Formal Specification of Security Guidelines for Program Certification

Zeineb Zhioua
EURECOM
zeineb.zhioua@eurecom.fr

Yves Roudier
UCA - I3S - CNRS
Université de Nice Sophia Antipolis
yves.roudier@i3s.unice.fr

Rabea Ameur-Boulifa
LTCI, Télécom ParisTech,
Université Paris-Saclay
rabea.ameur-boulifa@telecom-paristech.fr

***Abstract*—Secure software can be obtained out of two distinct processes: security by design, and security by certification. The former approach has been quite extensively formalized as it builds upon models, which are verified to ensure security properties are attained and from which software is then derived manually or automatically. In contrast, the latter approach has always been quite informal in both specifying security best practices and verifying that the code produced conforms to them. In this paper, we focus on the latter approach and describe how security guidelines might be captured by security experts and verified formally by developers. Our technique relies on abstracting actions in a program based on modularity, and on combining model checking together with information flow analysis. Our goal is to formalize the existing body of knowledge in security best practices using formulas in the MCL language and to conduct formal verifications of the conformance of programs with such security guidelines. We also discuss our first results in creating a methodology for the formalization of security guidelines.**

***Index Terms*—Security Guidelines, Security Best Practices, Program Certification, Information Flow Analysis, Model Checking, Labelled Transition Systems**

I. Introduction

Organizations and companies develop very complex software today. Errors and flaws can be introduced at different phases of the software development lifecycle and lead to exploitable vulnerabilities. Establishing a set of security objectives helps produce secure software. However, ensuring that some program complies with those security objectives is a challenging task. Two techniques can be adopted: security by design and security by certification. The former technique, also called model-driven engineering for security, has been extensively developed in the academic community, and follows a specify, model, verify, and then implement approach. It is well suited to classical software development featuring for instance an organized V-cycle, and security objectives generally correspond to security properties expected from software components or communication protocols (see for instance [1]). In contrast, the latter technique follows a build, then detect flaws approach, based on the description of security best practices and programming style and idioms. Apart from the safety-critical domain, this approach has received a much wider acceptance in the industry because security is often addressed from a developer point of view, in a programmatic style, rather than from a security architect perspective, as through a comprehensive model. A number of security guidelines is defined forming a program-oriented security policy that the developer has to adhere to. This model is also adapted to very different styles of software engineering including agile methods. Unfortunately, this technique lacks automation and formality in the way best practices are specified and verified.

In practice, companies following the security by certification approach perform review processes and conduct audit sessions (e.g.,[2]) comprising a wide range of experiments to verify the compliance with general and security related policies. A number of certification processes like the Common Criteria methodology, for instance, account for such reviews in the way security objectives and the target of certification are specified. Manual reviews can be time-consuming and very costly to companies in terms of human and financial resources, and they can fall short in detecting security policy violations. From a developer perspective, interpreting and applying security policies is not trivial, as those policies, translated into security guidelines, are often written in an informal style, use ambiguous language, and require domain expertise to interpret them [3].

In this paper, we focus on how to capture the security guidelines defined by security experts that developers have to follow in the security-by certification approach. Most notably, our ambition is firstly to help security experts disambiguate the complex guidelines described above by formalizing them. We make use of logical formulas based on the MCL language in order to express such guidelines. Our technique relies on building formulas based on actions abstracted from the program API, that is the different software modules and their interfaces.

We secondly aim to assist developers in their task by automating and speeding the conformance tests that must normally be carried out after development. Such tests usually take weeks before a program can be deemed secure or simply thrown back to the developer for rework. Such situations typically arise in the framework of mobile applications stores, that usually take up to a few days of code review. To this end, our verification technique analyzes MCL formulas defining security guidelines through a combination of information flow analysis and model checking. This notably makes it possible to analyze not only explicit, but also implicit data dependencies. In addition, keeping our analysis as close as possible to the

978-1-5386-1925-4/17/$31.00

actual program code makes it easier to provide a meaningful feedback to developers as we show in an example discussed in Section III.

