Exemplary Formalization of Secure Coding Guidelines

Technical Report TUD-CS-2010-0060

March 2010

Markus Aderhold [1]
Jorge Cuéllar [2]
Heiko Mantel [1]
Henning Sudbrock [1]

[1] Technische Universität Darmstadt
[2] Siemens AG, München

TECHNISCHE UNIVERSITÄT DARMSTADT

Modeling and Analysis of Information Systems

SIEMENS

Contents

1 Introduction

Programming errors are the cause for a multitude of security vulnerabilities in software systems. This makes common attacks like, for instance, buffer overflow exploits, cross-site request forgery, cross-site scripting, or SQL injections possible. Secure coding guidelines describe programming practices that support the developer in improving the quality of software systems with respect to security so that the number of security vulnerabilities is reduced.

This report describes the results of a collaboration between Siemens AG and TU Darmstadt. The objective of this collaboration was the exemplary formalization of secure coding guidelines. The resulting formalizations shall provide reference points for secure coding that are more precise than the informal descriptions of secure coding guidelines (like the CERT Secure Coding Standards such as [Sea08]). Another goal was to investigate the feasibility of formalizations of secure coding guidelines in different domains (such as *maintaining session control* or *using cryptography properly*).

In this report, we provide formalizations for seven secure coding guidelines from different domains. Our formalization facilitates a structured presentation of secure coding guidelines: Firstly, the formalization concisely exposes the program actions that are relevant to each guideline. Secondly, it precisely defines the causal and temporal relations between these program actions as required by the guideline. This structured presentation may simplify the correct application of secure coding guidelines by developers. In addition, our formalizations reveal unclarities in the informal descriptions of secure coding guidelines. Based on those insights we provide several recommendations to eliminate these unclarities.

The formalizations lend themselves to further potential uses such as automated code inspection with respect to secure coding guidelines, for example. An outlook on two such possibilities for the future, namely runtime monitoring and verification, is also given in this report.

Structure of this report. In Section 2, we introduce the specification formalism that we employ for the formalizations. Section 3 constitutes the core of this report and describes our formalizations for seven secure coding guidelines. In Section 4, we discuss further potential uses of the formalizations. Section 5 highlights our results and concludes with an outlook on further perspectives to exploit the formalization of secure coding guidelines.

2 Specification Formalism

Informal descriptions of secure coding guidelines are typically formulated at different levels of abstraction. On the concrete level, the descriptions refer to the execution of commands like, for instance, "invoking the `equals`-method of a Java object". Other execution steps are described on a more abstract level, like, for instance, "storing input" or "detecting suspicious user behavior". In this section, we describe a formalism that facilitates the formal specification of secure coding guidelines in a uniform manner despite this varying degree of abstraction.

In our formalizations of secure coding guidelines we use *labels* to describe execution steps of a program both on the concrete level and on more abstract levels. Labels are expressions which describe the execution steps that occur during a program run. Each execution step is associated with one label describing the command that is executed (the *concrete label*) and may additionally be associated with one or more labels describing the execution step on a more abstract level (the *abstract labels*). As an example, consider the Java command `a = reader.read()`. The execution of this command is associated with the concrete label "*a = reader.read()*". If the object `reader` provides user input, then the execution of the command implies that user input is stored in variable *a*. We could express this more abstract fact by an abstract label *writeInputTo*(*a*) that we additionally associate with the execution of the command. In order to associate a label with complex commands, these complex commands can be split up into several lines. In this way, each line can be associated with a specific label. For instance, a line break after the symbol "=" could be inserted into the line `a = reader.read()` in order to associate the execution steps corresponding to the variable assignment and to the method call, respectively, with distinct labels.

The association of abstract labels with execution steps is usually performed implicitly in the developer's mind when applying a secure coding guideline. As this procedure is error-prone and time consuming, we suggest to make the mapping from execution steps to abstract labels explicit. Our formalizations provide a basis for such an explicit mapping by concisely describing the relevant execution steps for a given secure coding guideline. Recording and using explicit mappings shall help developers to correctly apply a secure coding guideline with less effort and shall reduce the risk of mistakes.

