Tracking Dependent Information Flows

Zeineb Zhioua[1], Yves Roudier[2], Rabea Boulifa Ameur[3], Takoua Kechiche[4] and Stuart Short[4]
[1]*SAP Labs France/EURECOM*
[2]*I3S-CNRS, Universite de Nice Sophia Antipolis*
[3]*Telecom ParisTech*
[4]*SAP Labs France*
zeineb.zhioua@sap.com, yves.roudier@i3s.unice.fr, rabea.ameur-boulifa@telecom-paristech.fr, takoua.kechiche@sap.com, stuart.short@sap.com

Keywords: Security Guidelines, Formal Specification, Model Checking, Information Flow Analysis, Program Dependence Graph, Labeled Transition System

Abstract: Ensuring the compliance of developed software with security requirements is a challenging task due to imprecision on the security guidelines definition, and 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 life cycle by combining the model checking together with information flow analysis. We formally specify security guidelines that involve dependent information flows as a basis to lead formal verification through model checking, and provide precise feedback to the developer.

1 INTRODUCTION

Security guidelines are mainly meant to specify 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 security, and consequently, to reduce the program exposure to vulnerabilities when delivered and running in the execution environment. Security guidelines are defined by the security expert(s) in the *Requirements* phase of the development lifecycle, and include essential elements (Chen, 2011) such as the object or the asset to be protected (user private information for example), the goal (required security property), and the security mechanisms to be applied in order to ensure the requirement satisfiability. Security guidelines have also been defined by different organizations such as CERT Coding Standard (CERT, b) and OWASP (OWASP, c) (OWASP, b). They introduce good programming practices to be followed by developers to ensure the security of sensitive assets. However, guidelines suffer from ambiguities and the lack of precision (Zhioua et al., 2016), and are usually presented in an informal and imprecise way. Huge effort was carried out to build the guidelines catalogs, but to provide the means allowing the automatic verification of the adherence to those guidelines that require security expertise to interpret, implement and verify them.

We propose in this paper our approach that provides the means to formally specify the security guidelines, and to verify their satisfiability using formal proofs.

The paper is organized as follows; Section 2 provides the motivation behind this work. In Section 3, we depict our approach and present its main phases. Section 4 illustrates the enhancements we carried out on the Program Dependence Graph construction. In Section 5, we formalize a security guideline in MCL Formalism (Model Checking Language), and we validate our formal specification and verification on a concrete example. Section 6 discusses some limitations of our approach, followed by a discussion on existing approaches that dealt with guidelines specification. Section 8 concludes the paper.

2 MOTIVATION

2.1 Information Flow Analysis

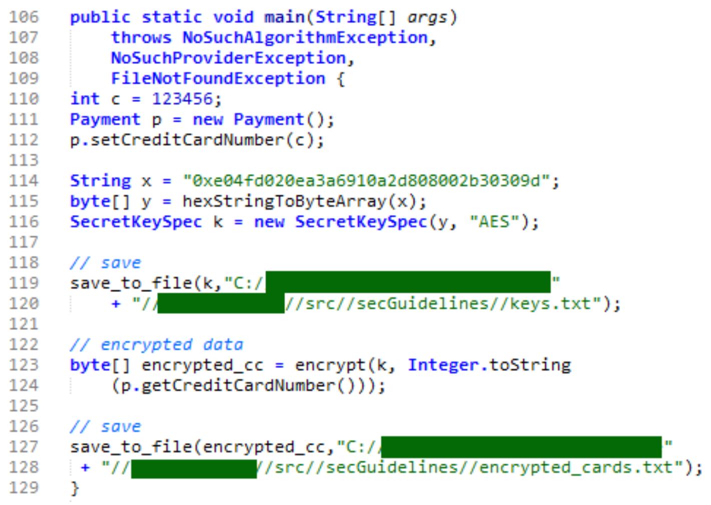
Different security mechanisms, such as access control and encryption allow to protect sensitive data, but they fall short in providing assurance about where and how the data will propagate, where it will be stored, or where it will be sent or processed. This entails the need for controlling information flow using static code analysis. This same idea is emphasized by Andrei Sabelfeld, and Andrew C. Myers (Sabelfeld and Sands, 2009), who deem necessary to analyze how the information flows through the program. The main objective of information flow analysis (Denning and Denning, 1977) is to verify that the program satisfies data confidentiality and integrity policies.