To illustrate our discussion, we will use throughout the paper the sample program depicted in Figure I. In this program, user credentials (user name and password) are provided as inputs (lines 64 and 65).

```
58      BufferedReader reader = new BufferedReader
59              (new InputStreamReader(System.in));
60      System.out.println("User name : ");
61      System.out.println("Password : ");
62
63      // input
64      user.setUsername(reader.readLine());
65      user.setPassword(reader.readLine());
66
67      // copy password
68      String xx = user.getPassword();
69
70      // hash
71      MessageDigest hash = MessageDigest.getInstance("MD5");
72      byte[] bytes_password = user.getPassword().getBytes("UTF-8");
73      byte[] hash_password = hash.digest(bytes_password);
74
75      user.setPassword(hash_password.toString());
76
77      // log
78      logMessage = "user name = " + user.getUsername() + ","
79              + " password = " + user.getPassword() + " " + xx;
80      logger.log(Level.INFO, logMessage);
```

Fig. 1. Account creation sample code

Suppose that we want to verify whether the sample code adheres to the guideline MSC62-J: *"Store passwords using a hash function"* from the CERT secure coding standards. If we want to verify if the sample code is compliant with this specific guideline that recommends to hash passwords before storing them, we would notice the following: the password variable was assigned to *user.setPassword* at line 65. Then, this sensitive data was hashed using the MD5 method at line 73. The developer then invoked the logging operation (*logger.log*) at line 80, which is also seen as storage operation on log files. We might then draw different conclusions: from pure control flow angle, the guideline in terms of methods invocation was met. The password was hashed before being stored (logged), and this is the flow that the guideline recommends. However, if we take a closer look at the propagation of the password data, we would notice that it was assigned to another variable named *xx* at line 68. It is true that the digest of the password was stored, but the password in plain text contained in the variable *xx* was also stored, which constitutes a clear violation of the guideline that will not be detected if we do not include the analysis on the information flow level. Nowadays, formal methods, in particular formal verification are increasingly being used to enforce security and safety of programs.

The paper flow is as follows: we provide in Section 2 an overview of the proposed approach for the specification and verification of security guidelines. This section is then followed by a presentation of security guidelines, and their specification in the Model Checking Language (MCL) in Section 3. We validate our formal specification and verification on a concrete example. In Section 5, we present existing approaches that dealt with guidelines specification. Section 6 concludes the paper and discusses the limitations as well as possible directions of our work.

II. Framework

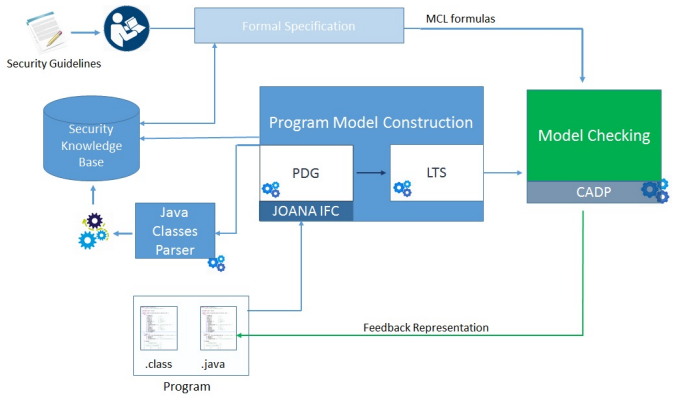

Fig. 2. Prototype for the automatic verification of security guidelines

Figure 3 presents the architecture of our framework that illustrates the proposed approach to help automating the systematic verification of the security guidelines. The figure highlights the relevant steps towards fulfilling the transformation of security guidelines from natural language to exploitable formulas that can be automatically verified over the program to analyze. Our framework is composed of mainly two big parts: the **model construction** that consists in constructing an abstraction of the program to analyze, and the **automatic verification** of the security guidelines. The detailed description of the framework is illustrated in [4].

First of all, we need to make the distinction between the main actors in our framework; the *security expert* and the *developer*. The former plays a major role in the formal specification of the security guidelines, and their transformation into mathematics-based formulas that are supported by standard model checking tools. The latter invokes the framework to verify the compliance of his developed software with the security guideline.

We refer to CERT Oracle Coding Standard for Java [5] [6] as the source of the Java good programming practice we consider as example in this paper. We show then that this guideline can be modeled in a formalism, and can be formally verified using a model checking tool.

In this paper, we focus mainly on the formal graph construction as well as on the formal specification and the formal verification of the security guidelines. The fine-grained description of the approach is provided in [4] [7]. We describe in this section the architecture of our framework in terms of implemented components and the tools we have used in order to automatically verify the compliance with the security guidelines expressed in a formal language.

The different components of our framework are automatic, except the first step that is carried out by the *security expert*.

First, he transforms the informal security guidelines from natural language into exploitable formulas expressed in the Model Checking Language (Section III.A). The security expert extracts the key-concepts or the key-words from the informal textual description of the guideline, and builds upon them the formulas. The key-words are referred to as *labels*.

As the reader can notice, this operation is manual, and requires security expertise to extract the context and the key-concepts from the guidelines informal description, and then to build upon them the formulas.

