ASTES

Advances in Science, Technology and Engineering Systems Journal
Vol. 3, No. 1, 38-48 (2018)
www.astesj.com
Special issue on Advancement in Engineering Technology

ASTES Journal
ISSN: 2415-6698

Framework for the Formal Specification and Verification of Security Guidelines

Zeineb Zhioua[*,1], Rabea Ameur-Boulifa[2], Yves Roudier[3]

[1]*EURECOM, Digital Security, France*

[2]*LTCI, Télécom ParisTech, Université Paris-Saclay, France*

[3]*I3S - CNRS - Université de Nice Sophia Antipolis, France*

ARTICLEINFO

Article history:
Received: 31 October, 2017
Accepted: 09 January, 2018
Online: 30 January, 2018

Keywords:
Security Guidelines
Formal specification
Model Checking
Information Flow Analysis
Program Dependence Graph

ABSTRACT

Ensuring the compliance of developed software with general and application-specific security requirements is a challenging task due to the lack of automatic and formal means to lead this verification. In this paper, we present our approach that aims at integrating the formal specification and verification of security guidelines in early stages of the development lifecycle by combining both the model checking analysis together with information flow analysis. We present our framework that is based on an extension of LTS (labelled transition Systems) by data dependence information to cover the end-to-end specification and verification of security guidelines.

1 Introduction

This paper is an extension of work originally presented in Pacific Rim International Symposium on Dependable Computing (PRDC 2017) [1]. About 64% of the 2500+ vulnerabilities in the National Vulnerability Database NVD were due to programming mistakes [2], and the majority of software vulnerabilities are caused by coding errors. Flaws and errors can be introduced during the different phases of the software development lifecycle, from design to development. The missed programming errors can turn into security vulnerabilities at run-time, and can be exploited by intruders who may cause serious damage to the software critical assets and resources. The undetected flaws can cause a cost increase, comprising maintenance and flaw correction fees. Using code analysis tools would avoid such issues and help produce safe and secure software. The last decades have witnessed the development of many analysis techniques that aim at detecting security vulnerabilities in the early stages of development lifecycle; however, most attention was devoted to control-flow, somehow ignoring the data dependencies source of vulnerabilities that can remain undetected.

On the one hand, it is important for developers to be aware of domain-specific requirements as failure to pass the verification and validation phase, and subsequent application corrections and maintenance may be costly, time-consuming, and affect the company's reputation. On the other hand, security guidelines should be expressed in a way that allows their understanding and easy implementation for developers who may not be security experts to develop and deliver secure software.

Security guidelines or security best practices serve as recommendations to developers to reduce the application exposure to security issues, and to ensure that the developed system will behave as expected in hostile environments.

The problems described above entail the need to add on top of the development process different types of verification such as the compliance with the general and application-specific security requirements. Applying formal methods in the different phases of the software engineering process can help further understanding of the system, and detect design flaws rather early in the development.

In our papers [3,4] we conducted a survey on the static code analysis methods with the objective of identifying the approach that best fits our needs in terms of information flow properties detection and

[*]Corresponding Author: Zeineb Zhioua, zeineb.zhioua@eurecom.fr

https://dx.doi.org/10.25046/aj030106

validation, as well as on the system abstraction models that constitute a strong basis in order to carry out the analysis.

The main problem we are tackling in this paper is how to automatically verify the systematic application and compliance of (being) developed software with security requirements expressed in natural language. This problem requires the transformation of the guidelines written informally into a precise formalism by security expert(s), and this is a very tedious task. Nowadays, formal methods, in particular formal verification, are increasingly being used to enforce security and safety of programs.

We propose a framework that first provides security experts with the means to express security guidelines in a more formal way than plain text. Then, our framework verifies the adherence to the guidelines over an abstraction of the program, and provides understandable and clear feedback to the developer to indicate the exact program location where the error occurred. The innovation of our framework relies on the combination between model checking and data dependencies together with the analysis of the system behavior without actually executing it. We focus on the current version of our framework on the Java programming language.