2.2 Security Guidelines

The OWASP Foundation (OWASP, a) 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 (Zhioua et al., 2016). In the OWASP Storage Cheat Sheet (OWASP, b), OWASP introduces the guideline ”Store unencrypted keys away from the encrypted data” [1] explaining the encountered risks when the encryption key is stored in the same location as encrypted data. This guideline recommends to store encryption key and the encrypted data in different locations. As the reader can see, the guideline involves 2 information flows (encryption key and encrypted data) that are dependent. Their specification and identification on the code level require an advanced information flow analysis capable of handling complex information flows, such is the case for this guideline. 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.

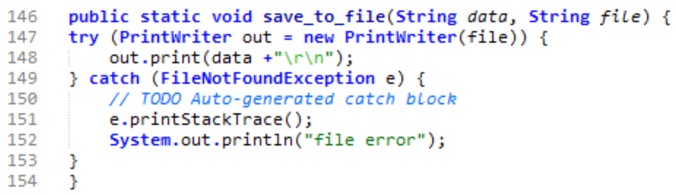
2.3 Sample Code

Let’s analyze the sample code in Figure 1 to verify whether the guideline ”Store unencrypted keys away from the encrypted data” is met or not. The developer Bob encrypts the secret data credit card number, and stores the cipher text into a file. At line 115, Bob creates a byte array y used as parameter for the instantiation of a SecretKeySpec named k (line 116). At line 119, Bob stores key k in a file, through the invocation of method *save_to_file* (Figure 2). Once created, key k is provided as parameter to the method *save_to_file(String data, String file)* (Figure 2) Bob 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 2). As the reader can see, 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 *encypted_cc* are stored in separate files. However, the two files are located in the same file system, which constitutes a violation of the guideline. The specification and identification of the dependent information flows between key k, *encrypted_cc* and file location is a challenging task that we could achieve through the framework we propose in this paper.



```
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



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

Figure 2: save_to_file method

3 APPROACH

We propose our framework that enables the formalization of security guidelines and the exploitation

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

of this specification in the implementation and Verification&Validation phases of the engineering process. Our framework is based on formal proofs for the translation of security requirements into good programming practices. We focus mainly on verifying whether those good programming practices are met or not. Figure 3 illustrates our approach and highlights the relevant phases for the transformation of security guidelines from natural language into exploitable formulas that can be automatically verified over the program to analyze.

We aim at separating the duties and make the distinction between the main stakeholders in our framework; the *security expert(s)* and the *developer*.

In a preliminary phase, the *security expert(s)* carries out the translation of security guidelines from natural language into a formal way (Section 3.1). This phase results in generic security guidelines that can be instantiated on different programs logics.

The developer writes the code and invokes the framework that verifies whether the rules specified by the security expert are correctly applied in his software. First, the framework constructs the program model (Section 3.2), that is used as a basis to lead the formal verification of the security guidelines (Section 3.3). We stress that the developer does not have to deal with the specified formulas.

We aim at reducing the intervention of the security expert for the identification on the code of the critical data that are at the heart of the security guidelines, such as encryption key in our example. However, in some other cases, the identification of sensitive data requires knowledge and awareness about the code logic and semantics.