Automating this operation would be an achievement, as it will help automating the full end-to-end tool chain, and bridge the gap between the informal guidelines and their automatic verification from the developer side. The reason why we did not automate this task is that we do not have a model against which we can verify the correctness of the formulas. In addition, performing a deep semantic security text analytics on the guideline's description is not in the scope of our work. In this step, we strongly rely on the expertise of the security expert to formalize the guidelines. Validating the specification with a community of security experts may be a way forward to ensure the correctness of the proposed formulas.

The remaining components of our framework are automatic, and consist of:

a) Program Model Construction:

- Augmented Program Dependence Graph: this component builds the program dependence graph (PDG) from the Java bytecode (.class) using the JOANA IFC tool [8]. We have chosen the Program Dependence Graph (PDG) as the abstraction model for its ability to represent both control and (explicit/implicit) data dependencies. The generated PDG is then annotated by the **PDG Annotator** with specific annotations (labels in the MCL formulas). The automatic annotations are handled in the Security Knowledge Base [4]. We run the information flow analysis using the JOANA IFC, that is formally proven [8] 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**.

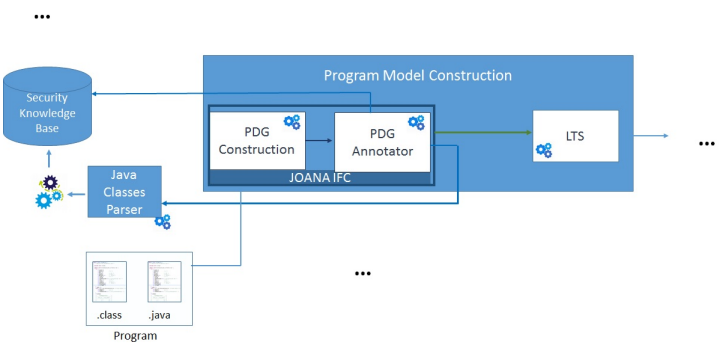

Fig. 3. Model Construction: From program sources to LTS

- LTS Construction: this component translates automatically the **Augmented PDG** into a parametrized 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 [9] takes as input the URL of the Java class official documentation [10] [11], 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.

b) Model Checking:

- Model Checking: we carry out the model-checking analysis on the pLTS that we generate from the Augmented PDG. We made use of the checker EVALUATOR of the CADP toolsuite [12] to automatically verify the security guideline expressed in MCL. The output of this component indicates whether the guideline is met, or it is violated, and the violation traces are returned.
- Feedback Representation: this component exploits the output of the "Model Checker" to provide a precise and useful feedback to the developer to understand the source of the violation, and possibly how to fix it.

III. Formal Models

In order to check automatically whether a program satisfies a given guideline, both the program and the guideline are formulated in precise mathematical languages.

A. From informal guidelines to formal specification

Security guidelines or security good programming practices are managed by different organizations such as CERT Coding Standard [6], OWASP [13], [14] , Apple App Store [15], etc. The CERT Coding Standards have been adopted by corporate companies such as Oracle and Cisco [16], and contributed also to the development of the Source Code Analysis Laboratory (SCALe) [17]. In order to have a clear understanding of the security guidelines, we conducted a deep analysis on good and bad programming practices [3] referring to positive and negative security patterns respectively. This work led to the classification of the guidelines regarding the flow type they induce and the verification operation that should be performed in order to verify the adherence or not to the specific guideline.

Our methodology requires first to formalize the security guidelines. This phase produces formal specification using formal language. The formal specification is typically a mathematics-based description of guidelines using mathematical logic. As pointed out in [3], most guidelines provide recommended safeguards that developers should follow in order to ensure compliance with the

data protection. Their distinctive features is that they are typically dealing with data parameters, and are generally abstracted away in formal models as verification problems are undecidable for infinite systems. The MCL logic [18] is the relevant language that addresses this crucial matter: representing and handling data. It allows reasoning naturally about systems described in value-passing process algebras such as LOTOS. MCL is an extension of the alternation-free regular μ-calculus with facilities for manipulating data in a manner consistent with their usage in the system definition. The MCL formulas are logical formulas built over regular expressions using boolean operators, modalities operators (necessity operator denoted by $[\ ]$ and the possibility operator denoted by $\langle\ \rangle$) and maximal fixed point operator (denoted by μ). By using data, one can specify state machines with an infinite action alphabet, in which quantification over data can be used, and propositions and variables may have data parameters.