Below we provide a sample code (Figure 1) that presents an implicit violation of the guideline from the *OWASP Cryptographic Storage Cheat Sheet* [4]:"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. Let us look at the details of the code. The developer encrypts the secret data credit card number, and stores the cipher text into a file. At line 115, the developer creates a byte array *y* used as parameter for the instantiation of a SecretKeySpec named *k* (line 116). At line 119, the key *k* is stored in a file, through the invocation of method *save_to_file* (Figure 1). Once created, key *k* is provided as parameter to the method *save_to_file(String data, String file)* (Figure 1) The developer then encrypts the secret variable creditCardNumber using method *private static byte[] encrypt(Key k, String text)* which uses key *k* as parameter. The encrypted data is then stored using method *save_to_file(String data, String file)* (Figure 1). One can conclude that the guideline is met, as key *k* and *encypted_cc* are stored in separate files. But if we take a closer look, we would notice that the two files are located in the same file system. Hence, the code violated the guideline.

In this paper, we go through the approach that we propose in order to help solving the difficulty of capturing implicit and subtle dependencies that can be source of security guidelines violations.

```
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,"C:/[redacted]"
120      + "//[redacted]//src//secGuidelines//keys.txt");
121
122  // encrypted data
123  byte[] encrypted_cc = encrypt(k, Integer.toString
124      (p.getCreditCardNumber()));
125
126  // save
127  save_to_file(encrypted_cc,"C:/[redacted]"
128   + "//[redacted]//src//secGuidelines//encrypted_cards.txt");
129  }
```

Figure 1: Sample code for the encryption of credit card number

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

Figure 2: Source code of save_to_file method

The flow of this paper is as follows: Section 2 introduces the security guidelines and discusses the motivation behind this work. In Section 3, we explain in detail the approach we carried out. We highlight several issues related with the guidelines presentation to developers in Section 4. In Section 5, we introduce the notion of information flow analysis that we make use of in our framework for the detection of implicit dependencies. In Section 6, we present our model construction methodology. We provide in Section 7 a comprehensive application of our formalism to specify and to verify the selected guideline. In section 8, we outline the related work, followed by a discussion in Section 9. And finally, Section 10 concludes the paper and discusses future work.

[1] https://www.owasp.org/index.php/Cryptographic_Storage_Cheat_Sheet#Rule_-_Store_unencrypted_keys_away_from_the_encrypted_data

2 Security Guidelines

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 meet 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.

Official sources, such as OWASP [5], Oracle [6], CERT [7], NSA [8], NIST [9] propose rules and examples of good/bad programming practices. The presentation of the security guidelines differ from one source to another. For instance, CERT Oracle Coding Standard for Java [7] provides for each guideline a textual description, followed by a compliant sample code, and another sample code violating the guideline. OWASP [5] provides for most guidelines a detailed description, and examples of compliant and non-compliant solutions.

Motivation The OWASP Foundation[5] for instance introduces a set of guidelines and rules to be followed in order to protect data at rest. However, the guidelines are presented in an informal style, and their interpretation and implementation require security expertise, as stressed in [3]. In the *OWASP Storage Cheat Sheet* [5], OWASP introduces the guideline "Store unencrypted keys away from the encrypted data" [5] explaining the encountered risks when the encryption key is stored in the same location as encrypted data. This guideline recommends not to store encrypted 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 HeartBleed[2] [10] attack that occurred couple of years ago (April 2014), and that allowed an attacker to read the memory, steal users credentials directly from the systems protected by the vulnerable version of OpenSSL. This example emphasizes the critical attacks that can be performed if the guideline is not respected. OWASP provides a set 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.

In the sample code that we provided in Section 1, we showed how we could implicitly violate the guideline. Detecting this violation is not trivial, as it includes subtle dependencies that should be analyzed with due consideration. This is the main objective of this paper, and we could achieve this through the approach that we depict in details in the next Section.

3 Approach

In this section, we go through the details of our approach that aims at filling the gap between the informal description of security guidelines presented in natural language, 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 Section 4.

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 us 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.

[2] http://heartbleed.com/

The big picture of the proposed approach is shown in Figure 3, highlighting the relevant steps towards fulfilling the transformation of security guidelines written in natural language into exploitable formulas that can be automatically verified over the program to analyze.

The crucial part of the work is the explicit mapping between abstract security guidelines formal specification, and concrete statements on the code.

In order to make the process more concrete, a *separation of duties* needs to be made. We make the distinction between the *security expert* and the *developer*. The former carries out the formal specification of the security guidelines and their translation from natural language to formulas and patterns. He establishes also the mapping between the abstract labels and possible Java language instructions. The latter invokes the framework that makes use of this specification to make the mapping between abstract labels and the program logic, and then to verify the compliance of his developed software with the security requirement.