For the formalizations, we employ a variant of the specification formalism called *Linear Temporal Logic* (abbreviated by *LTL*) where we use labels as propositions. This formalism allows us to make precise statements about occurrences of and dependencies between execution steps occurring during a program run.

In Section 2.1, we provide a brief introduction to LTL with several examples. Section 2.2 introduces *Labeled Transition Systems* to model program executions and establishes the connection between LTL specifications and program executions. Our recapitulation of LTL is tailored to the formalizations in Section 3; a general textbook introduction into LTL can be found in [HR04], for example. LTL formulas offer a promising basis for inspecting programs or monitoring program executions with respect to formally specified secure coding guidelines. We discuss these perspectives in Section 4.

2.1 Linear Temporal Logic (LTL)

LTL formulas are composed of a small set of temporal operators ($\Box$, $\Diamond$, $\bigcirc$, and $\mathcal{U}$), standard operators from propositional logic ($\neg$, $\wedge$, $\vee$, and $\longrightarrow$), and labels as atomic propositions. In the following, we illustrate how to read LTL formulas with several concrete examples and provide a formal definition of LTL formulas afterwards.

1. $\Box(\neg \mathit{UnhandledNullPointerException})$

   Intuitively, this LTL formula specifies that a program will never throw an unhandled null pointer exception.

   This is formalized by using the temporal operator "$\Box$" (read: "globally") that specifies that a property must hold at every execution step and the abstract label *UnhandledNullPointerException* describing execution steps where an unhandled null pointer exception is thrown. Thus the above formula specifies that no execution step may correspond to an unhandled null pointer exception (the operator "$\neg$" formalizes negation, i.e., the formula $\neg A$ states that the formula A must not be satisfied).

2. $\Box\Big(\mathit{FailedAuthn} \longrightarrow \bigcirc \mathit{AuthnException}\Big)$

   Intuitively, this LTL formula specifies that whenever an authentication attempt fails, then the next execution step must correspond to an authentication exception.

   The "$\bigcirc$"-operator (read: "next") specifies that a property must hold at the next execution step. The formula uses the labels *FailedAuthn* describing a failed authentication attempt and *AuthnException* describing that an authentication exception is thrown. The whole formula is read as follows: At all execution steps, if the execution step corresponds to a failed authentication, then the subsequent execution step must correspond to an authentication exception (the operator "$\longrightarrow$" formalizes implication, i.e., the formula $A \longrightarrow B$ states that if A is satisfied then B must also be satisfied).

3. $(\neg \mathit{Connection.send})\ \mathcal{U}\ \mathit{Connection.open}$

   Intuitively, this LTL formula specifies that data will never be sent on a connection unless the connection has been opened.

   The temporal formula "$A\ \mathcal{U}\ B$" (read: "A unless B") specifies that A must hold at each execution step, but if B holds at some execution step, A does not need to hold any longer from that step on. (For the reader familiar with LTL, note that the Unless-operator differs from the Until-operator, as it does not require that B must hold at some execution step.) The two labels *Connection.send* and *Connection.open* describe execution steps corresponding to sending data on a connection and opening a connection, respectively. The whole formula is read as follows: No execution step may correspond to sending on a connection up to an execution step corresponding to opening the connection.

4. $\Big((\neg \mathit{Connection.send})\ \mathcal{U}\ \mathit{Connection.open}\Big) \wedge$
$$\Box\Big(\mathit{Connection.close} \longrightarrow ((\neg \mathit{Connection.send})\ \mathcal{U}\ \mathit{Connection.open})\Big)$$
Intuitively, this LTL formula provides a refined specification for the correct usage of a connection than the specification in the previous example. The same condition as in the previous example must be satisfied, but a further requirement is added: Whenever the connection is closed, the program must not send data on the connection unless the connection is opened again (the operator "$\wedge$" formalizes conjunction, i.e., the formula $A \wedge B$ states that both formulas A and B must be satisfied).