One crucial step of the work is the explicit mapping between abstract security guidelines formal specification, and concrete statements on the code. This is handled in the Security Knowledge Base (Section 3.4). In our framework, we do not aim at proving the program correct, but to verify that it adheres to specific security guidelines written, formulated and formalized by security expert(s).

The proposed framework is depicted in Figure 3:

3.1 Step 1: Formal specification of security guidelines

We make the strong assumption that the **security expert** formally specifies the security guidelines by extracting the key elements (the labels), and builds upon them the formulas/patterns based on formalism. The built formulas can be supported by standard model checking tools. One crucial operation in this phase is the specification of simple as well as dependent information flows, such is the case for the guideline we consider in this paper. The outcome of this phase is generic security guidelines that can be instantiated on different codes. 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 (Section 3.4).

3.2 Step 2: Construction of Augmented Program Dependence Graph

We aim at constructing a data structure enabling the representation and the extraction of multiple information flows at the same time. The outcome of this step is a Program Dependence Graph (PDG) augmented with information and details obtained from deep dependency analysis on the program. The standard PDG contains both control and data dependencies between program instructions, and has the ability to represent information flows in the program.

We make use of the JOANA IFC tool (Graf et al., 2013) (Graf et al., 2015) to construct the standard PDG. The generated PDG is then augmented with details and information extracted from the verification of security guidelines formulas and patterns.

In our framework, we carry out the information flow analysis using the JOANA tool (Graf et al., 2013) 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. JOANA analyzes java byte code for the non-interference property, which demands that public events should not be influenced by secret data. The analysis performed was formally proven using Isabelle (Wasserrab et al., 2009).

Information Flow Analysis aims at capturing the different dependencies that may occur between the different PDG nodes, hence, augmenting the generated standard PDG with relevant details, such as annotations mapping the PDG nodes to abstract labels of the security guidelines. Possible mappings between Java APIs and abstract labels are handled in the **Security Knowledge Base** (Section 3.4).

3.3 Step 3: Formal Verification

This step aims at constructing from the augmented PDG a formal graph that is accepted by model checking tools: Security Labeled Transition System **Sec_LTS**, which is an augmented LTS (Labeled Transition System) accepted by a model checking tool.

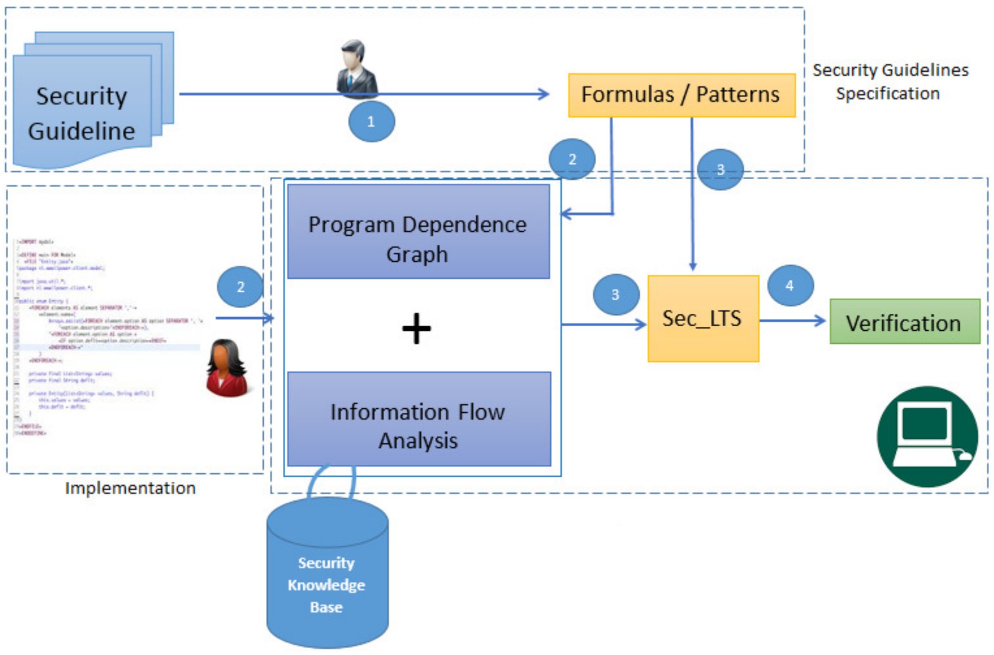


Figure 3: End-to-end Approach

This step is depicted in details in Section 5.2. 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 Security_LTS program representation model through model checking.

The verification phase can have the following outcomes:

- The security guideline is valid over all the feasible paths.
- The security guideline is violated, and the violation traces are returned.

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.

3.4 Security Knowledge Base