The idea we propose, as depicted in Figure 3, is the following:

3.1 Formal Specification of Security Guidelines

Starting from the guidelines presented in an informal manner, we make the strong assumption that the **security expert** formally specifies the security guidelines by extracting the key elements, and builds the formulas or patterns based on formalism. The established formulas or patterns can be supported by standard model checking tools. We present in Section 7 how the guideline that we consider in this paper can be modeled in a formalism, and can be formally verified using a model checking tool.

3.2 Program Model Construction

Choosing a program representation depends on the intended application. In our case, the program should be abstracted in a way that preserves its properties, such as the explicit and the implicit dependencies, hence allowing the performance of deep information flow analysis. In our framework, we have chosen the Program Dependence Graph (PDG) (see section 6) as the representation model, for its ability to represent both control and data dependencies. The generated PDG is then augmented with details and information extracted from the formulas and patterns of the security guidelines. We performed Information Flow Analysis over the constructed PDG in order to augment it with further security-related details. This analysis aims at capturing the different dependencies that may occur between the different PDG nodes, hence, augmenting the generated PDG with relevant details, such as annotations mapping the PDG nodes to abstract labels of the security guidelines. Then, we generate form the augmented PDG, a Labelled Transition System tha is accepted by model checking tools.

3.3 Verification

As previously mentioned, security guidelines will be modeled in the form of sequence of atomic propositions or statements representing the behavior of the system. The security guidelines will then be verified over the Labelled Transition System that we generate from the PDG augmented with implicit and subtle dependencies. The verification phase can have the following outcomes:

- The security guideline is valid over all the feasible paths
- The security guideline is violated

The first case can be advanced further, meaning that the verification can provide more details to the developer (or the tester) about circumstances under which the security guideline is valid. In the second case, recommendations to make the necessary corrections on the program can then be proposed . The concrete mapping between the abstract propositions in the formal security guidelines and the program model is managed in the Security Knowledge Base (Section 9).

4 Security Guidelines Analysis

Security guidelines are usually presented in an informal and unstructured way. Their presentation differs from one source to another, which can be misleading to developers. We did the effort of analyzing the guidelines from different sources, and we raised different problems that we have discussed in [4]. Upon this survey, we noticed a lack of precision and a total absence of automation. From developers perspective, the understanding and interpretation of guidelines is not a trivial task, as there is no formalization that exposes the necessary program instructions for each guideline, or that explains how to apply them correctly in their software. In order to overcome this weakness, we come up with a centralized database that gathers the possible mappings between guidelines and Java instructions in the Security Knowledge Base. In the OWASP Secure Coding Practices guide [5], a set of security guidelines 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 not tied to a specific programming language. Another programming practices guide we can consider for instance is the CERT Oracle Coding Standard for Java[7] ; 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. 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.

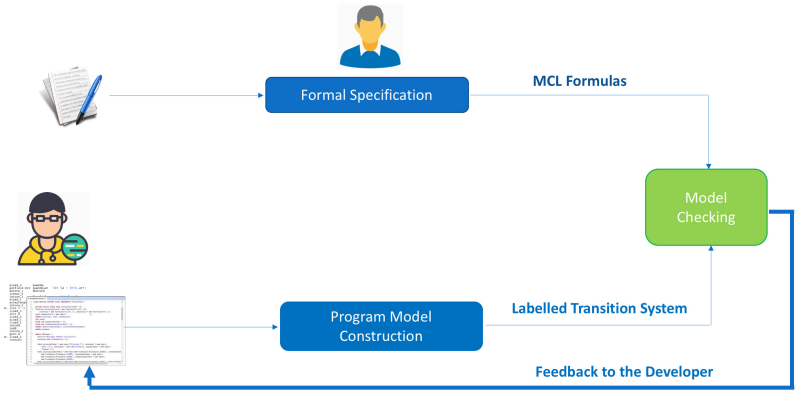

Figure 3: Framework for the formal specification and verification of Security Guidelines

5 Information Flow Analysis

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. One might argue about using known security mechanisms such as access control to control the information propagation in a program. This aspect is of a paramount importance when dealing with information security. However, from a historic point of view, access control mechanisms, are used to verify the access rights at the point of access, and then, to allow or deny the access to the asset over which the mechanism is set. Access control mechanisms, just like 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 [11], 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.