5. $\Box\Big(\mathit{Connection.open} \longrightarrow \bigcirc((\neg \mathit{Connection.open})\ \mathcal{U}\ \mathit{Connection.close})\Big)$
Intuitively, this LTL formula specifies that a connection which has already been opened must not be opened again.
The formula is read as follows: Whenever a connection is opened, from the next execution step on the connection must not be opened again unless it has been closed (specified by $\bigcirc((\neg \mathit{Connection.open})\ \mathcal{U}\ \mathit{Connection.close})$).

6. $\Box\Big(\mathit{Connection.open} \longrightarrow \Diamond \mathit{Connection.close}\Big)$
Intuitively, this LTL formula specifies that whenever a connection is opened, it must eventually be closed.
The specification is formalized by using the temporal operator "$\Diamond$" (read: "eventually") which states that the program must eventually behave as specified by the formula following the $\Diamond$-operator. In this example, the specification states that at all times, if the connection is opened, it must eventually be closed.

A table listing the intuitive meanings of the temporal operators in a concise form is provided in Section 2.2. Formally, LTL formulas are defined as follows:

Definition 1 (Syntax of LTL Formulas)**.** An *LTL formula* is an expression that is constructed according to the following rules:
- If l is a label, then l is an LTL formula.
- If P is an LTL formula, then $\neg(P)$, $\Box(P)$, $\bigcirc(P)$, and $\Diamond(P)$ are LTL formulas.
- If P and Q are LTL formulas, then $(P \wedge Q)$, $(P \vee Q)$, $(P \longrightarrow Q)$, and $(P\ \mathcal{U}\ Q)$ are LTL formulas.

As in the preceding examples, we often omit parentheses to keep the formulas readable.

2.2 Linking LTL Specifications with Programs

In the previous section, we illustrated by example how LTL formulas and labels can be employed to specify the behavior of programs. In this section, we describe the connection between an LTL specification and a program in more detail. This connection is important in order to state precisely whether a program satisfies an LTL specification.

We describe the possible program executions via so-called *Labeled Transition Systems*. Labeled transition systems can be used to describe executions of programs written in programming

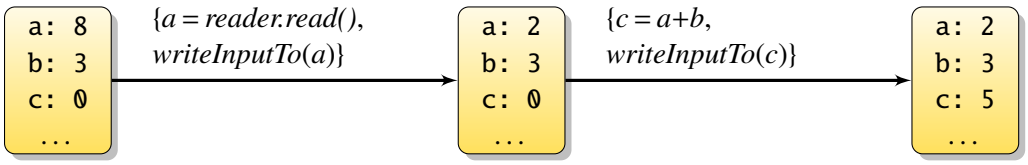

Figure 1: Excerpt of a labeled transition relation

languages such as Java (see, e.g., [KN06, DE99] for Java semantics in the form of transition systems). Formally, we define labeled transition systems as follows:

Definition 2 (Labeled transition systems)**.** A *labeled transition system* is a tuple $(S, S_0, L, \rightarrow)$ consisting of a set of program states S, a set of initial program states $S_0 \subseteq S$, a set of labels L, and a labeled transition relation $\rightarrow \subseteq S \times (\mathcal{P}(L) \setminus \{\emptyset\}) \times S$ (where $\mathcal{P}(L)$ denotes the powerset of L and $\emptyset$ denotes the empty set).

Intuitively, *program states* describe the current execution state of a program (e.g., the current heap, the current frame stack, and the pointer to the next program instruction for a Java program), and *initial program states* describe those execution states in which program executions may start. The *labels* are expressions that provide information about execution steps either on a concrete or on a more abstract level. The *transition relation* $\rightarrow$ describes the possible execution steps. If $(s, L', s') \in \rightarrow$, this means that there is an execution step from program state s to program state s', and that this execution step is described by the labels in L'. Note that $L' \subseteq L$ is a nonempty *set* of labels, because each execution step is annotated with at least one label; i.e., it can be annotated with multiple labels (one concrete label and zero or more abstract labels).

