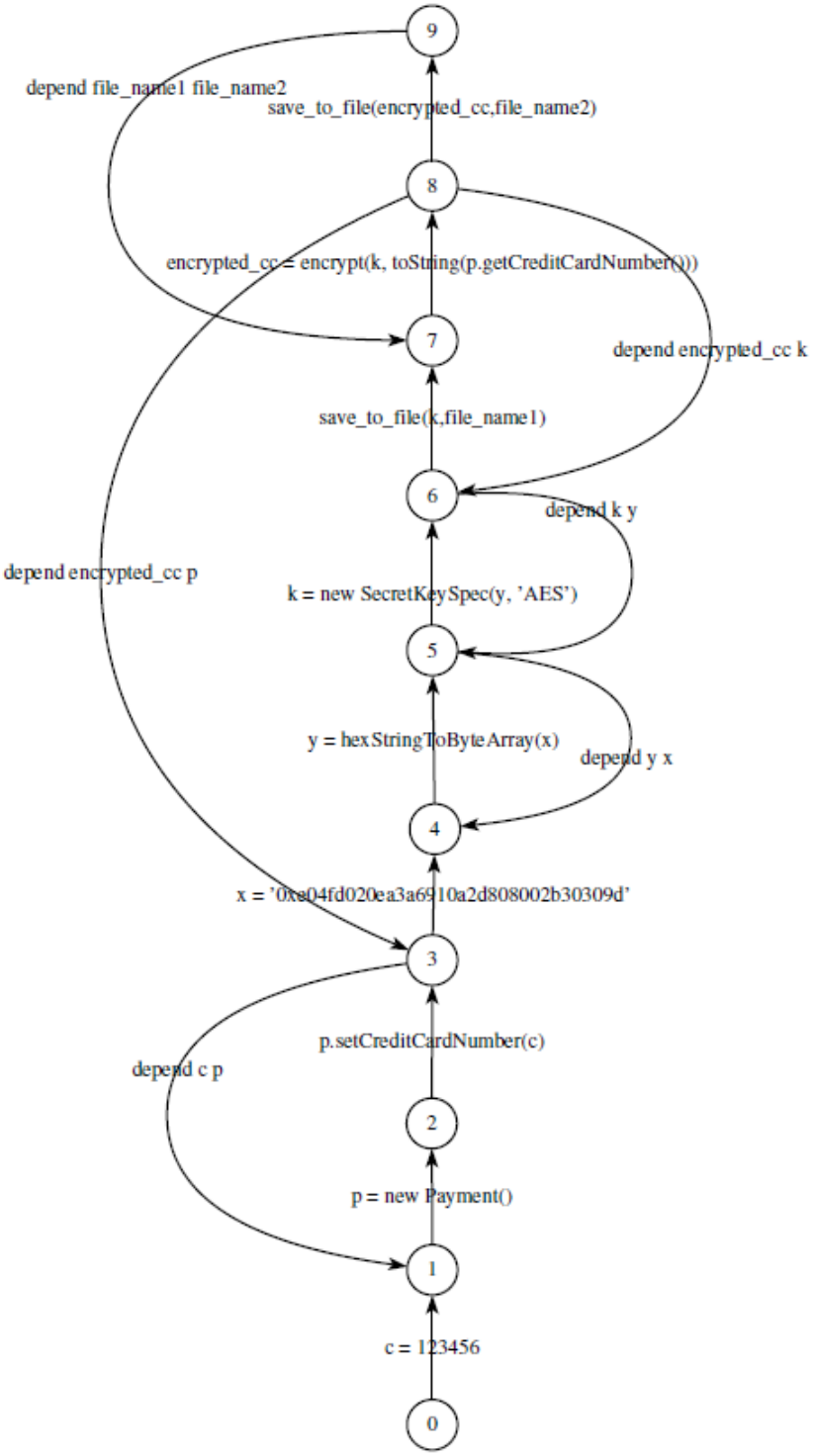


Figure 10.4: pLTS généré à partir du PDG augmenté

ligne directrice, et d'autre part, le pLTS.

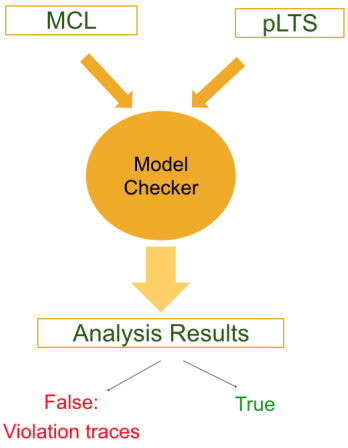


Figure 10.5: Model Checking

Nous pouvons effectuer l'analyse de contrôle de modèle directement sur le pLTS, que nous pouvons simplifier et réduire davantage. *hiding* et *renaming* sont utilisés pour calculer un système de transition étiqueté minimisé, qui est un modèle "opérable" (voir Figure ref fig: MinLTS). Premièrement, certaines actions non pertinentes (pour les propriétés analysées) sont "cachées"; ils sont remplacés par des actions *tau* (notées i dans la figure ref fig: MinLTS). Deuxièmement, nous renommons les actions par leurs synonymes (points d'entrée) dans la base de connaissances de sécurité.

We made use of the checker EVALUATOR of the CADP toolsuite [97] to verify the property OWASP: Store unencrypted keys away from the encrypted data that we have formalized in MCL as follows:

```
[true*.{create_key ?key:String}.true*.
(({save !key ?loc1:String}.true*.
{encrypt ?data:String !key}.true*.
{save !data ?loc2:String}.true*.
{depend !loc1 !loc2}
   |{depend !key ?key1:String}.{save !key1
?loc1:String}.true*.{encrypt ?data:String
!key1}.true*.{save !data ?loc2:String}.
true*.{depend !loc1 !loc2})
|
  ({encrypt ?data:String !key}.true*.
  {save !key?loc1:String}.true*.
```

```
  {save !data ?loc2:String}.true*.
  {depend !loc1 !loc2}
   | {depend !key ?key1:String}.
  {encrypt ?data:String !key1}.
true*.{save !key1 ?loc1:String}.true*.
{save !data ?loc2:String}.true*.
{depend !loc1 !loc2}))] false
```

La vérification de cette propriété sur le résultat pLTS de la figure est fausse, indiquant que la directive est violée en raison de la dépendance implicite de l'emplacement du fichier indiquant que les deux emplacements de fichiers se trouvent dans le même système de fichiers. En plus d'un faux, le vérificateur de modèle produit une trace illustrant la violation de l'état initial, comme illustré dans la figure "Trace de violation".

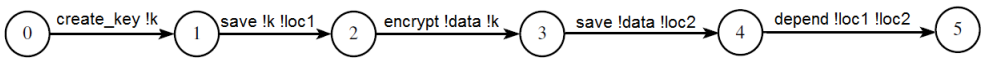


Figure 10.6: Trace de violation

10.5.5.4 Conclusion

Dans ce chapitre, nous avons présenté la méthodologie que nous avons adoptée pour construire le système de transition étiqueté (paramétrique) (pLTS) à partir du graphique de dépendance de programme augmenté. La construction PDG suit une approche plutôt conservatrice, et présente une sur-approximation du comportement du programme, ce qui peut conduire à de fausses alarmes. Nous avons discuté des efforts que nous avons déployés afin de capturer les dépendances implicites et subtiles entre les instructions du programme et les variables à travers l'analyse du flux d'information qui capture ces dépendances implicites. Nous avons montré comment les nouvelles dépendances que nous avons introduites et ajoutées au PDG ont permis d'optimiser l'analyse et d'augmenter la précision. Nous avons insisté sur l'importance du PDG augmenté comme représentation intermédiaire utilisée pour générer le pLTS. Quelqu'un pourrait discuter de la génération du système de transition étiqueté directement à partir du code source du programme (comme Alvis [94]). Cependant, l'adoption d'une approche similaire ne permettrait pas de capturer les dépendances directes et transitives, et manquera définitivement les dépendances implicites que l'analyse des flux d'information capture et augmente avec le PDG. Nous avons utilisé l'analyse de code statique avec la vérification du

modèle pour la vérification automatique des consignes de sécurité. La combinaison de ces deux approches différentes, qui constitue une innovation majeure de ce travail, a conduit à l'amélioration de la précision de l'analyse.

10.6 Conclusion et perspectives

10.6.1 Réalisations

Nous avons présenté une première preuve de concept concernant la faisabilité de notre approche qui vise à étendre la vérification et la validation des directives sur les différentes phases du cycle de vie du développement logiciel. Nous avons proposé une première tentative pour combler la lacune de la vérification formelle des directives fournies de manière informelle à travers la chaîne complète que nous avons mise en œuvre et qui couvre la vérification de bout en bout de la conformité aux directives de sécurité au niveau du code. L'innovation de notre travail est la combinaison d'approches existantes qui sont destinées à remplir différents objectifs. Nous avons fait l'effort de combiner l'analyse de code statique avec la vérification de modèle. Nous avons montré comment nous pouvons améliorer la précision de l'analyse à travers les nouvelles dépendances que nous avons introduites, et qui ont permis de capturer des dépendances très subtiles et implicites.