Security Knowledge Base is a centralized repository gathering the labels of the formulas mapped to APIs, instructions, libraries or programs. This will help the automatic detection of labels on the system model. 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 expert. We populated the **Security Knowledge Base** using a Java classes parser that we developed [2]; for the different Java classes used in the program to analyze, we launch programmatically 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 is performed a semi-automatic semantic analysis to detect key elements from the Java methods details (return type, method description, etc.), such as the key-word *secure*, *key*, *print*, *input*, etc. Then, the security expert establishes the mapping of those key words used to build the formulas 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

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

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.

4 AUGMENTED PDG

The starting key element for this step is the standard PDG generated by the JOANA tool (Graf et al., 2015) from the program bytecode. 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 JOANA failed to capture some of them. For instance, We noticed that there are implicit dependencies that are not captured (such as Java Reflection dependencies), and they constitute the source of the undetected violations, such is the case for storage location we consider in this paper. We carried out the effort of enhancing the PDG and capturing the missing dependencies that we translate into edges on the PDG.

4.1 Automatic annotations

The JOANA tool proposes two kinds of annotations together with their security levels specifying the source (*SOURCE*) and the target (*SINK*) of the information flow, in addition to the *DECLASS* annotation allowing to reduce the security level of the annotated node. We made modifications on the source code of JOANA tool, and added customized annotations referring to the abstract labels such as *hash*, *userInput*, *isPassword*, *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, and here we refer to the **Security Knowledge Base** (Section 3.4) that already contains the concrete possible mappings between known APIs, methods, methods parameters mapped to the abstract labels of the security guidelines specification. For instance, the *SecretKeySpec* object *k* is instantiated at line 116. The constructor *SecretKeySpec(byte[],String)* stands out in our Security Knowledge Base as method invocation mapped to the abstract label *create_key*. Hence, the variable *k* (line 116) is annotated ***create_key***. In the sample code of Figure 1, 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 en-

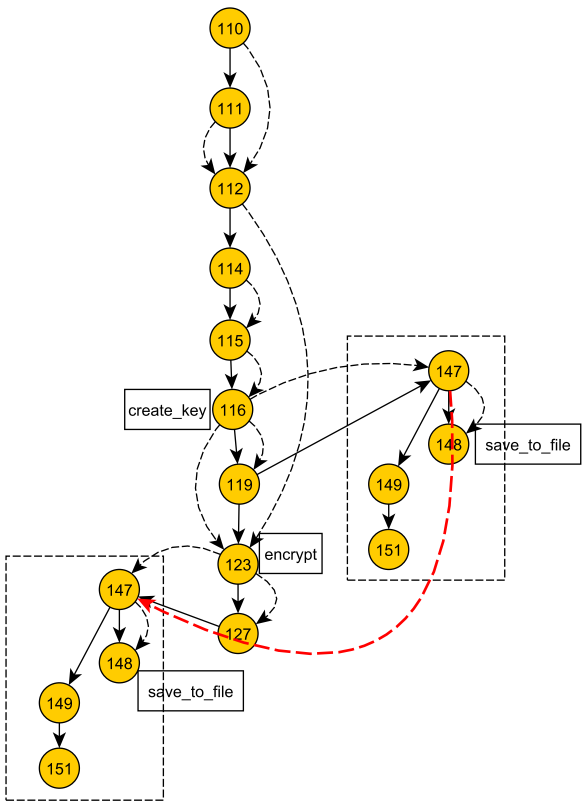


Figure 4: Augmented Program Dependence Graph for the sample code. Strong edges represent the control flows, the dashed edges refer to explicit and implicit data flows. Nodes are labeled with their corresponding instructions line numbers

cryption operation or storage. The semantic analysis is not in the scope of this paper. We can also be faced with the case where the methods are declared with insignificant names, which makes the automatic annotations unfeasible. In Figure 3, we provide the method save_to_file. As the reader can see, at line 148, Bob invokes the method *PrintWriter.print(String)* that is mapped in the Security Knowledge Base to the label save_tofile. The augmented PDG is represented in Figure 4.

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.

4.2 Multiple annotations on the same node

JOANA tool offers the possibility to annotate a node with one single kind 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 same data (the same node) is at the heart of more than one guideline.

4.3 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.

4.4 File location dependency detection