The main objectives of the information flow control [12] are to preserve the confidentiality and integrity of data; the former objective consists in guaranteeing that confidential data don't leak to public variables. As for the second objective, it consists in verifying that critical data is independent from public variables/output. Information flow control analyzes the software with the objective of verifying its compliance and conformance to some security policies. Different approaches have been proposed for Information Flow Control, where we can distinguish between *language-based* and *type-based* information flow control. The former has the advantage of exploiting the program source code and the programming language specificity, but falls short in covering different aspects such as physical side channels that is covered by other approaches[13] and execution environment properties. Language-based security mechanisms have been treated in the literature, including the bytecode verifiers and sandbox model. Those mechanisms enforce security through the Java language, but only the bytecode verifiers make use of the static code analysis. Type-based information flow control, on the other hand, basically makes use of the typing rules that capture illegal flows of information throughout a program, however, they are neither flow-insensitive, context-sensitive nor object-sensitive, which leads to imprecision, which in turn leads to false alarms.

6 Program Model Construction

The starting key element for this step is the standard PDG that we generate from the Java program bytecode using the JOANA tool [14]. In this PDG, control and (explicit/implicit) data dependencies are captured, which constitutes a strong basis to perform a precise analysis. 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 doesn't 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. We depict in Figure 4 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 [14]. 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

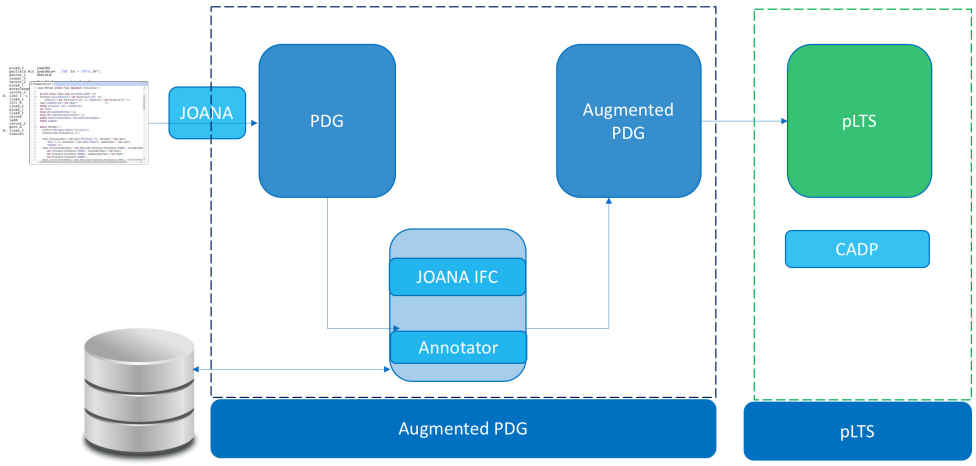

Figure 4: Methodology for the model construction: From program sources to the Augmented Program Dependence Graph, to the Labelled Transition System

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. We run the information flow analysis using the JOANA IFC, that is formally proven [14] 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**. We show in Figure 5 the Augmented PDG of the sample code (Figure 1) that we consider in this paper.

- 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 [3] takes as input the URL of the Java class official documentation, 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.

6.1 Program Dependence Graphs

PDG (Program Dependence Graph) is a language-independent representation of program. This model was first proposed by Ferrante et al. [15] as a program representation taking into consideration both control and data relationships in a program. Formally, the PDG is a directed graph whose nodes correspond to program statements and whose edges model dependencies in the program. Those dependencies can be classified as either control or data dependencies. The nodes are predicates (variable declarations, assignments, control predicates) and edges are data and control dependencie representation; both types are computed using respectively control-flow and data-flow analysis.

PDGs have the ability to represent the information flow in a program, and have different properties, such as being *flow-sensitive*, *context-sensitive* and *object-sensitive* [13]. 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 a separate node in the graph, and will be analyzed separately. In other words, the methods calling context is considered, and this increases precision. The object sensitivity, on the other hand, is the ability to extend 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, hence the analysis will be extended to the attributes level.
PDG abstracts away irrelevant details, such as independent and non-interacting program statements, that represent the unfeasible paths.

In our framework, we make use of the JOANA tool, that stands for Java Object sensitive Analysis [16] for the construction of the PDG. JOANA [14] is a framework that statically analyzes the byte code of Java programs; the tool first generates from the program byte code a PDG, which constitutes an over-approximation of the information flow in the program to analyze. The PDG contains apart from the nodes representing the statements and the variable declarations in the

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

program, contains also edges referring to control and data dependencies between nodes. The dependencies represent explicit dependencies as well as transitive and implicit dependencies.