Nous avons également fait l'effort d'étudier les directives de sécurité provenant de différentes sources, et de les classer par catégories en fonction du type de vérification qui devrait être effectué afin de vérifier l'adhérence du logiciel développé. Afin de supprimer les ambiguïtés, nous avons adopté un formalisme, MCL, pour spécifier les lignes directrices et préparer le terrain pour leur vérification formelle par la vérification des modèles. Les méthodes formelles peuvent ajouter une valeur ajoutée significative à la recherche de violations de sécurité subtiles mais graves et implicites que les moyens traditionnels ne parviennent pas à détecter.

Nous avons souligné la difficulté rencontrée lorsque la directive de sécurité implique des flux d'informations dépendants qui ne peuvent pas être spécifiés séparément. Notre framework utilise cette spécification pour effectuer la vérification du modèle du système de transition étiqueté que nous avons construit à partir du graphe de dépendance du programme que nous avons augmenté avec des détails tels que les annotations personnalisées et les dépendances implicites que nous pouvons capturer via la base de connaissances de sécurité. est une autre réalisation de notre travail. La base de connaissances sur la sécurité contient un référentiel de directives et de schémas de sécurité formalisés qui constituent la base d'un traitement central et d'une gestion des bonnes et des mauvaises pratiques de programmation. En outre, la base de connaissances de sécurité contient un large éventail d'API Java, y compris des méthodes liées à la sécurité. Ces API sont décorées avec leur signa-

ture, description et mappées aux annotations de flux d'informations traditionnelles (source, sink) et leur niveau de sécurité qui indique leur niveau de confidentialité / intégrité (high = secret / private, low = public). Ce référentiel peut être utilisé à la fois par le (s) expert (s) de sécurité et les programmeurs, car il permet de centraliser les connaissances liées à la sécurité partagées par les différents experts en sécurité qui formalisent les consignes de sécurité. Cela peut constituer une base solide pour sensibiliser les développeurs et les sensibiliser à la sécurité afin de mieux comprendre les bonnes pratiques de programmation, ce qui améliore la qualité du code en matière de sécurité.

La sortie de la phase de vérification indique si la directive est respectée ou si elle est violée et les traces de violation sont renvoyées. En utilisant cette sortie, nous serons en mesure de fournir un feedback précis et utile au développeur pour comprendre la source de la violation, et éventuellement comment la réparer.

10.6.2 Perspectives

Nous partageons dans cette section les directions possibles pour ce travail de thèse. Nous avons un large éventail d'idées que nous n'avons pas pu couvrir par manque de temps.

10.6.2.1 Vérification dynamique des consignes de sécurité

Puisque l'analyse de code statique a plusieurs limites en termes de précision et de solidité. Il produit généralement de fausses alarmes ou manque des violations qui peuvent être laissées non détectées et causer des dommages sérieux une fois que le logiciel est déployé. De plus, l'analyse de code statique ne couvre pas les problèmes de configuration et d'environnement d'exécution, car ils ne font pas partie du code source. L'exécution d'une vérification dynamique des consignes de sécurité constituerait un moyen de contrôler la satisfaction des directives dans l'environnement d'exécution. Nous pouvons combiner la surveillance dynamique avec des annotations de sécurité que nous pouvons propager sur la représentation dynamique du programme. Des travaux antérieurs dans le domaine du flux d'informations dynamiques ont été proposés [130]. [16] se concentre sur le suivi dynamique des politiques de déclassification. [17] combine la surveillance dynamique avec la sensibilité au flux.

10.6.2.2 Système de notation pour l'adhésion du programme aux consignes de sécurité

Parmi les orientations futures possibles de notre travail est d'établir un système de notation basé sur la conformité aux directives de sécurité. L'idée est de disposer d'un corpus centralisé rassemblant les différentes lignes directrices visant à soutenir un nouveau schéma de certification qui va au-delà de la fourniture de réponses binaires, et fournit un score de conformité par rapport aux lignes directrices à vérifier. vérifié L'approche adoptée dans le projet OPTET a permis de couvrir différentes limites dans les marchés de logiciels, et a proposé des innovations en termes de découverte de services basés sur des critères et des contraintes de sécurité. Parmi les innovations d'OPTET, il faut évaluer les fonctionnalités de sécurité des services et les rendre visibles aux utilisateurs. Ces caractéristiques de sécurité sont représentées sous la forme d'un certificat dans un format lisible par machine, similaire au schéma de certification proposé dans les Projets Européens *Service de Sécurité Avancé pour SOA* (ASSERT4SOA) [13] et *FI -WARE* projets. Le schéma de certification proposé suppose l'intervention d'un tiers de confiance: l'Autorité de Certification qui conduit une évaluation basée sur des tests des propriétés de sécurité des services appartenant à la classification bien connue: confidentialité, intégrité et disponibilité.

10.6.2.3 Retour au développeur

Nous avons travaillé à proximité du code source du programme, et plus proche du développeur afin de le guider dans le processus de développement et d'assurer la qualité de son code en matière de sécurité. Il est vrai que nous avons fait l'effort de fournir une rétroaction claire au développeur indiquant si la ligne directrice a été respectée ou non. Dans ce dernier cas, nous retournons la ou les traces de violation sur le pLTS qui correspondent aux chemins sur le PDG qui à leur tour correspondent aux instructions du programme exact. Différentes perspectives peuvent être envisagées pour améliorer davantage le résultat de la vérification. Nous pouvons par exemple penser à rendre la sortie de vérification sur l'environnement de développement intégré (IDE), qui est encore plus proche du développeur plutôt que d'exécuter l'analyse sur un autre outil. Cela peut avoir plusieurs avantages tels que la sécurité-éducation du programmeur.

Une autre direction que nous pouvons envisager est de fournir également les conditions dans lesquelles la ligne directrice peut être respectée ou être violée. Le travail de [79] et de [139] nous a rappelé l'idée d'augmenter les bords du PDG avec des conditions précises pour le flux d'information. Cela augmenterait la précision de l'analyse, car elle permet d'affiner davantage le PDG. Il est vrai que

sur le PDG, seuls les chemins réalisables sont représentés, mais le raffinement de la condition de chemin déterminera si un tel chemin est impossible même s'il est représenté sur le PDG. L'application d'une telle approche réduirait la taille du PDG, en supposant que les chemins impossibles seront supprimés du graphique. Par conséquent, la taille du système de transition étiqueté généré sera également réduite, ce qui représentera un pas en avant dans le problème d'explosion d'espace d'état de notre infrastructure.

En ce qui concerne la représentation de la rétroaction au développeur, nous pouvons améliorer la représentation du résultat de la vérification d'une manière claire et précise, ce qui permet au développeur de comprendre la source de la violation soulevée. Les travaux futurs comprennent la représentation de la sortie de vérification de modèle sur le diagramme de dépendance de programme et sur le niveau de code dans l'environnement de développement intégré. Nous pouvons également tirer profit du principe de vérification de modèle: *correct et test again*. En d'autres termes, le retour peut être interactif permettant au développeur de faire des corrections et de vérifier à nouveau, jusqu'à ce qu'il se retrouve avec un programme conforme. Cela peut constituer une autre facette de la sécurité-éducation pour les développeurs.

10.6.3 Conclusion

À travers ce travail de thèse, nous avons voulu souligner les avantages que nous pouvons tirer de la combinaison de différentes approches. Nous avons montré la force de l'analyse des flux d'informations sur la structure PDG pour la détection de flux d'information subtils et implicites que nous pourrions capturer à travers les nouvelles dépendances que nous avons introduites. La combinaison des différentes techniques et outils existants est une innovation majeure de notre travail, et nous tirons profit de la précision et de l'automatisation de la vérification des modèles, ainsi que de la détection implicite des détails sur les sources du programme. La spécification des consignes de sécurité a permis de supprimer les ambiguïtés et d'améliorer la précision de la description textuelle. Nous voulions également trouver des espaces communs pour les différentes communautés: sécurité, contrôle du flux d'information, ingénierie des exigences et méthodes formelles. Nous croyons fermement que ce travail pourrait être avancé pour couvrir un large éventail de directives de sécurité, et être appliqué sur des logiciels du monde réel.

Appendices

Appendix A

Contributions to the OPTET Project

I started my PhD work within SAP Labs France and EURECOM/Telecom ParisTech in June 2013, and have been working since then on the OPTET Project. My contributions ranged from research, to design and development activities.