Figure 1 shows an excerpt of a labeled transition relation. Program states are depicted by boxes. Each box shows the current values of the variables `a`, `b`, and `c`. The ellipses indicate that the states contain further information which is not shown in this excerpt (like, for instance, the values of further variables and a pointer to the command that will be executed in the next execution step). The transitions are depicted by labeled arrows. In the example, each transition is annotated with a set containing two labels: The first label in each set is a concrete label (i.e., describing the execution of a command) and the second label is an expression that describes the corresponding execution step on a more abstract level. The abstract labels of the form *writeInputTo*(v) describe execution steps that modify the content of variable v such that the new value depends on data that has been provided as input to the program.

Labeled transition systems describe all possible execution sequences in the following sense:

Definition 3 (Execution sequences)**.** An *execution sequence* of the labeled transition system $(S, S_0, \rightarrow, L)$ is a pair $\sigma = ((s_i)_{i\in\{0,1,\ldots\}}, (L_i)_{i\in\{0,1,\ldots\}})$ of infinite sequences, written as

$$s_0 \xrightarrow{L_0} s_1 \xrightarrow{L_1} \ldots$$

in the following, where $s_0, s_1, \ldots \in S$ are program states and $L_0, L_1, \ldots \in \mathcal{P}(L)$ are sets of labels such that the following two conditions are satisfied:

1. For all $i \in \{0, 1, \ldots\}$,
   i) either $(s_i, L_i, s_{i+1}) \in \rightarrow$ (note that $L_i \neq \emptyset$ holds in this case)
   ii) or $L_i = \emptyset$, $s_{i+1} = s_i$, and there do not exist $L'_i \in \mathcal{P}(L) \setminus \{\emptyset\}$ and $s'_i \in S$ such that $(s_i, L'_i, s'_i) \in \rightarrow$.
2. For all $i \in \{0, 1, \ldots\}$, if $L_i = \emptyset$, then $L_j = \emptyset$ for all $j > i$.

We write σ^i for the execution sequence that results from σ by removing the first i transitions and states (i.e., $\sigma^0 = \sigma$ and, for instance, $\sigma^3 = s_3 \xrightarrow{L_3} s_4 \xrightarrow{L_4} \ldots$).

Execution sequences either represent non-terminating program executions or terminating program executions. For non-terminating program executions, all the label sets L_i are nonempty and each transition $s_i \xrightarrow{L_i} s_{i+1}$ is specified by the transition relation of the labeled transition system (compare condition 1.i)). For terminating program executions, there is some index i such that the transition relation $\rightarrow$ does not specify any possible transition from the state s_i. In this case, all subsequent states in the execution sequence are equal to s_i, and all subsequent label sets are empty (compare conditions 1.ii) and 2).

The following definition establishes the connection between execution sequences and LTL formulas by specifying which execution sequences satisfy a given LTL formula, and which execution sequences do not satisfy this formula:

Definition 4 (Semantics of LTL formulas)**.** The execution sequence $\sigma = s_0 \xrightarrow{L_0} s_1 \xrightarrow{L_1} \ldots$ *satisfies* the LTL formula P, written $\sigma \models P$, if and only if:

$P = l$	and $l \in L_0$,
$P = \Box Q$	and $\sigma^i \models Q$ for all $i \in \{0, 1, 2, \ldots\}$,
$P = \bigcirc Q$	and $\sigma^1 \models Q$,
$P = \Diamond Q$	and $\sigma^i \models Q$ for some $i \in \{0, 1, 2, \ldots\}$,
$P = Q \,\mathcal{U}\, R$	and either $\sigma^i \models Q$ for all $i \in \{0, 1, 2, \ldots\}$, or there exists some $i \in \{0, 1, 2, \ldots\}$ such that $\sigma^i \models R$ as well as $\sigma^j \models Q$ for all $j < i$,
$P = \neg Q$	and $\sigma \models Q$ does not hold,
$P = Q \wedge R$	and both $\sigma \models Q$ and $\sigma \models R$ hold,
$P = Q \vee R$	and $\sigma \models Q$ or $\sigma \models R$ holds, or
$P = Q \longrightarrow R$	and $\sigma \models R$ whenever $\sigma \models Q$.