JOANA's strength relies on the ability to track how information propagates through a program, and captures both the explicit and implicit information flows.

6.2 Augmented Program Dependence Graph

We extend the definition of program dependence graphs to accommodate propositions over the set of program variables. Once the PDG is built, we compute all the propositions that are defined over the program. These propositions are parametrized by the variables defined within the program. Each node is annotated with a set of propositions.

There are two categories of information which can all be used as annotations. The first category is obtained by identifying the set of standard instructions: variables assignments, method calls, etc. at a given node. Example of methods are *encypt*, *hash*, *log*, *normalize*, *sanitize*. Second category is dedicated to relationship between variables, the dependencies in terms of explicit and implicit data dependencies between variables in a program.

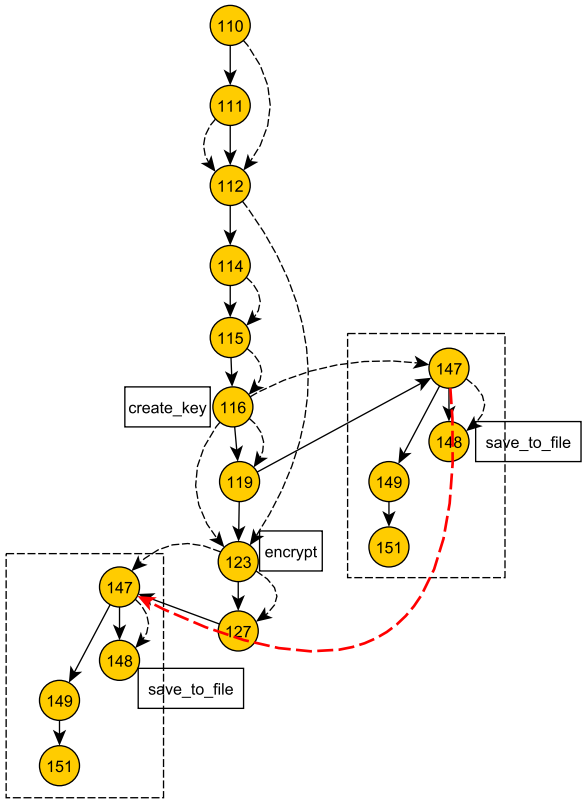

Figure 5: Augmented Program Dependence Graph for the sample code. Strong edges represent the control flows, the dashed edges refer to explicit and implicit data flows. Nodes are labeled with their corresponding instructions line numbers

Note that referring to our security knowledge base, different annotations on the PDG were pre-computed, like for instance the *save*, *userInput* and the *encyption_key*.

Automatic annotations. First, we have created our own annotations based on the atomic propositions of the security guidelines formulas. We made modifications on the source code of JOANA, and added the annotations *hash*, *userInput*, *Password*, *encrypt*, etc. in addition to the predefined annotations *SOURCE* and *SINK*. As shown in Figure 5, different program nodes are annotated with abstract labels, such as node 65 annotated as *Password*, node 73 annotated as *hash* and node 80 annotated as *log* or *store*. The log annotation was pre-computed after the PDG is built, meaning that the security knowledge base was accessed to fetch the concrete possible mappings between known APIs, methods, methods parameters mapped to the abstract labels the formulas are built upon. The mapping between the method invocation *logger.log* and the label *log* is already established. Same for the *hash* label. However, for the *Password*, the automatic annotation requires a semantic analysis to be performed over the code in order to determine the variable names matching password. The semantic analysis is not in the scope of this paper.

Annotations validation by the developer. Once the automatic detection of the atomic propositions on the PDG is performed, the intervention of developer is required to validate the added annotations. There might also be the case where the developer creates a method implementing the hash functionality, then the detection of the label *hash* on the program model will fail. In the sample code, the logging, which is one simple possible storing operation, was invoked. The developer, in our example, annotated the node 65 as *Password*, and the node 80 as *store* (in addition to the *log* automatic annotation).

6.3 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 [17], Z [18], B [19] 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.

Definition 1 (pLTS) *A parametrized 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 parametrized 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 parametrized, i.e, transitions are labelled 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.