A.1 State of the Art on Cloud and Mobile Software Marketplaces

In the Deliverable D5.1.1 I [136] have actively contributed to the state-of-the art on software marketplaces, covering mobile as well as cloud marketplaces with the objective of identifying weaknesses in terms of security mechanisms where major innovation can be performed. The focus was mainly on establishing the requirements for the marketplace model. My contributions covered also the identification of the stakeholders in the marketplace model and the possible interactions that they may have.

I have also contributed to the Deliverable D5.3 and still with a state-of-the art on software marketplaces, but with a focus on the delivery and the deployment functionalities and how they are implemented from a security perspective.

A.2 Design of the Software Marketplace Model

To have concrete elements, I proceeded to design the marketplace mockups, and organized together with my SAP colleagues a focus group session to get users feedback on the mockups from ergonomic point-of-view. This helped shaping the design of the marketplace model, and prepared the ground for the design in terms of diagrams illustrating the possible interactions between the different stakeholders, and also the interactions between components in this specific scenario.

In the Deliverable D5.1.2, my contributions covered the identification of the functional requirements of the marketplace model through the consideration of different personas covering different scenarios. For each of the personas, we have identified the involved components and how those components should communicate in order to achieve specific tasks.

A.3 Prototyping Activities

I had also the chance of being part of the prototyping activities of the marketplace model, and also to the test plan document. My contributions covered also the mapping of the users security requirements and the applications security features, based on attributes and metrics. The result is a list of compliant applications that are ordered with respect to their compliance score.

A.4 Contribution to the OPTET Follow-up Activities

The main objective was to integrate information flow analysis for the verification of security guidelines. I contributed to the design of this project, as well to the supervision of our intern, Ms Takoua Kechiche. During six months, we have been working together on the extension of the JOANA tool. Part of this project was included in our paper [156].

Appendix B

Approach for the Formal Specification and Verification of Security Guidelines

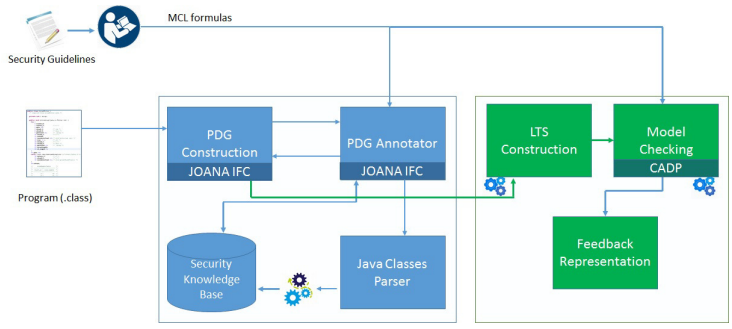


Figure B.1: Prototype for the automatic verification of security guidelines

* PDG Constructor: this component builds the standard program dependence graph from the Java bytecode (.class) which is provided as input. We refer to the JOANA IFC tool [72] that we made use of for many aspects, including the construction of the PDG. We have chosen the Program Dependence Graph (PDG) as the abstraction model, for its ability to represent both control and (explicit/implicit) data dependencies.
* PDG Annotator: given the standard PDG constructed in the previous step, the PDG Annotator adds annotations on the constructed PDG. The annotations (labels in the MCL formulas) are added automatically, and this was made possible through the Security Knowledge Base. We run the information flow analysis on the annotated PDG in order to propagate the annotations, and to capture the explicit as well as implicit dependencies using the JOANA IFC

tool that is formally proven [72]. The PDG Annotator provides as output the **Augmented PDG**.

* Java Classes Parser: with the objective of gathering the different native Java classes details, we developed a parser [157]. This component takes as input the URL of the Java class official documentation [115, 114]. The parser proceeds as follows: for the given Java class URL, it parses the HTML code (javadoc) and extracts all the relevant class details: the class name, the class it inherits from, the class description, the attributes, the constructor(s), the methods signatures, their return type and their parameters. This component populates the Security Knowledge Base with the extracted class details.
* Security Knowledge Base (SKB): apart from the Java classes details, this database contains the different labels that the security expert makes use of in order to build the security guidelines formulas. We carried out the effort of performing a semi-automatic and very basic semantic analysis, basically a keyword matching that aims at assigning to the different Java methods specific labels. For example, the different methods that contain in their description the word "print" are assigned the label "print". Similarly, the label "isKey" is assigned to the methods whom descriptions include the expressions "construct key" or "create key". When building the formulas, the security expert makes use of the labels from the Security Knowledge Base. We need also to mention that this dictionary is rich yet extensible, as security notions and concepts evolve, and the security expert(s) can introduce the new security concepts through new labels that he can add in the SKB.
* LTS Construction: this component translates automatically the **Augmented PDG** into a Labelled Transition System; it retrieves the Augmented PDG details (nodes, edges, annotations, security levels, etc.) and builds the Labelled Transition System that is accepted by model checking tools.
* Model Checking: we carry out the model-checking analysis on the LTS that we generate from the Augmented PDG. We made use of the checker EVALUATOR of the CADP toolsuite [97] to automatically verify the security guideline expressed in MCL. The output of this component indicates whether the guideline is met, or it is violated, and the violation traces are returned.
* Feedback Representation: this component exploits the output of the "Model Checker" to provide a precise and useful feedback to the developer to understand the source of the violation, and possibly how to fix it. This component represents the violation traces on the Augmented PDG constructed from the source code, and this provides a clear feedback to the developer explaining the source of the violation.

Appendix C

Formal Specification of Security Guidelines

Having identified the key concepts from the guideline textual description, Brian proceeds to building the more natural and intuitive MCL formula.

IDS03-J:*Do not log unsanitized user input*

```
[true*.{userInput ?msg:String}.(not ({sanitize !msg}))*. {log !msg}] false
```

MSC62-J:*Store passwords using a hash function* **[40]** For instance, the guideline MSC62-J stating : *"Store passwords using a hash function"*, will be encoded directly by the following formula MCL:

```
[true*.{setPassword ?msg:String}.(not ({hash !msg}))*. {log !msg}] false
```

This basic formula specifies that for each possible value of *msg* (the identifier of a password), the action *store !msg* denoting the storage of the corresponding *msg* stored cannot be reached before *hash !msg* action.

```
[true*.{create_key ?key:String}.true*.({save !key ?loc:String}.true*.
{encrypt ?data:String !key}.true*. {save !data !loc}.true*)] false
```

MSC03-J:*Never hard code sensitive information* **[35]**

```
[true*.{isSensitive ?data:String}*. {clearText(!data)}] false
```

IDS07-J:*Sanitize untrusted data passed to the Runtime.exec() method* **[38]**

```
[true*.{isUntrusted ?msg:String}.(not ({sanitize !msg}))*.
{Runtime_exec (!msg)}] false
```

IDS15-J:*Do not allow sensitive information to leak outside a trust boundary* **[29]**

```
[true*.{isSensitive ?data:String}*. {clearText(!data) \vee display(!data)
\vee write(!data)}] false
```

OWASP1: Store unencrypted keys away from the encrypted data

```
[true*.{create_key ?key:String}.true*.
({save !key ?loc1:String}
.true*.{encrypt ?data:String !key}.true*.
{save !data ?loc2:String}.true*.{depend !loc1 !loc2}
|
{encrypt ?data:String !key}.true*.
{save !key ?loc1:String}.
true*.{save !data ?loc2:String}.true*
.{depend !loc1 !loc2})] false
```

This formula presents five actions: the action *{create_key ?key:String}* denoting encryption key *key* (of type *String*) is created, the actions *{save !key ?loc1:String}*, *{save !data ?loc2:String}*, *{encrypt ?data:String !key}* denoting respectively the storage of the corresponding *key* in location *loc1*, the storage of the corresponding *data* in location *loc2*, the encryption of *data* using *key*, and the particular action *true* denoting any arbitrary action. Note that actions involving data variables are enclosed in braces ({ }).