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

Consider the guideline MSC62-J[6] stating "***Store passwords using a hash function***". It represents a typical safety property and means that the storage of a password should not be reached if the password was not hashed beforehand. This can be expressed by the following MCL formula:

$$[\texttt{true}*.\{\texttt{setPassword}\,?\texttt{msg:String}\}.\texttt{true}*.\{\texttt{store}\,!\texttt{msg}\}]\,\texttt{false}$$

The formula expresses a bad sequence of actions that we want to prevent. This sequence is specified by the following regular expression:

$$\texttt{true}*.\{\texttt{setPassword}\,?\texttt{msg:String}\}.\texttt{true}*.\{\texttt{store}\,!\texttt{msg}\}.$$

This expression tries to match sequences that begin with zero or more of any action (denoted true$*$) followed by a password creation action ($\{\texttt{setPassword}\,?\texttt{msg:String}\}$) followed by any action, and end with a storage action ($\{\texttt{store}\,!\texttt{msg}\}$). This logic also enables to express liveness properties by using the fixed point formula.

Table I shows examples of MCL formulas representing the specification of security guidelines we collected (listed informally in [3]) from several sources. Most of these programming practices that programmers should be aware of, are safety properties that can be verified by a model checker. They can be specified as security patterns usable within the validation phase of the software development lifecycle.

Similarly to Dywer's work [19], we propose a catalog of patterns for security guidelines for facilitating the programmer's task at the verification level. These patterns identified in [18] will allow programmers who are not experts in formal language to read formal specification. The automatic generation of formula from a given guideline requires first an instantiation of the labels, and this operation induces a heavy load on the programmer. We overcame this difficulty by automating the mapping of the program instructions to the labels on which the MCL guidelines formulas are built; the automatic mapping is carried out by the PDG Annotator through the Security Knowledge Base.

The proposed formalization in MCL language presents the key concepts or actions required by each of the guidelines. The challenge is to define those concepts and actions and have a unified yet extensible set of keywords that can be used by security expert(s) for the formal specification. We tried to reduce the security expert intervention by introducing the Security Knowledge Base which is a central repository containing the different labels serving to compose the guidelines, together with their description and semantic meaning. The security expert maps the labels composing the security patterns to Java methods, objects, instructions, etc.

B. Security Guidelines Verification

The model checking technique [20] involves several algorithms for the verification of systems. In our approach, the system to be verified is modeled as a finite transition system and the property is expressed as a formula of temporal logics. More specifically, this technique relies on translating the system to be verified to formal models that are precise in meaning and amenable to formal analysis, in particular accepted by model checking algorithms. In the context of our work, the models are parametrized Labelled Transition System (pLTS). pLTS extends the general notion of Labelled Transition Systems (LTS) [21] by adding parameters and value-passing features. These additional parameters were strongly required for the verification of security guidelines that we treat in our work. A parameterized LTS is an LTS with parameterized actions, with a set of parameters and variables attached to each transition.

Definition 1 (pLTS): A pLTS is a tuple $(S, s_0, L, \rightarrow)$ where:

- S is a set of states.
- $s_0 \in S$ is the initial state.