For a labeled transition system $(S, S_0, \rightarrow, L)$ and an LTL formula P we write $(S, S_0, \rightarrow, L) \models P$ if *each* execution sequence $\sigma = s_0 \xrightarrow{L_0} s_1 \xrightarrow{L_1} \ldots$ of $(S, S_0, \rightarrow, L)$ with $s_0 \in S_0$ satisfies P.

The intuitive meaning of the temporal operators for execution sequences of labeled transition systems are summarized in Table 1.

In the following sections, it will become evident that LTL formulas provide a suitable specification formalism for the formalization of secure coding guidelines. Section 4 gives an outlook on how such specifications can be exploited further for the inspection of programs with respect to secure coding guidelines.

Temporal formula	**Intuitive meaning**
$\Box P$ (read: Globally P)	The property specified by P must be satisfied at all execution steps.
$\bigcirc P$ (read: Next P)	The property specified by P must be satisfied at the next execution step.
$\Diamond P$ (read: Eventually P)	The property specified by P must be satisfied at some future execution step.
$P \mathcal{U} Q$ (read: P unless Q)	The property specified by P must either be satisfied in all future execution steps, or up to an execution step where the property specified by Q is satisfied.
P (without temporal construct)	The property specified by P must be satisfied at the first execution step. If $P = l$, where $l \in L$ is a label, then P specifies that the first execution step must be annotated with the label l.

Table 1: Intuitive meaning of the temporal operators that are used in this report

3 Formalizations for Selected Secure Coding Guidelines

This section contains formalizations for seven secure coding guidelines. These guidelines were selected from a list of exemplary guidelines that have been kindly provided by Siemens for examination purposes [Sie09]. They are similar (but not identical) to Siemens-internal secure coding guidelines that were developed as recommendations for programmers. Please note that this paper makes no claim about the use of secure coding guidelines within any specific Siemens products. Siemens has identified several thematic domains of secure coding, which we call *secure coding domains* in the following. The selected guidelines belong to seven different secure coding domains. Table 2 gives an overview of the selected secure coding guidelines and the corresponding secure coding domains.

Secure coding guidelines often consist of several *secure coding aspects*. For instance, consider the secure coding guideline "Use prepared statements for database queries" (Section 3.3). Examples for secure coding aspects of this guideline are "Make sure that user input is only bound to parameters in the prepared statement. It must not affect the logic of the query" and "When utilizing stored procedures or prepared queries, make sure to use prepared statements also in the definition of the procedures or queries." We formalized one such secure coding aspect for each of the selected secure coding guidelines.

Structure of the formalizations. Sections 3.1–3.7 contain the formalizations of seven secure coding aspects. Each section starts with the title and the description of the corresponding secure coding guideline as well as the secure coding aspect that is formalized. Subsequently, a more precise formulation of the secure coding aspect is provided if necessary. The actual formalization of the secure coding aspect is then given as an LTL formula, which is displayed inside a frame.

For each formalization, a table lists all relevant abstract labels, describes their intuitive meaning, and gives examples for execution steps that typically should be associated with these labels. Each section also contains at least one application scenario that illustrates which execution steps should be associated with which abstract label for concrete Java programs.

In case that the formalization revealed imprecisions in the informal description of a given secure coding aspect, we provide recommendations for making the informal description more precise at the end of the section. These recommendations are accompanied by corresponding formalizations in the form of LTL formulas.

How to use the formalizations in practice. In order to check if some source code complies with the formalized secure coding aspect of a given secure coding guideline, we suggest a two-stage approach: First, the pieces of code should be identified that correspond to the labels used in the formalization. This can be done with the help of the respective table provided in this report, which lists and describes the labels that are relevant for the secure coding aspect. In the second step, one should convince oneself that the pieces of code that were identified in the first step are executed in the required order *in any conceivable case*. This should be done using the respective LTL formula. Alternatively, one could decide to use a precise informal description of the secure coding aspect instead of the formula in the last step. The precise informal descriptions suggested in the rest of this section could be used for this alternative, less formal approach.