IDS07-J:*Sanitize untrusted data passed to the Runtime.exec() method* [38]

```
[true*.{isUntrusted ?msg:String}.(not ({sanitize !msg}))*.
{Runtime_exec (!msg)}] false
```

IDS15-J:*Do not allow sensitive information to leak outside a trust boundary* [29]

```
[true*.{isSensitive ?data:String}*. {clearText(!data) \vee display(!data)
\vee write(!data)}] false
```

MSC03-J:*Never hard code sensitive information* [35]

```
[true*.{isSensitive ?data:String}*. {clearText(!data)}] false
```

Appendix D

Security Knowledge Base

D.1 SQL Script

```
--
-- File generated with SQLiteStudio v3.1.1 on lun. juin 5 01:51:21
    2017
--
-- Text encoding used: UTF-8
--
PRAGMA foreign_keys = off;
BEGIN TRANSACTION;

-- Table: abs_guideline
DROP TABLE IF EXISTS abs_guideline;

CREATE TABLE abs_guideline (
    id_ag         INTEGER PRIMARY KEY AUTOINCREMENT,
    name_ag       TEXT,
    type_ag       TEXT,
    description_ag TEXT
);


-- Table: abs_label
DROP TABLE IF EXISTS abs_label;

CREATE TABLE abs_label (
    id_label   INTEGER PRIMARY KEY AUTOINCREMENT,
    text       TEXT,
    description TEXT,
    source     INTEGER,
```

```
    sink       INTEGER,
    declass    INTEGER,
    sec_level  TEXT
);

-- Table: attribute
DROP TABLE IF EXISTS attribute;

CREATE TABLE attribute (
    id      INTEGER PRIMARY KEY AUTOINCREMENT,
    name    STRING,
    sc_type STRING,
    bc_type STRING,
    sec_flag INTEGER
);


-- Table: class
DROP TABLE IF EXISTS class;

CREATE TABLE class (
    id          INTEGER PRIMARY KEY AUTOINCREMENT,
    name        STRING,
    fullname    STRING,
    provider    STRING,
    type        STRING,
    bc_name     STRING,
    library     STRING,
    user_defined INTEGER DEFAULT (0)
);


-- Table: conc_formula
DROP TABLE IF EXISTS conc_formula;

CREATE TABLE conc_formula (
    id_cf     INTEGER PRIMARY KEY AUTOINCREMENT,
    formula_cf TEXT,
    id_af     INTEGER REFERENCES formula (id)
);


-- Table: conc_label
DROP TABLE IF EXISTS conc_label;
```

```
CREATE TABLE conc_label (
    id_cl         INTEGER PRIMARY KEY AUTOINCREMENT,
    text_cl       TEXT,
    description_cl TEXT
);


-- Table: formula
DROP TABLE IF EXISTS formula;

CREATE TABLE formula (
    id      INTEGER PRIMARY KEY AUTOINCREMENT,
    formula TEXT
);


-- Table: label_to_method
DROP TABLE IF EXISTS label_to_method;

CREATE TABLE label_to_method (
    id_label INTEGER REFERENCES abs_label (id_label),
    id_method INTEGER REFERENCES method (id)
);


-- Table: label_to_param
DROP TABLE IF EXISTS label_to_param;

CREATE TABLE label_to_param (
    id_label INTEGER REFERENCES abs_label (id_label),
    id_param INTEGER REFERENCES method_param (id)
);


-- Table: library
DROP TABLE IF EXISTS library;

CREATE TABLE library (
    id   INTEGER PRIMARY KEY AUTOINCREMENT,
    name STRING
);
```

```
-- Table: method
DROP TABLE IF EXISTS method;

CREATE TABLE method (
    id          INTEGER PRIMARY KEY AUTOINCREMENT,
    class       STRING,
    signature   STRING,
    sc_return   STRING,
    bc_return   STRING,
    description STRING,
    sec_flag    INT,
    user_defined INTEGER
);


-- Table: method_param
DROP TABLE IF EXISTS method_param;

CREATE TABLE method_param (
    id          INTEGER PRIMARY KEY AUTOINCREMENT,
    method_id   INTEGER REFERENCES method (id),
    [order]     INTEGER,
    sc_type     STRING,
    bc_type     STRING,
    sec_flag    INTEGER,
    description STRING,
    user_defined INTEGER DEFAULT (0)
);

-- Table: method_sc_bc
DROP TABLE IF EXISTS method_sc_bc;

CREATE TABLE method_sc_bc (
    id       INTEGER PRIMARY KEY AUTOINCREMENT,
    sc_method STRING,
    bc_method STRING
);


-- Table: type
DROP TABLE IF EXISTS type;

CREATE TABLE type (
    id          INTEGER PRIMARY KEY AUTOINCREMENT,
```

```
    sc_type     STRING,
    bc_type     STRING,
    user_defined INTEGER DEFAULT (0)
);

-- Table: algorithm
DROP TABLE IF EXISTS algorithm;

CREATE TABLE algorithm (
    id          INTEGER PRIMARY KEY,
    full_name   TEXT,
    name        TEXT,
    observation
);


-- View: method_label_secLevel
DROP VIEW IF EXISTS method_label_secLevel;
CREATE VIEW method_label_secLevel AS
    SELECT *
      FROM method m,
           abs_label l,
           label_to_method lm
     WHERE m.id = lm.id_method AND
           l.id_label = lm.id_label;

CREATE TABLE algorithm (
    id          INTEGER PRIMARY KEY,
    full_name   TEXT,
    name        TEXT,
    observation
);

COMMIT TRANSACTION;
PRAGMA foreign_keys = on;
```

D.2 Graphical Representation

We present in this section the different tables that we have created in our Security Knowledge Base as well as their relationships in terms of references.

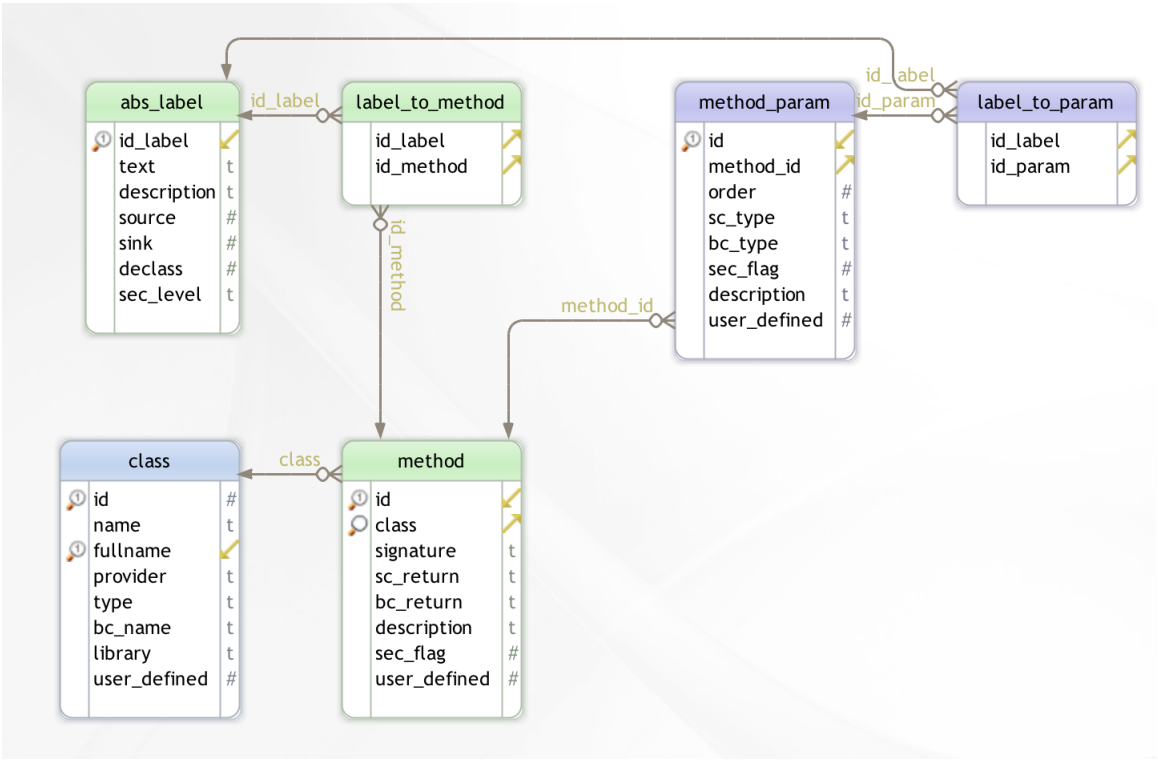


Figure D.1: Security Knowledge Base: Graphical representation of the Java classes and methods, mapped to labels

D.3 Sample Data from the Security Knowledge Base

We provide in this section sample data from the Security Knowledge Base. As the reader can notice, the Java APIs are mapped to their corresponding label (column "text"). We have also taken care of mapping the labels to the security-related annotations: source, sink or declass, together with their security level. Note that the labels are used to build the MCL (Model Checking Language) formulas of the security guidelines.