TABLE I
EXAMPLES OF CERT GUIDELINES FORMULATED IN MCL LANGUAGE

Code	Guidelines and corresponding MCL formulas
IDS01-J	*Normalize strings before validating them* $[\texttt{true}*.(\neg(\{\texttt{normalize}\,?\texttt{msg:String}\}))*.\{\texttt{validate}\,!\texttt{msg}\}]\texttt{false}$
IDS03-J	*Do not log unsanitized user input* $[\texttt{true}*.\{\texttt{userInput}\,?\texttt{msg:String}\}.(\neg(\{\texttt{sanitize}\,!\texttt{msg}\}))*.\{\texttt{log}\,!\texttt{msg}\}]\texttt{false}$
OWASP	*Store unencrypted keys away from the encrypted data* $\forall \texttt{loc1},\texttt{loc2}\ :\texttt{String}.(<\texttt{true}*.\{\texttt{create\_key}\,?\texttt{key:String}\}.\texttt{true}*.\{\texttt{save}\,!\texttt{key}\,!\texttt{loc1}\}.\texttt{true}*.$ $\{\texttt{encrypt}\,?\texttt{data:String}\,!\texttt{key}\}.\texttt{true}*.\{\texttt{save}\,!\texttt{data}\,!\texttt{loc2}\}>\texttt{true}\Rightarrow[\texttt{true}*.\{\texttt{depend}\,!\texttt{loc1}\,!\texttt{loc2}\}]\texttt{false})$
IDS07-J	*Sanitize untrusted data passed to the Runtime.exec() method* $[\texttt{true}*.\{\texttt{isUntrusted}\,?\texttt{msg:String}\}.(\neg(\{\texttt{sanitize}\,!\texttt{msg}\}))*.\{\texttt{invokemethod}\,''\texttt{Runtime}''\,''\texttt{exec}''\,!\texttt{msg}\}]\texttt{false}$
IDS08-J	*Sanitize untrusted data included in a regular expression* $[\texttt{true}*.\{\texttt{isUntrusted}\,?\texttt{msg:String}\}.(\neg(\{\texttt{sanitize}\,!\texttt{msg}\}))*.\{\texttt{regex}\,!\texttt{msg}\}]\texttt{false}$
CWE 129	*Improper Validation of Array Index* $[\texttt{true}*.\{\texttt{isArrayIndex}\,?\texttt{index:int}\}.(\neg(\{\texttt{validate}\,!\texttt{index}\}))]\texttt{false}$
MSC03-J	*Never hard code sensitive information* $[\texttt{true}*.\{\texttt{isSensitive}\,?\texttt{msg:String}\}.(\neg(\{\texttt{obfuscate}\,!\texttt{msg}\})]\texttt{false}$
MET53-J	*Ensure that the clone() method calls super.call()* $[\texttt{true}*.\{\texttt{invokemethod}\,''\texttt{clone}''\}]\mu\texttt{Y}.\langle\texttt{true}\rangle\texttt{true}\wedge[\neg\{\texttt{invokemethod}\,''\texttt{super.clone}''\}]\texttt{Y}$
MET56-J	*Do not use Object.equals() to compare cryptographic keys* $[\texttt{true}*.\{\texttt{isKey}\,?\texttt{key1:String}\}.\texttt{true}*.\{\texttt{isKey}\,?\texttt{key2:String}\}.$ $\{\texttt{call}\,''\texttt{Object.equals}''\,!\texttt{key1}\,!\texttt{key2}\}\vert\{\texttt{call}\,''\texttt{Object.equals}''\,!\texttt{key2}\,!\texttt{key1}\}]\texttt{false}$
MSC62-J	*Store passwords using a hash function* $[\texttt{true}*.\{\texttt{setPassword}\,?\texttt{msg:String}\}.\texttt{true}*.\{\texttt{store}\,!\texttt{msg}\}]\texttt{false}$
EXP02-J	*Do not use the Object.equals() method to compare two arrays* $[\texttt{true}*.\{\texttt{isArray}\,?\texttt{ar1:String}\}.(\texttt{isArray}\,?\texttt{ar2}:\texttt{String}).\{\texttt{invokemethod}''\texttt{Object}''\,''\texttt{equals}''\,!\texttt{ar1}\,!\texttt{ar2}\}]\texttt{false}$
OBJ10-J	*Do not use public static non final fields* $[\texttt{true}*.\{\texttt{isPublic}\,?\texttt{data:String}\}.(\texttt{isStatic}\,!\texttt{data}).(\neg\{\texttt{isFinal}\,!\texttt{data}\})]\texttt{false}$

- L is the set of labels encoding the set of instructions that a program can perform: $x_j := e_j$ encoding an assignment of the variable x_j, e_j encoding an expression built over program variables and $m\overrightarrow{p}$ encoding a call to the method m with a finite set of arguments $\overrightarrow{p}$.
- $\rightarrow \subseteq S \times L \times S$ is the transition relation.

Informally, a pLTS describes mainly the behavior of a program as a set of reachable states and actions (instructions) that trigger a change of state. Parameterized LTS have a rich structure, for they take care of value passing in the instructions, of assignment of variables, of expressions and parameters of method calls. In fact, the states express the possible values of the program counter, and they indicate whether a state is an entry point of a method (initial state), a sequence state, a call to another method, a reply point to a method call, or a state in which the method terminates. Each transition describes the execution of a given instruction, the labels represent the instructions code. Furthermore, the pLTS we extract from program sources subsume data dependencies both explicit and implicit between all the variables in the program. During the PDG construction, we adopt a known technique used in taint analysis, and consists of renaming [22] the program variable; we rename each definition of a variable x to a different name and rename every use of x by the new name, to ensure that operations carried out on this specific data keep the same variable name.

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

Figure4 shows the graphical representation of the pLTS model corresponding to the program of Figure I. As shown in the middle of the pLTS a linear trace encoding naturally all the instructions of the program source. It depicts also all the data flow relationships that occur between the variables in the program source. For this, a special action, called the *depend* action is introduced. A transition labelled with an action $depend\, d_1\, d_2$ means that there is data dependence between the variable (d_2) attached to source state and the variable (d_1) attached to destination state.