In the guideline ***Store unencrypted keys away from encrypted data***, we encounter the imprecise and implicit 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. For example, the set **save** contains labels such as *save_to_file*, *save_to_database*, *save_to_array*, etc. 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 Figure 4.

5 FORMAL VERIFICATION

With the objective of proposing a framework that provides help and guidance to developer in verifying that his program satisfies given security guidelines, we translate java programs into a formal description (e.g., finite state machines, process algebra, etc.), which is precise in meaning and amenable to formal analysis. As our main objective is to automatically verify programs, 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. Indeed, model checking is an automatic technique for verifying behavioral properties of a system model by an exhaustive enumerating of its states. In order to carry out this operation, we need first to express security guidelines in the suitable formalism.

5.1 Formal Specification of Security Guidelines

For expressing the properties, we use MCL logic (Mateescu and Thivolle, 2008). 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 formula are logical formula built over regular expressions using boolean operators, modalities operators (necessity operator denoted by $[\,]$ and the possibility operator denoted by $\langle\,\rangle$) 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.

5.2 From Program Dependence Graph to Labeled Transition System

We focus in this section on the transformation of augmented PDG into a formal model allowing to proceed to the formal verification using model checking techniques.

As usual in the setting of distributed and concurrent applications, we give behavioral semantics of analyzed programs in terms of a set of interacting finite state machines, called LTS (Arnold, 1994).

Definition 1 (Labeled Transition System)**.** *A Labeled Transition System (LTS for short) is a 4-tuple* $\langle Q, q_0, L, \rightarrow\rangle$ *where*

- *Q is a finite set of states;*
- *$q_0 \in Q$ is the initial state;*
- *L is a countable set of labels;*
- *$\rightarrow \subseteq Q \times L \times Q$ is the transition relation.*

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.

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 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.

It is important to note that the augmented PDG and the LTS are isomorphic, in a sense that both graphs have the same number of edges and the same number of nodes. LTS is similar to the augmented PDG, except that the LTS labels are on edges (transitions) and not on nodes. The same actions (instructions) on the PDG nodes are translated into transitions on the LTS, 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 for faithfulness of the analysis support, and also to export the analysis results (violating traces) on the PDG built from the source code. Figure 5 represents the LTS corresponding to the augmented PDG of Figure 4.

5.3 Model Checking

We can carry out the model-checking analysis directly on the LTS. For the sake of simplicity, *hiding* and *renaming* are used to compute a minimized Labeled Transition System, which is an "operable" model (see Figure 6). First, some irrelevant actions (for the analyzed properties) are "hidden";they are replaced by τ actions (denoted *i* in Figure 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 (Lang et al., 2002) to verify the property we formalized in Section 5.1.

The verification 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 7.

We are able to track all variables within our model, in fact even if variable changes name in the code we can from original guideline generate automatically an appropriate formula to check the properties with the new name. For instance, the property can be 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})
```