#	id INTEGER	class STRING	signature STRING	description STRING	text TEXT	sink INTEGER	source INTEGER	declass INTEGER	sec_level TEXT
1	251	BufferedReader	read()	Reads a single character.	userInput	NULL	1	NULL	h
2	252	BufferedReader	read(char[] cbuf, int off, int len)	Reads characters into a portion of an array.	userInput	NULL	1	NULL	h
3	253	BufferedReader	readLine()	Reads a line of text.	userInput	NULL	1	NULL	h
4	254	BufferedReader	ready()	Tells whether this stream is ready to be read.	userInput	NULL	1	NULL	h
5	327	String	concat(String str)	Concatenates the specified string to the end of this string.	isString				
6	331	String	copyValueOf(char[] data)	Returns a String that represents the character sequence in the array specified.	isString				
7	332	String	copyValueOf(char[] data, int offset, int count)	Returns a String that represents the character sequence in the array specified.	isString				
8	336	String	format(Locale l, String format, Object... args)	Returns a formatted string using the specified locale, format string, and arguments.	isString				
9	337	String	format(String format, Object... args)	Returns a formatted string using the specified format string and arguments.	isString				
10	348	String	intern()	Returns a canonical representation for the string object.	isString				
11	359	String	replace(char oldChar, char newChar)	Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.	isString				
12	360	String	replace(CharSequence target, CharSequence replacement)	Replaces each substring of this string that matches the literal target sequence with the	isString				

				specified literal replacement sequence.					
13	361	String	replaceAll(String regex, String replacement)	Replaces each substring of this string that matches the given regular expression with the given replacement.	isString				
14	362	String	replaceFirst(String regex, String replacement)	Replaces the first substring of this string that matches the given regular expression with the given replacement.	isString				
15	368	String	substring(int beginIndex)	Returns a new string that is a substring of this string.	isString				
16	369	String	substring(int beginIndex, int endIndex)	Returns a new string that is a substring of this string.	isString				
17	371	String	toLowerCase()	Converts all of the characters in this String to lower case using the rules of the default locale.	isString				
18	372	String	toLowerCase(Locale locale)	Converts all of the characters in this String to lower case using the rules of the given Locale.	isString				
19	373	String	toString()	This object (which is already a string!) is itself returned.	isString				
20	374	String	toUpperCase()	Converts all of the characters in this String to upper case using the rules of the default locale.	isString				
21	375	String	toUpperCase(Locale locale)	Converts all of the characters in this String to upper case using the rules of the given Locale.	isString				

22	376	String	trim()	Returns a copy of the string, with leading and trailing whitespace omitted.	isString				
23	377	String	valueOf(boolean b)	Returns the string representation of the boolean argument.	isString				
24	378	String	valueOf(char c)	Returns the string representation of the char argument.	isString				
25	379	String	valueOf(char[] data)	Returns the string representation of the char array argument.	isString				
26	380	String	valueOf(char[] data, int offset, int count)	Returns the string representation of a specific subarray of the char array argument.	isString				
27	381	String	valueOf(double d)	Returns the string representation of the double argument.	isString				
28	382	String	valueOf(float f)	Returns the string representation of the float argument.	isString				
29	383	String	valueOf(int i)	Returns the string representation of the int argument.	isString				
30	384	String	valueOf(long l)	Returns the string representation of the long argument.	isString				
31	385	String	valueOf(Object obj)	Returns the string representation of the Object argument.	isString				
32	128	Logger	config(String msg)	Log a CONFIG message.	log	1	*NULL*	*NULL*	1
33	129	Logger	entering(String sourceClass, String sourceMethod)	Log a method entry.	log	1	*NULL*	*NULL*	1
34	130	Logger	entering(String sourceClass, String sourceMethod, Object param1)	Log a method entry, with one parameter.	log	1	*NULL*	*NULL*	1
35	131	Logger	entering(String sourceClass, String sourceMethod, Object[] params)	Log a method entry, with an array of parameters.	log	1	*NULL*	*NULL*	1
36	132	Logger	exiting(String sourceClass, String sourceMethod)	Log a method return.	log	1	*NULL*	*NULL*	1
37	133	Logger	exiting(String sourceClass, String sourceMethod,	Log a method return, with	log	1	*NULL*	*NULL*	1

			Object result)	result object.					
38	134	Logger	fine(String msg)	Log a FINE message.	log	1	*NULL*	*NULL*	1
39	135	Logger	finer(String msg)	Log a FINER message.	log	1	*NULL*	*NULL*	1
40	136	Logger	finest(String msg)	Log a FINEST message.	log	1	*NULL*	*NULL*	1
41	175	MessageDigest	digest()	Completes the hash computation by performing final operations such as padding.	hash	*NULL*	*NULL*	1	*NULL*
42	176	MessageDigest	digest(byte[] input)	Performs a final update on the digest using the specified array of bytes, then completes the digest computation.	hash	*NULL*	*NULL*	1	*NULL*
43	177	MessageDigest	digest(byte[] buf, int offset, int len)	Completes the hash computation by performing final operations such as padding.	hash	*NULL*	*NULL*	1	*NULL*
44	187	MessageDigest	update(byte input)	Updates the digest using the specified byte.	hash	*NULL*	*NULL*	1	*NULL*
45	188	MessageDigest	update(byte[] input)	Updates the digest using the specified array of bytes.	hash	*NULL*	*NULL*	1	*NULL*
46	189	MessageDigest	update(byte[] input, int offset, int len)	Updates the digest using the specified array of bytes, starting at the specified offset.	hash	*NULL*	*NULL*	1	*NULL*
47	190	MessageDigest	update(ByteBuffer input)	Update the digest using the specified ByteBuffer.	hash	*NULL*	*NULL*	1	*NULL*
48	413	PrintStream	print(boolean b)	Prints a boolean value.	display	1			1
49	414	PrintStream	print(char c)	Prints a character.	display	1			1
50	415	PrintStream	print(char[] s)	Prints an array of characters.	display	1			1
51	416	PrintStream	print(double d)	Prints a double-precision floating-point number.	display	1			1
52	418	PrintStream	print(int i)	Prints an integer.	display	1			1
53	419	PrintStream	print(long l)	Prints a long	display	1			1

				integer.					
54	420	PrintStream	print(Object obj)	Prints an object.	display	1			1
55	421	PrintStream	print(String s)	Prints a string.	display	1			1
56	425	PrintStream	println(boolean x)	Prints a boolean and then terminate the line.	display	1			1
57	426	PrintStream	println(char x)	Prints a character and then terminate the line.	display	1			1
58	427	PrintStream	println(char[] x)	Prints an array of characters and then terminate the line.	display	1			1
59	428	PrintStream	println(double x)	Prints a double and then terminate the line.	display	1			1
60	429	PrintStream	println(float x)	Prints a float and then terminate the line.	display	1			1
61	430	PrintStream	println(int x)	Prints an integer and then terminate the line.	display	1			1
62	431	PrintStream	println(long x)	Prints a long and then terminate the line.	display	1			1
63	432	PrintStream	println(Object x)	Prints an Object and then terminate the line.	display	1			1
64	433	PrintStream	println(String x)	Prints a String and then terminate the line.	display	1			1
65	152	Logger	log(Level level, String msg)	Log a message, with no arguments.	log	1	*NULL*	*NULL*	1
66	153	Logger	log(Level level, String msg, Object param1)	Log a message, with one object parameter.	log	1	*NULL*	*NULL*	1
67	154	Logger	log(Level level, String msg, Object[] params)	Log a message, with an array of object arguments.	log	1	*NULL*	*NULL*	1
68	155	Logger	log(Level level, String msg, Throwable thrown)	Log a message, with associated Throwable information.	log	1	*NULL*	*NULL*	1
69	156	Logger	log(LogRecord record)	Log a LogRecord.	log	1	*NULL*	*NULL*	1
70	157	Logger	logp(Level level, String sourceClass, String sourceMethod, String msg)	Log a message, specifying source class and method,	log	1	*NULL*	*NULL*	1

				with no arguments.					
71	158	Logger	logp(Level level, String sourceClass, String sourceMethod, String msg, Object param1)	Log a message, specifying source class and method, with a single object parameter to the log message.	log	1	*NULL*	*NULL*	1
72	159	Logger	logp(Level level, String sourceClass, String sourceMethod, String msg, Object[] params)	Log a message, specifying source class and method, with an array of object arguments.	log	1	*NULL*	*NULL*	1
73	160	Logger	logp(Level level, String sourceClass, String sourceMethod, String msg, Throwable thrown)	Log a message, specifying source class and method, with associated Throwable information.	log	1	*NULL*	*NULL*	1
74	161	Logger	logrb(Level level, String sourceClass, String sourceMethod, String bundleName, String msg)	Log a message, specifying source class, method, and resource bundle name with no arguments.	log	1	*NULL*	*NULL*	1
75	162	Logger	logrb(Level level, String sourceClass, String sourceMethod, String bundleName, String msg, Object param1)	Log a message, specifying source class, method, and resource bundle name, with a single object parameter to the log message.	log	1	*NULL*	*NULL*	1
76	163	Logger	logrb(Level level, String sourceClass, String sourceMethod, String bundleName, String msg, Object[] params)	Log a message, specifying source class, method, and resource bundle name, with an array of object arguments.	log	1	*NULL*	*NULL*	1
77	164	Logger	logrb(Level level, String sourceClass, String sourceMethod, String bundleName, String msg, Throwable thrown)	Log a message, specifying source class, method, and resource bundle name, with associated Throwable information.	log	1	*NULL*	*NULL*	1
78	170	Logger	severe(String msg)	Log a SEVERE	log	1	*NULL*	*NULL*	1