Once the model is built, it can be used for verification

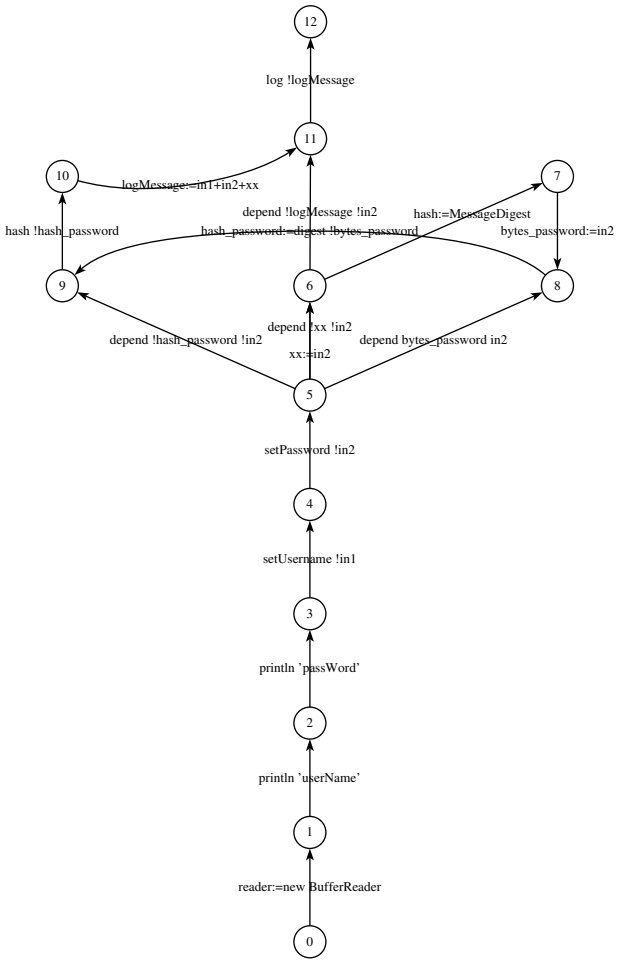

Fig. 4. pLTS for the program code from Figure I

by model checking. Verifying a property against a model consists in comparing the behavior specified in the property with that permitted by the model, in our case the pLTS. For example, we express the security guideline MSC62-J (given in Section I) by the following MCL:

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

This formula specifies that for each possible value of `msg` (the identifier of a password), the logging action with this password as it stands without hashing, cannot be reached. Moreover, this is not possible for all the variables which depend on this password. As one can notice, we use deliberately `log` action instead of store. One of the features of log framework is to store Java log messages to a database. As the sensitive information `password` can flow through the variable `xx`; we capture the explicit information flow in the second part of the alternation: $(\{\texttt{depend}\,?\texttt{msg1:String}\,!\texttt{msg}\}.\texttt{true}*.\{\texttt{log}\,!\texttt{msg1}\})$.

We are totally aware that the consideration of the dependent information flows in the specification of the guideline might not be trivial to the security expert when formalizing the guideline. In order to cover this specific issue, and also to increase the automation of the guidelines specification, we are currently implementing a script that computes the possible combinations of the different parameters dependences in the guideline minimal formula. By using the model checker EVALUATOR of the CADP toolbox [12], the formula is evaluated to `FALSE`, which means the property is not satisfied and a counterexample is produced, and reported as a trace illustrating the violation that occurred starting from the initial state (Figure 5).

IV. Discussion

We proposed a formalization of security guidelines in MCL capturing the main actions and key concepts presented in the guidelines informal textual description. MCL provides a basis for security experts to build the formulas. The objective is to increase the precision and strip away the ambiguities when interpreting the guideline. We were able to formalize a number of guidelines, and proceed to the formal verification through model checking. The performed formalization revealed ambiguities, unclarities and imprecision in the keywords, for example in the guideline MET03-J: ***Methods that perform a security check must be declared private or final*** [6], the extraction of precise key concepts (security check) is not trivial for the lack of precision in the guideline description.

We were not able to provide a precise formalization for other guidelines such as IDS15-J: ***Do not allow sensitive information to leak outside a trust boundary*** [6]. The notion of trust boundary can lead to misinterpretation, hence to improper implementation. From a developer perspective, it is tough to identify all the sensitive information in his program, the complexity of this operation increases with the complexity and the program size. The notion of *trust boundaries* is another ambiguity we found in different guidelines. This notion is ambiguous and its imprecision can be perceived from the difficulty of defining the system boundaries.

The guideline ***Store unencrypted keys away for encrypted data*** is imprecise and contains the implicit notion of location that can be expressed in different manners such as file location, insert in database, add to an array, etc. We have discussed this guideline in [4].