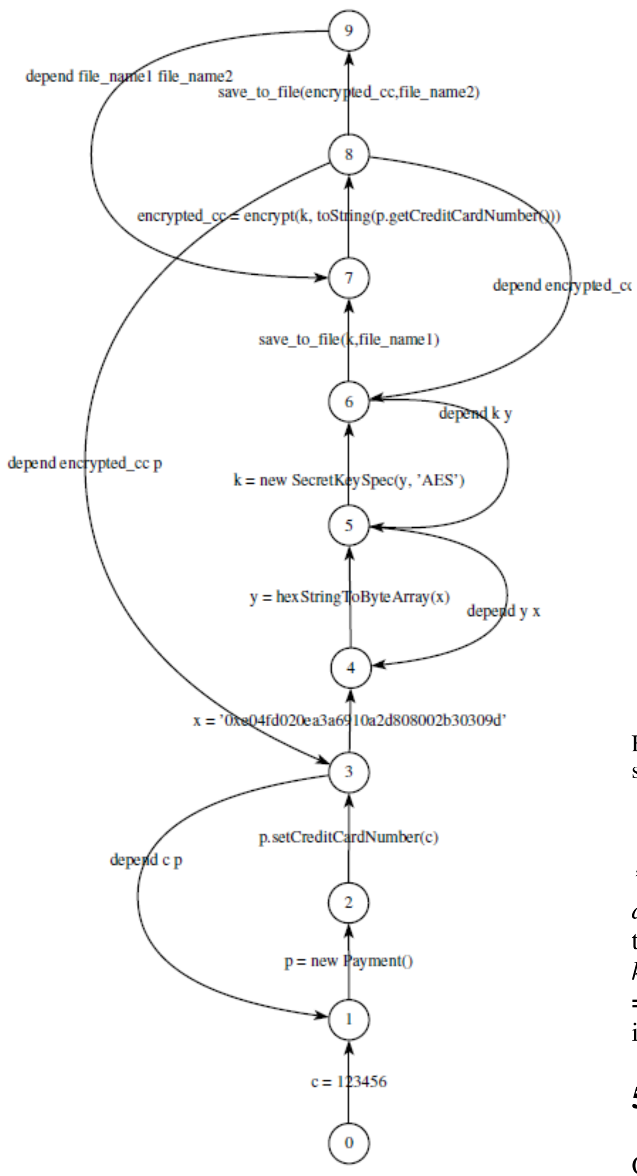

9
depend file_name1 file_name2
save_to_file(encrypted_cc,file_name2)
8
encrypted_cc = encrypt(k, toString(p.getCreditCardNumber()))
7
depend encrypted_cc
save_to_file(k,file_name1)
6
depend k y
depend encrypted_cc p
k = new SecretKeySpec(y, 'AES')
5
y = hexStringToByteArray(x)
depend y x
4
x = '0x804fd020ea3a6910a2d808002b30309d'
3
p.setCreditCardNumber(c)
depend c p
2
p = new Payment()
1
c = 123456
0

Figure 5: Labeled Transition System for the sample code given in Figure 1

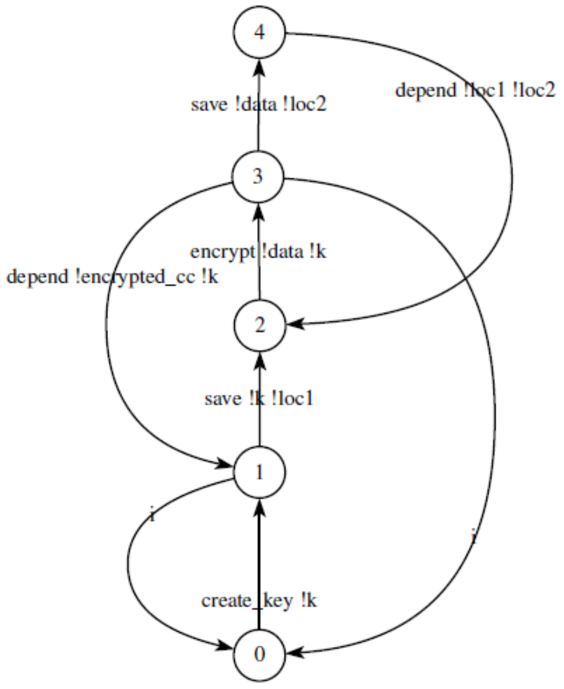


Figure 6: Minimized Labeled Transition System for the sample code given in Figure 1

```
|
  ({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
```