				message.					
79	171	Logger	throwing(String sourceClass, String sourceMethod, Throwable thrown)	Log throwing an exception.	log	1	*NULL*	*NULL*	1
80	172	Logger	warning(String msg)	Log a WARNING message.	log	1	*NULL*	*NULL*	1
81	150	Logger	info(String msg)	Log an INFO message.	log	1	*NULL*	*NULL*	1
82	598	AccessibleObject	setAccessible(boolean flag)	Set the accessible flag for this object to the indicated boolean value.	reflection	1	*NULL*	*NULL*	1
83	597	AccessibleObject	setAccessible(AccessibleObject[] array, boolean flag)	Convenience method to set the accessible flag for an array of objects with a single security check (for efficiency).	reflection	1	*NULL*	*NULL*	1
84	585	Field	getName()	Returns the name of the field represented by this Field object.	reflection	1	*NULL*	*NULL*	1
85	582	Field	getInt(Object obj)	Gets the value of a static or instance field of type int or of another primitive type convertible to type int via a widening conversion.	reflection	1	*NULL*	*NULL*	1
86	581	Field	getGenericType()	Returns a Type object that represents the declared type for the field represented by this Field object.	reflection	1	*NULL*	*NULL*	1
87	580	Field	getFloat(Object obj)	Gets the value of a static or instance field of type float or of another primitive type convertible to type float via a widening conversion.	reflection	1	*NULL*	*NULL*	1
88	579	Field	getDouble(Object obj)	Gets the value of a static or instance field of type double or of another primitive type convertible to type double	reflection	1	*NULL*	*NULL*	1

				via a widening conversion.					
89	577	Field	getDeclaredAnnotations()	Returns all annotations that are directly present on this element.	reflection	1	*NULL*	*NULL*	l
90	576	Field	getChar(Object obj)	Gets the value of a static or instance field of type char or of another primitive type convertible to type char via a widening conversion.	reflection	1	*NULL*	*NULL*	l
91	575	Field	getByte(Object obj)	Gets the value of a static or instance byte field.	reflection	1	*NULL*	*NULL*	l
92	574	Field	getBoolean(Object obj)	Gets the value of a static or instance boolean field.	reflection	1	*NULL*	*NULL*	l
93	572	Field	get(Object obj)	Returns the value of the field represented by this Field, on the specified object.	reflection	1	*NULL*	*NULL*	l
94	564	Method	getModifiers()	Returns the Java language modifiers for the method represented by this Method object, as an integer.	reflection	1	*NULL*	*NULL*	l
95	565	Method	getName()	Returns the name of the method represented by this Method object, as a String.	reflection	1	*NULL*	*NULL*	l
96	605	SecretKeySpec	SecretKeySpec(byte[] key, int offset, int len, String algorithm)	Constructs a secret key from the given byte array, using the first len bytes of key, starting at offset inclusive.	create_key	*NULL*	1	*NULL*	h
97	604	SecretKeySpec	SecretKeySpec(byte[] key, String algorithm)	Constructs a secret key from the given byte array.	create_key	*NULL*	1	*NULL*	h
98	121	KeyPairGenerator	genKeyPair()	Generates a key pair.	create_key	*NULL*	1	*NULL*	h
99	120	KeyPairGenerator	generateKeyPair()	Generates a	create_key	*NULL*	1	*NULL*	h

				key pair.					

Document generated by SQLiteStudio v3.1.1 on mer. juil. 12 20:43:38 2017

Appendix E

Techniques on the PDG

E.0.1 Techniques

The PDG of a program P is denoted G_p.

E.0.1.1 Slicing

Slicing is a known technique that consists in simplifying programs and eliminating the parts that have no influence on the part of interest. The part of interest is called in the *slicing criterion*, and is typically defined by a pair; program location and variables of interest. The computation of the parts of the program that affect directly on transitavely the slicing criterion C result in the *program slice* with respect to the slicing criterion C. A slice S of a program is defined as an executable reduced program computed from a program P [148]. Another definition joins the same concept introduced by Weiser except that the slice is not necessarily executable, and that it is a subset of the program instructions that influence the slicing criterion C. Weiser [148] was the first to introduce the concept of slicing, and presnted it as the natural abstraction that people make when they debug programs. Ottenstein and Ottenstein [118] defined slicing as a reachability problem in dependence graph representations. [24] [88] discusses the interprocedural slicing when the same method is called in multiple sites. We distinguish also between static and dynamic slice. The former is computed without reasoning about the data provided as input, however the latter is based on explicit test cases that make assumptions on input data, and is valid only for a particular execution.
As we deal in our work with static analysis, static slicing stands out as the relevant slicing technique that we carry out in order to fulfill several objectives, including:

- automatic detection of sinks and sources 7.4.1

· information flow analysis and tracking

The computation of slices is highly dependent on control and data dependences, which makes the PDG ideal for this technique. The slicing theorem proposed by [85]:

Slicing theorem 1. *If there is no PDG path $a \rightarrow * b$, it is guaranteed that there is no information flow from statement a to statement b in any program run.*

This theorem guarantees sequential non-interference but falls short in covering covert channels.

E.0.1.2 Backward Slicing

The backward slice with respect to a variable x at a program point p is the set of paths that may influence the variable v at that point p.
BS(x) = y | y →*x

Backward slicing allows to capture all the statements that can influence directly or indirectly the variable v at the program point p.

E.0.1.3 Forward Slicing

The forward slice with respect to a variable x at a program point p is the set of paths that the variable v may influence at the point p.
FS(x) = y | x →*y

E.0.1.4 Chopping

Chopping [21] extracts the statements involved in a transitive dependence from the source criterion to the target criterion. A *chop(s,t)* of a program source is the set of statements that propagate influence from s to t [8]. This represents the set of nodes that are part of a path from s to t in the PDG.

Bibliography

[1] App AppStore Developer Program.

[2] Class PDFWalaIR.

[3] Common criteria part 1: introduction and general model, 2012.

[4] Grammatech: Dependence graphs and program slicing.

[5] Markus Aderhold, Jorge Cuéllar, Heiko Mantel, and Henning Sudbrock. Exemplary formalization of secure coding guidelines. Technical report, Technical Report TUD-CS-2010-0060, TU Darmstadt, Germany, 2010.

[6] Confluence Administrator. Sei cert coding standards. 2017.

[7] Fatimah Akeel, Asieh Salehi Fathabadi, Federica Paci, Andrew Gravell, and Gary Wills. Formal modelling of data integration systems security policies. *Data Science and Engineering*, pages 1–10, 2016.

[8] Irem Aktug and Katsiaryna Naliuka. Conspec – a formal language for policy specification. *Electronic Notes in Theoretical Computer Science*, 197(1):45 – 58, 2008. Proceedings of the First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM 2007).

[9] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[10] John R Allen and Ken Kennedy. A program to convert fortran to parallel form. *Rice MASC TR82-6, Rice University,(March 1, 1982)*, 1982.

[11] Johnson Andrew, Waye Lucas, and Moore Scott. Exploring and enforcing security guarantees via program dependence graphs. *PLDI 2015 Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–302, june 2015.

[12] Marco Anisetti, Claudio Ardagna, Franco Guida, Sigrid Gürgens, Volkmar Lotz, Antonio Maña, Claudia Pandolfo, Jean-Christophe Pazzaglia, Gimena Pujol, and George Spanoudakis. Assert4soa: toward security certification of service-oriented applications. In *On the Move to Meaningful Internet Systems: OTM 2010 Workshops*, pages 38–40. Springer, 2010.

[13] Marco Anisetti, Claudio A. Ardagna, Michele Bezzi, Ernesto Damiani, Samuel Paul Kaluvuri, and Antonino Sabetta. *A Certification-Aware Service-Oriented Architecture*, pages 147–170. Springer New York, New York, NY, 2014.

[14] Apple. App store review guidelines.

[15] Apple. App store review guidelines-privacy.

[16] Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Computer Security Foundations Symposium, 2009. CSF'09. 22nd IEEE*, pages 43–59. IEEE, 2009.

[17] Thomas H Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. *ACM Sigplan Notices*, 44(8):20–31, 2009.

[18] Serge Autexier, Dieter Hutter, Bruno Langenstein, Heiko Mantel, Georg Rock, Axel Schairer, Werner Stephan, Roland Vogt, and Andreas Wolpers. VSE: formal methods meet industrial needs. *STTT*, 3(1):66–77, 2000.