7 Verification

With the objective of achieving our main goal consisting in helping a programmer verify that his program satisfies given security guidelines, we translate the augmented PDG into a formal description, which is precise in meaning and amenable to formal analysis. As usual, in the setting of distributed and concurrent applications, we provide behavioral semantics of analyzed programs in terms of a set of interacting finite state machines, called LTS [20]. An LTS is a structure consisting of states with transitions, labeled with actions between them. The states model the system states; the labeled transitions model the actions that a system can perform. Considered LTS are specific; their actions have a rich structure, for they take care of value passing actions and of assignment of state variables. They encode in a natural way the standard instructions of PDGs (as shown in Figure 5). Besides the classical behavior of a PDG, we encode in our LTS the result of tracking of explicit and implicit dependencies between program instructions. These dependencies are encoded by transitions labeled with the action *depend input_data output_data* (see Figure 6), allowing one to prove information flow properties.

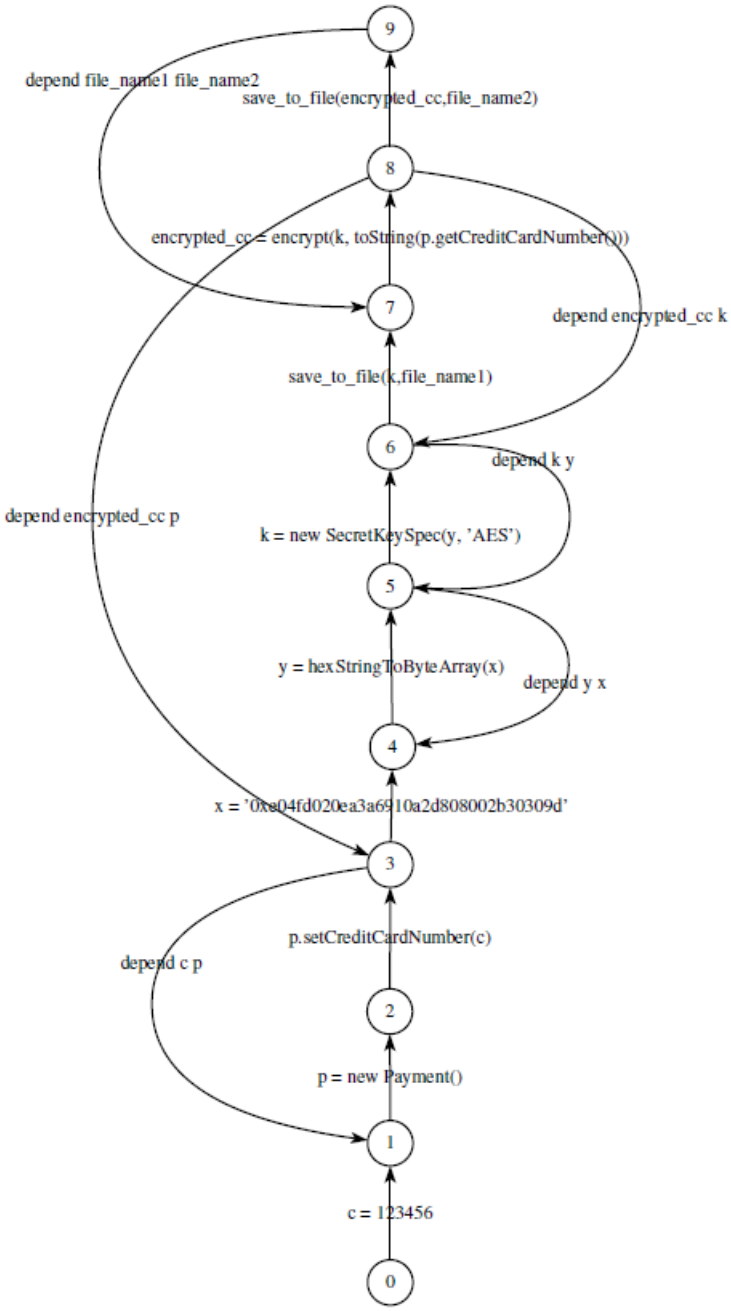

Figure 6: Labelled Transition System for the sample code given in Figure 1

Once the behavioral models are generated, we use model checking technique to automatically verify correctness of guidelines against the model.