This MCL formula encodes the same guideline *"Store unencrypted keys away from the encrypted data"*. It allows to specify further dependencies and their combinations; it checks also the case where key *k* is renamed (by introducing the instruction ***String x = k.toString()*** between line 116 and 118, and renaming *k* by *x* in the rest of the code).

5.4 Feedback to the developer

One crucial step in our framework is the feedback representation to the developer in a way that allows him to understand the source of the violation, and to be able to make the needed corrections to fix it. It is then necessary to export the model checking output on the PDG built from the source code, as it is the closest representation of the program code.

JOANA tool generates the PDG from the Java byte code, and performs the analysis on the PDG. However, interpreting the results on the byte code level is not straightforward, neither is its mapping to the source code. With the objective of covering this shortcoming, we built a PDG from the program source code, and we made possible the bidirectional mapping between the PDG source code and the PDG byte code. The two graphs are augmented as explained in Section 4.

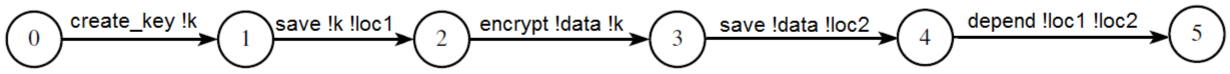


Figure 7: Violation trace

The verification carried out on the Sec_LTS returns the trace(s) violating the security guideline; those results are then returned on the PDG we constructed from the source code to provide more details to the developer about where and why the violation occurred. The returned trace (Figure 7) shows the exact path where the guideline violation occurred.

6 LIMITATION

The framework we proposed in this paper constitutes a proof-of-concept regarding the feasibility of our approach. However, our framework falls short in specifying guidelines involving imprecise and ambiguous notions such as guideline IDS15-J:***Do not allow sensitive information to leak outside a trust boundary*** (CERT, a) from CERT Coding Standard source (CERT, b). The notion of *trust boundary* requires well defined system boundaries, which is not trivial when it comes to distributed applications.

Other guidelines involve quantitative information flow theories, and this is also some other limitation of the formal specification we cover in our framework. For instance, the guideline ***Limit quantity of data encrypted with one key*** 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 and verification method falls short in covering this guideline for the dynamic aspect it involves, and that can't be captured statically.

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.

7 RELATED WORK

The specification and verification of security guidelines have also been addressed in the literature.

In the technical report (Aderhold et al., 2010) of the joint work between TU Darmstadt and Siemens AG, the authors 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 leverage 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 developers.

SecureDIS (Akeel et al., 2016) 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 at 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 (Wilander and Fak, 2005) (Wilander and Fak,),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.

The Jif (Myers and Liskov, 2000) language implements type-checking that makes use of Decentralized Label Model (DLM) (Myers and Liskov, 1997); it allows to define a set of rules to be followed by programs to prevent information leakage. Jif programs are type-checked at compile-time, which ensures type-safety as well as that rules are applied. However, the labels, which define policies for use of the data, apply only to a single data value, and are not checked at run-time.

Dimitrova et al. (Dimitrova et al., 2012) proposed an approach that integrates the dynamic analysis into the monitoring of information flow properties. The authors proposed an extension to LTL *SecLTL* taking into account the information flow properties such as non-interference and declassification. *SecLTL* was then used as a specification for model checking. 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.