[19] Gilles Barthe, Lennart Beringer, Pierre Crégut, Benjamin Grégoire, Martin Hofmann, Peter Müller, Erik Poll, Germán Puebla, Ian Stark, and Eric Vétillard. Mobius: Mobility, ubiquity, security. In *International Symposium on Trustworthy Global Computing*, pages 10–29. Springer, 2006.

[20] Andreas Bauer, Martin Leucker, and Jonathan Streit. Salt—structured assertion language for temporal logic. In *International Conference on Formal Engineering Methods*, pages 757–775. Springer, 2006.

[21] Lujo Bauer, Jarred Ligatti, and David Walker. More enforceable security policies. In *Proceedings of the Workshop on Foundations of Computer Security (FCS'02), Copenhagen, Denmark*. Citeseer, 2002.

[22] D Elliott Bell and Leonard J LaPadula. Secure computer systems: Mathematical foundations. Technical report, DTIC Document, 1973.

[23] Kenneth J Biba. Integrity considerations for secure computer systems. Technical report, DTIC Document, 1977.

[24] David W. Binkley and Keith Brian Gallagher. Program slicing. *Advances in Computers*, 43:1 – 50, 1996.

[25] Barry W Boehm et al. *Software engineering economics*, volume 197. Prentice-hall Englewood Cliffs (NJ), 1981.

[26] Joachim Breitner, Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. *On Improvements of Low-Deterministic Security*, pages 68–88. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[27] Karen H Brown. Security requirements for cryptographic modules. *Fed. Inf. Process. Stand. Publ*, pages 1–53, 1994.

[28] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 147–160, Berkeley, CA, USA, 2006. USENIX Association.

[29] CERT. Do not allow sensitive information to leak outside a trust boundary.

[30] CERT. Do not log unsanitized user input.

[31] CERT. Do not use the object.equals() method to compare two arrays.

[32] CERT. Ensure that the clone() method calls super.clone().

[33] CERT. Exclude unsanitized user input from format strings.

[34] CERT. Methods that perform a security check must be declared private or final.

[35] CERT. Never hard code sensitive information.

[36] CERT. Normalize strings before validating them.

[37] CERT. Sanitize untrusted data included in a regular expression.

[38] CERT. Sanitize untrusted data passed to the runtime.exec() method.

[39] CERT. Sei cert oracle coding standard for java.

[40] CERT. Store passwords using a hash function.

[41] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79, 2004.

[42] Brian V Chess. Improving computer security using extended static checking. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 160–173. IEEE, 2002.

[43] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In *Informatics*, pages 176–194. Springer, 2001.

[44] Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*, pages 1–30. Springer, 2011.

[45] Michael R Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K Micinski, Markus N Rabe, and César Sánchez. Temporal logics for hyperproperties. *POST*, 8414:265–284, 2014.

[46] James C Corbett, Matthew B Dwyer, and John Hatcliff. Expressing checkable properties of dynamic systems: the bandera specification language. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(1):34–56, 2002.

[47] Antoine Coutant. French scheme cspn to cc evaluation http://www. your-creativesolutions. nl/iccc13/p/cc% 20and% 20n ew% 20techniques/antoine% 20coutant% 20-% 20cspn% 20to% 20cc% 20evaluation. pdf. *Last accessed*, 12:12, 2016.

[48] Facebook Ireland Ltd Data Protection Commissioner. Report of re-audit. September 2012.

[49] Nicolette De Francesco, Antonella Santone, and Luca Tesei. Abstract interpretation and model checking for checking secure information flow in concurrent systems. *Fundam. Inf.*, 54(2-3):195–211, April 2003.

[50] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.

[51] Rayna Dimitrova, Bernd Finkbeiner, and Markus N. Rabe. *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change: 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I*, chapter Monitoring Temporal Information Flow, pages 342–357. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[52] Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488. ACM, 2014.

[53] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.

[54] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Pios: Detecting privacy leaks in ios applications. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011.

[55] Mariki M. Eloff and Sebastiaan H Von Solms. Information security management: an approach to combine process certification and product evaluation. *Computers & Security*, 19(8):698–709, 2000.

[56] William Enck. *Defending Users against Smartphone Apps: Techniques and Future Directions*, pages 49–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[57] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security.

[58] Úlfar Erlingsson and Fred B. Schneider. Sasi enforcement of security policies: A retrospective. In *Proceedings of the 1999 Workshop on New Security Paradigms*, NSPW '99, pages 87–95, New York, NY, USA, 2000. ACM.

[59] Jean Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu. Cadp a protocol validation and verification toolbox. In *Computer Aided Verification*, pages 437–440. Springer, 1996.

[60] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[61] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Pldi 2002: Extended static checking for java. *SIGPLAN Not.*, 48(4S):22–33, July 2013.

[62] Ira R Forman, Nate Forman, and John Vlissides Ibm. Java reflection in action. 2004.

[63] K. Gallaba, A. Mesbah, and I. Beschastnikh. Don't call us, we'll call you: Characterizing callbacks in javascript. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10, Oct 2015.

[64] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. *CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes*, pages 372–387. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[65] Dennis Giffhorn and Gregor Snelting. Probabilistic noninterference based on program dependence graphs. Technical Report 6, Karlsruhe Institute of Technology, April 2012.

[66] Dennis Giffhorn and Gregor Snelting. A new algorithm for low-deterministic security. *International Journal on Information Security*, 2014. to appear.

[67] Li Gong, Gary Ellison, and Mary Dageforde. Inside java 2 platform security. 2003.

[68] Jurgen Graf. Speeding up context-, object-and field-sensitive sdg generation. In *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on*, pages 105–114. IEEE, 2010.

[69] Jürgen Graf, Martin Hecker, and Martin Mohr. Using joana for information flow control in java programs - a practical guide. In *Proceedings of the 6th Working Conference on Programming Languages (ATPS'13)*, Lecture Notes in Informatics (LNI) 215, pages 123–138. Springer Berlin / Heidelberg, February 2013.

[70] Jürgen Graf, Martin Hecker, and Martin Mohr. Using joana for information flow control in java programs - a practical guide. In *Proceedings of the 6th Working Conference on Programming Languages (ATPS'13)*, Lecture Notes in Informatics (LNI) 215, pages 123–138. Springer Berlin / Heidelberg, February 2013.

[71] Jürgen Graf, Martin Hecker, Martin Mohr, and Benedikt Nordhoff. Lock-sensitive interference analysis for java: Combining program dependence graphs with dynamic pushdown networks. In *Proc. 1st International Workshop on Interference and Dependence*, 2013.

[72] Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. Checking applications using security apis with joana. July 2015. 8th International Workshop on Analysis of Security APIs.

[73] Orna Grumberg and Helmut Veith. *25 years of model checking: history, achievements, perspectives*, volume 5000. Springer, 2008.

[74] B. Hamid, S. Gürgens, and A. Fuchs. Security patterns modeling and formalization for pattern-based development of secure software systems. *Innovations in Systems and Software Engineering*, 12(2):109–140, Jun 2016.

[75] Christian Hammer. *Information flow control for Java.* PhD thesis, PhD thesis, Universität Karlsruhe (TH), 2009.

[76] Christian Hammer. *Information Flow Control for Java - A Comprehensive Approach based on Path Conditions in Dependence Graphs.* PhD thesis, Universität Karlsruhe (TH), Fak. f. Informatik, July 2009. ISBN 978-3-86644-398-3.

[77] Christian Hammer. Experiences with pdg-based ifc. In *Proceedings of the Second International Conference on Engineering Secure Software and Systems*, ESSoS'10, pages 44–60, Berlin, Heidelberg, 2010. Springer-Verlag.

[78] Christian Hammer. *Experiences with PDG-Based IFC*, pages 44–60. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[79] Christian Hammer, Jens Krinke, and Gregor Snelting. Information flow control for java based on path conditions in dependence graphs. In *IEEE International Symposium on Secure Software Engineering*, pages 87–96, 2006.

[80] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, December 2009.

[81] Jin Han, Su Mon Kywe, Qiang Yan, Feng Bao, Robert Deng, Debin Gao, Yingjiu Li, and Jianying Zhou. *Launching Generic Attacks on iOS with Ap-*

proved Third-Party Applications, pages 272–289. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[82] Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *ACM SIGPLAN Notices*, volume 44, pages 226–238. ACM, 2009.

[83] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[84] S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 146–157, New York, NY, USA, 1988. ACM.

[85] S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 146–157, New York, NY, USA, 1988. ACM.

[86] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.

[87] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[88] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, January 1990.

[89] M. Huisman and A. Tamalet. A formal connection between security automata and jml annotations. In M. Checkik and M. Wirsing, editors, *Fundamental Approaches to Software Engineering*, volume 5503 of *Lecture Notes in Computer Science*, pages 340–354, Berlin, 2009. Springer Verlag.

[90] Paul Hyde. *Java thread programming*, volume 1. Sams Indianapolis, 1999.