For expressing the properties, we adopt MCL logic [21]. MCL (Model Checking Language) is an extension of 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 formula are logical formula built over regular expressions using boolean operators, modalities operators (necessity operator denoted by [ ] and the possibility operator denoted by ⟨ ⟩) and maximal fixed point operator (denoted by μ). For instance, the guideline *"Store unencrypted keys away from the encrypted data"* will be encoded directly by the following formula MCL:

```
[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 ({ }). Another particular action that we make use of in this formula is {*depend !loc1 !loc2*}, denoting the implicit dependency between the file locations *loc1* and *loc2*; we captured this implicit dependency through advanced information flow analysis on the code.

This formula means that for all execution traces, undesirable behavior never occurs (false). The unexpected behavior is expressed by this sequence of actions: if encryption key *k* is saved in *loc1*, and *k* is used to encrypt *data* that is afterwords stored in *loc2*, then if *loc1* and *loc2* are dependent, the guideline is violated. The second undesirable behavior, expressed in the second sequence of the formula, means that if encryption of data using *k* occurs before the storage of *k* in *loc1*, and if *loc1* and *loc2* are dependent, then the guideline is violated.

We made use of the checker EVALUATOR of the CADP toolbox [22] to verify the property. From a behavioral point of view, the verification result is true, indicating that the guideline is verified. However, from a security point of view the answer should be false, as the variable *xx* containing the password in plain text, was leaked to the logging operation through and the implicit flow between this variable and the logging operation. To guarantee the reliability of the analysis, one needs to check secret/sensitive variables and depending variables as in the presented formula.

No surprise the answer is false. In addition to a *false*, the model checker produces a trace illustrating the violation from the initial state, as in Figure 7.

8 Related Work

Prior work in the area of information-flow security [23] has been developed during the last decades. A line of work [24] [25] adopts the Extended Static Checking, a specific technique for finding source code errors at compile-time. Eau Claire [21] framework operates as follows; it translates C program into Guard 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. [26] 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 [27] 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 [28] 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 it 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.

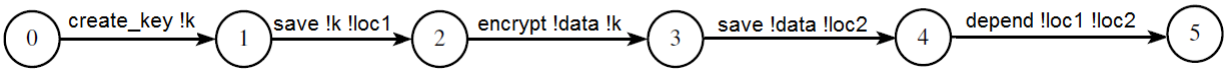

Figure 7: Path violating the guideline

GraphMatch [29] 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 [30] 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 specify constraints on how user data can be handled, through the clauses ACCEPT and DENY [**?**]. 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.

9 Discussion

We tackled in this paper the problem of verifying the adherence of the developed software to security guidelines that are presented in formal language. We raised the main issue regarding the interpretation, implementation and verification of the guidelines that are written in natural language, which might be subject to misinterpretation by developers. We worked towards stripping away ambiguities in [19] capturing implicit information flows that can be source of information leakage. We provided a centralized repository (Security Knowledge Base) gathering the security guidelines and patterns that we have discussed in detail in [19]. Apart from the patterns, we have centralized the labels that are used to build the formulas, mapped to the traditional information annotations (*SOURCE*, *SINK* and *DECLASS*) together with their security level (*HIGH* / *LOW*).

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 [4] 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 made 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 us 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**.

As part of the Security Knowledge Base, we have also considered the vulnerabilities the program could eventually be exposed to if the guideline is violated. The Security Knowledge Base is rich yet extensible repository, that can be extended if new security concepts are introduced. For instance, the same guideline might be expressed through different MCL formulas and using different terms that are semantically equivalent. Let us take the example of the guideline *"Store unencrypted keys away from the encrypted data"* that we have previously formulated using those keywords: *create_key*, *save* and *encrypt*. Among the keywords contained in the dictionary that we provide to the security expert, 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.

10 Conclusion

In this paper, 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

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