If we go back to the sample code of Figure I and invert the statements order as follows: We declare a String variable named `x` that contains the password's value. We assign it to the log message `logMessage` = `x`;. We invoke the logging operation on this `logMessage` (`logger.log(Level.INFO, logMessage);`). Then, we as-

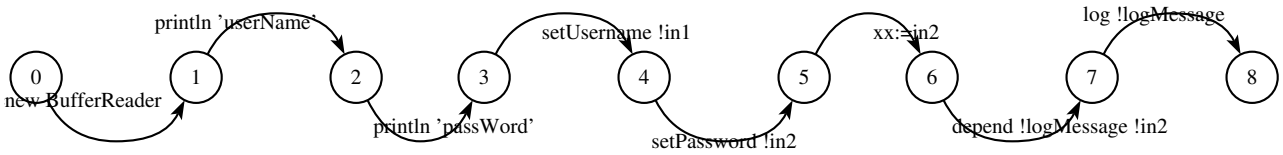

Fig. 5. Trace expressing the counter-example

sign this variable x to the attribute password of the object User (`User.setPassword(logMessage)`). This actions flow comprises an implicit violation of the guideline, as it carries out the logging of a data that seems at a first glance non-sensitive, in addition to the fact that semantically speaking, has a non-significant name x. Our formalization does not cover this specific scenario, as we use as intermediate representation the PDG which is flow-sensitive, meaning that it considers the order of statements. We tested this sample code using the JOANA IFC tool, and it did not capture the implicit flow. We are currently working on addressing this issue through PDG traversal and annotations propagation (forward and backward propagation). *State space explosion* [23], [24] is another limitation that our framework might be faced to. *State space explosion* is a widely known limitation of model checking, and it occurs in programs presenting a wide range of interacting components and data structures. In these programs, the number of states can be very large, which results in an important amount of time and required memory resources. As usual in this setting, we can use slicing techniques on the Augmented Program Dependence Graph to reduce the number of nodes and edges, and limit the size of the subgraph. We can also use slicing and abstraction techniques on the pLTS.

V. Related Work

The specification of security guidelines has also been addressed in the literature. In the technical report [25] 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 model security policies, and put more focus on actions rather than data. The mapping between the labels of LTL formulas and the program instructions is performed by the developers, which is an overhead to them. The Verification Support Environment [26] 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.
Aoraï plugin [27] 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.
Schneider [28] carries out a rather dynamic approach for the verification of a class of security policies known as EM (Execution Monitoring). He represents safety properties as *security automaton*. The automata serve as a basis to terminate the program once the security policy is violated. In the same line of work [29] proposed an extension of Schneider's security automata, and defined edit and suppress automata that enforce security policies through the modification of the security automata, hence to instrument programs. In the same line of work, [30] defines Security Automata SFI Implementation (SASI) which is a tool that enforces security policies. [31] proposes a translation from security automata to the annotations on the code level, and that serve to present pre- and post-conditions on the methods. SecureDIS [32] makes use of model checking together with theorem-proving to verify and generate the proofs. The authors adopt the Event-B method to specify the system and the security policies. In this work, it is not clear how the policies parameters are mapped to the system assets. In addition, the policy verification and enforcement are not extended on the program level.
GraphMatch [33] is a code analysis tool/prototype for security policy violation detection. GraphMatch considers examples of positive and negative security properties, 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 [34] language implements type-checking that makes use of the Decentralized Label Model (DLM) [35]; it allows defining 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.
PIDGIN [36] introduces an approach close 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 to verify policies satisfiability. The parameters of the queries are labels of PDG, which supposes that the developer is fully aware of the complex structure of PDGs, identifies 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.

VI. Conclusion and Future Work

In this paper, we proposed an approach for formalizing security guidelines from different sources with the objective of stripping away ambiguities. This first effort led to the specification of a set of guidelines, and to a proof-of-concept. Our approach aims at extending the verification and validation of guidelines at the different phases of the software development lifecycle. Our formalization can reduce the risk of misinterpretation of guidelines and improve its applicability while reducing the overhead on the developer. The output of the verification phase indicates whether the guideline is met, or if it is violated, in which case a violation trace is returned. Based on this output, the developer obtains a precise and useful feedback specifying the source of the violation. Our work strongly relies on the correctness of the PDG structure which we first generate by the JOANA IFC tool and that we augment with relevant details. PDG adopts conservative approximation of the program information flows. Our future work will focus on further experimental validations of our proposal and the development of a centralized repository for modular and composable security guidelines.

References

[1] Y. Roudier and L. Apvrille, "SysML-Sec: A model driven approach for designing safe and secure systems," *Model-Driven Engineering and Software Development (MODELSWARD), 2015 3rd International Conference*, pp. 655–664, Feb. 2015.

[2] D. P. Commissioner, "Facebook Ireland Ltd: Report of Re-Audit," 2012.

[3] Z. Zhioua, Y. Roudier, S. Short, and R. Ameur-Boulifa, "Security guidelines: Requirements engineering for verifying code quality," in *ESPRE 2016, 3rd International Workshop on Evolving Security and Privacy Requirements Engineering, September 12th*, 2016.