[91] Gartner Inc. Gartner says sas 70 is not proof of security, continuity or privacy compliance. 2010.

[92] Samuel Paul Kaluvuri. *Security assurance of web services through digital security certification.* PhD thesis, Thesis, 11 2014.

[93] Bogdan Korel. The program dependence graph in static program testing. *Information Processing Letters*, 24(2):103 – 108, 1987.

[94] Leszek Kotulski, Marcin Szpyrka, and Adam Sedziwy. *Labelled Transition System Generation from Alvis Language*, pages 180–189. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[95] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, pages 207–218, New York, NY, USA, 1981. ACM.

[96] Butler W Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[97] F. Lang, H. Garavel, and R. Mateescu. An overview of cadp 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4, August 2002.

[98] Kevin Lano. *The B language and method: a guide to practical formal development*. Springer Science & Business Media, 2012.

[99] Gary T Leavens, Albert L Baker, and Clyde Ruby. Jml: a java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA'98)*, pages 404–420. Citeseer, 1998.

[100] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

[101] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for java. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 139–160, Berlin, Heidelberg, 2005. Springer-Verlag.

[102] Fred Long, Dhruv Mohindra, Robert C Seacord, Dean F Sutherland, and David Svoboda. *The CERT Oracle Secure Coding Standard for Java*. Addison-Wesley Professional, 2011.

[103] Radu Mateescu and Damien Thivolle. A model checking language for concurrent value-passing systems. In *Proceedings of the 15th International Symposium on Formal Methods*, FM '08, pages 148–164, Berlin, Heidelberg, 2008. Springer-Verlag.

[104] Masahiro Matsubara, Kohei Sakurai, Fumio Narisawa, Masushi Enshoiwa, Yoshio Yamane, and Hisamitsu Yamanaka. Model checking with program slicing based on variable dependence graphs. In *Proceedings First International Workshop on Formal Techniques for Safety-Critical Systems, FTSCS 2012, Kyoto, Japan, November 12, 2012.*, pages 56–68, 2012.

[105] Richard M. Murray. Linear temporal logic (ltl), 2012.

[106] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. *SIGOPS Oper. Syst. Rev.*, 31(5):129–142, October 1997.

[107] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, October 2000.

[108] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM.

[109] George C Necula and Peter Lee. Research on proof-carrying code for untrusted-code security. In *IEEE symposium on security and privacy*, volume 204, 1997.

[110] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.

[111] NIST. Cwe: 129 improper validation of array index.

[112] NSA. Juliet test suite.

[113] National Institute of Standards, Technologies, and Elaine Barker. Recommendation for key management. 2016.

[114] Oracle. Java platform, standard edition 7 api specification.

[115] Oracle. Java platform, standard edition 8 api specification.

[116] Oracle. Secure coding guidelines for java se. 2014.

[117] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Not.*, 19(5):177–184, April 1984.

[118] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Not.*, 19(5):177–184, April 1984.

[119] OWASP. Cryptographic storage cheat sheet.

[120] OWASP. Cryptographic storage cheat sheet.

[121] OWASP. Owasp secure coding practices quick reference guide.

[122] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *Computer aided verification*, pages 377–390. Springer, 1994.

[123] Marco Pistoia, Satish Chandra, Stephen J Fink, and Eran Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Systems Journal*, 46(2):265–288, 2007.

[124] Strategic Planning. The economic impacts of inadequate infrastructure for software testing. 2002.

[125] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[126] Ben Potter, David Till, and Jane Sinclair. *An Introduction to Formal Specification and Z*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1996.

[127] Markus N Rabe. *A temporal logic approach to iInformation-flow control*. PhD thesis, Saarländische Universitäts-und Landesbibliothek, 2016.

[128] Derek Rayside. "points-to analysis. *MIT Open-CourseWare*, 2005.

[129] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. *ACM SIGSOFT Software Engineering Notes*, 19(5):11–20, 1994.

[130] Alejandro Russo, Andrei Sabelfeld, and Andrey Chudnov. Tracking information flow in dynamic tree structures. *Computer Security–ESORICS 2009*, pages 86–103, 2009.

[131] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[132] Andrei Sabelfeld and David Sands. *Declassification: Dimensions and Principles*, pages 255–269. IEEE, 2005.

[133] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

[134] Craig Schlenoff, Michael Gruninger, Florence Tissot, John Valois, Joshua Lubell, and Jintae Lee. The process specification language (psl): Overview and version 1.0 specification, 2000.

[135] Fred B Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.

[136] Stuart Short, Francesco Di Cerbo, Zeineb Zhioua, Symeon Meichanetzoglou, Thomas Vilarinho, Bassem Nasser, Sandro Hartenstein, Holger Könnecke, Sachar Paulus, and Sébastien Keller. D5.1.1: Requirements for a specification of trustworthy service marketplace.

[137] Gang Shu, Boya Sun, Tim AD Henderson, and Andy Podgurski. Javapdg: A new platform for program dependence analysis. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 408–415. IEEE, 2013.

[138] Saurabh Sinha, Mary J. Harrold, and Gregg Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *International Conference on Software Engineering*, pages 432–441, 1999.

[139] Gregor Snelting. Combining slicing and constraint solving for validation of measurement software. *Static Analysis*, pages 332–348, 1996.

[140] Gregor Snelting, Dennis Giffhorn, Jürgen Graf, Christian Hammer, Martin Hecker, Martin Mohr, and Daniel Wasserrab. Checking probabilistic noninterference using joana. *it-Information Technology*, 56(6):280–287, 2014.

[141] Gregor Snelting, Torsten Robschink, and Jens Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(4):410–457, 2006.

[142] Nicolas Stouls and Virgile Prevosto. Aorai plugin tutorial – frama c.

[143] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. In *ACM Sigplan Notices*, volume 44, pages 87–97. ACM, 2009.

[144] Nikola Trčka, Wil M. P. van der Aalst, and Natalia Sidorova. Data-flow anti-patterns: Discovering dataflow errors in workflows. *21st International Conference, CAiSE 2009, Amsterdam, The Netherlands, June 8-12, 2009. Proceedings*, pages 425–439, June 2009.

[145] Chii-Ren Tsai, Virgil D. Gligor, and C. S. Shandersekaran. On the identification of covert storage channels in secure systems. *IEEE Trans. Softw. Eng.*, 16(6):569–580, June 1990.

[146] Antti Valmari. *The state explosion problem*, pages 429–528. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[147] Tielei Wang, Kangjie Lu, Long Lu, Simon P Chung, and Wenke Lee. Jekyll on ios: When benign apps become evil. In *USENIX Security Symposium*, volume 13, pages 559–572, 2013.

[148] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[149] Mark Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, July 1982.

[150] John Wilander and Pia Fak. Modeling and visualizing security properties of code using dependence graphs.

[151] John Wilander and Pia Fak. Pattern matching security properties of code using dependence graphs.

[152] Pierre Wolper. The tableau method for temporal logic: An overview. *Logique et Analyse*, pages 119–136, 1985.

[153] Nisansala Prasanthi Yatapanage. *Slicing Behavior Trees for verification of large systems*. PhD thesis, Griffith University, 2011.

[154] Akshada D Yevatkar and SA Bhura. Analysis of privacy compl data systems.

[155] Roudier Yves and Apvrille Ludovic. Sysml-sec: A model driven approach for designing safe and secure systems. *Model-Driven Engineering and Software Development (MODELSWARD), 2015 3rd International Conference*, pages 655–664, February 2015.

[156] Z. Zhioua, Y. Roudier, R. Ameur-Boulifa, T. Kechiche, and S. Short. Tracking dependent information flows. In *ICISSP 2017 : 3rd International Conference on Information Systems Security and Privacy*, Porto, Portugal, February 2017.

[157] Zeineb Zhioua. https://github.com/zeineb/java-classes-parser.

[158] Zeineb Zhioua, Yves Roudier, and Rabea Boulifa Ameur. Formal specification and verification of security guidelines. In *Dependable Computing (PRDC), 2017 IEEE 22nd Pacific Rim International Symposium on*, pages 267–273. IEEE, 2017.

[159] Zeineb Zhioua, Yves Roudier, and Rabea Ameur Boulifa. Formal specification of security guidelines for program certification. IEEE, 2017.

[160] Zeineb Zhioua, Yves Roudier, S Short, and Rabea Boulifa Ameur. Security guidelines: Requirements engineering for verifying code quality. In *ESPRE 2016, 3rd International Workshop on Evolving Security and Privacy Requirements Engineering, September 12th, 2016, Beijing, China, co-located with the 24th IEEE International Requirements Engineering Conference*, Beijing, CHINA, 09 2016.

[161] Zeineb Zhioua, Stuart Short, and Yves Roudier. Towards the verification and validation of software security properties using static code analysis. *Int. J. Comput. Sci. Theor. App*, 2(2):23–34, 2014.