presented presents ambiguities, and this might lead to misinterpretation. Formalizing the guidelines would help strip away ambiguities, and prepare the ground for the formal verification. We stressed on the need for 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 leads this verification.

References

1. Zeineb Zhioua and Stuart Short and Yves Roudier, "Towards the Verification and Validation of Software Security Properties Using Static Code Analysis", in International Journal of Computer Science: Theory and Application,
2. Jon Heffley, Pascal Meunier, "Can Source Code Auditing Software Identify Common Vulnerabilities and Be Used to Evaluate Software Security?", in Proceedings of the 37th Hawaii International Conference on System Sciences-2004, https://doi.org/10.1109/HICSS.2004.1265654
3. Zeineb Zhioua and Stuart Short and Yves Roudier, "Static Code Analysis for Software Security Verification: Problems and Approaches", in 2014 IEEE 38th Annual International Computers, Software and Applications Conference Workshops,
4. Zeineb Zhioua and Yves Roudier and Stuart Short and Rabea Ameur-Boulifa, "Security Guidelines: Requirements Engineering for Verifying Code Quality", in ESPRE 2016, 3rd International Workshop on Evolving Security and Privacy Requirements Engineering
5. OWASP, Cryptographic Storage Cheat Sheet
6. Oracle, Secure Coding Guidelines for Java SE
7. CERT, SEI CERT Oracle Coding Standard for Java
8. NSA, Juliet Test Suite
9. National Institute of Standards and Technologies and Elaine Barker, "Recommendation for Key Management", 2016, http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf
10. Durumeric, Zakir and Kasten, James and Adrian, David and Halderman, J Alex and Bailey, Michael and Li, Frank and Weaver, Nicolas and Amann, Johanna and Beekman, Jethro and Payer, Mathias and others, "The matter of heartbleed", in Proceedings of the 2014 Conference on Internet Measurement Conference
11. Sabelfeld, Andrei and Sands, David, "Declassification: Dimensions and Principles"
12. Denning, Dorothy E. and Denning, Peter J, "Certification of Programs for Secure Information Flow", https://doi.org/10.1145/359636.359712
13. Hammer, Christian and Krinke, Jens and Snelting, Gregor, "Information flow control for java based on path conditions in dependence graphs" in IEEE International Symposium on Secure Software Engineering
14. Jrgen Graf and Martin Hecker and Martin Mohr and Gregor Snelting, "Checking Applications using Security APIs with JOANA", in 8th International Workshop on Analysis of Security APIs
15. Ferrante, Jeanne and Ottenstein, Karl J. and Warren, Joe D., "The Program Dependence Graph and Its Use in Optimization", in ACM Trans. Program. Lang. Syst., July 1987, https://doi.org/10.1145/24039.24041
16. Jurgen Graf and 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)
17. Leavens, Gary T and Baker, Albert L and Ruby, Clyde, "JML: a Java modeling language" in Formal Underpinnings of Java Workshop (at OOPSLA98)
18. Potter, Ben and Till, David and Sinclair, Jane, "An Introduction to Formal Specification and Z"
19. Lano, Kevin, "The B language and method: a guide to practical formal development"
20. A. Arnold, "Finite transition systems. Semantics of communicating systems"
21. Sabelfeld, Andrei and Myers, Andrew C, "Language-based information-flow security", in IEEE Journal on selected areas in communications
22. Mateescu, Radu and Thivolle, Damien, "A Model Checking Language for Concurrent Value-Passing Systems" in "Proceedings of the 15th International Symposium on Formal Methods" , https://doi.org/10.1007/978-3-540-68237-0_12
23. Garavel, Hubert and Lang, Frederic and Mateescu, Radu and Serwe, Wendelin "CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes" in "Tools and Algorithms for the Construction and Analysis of Systems: 17th International Conference, TACAS 2011", https://doi.org10.1007/978-3-642-19835-9_33
24. Flanagan, Cormac and Leino, K. Rustan M. and Lillibridge, Mark and Nelson, Greg and Saxe, James B. and Stata, Raymie, "PLDI 2002: Extended Static Checking for Java", http://doi.acm.org/10.1145/2502508.2502520
25. Chess, Brian V "Improving computer security using extended static checking"
26. De Francesco, Nicolette and Santone, Antonella and Tesei, Luca "Abstract Interpretation and Model Checking for Checking Secure Information Flow in Concurrent Systems"
27. Serge Autexier and Dieter Hutter and Bruno Langenstein and Heiko Mantel and Georg Rock and Axel Schairer and Werner Stephan and Roland Vogt and Andreas Wolpers, "VSE formal methods meet industrial needs", https://doi.org/10.1007/s100099900022
28. Akeel, Fatimah and Salehi Fathabadi, Asieh and Paci, Federica and Gravell, Andrew and Wills, Gary, "Formal Modelling of Data Integration Systems Security Policies", https://doi.org/10.1007/s41019-016-0016-y
29. John Wilander and et al, "Pattern Matching Security Properties of Code using Dependence Graphs"
30. Andrew, Johnson and Lucas, Waye and Scott, Moore, "Exploring and Enforcing Security Guarantees via Program Dependence Graphs", https://doi.org/10.1145/2737924.2737957

"Formal specification of security guidelines for program certification"

"Abstract Interpretation and Model Checking for Checking Secure Information Flow in Concurrent Systems" in Proceedings of Fundamenta Informaticae, 2003