In their work "Detecting Temporal Logic Predicates in Distributed Programs Using Computation Slicing" (Sen and Garg, 2003), Alper Sen and Vijay K.Grag adopted an approach that models the possible executions of the program in finite traces of events, and performs "computational slicing", that is, slicing with respect to a global predicate. Their approach is based on the dynamic behavior of the program, which requires a sufficient number of test cases and is quite time-consuming, yet it cannot ensure a verification of the entire set of paths of the program to analyze.

Aoraï plugin (Stouls and Prevosto,) 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 of 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. The tool is only focused on the control dependencies between method calls, and the analysis is not extended to the data level.

PIDGIN (Andrew et al., 2015) introduces an approach similar to our work. The authors propose the use of PDGs to verify security guidelines. 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. 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. It is not also explained how the feedback will be presented to the developer, or how we might be guided through the correction phase.

8 CONCLUSION AND FUTURE WORK

We presented in this paper 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. We stressed the difficulty encountered when the security guideline involves dependent information flows that can't be specified separately. This requires security expertise to specify the dependent information flows. We make the strong assumption that the security expert extracts the key concepts from the guidelines textual descriptions and builds upon them the formulas using the MCL formalism. 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.

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. 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 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.

REFERENCES

Aderhold, M., Cu?llar, J., Mantel, H., and Sudbrock, H. (2010). Exemplary formalization of secure coding guidelines. Technical report, TU Darmstadt and Siemens AG.

Akeel, F., Salehi Fathabadi, A., Paci, F., Gravell, A., and Wills, G. (2016). Formal modelling of data integration systems security policies. *Data Science and Engineering*, pages 1–10.

Andrew, J., Lucas, W., and Scott, M. (2015). 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.

Arnold, A. (1994). *Finite transition systems. Semantics of communicating sytems*. Prentice-Hall. ISBN 0-13-092990-5.

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

CERT. Sei cert oracle coding standard for java.

Chen, Z., editor (2011). *Specification and Management of Security Requirements for Service-Based Systems.* Proquest, Umi Dissertation Publishing.

Denning, D. E. and Denning, P. J. (1977). Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513.

Dimitrova, R., Finkbeiner, B., and Rabe, M. N. (2012). *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.

Graf, J., Hecker, M., and Mohr, M. (2013). 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.

Graf, J., Hecker, M., Mohr, M., and Snelting, G. (2015). Checking applications using security apis with joana. 8th International Workshop on Analysis of Security APIs.

Lang, F., Garavel, H., and Mateescu, R. (2002). An overview of cadp 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4.

Mateescu, R. and Thivolle, D. (2008). 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. Springer-Verlag.

Myers, A. C. and Liskov, B. (1997). A decentralized model for information flow control. *SIGOPS Oper. Syst. Rev.*, 31(5):129–142.

Myers, A. C. and Liskov, B. (2000). Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442.

OWASP. About the open web application security project.

OWASP. Cryptographic storage cheat sheet.

OWASP. Owasp secure coding practices quick reference guide.

Sabelfeld, A. and Sands, D. (2009). Declassification: Dimensions and principles. *J. Comput. Secur.*, 17(5):517–548.

Sen, A. and Garg, V. K. (2003). Detecting temporal logic predicates in distributed programs using computation slicing. In *Principles of Distributed Systems, 7th International Conference, OPODIS 2003 La Martinique, French West Indies, December 10-13, 2003 Revised Selected Papers*, pages 171–183.

Stouls, N. and Prevosto, V. Aorai plugin tutorial – frama c.

Wasserrab, D., Lohner, D., and Snelting, G. (2009). On pdg-based noninterference and its modular proof. In *Proceedings of the 4th Workshop on Programming Languages and Analysis for Security*, pages 31–44. ACM.

Wilander, J. and Fak, P. Modeling and visualizing security properties of code using dependence graphs.

Wilander, J. and Fak, P. (2005). Pattern matching security properties of code using dependence graphs.

Zhioua, Z., Roudier, Y., Short, S., and Boulifa Ameur, R. (2016). 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.