[4] Z. Zhioua, Y. Roudier, R. Ameur-Boulifa, T. Kechiche, and S. Short, "Tracking dependent information flows," in *ICISSP 2017: 3rd International Conference on Information Systems Security and Privacy*, Porto, Portugal, Feb. 2017.

[5] C. Administrator, "SEI CERT coding standards," 2017.

[6] CERT, "SEI CERT oracle coding standard for Java."

[7] Z. Zhioua, Y. Roudier, and R. Ameur-Boulifa, "Formal specification and verification of security guidelines," in *Dependable Computing (PRDC), 2017 IEEE 22nd Pacific Rim International Symposium on*. IEEE, 2017, pp. 267–273.

[8] J. Graf, M. Hecker, M. Mohr, and G. Snelting, "Checking applications using security APIs with JOANA," July 2015, 8th International Workshop on Analysis of Security APIs.

[9] Z. Zhioua, "https://github.com/zeineb/java-classes-parser."

[10] Oracle, "Java platform, standard edition 8 API specification."

[11] ——, "Java platform, standard edition 7 API specification."

[12] F. Lang, H. Garavel, and R. Mateescu, "An overview of CADP 2001," *European Association for Software Science and Technology (EASST) Newsletter*, vol. 4, August 2002.

[13] OWASP, "Owasp secure coding practices quick reference guide."

[14] ——, "Cryptographic storage cheat sheet."

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

[16] CERT, "Cert secure coding professional certificates." [Online]. Available: "http://www.cert.org/go/secure-coding/"

[17] R. C. Seacord, W. Dormann, J. McCurley, P. Miller, R. Stoddard, D. Svoboda, and J. Welch, "Source code analysis laboratory (scale)," DTIC Document, Tech. Rep., 2012.

[18] R. Mateescu and D. Thivolle, "A model checking language for concurrent value-passing systems," in *Proceedings of the 15th International Symposium on Formal Methods*, ser. FM '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 148–164.

[19] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 21st International Conference on Software Engineering*, ser. ICSE'99. New York, USA: ACM, 1999, pp. 411–420.

[20] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[21] A. Arnold, *Finite transition systems. Semantics of communicating sytems*. Prentice-Hall, 1994.

[22] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 147–160.

[23] A. Valmari, *The state explosion problem*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 429–528.

[24] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem," in *LASER Summer School on Software Engineering*. Springer, 2011, pp. 1–30.

[25] M. Aderhold, J. Cuéllar, H. Mantel, and H. Sudbrock, "Exemplary formalization of secure coding guidelines," TU Darmstadt and Siemens AG, Tech. Rep., 03 2010.

[26] S. Autexier, D. Hutter, B. Langenstein, H. Mantel, G. Rock, A. Schairer, W. Stephan, R. Vogt, and A. Wolpers, "VSE: formal methods meet industrial needs," *STTT*, vol. 3, no. 1, pp. 66–77, 2000.

[27] N. Stouls and V. Prevosto, "Aorai plugin tutorial – Frama C."

[28] F. B. Schneider, "Enforceable security policies," *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 1, pp. 30–50, 2000.

[29] L. Bauer, J. Ligatti, and D. Walker, "More enforceable security policies," in *Proceedings of the Workshop on Foundations of Computer Security (FCS02), Copenhagen, Denmark*. Citeseer, 2002.

[30] U. Erlingsson and F. B. Schneider, "SASI enforcement of security policies: A retrospective," in *Proceedings of the 1999 Workshop on New Security Paradigms*, ser. NSPW '99. New York, USA: ACM, 2000, pp. 87–95.

[31] M. Huisman and A. Tamalet, "A formal connection between security automata and JML annotations," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, M. Checkik and M. Wirsing, Eds., vol. 5503. Berlin: Springer Verlag, 2009, pp. 340–354.

[32] F. Akeel, A. Salehi Fathabadi, F. Paci, A. Gravell, and G. Wills, "Formal modelling of data integration systems security policies," *Data Science and Engineering*, pp. 1–10, 2016.

[33] J. Wilander and P. Fak, "Pattern matching security properties of code using dependence graphs," in *In proceeding of the first international workshop on code based software security assessments*, 2005, pp. 5–8.

[34] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Trans. Softw. Eng. Methodol.*, vol. 9, no. 4, pp. 410–442, Oct. 2000.

[35] ——, "A decentralized model for information flow control," *SIGOPS Oper. Syst. Rev.*, vol. 31, no. 5, pp. 129–142, Oct. 1997.

[36] J. Andrew, W. Lucas, and M. Scott, "Exploring and enforcing security guarantees via program dependence graphs," *PLDI 2015 Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 291–302, Jun. 2015.