Section	Secure Coding Domain	Secure Coding Guideline
3.1	Validate the User Input	When calling system commands, ensure the execution logic cannot be manipulated by user input
3.2	Sanitize the Output	Sanitize outgoing data and replace special characters with HTML representation
3.3	Secure the Internal Flow	Use prepared statements for database queries
3.4	Secure the Login and Authentication Procedures	Limit login attempts, transmit passwords encrypted and store hash or encrypted only
3.5	Maintain Session Control	Invalidate session after logout, timeout or suspicious user-activity and destroy all related data on server side
3.6	Enforce a Strict Authorization Model	For every request, check and enforce user permission for the resource considering given parameters and current context
3.7	Use Cryptography Properly	Use storage mechanisms for key material provided by framework or operating system

Table 2: Selected secure coding guidelines

3.1 Validate the User Input

Secure Coding Guideline (provided by Siemens [Sie09]):

"When calling system commands, ensure the execution logic cannot be manipulated by user input"

"The execution of system commands should be generally avoided. Use the standard documented API or framework instead whenever possible. If the execution of system commands is inevitable, follow these rules:

(a) Validate every input thoroughly before passing it to system commands
(b) Sanitize any character that might manipulate the logic of the command. [...] Use whitelist filtering whenever possible.
(c) Use predetermined values for dynamic data if they are known
(d) Run command as a lower privileged user [...]."

Selected Secure Coding Aspect:

"(a) Validate every input thoroughly before passing it to system commands."

Precise formulation of the selected secure coding aspect: Whenever program input is passed to a system command or used for the computation of values that are passed to a system command, this program input must be validated before passing it to a system command or using it in computations.

Formalization of the selected secure coding aspect:

The labeled transition system $(S, S_0, L, \rightarrow)$ satisfies the selected secure coding aspect, if and only if the following property is satisfied for all $v \in \mathit{Mem}$:

$$(S, S_0, L, \rightarrow) \models \Box\Big(\mathit{inputForSysCmd}(v) \longrightarrow ((\neg \mathit{useInComputation}(v) \wedge \neg \mathit{passToSysCmd}(v)) \,\mathcal{U}\, \mathit{validate}(v))\Big)$$

The formalization uses four labels that are parametric in an element v of the set *Mem*. The elements of this set correspond to the memory locations used in a program. Note that memory locations do not necessarily correspond to program variables, as multiple program variables could reference the same memory location. For instance, if two Java variables reference the same object, then these variables correspond to the same element $v \in \mathit{Mem}$. The label *inputForSysCmd*(v) describes execution steps where input data is written into the memory location v

```
1  public void deleteFile() throws IOException {
2    String filename = new BufferedReader(System.in).readLine();
3    if (validFilename(filename))
4    {
5      String cmd = "rm -f " + filename;
6      Runtime.getRuntime().exec(cmd);
7    }
8  }
```

Listing 1: Validate input before passing it to system commands

and this input might either be passed to a system command or used to compute values that will be passed to a system command. The label *useInComputation*(v) describes execution steps that use the value of v in a computation. The label *passToSysCmd*(v) describes execution steps that pass the value of v to a system command. The label *validate*(v) describes execution steps that correspond to the successful completion of the validation of v.

The formalization specifies that whenever input data is stored in memory location v and this data or data depending on it is later passed to a system command, the value of v must be validated before being used in computations or being passed to a system command.

Exemplary mapping of labels to execution steps: Table 3 lists the abstract labels used in the formalization and provides an informal description of the execution steps that are described by each label. Moreover, the table illustrates how to map the abstract labels to execution steps by providing at least one example for each label.

Application scenario: As a concrete example consider the Java method `deleteFile` shown in Listing 1. The method removes a file whose filename is provided by user input. The removal is performed by a system command (more specifically, a Linux shell command): The value `("rm -f " + filename)` is assigned to the String variable `cmd`, and the execution of the method `exec()` of the `Runtime`-object with parameter `cmd` triggers the execution of the system command removing the file. Before its usage, the user-provided filename is validated.

In this scenario, the assignment to the variable `filename` in line 2 has to be annotated with the label *inputForSysCmd*($filename$). The usage of the variable `filename` in line 5 has to be annotated with the label *useInComputation*($filename$). The return from the method `validFilename` with return value `true` for the parameter `filename` has to be annotated with the label *validate*($filename$). The invocation of the method `Runtime.getRuntime().exec(cmd)` in line 6 has to be annotated with the label *passToSysCmd*(*cmd*).

The method `deleteFile` satisfies the formalization, because the content of the variable `filename` is validated before using it to compute the value `cmd` that is passed to a system command.

Conservative alternative formulation and formalization: Requiring the validation of all input The secure coding guideline requires that all inputs that are used to compute values passed to system commands must be validated, while inputs that are not used to compute such values need not be validated. In the formalization, this is taken care of by the label

Label	Description	Exemplary execution steps
inputForSysCmd(*v*)	Label indicating that input data is written into memory location *v* that (later on) will be directly passed to a system command or will be used for the computation of a parameter passed to a system command	1. Executing the assignment `v = req.getParameter("input")`, where `req` references an object of type `HttpServletRequest`, and *v* is used to construct a system command executed by the Servlet 2. Executing the assignment `v = BufferedReader.readln()`, where the `BufferedReader`-object provides input from the user console that is used as a parameter of a system command
useInComputation(*v*)	Label indicating that memory location *v* is used in a computation, except for computations that are performed to validate *v*	1. Executing the assignment `cmdparam = "rm␣-f␣"+v` 2. Executing the statement `escapeSpaces(v)`, where the method `escapeSpaces` escapes spaces in the String value *v*
passToSysCmd(*v*)	Label indicating that the value in memory location *v* is passed as a parameter to a system command	1. Invoking the method `Runtime.exec()` of the Java API with the parameter `("rm␣-f␣"+v)` 2. Invoking the method `stdin.println(v)`, where `stdin` references the *input* stream associated with a `Process`-object of the standard Java API
validate(*v*)	Label indicating that the value of memory location *v* is validated successfully	1. The method `p.matcher(v).matches()` returns `true`, where `p` references an instance of the Java class `Pattern` representing valid input strings 2. The method `whitelist.contains(v)` returns `true`, where `whitelist` references a Java object of type `Vector` containing admissible inputs

Table 3: Labels used in the formalization of the secure coding aspect

inputForSysCmd(v). This label describes only execution steps where input that will be passed to system commands is stored in the memory location v. However, if this label is forgotten at some execution steps or accidentally attached to a different execution step, this has severe consequences. If such errors occur, then programs passing unvalidated inputs to system commands could satisfy the formalization[1]. In comparison, if one annotates *too many* execution steps with that label, then the consequences are less severe: In such cases, programs satisfying the formalization do not pass unvalidated input to system commands, but rather perform too many validations by validating input data that is never passed to system commands.

Based on this observation, one stays on the safe side when one validates *all* input to a program. To formalize this conservative alternative of the secure coding guideline, we introduce the new label *input*(v). This label describes execution steps where input enters the program, no matter whether the input is used to compute values passed to system commands or not. Using this label, we formalize this alternative as follows:

The labeled transition system $(S, S_0, L, \rightarrow)$ satisfies the conservative alternative of the selected secure coding aspect, if and only if the following property is satisfied for all $v \in Mem$:

$$(S, S_0, L, \rightarrow) \models \Box\Big(input(v) \longrightarrow ((\neg useInComputation(v) \wedge \neg passToSysCmd(v)) \,\mathcal{U}\, validate(v))\Big)$$

Note that the above formula differs from the formalization of the selected secure coding aspect, as the label *inputForSysCmd*(v) is replaced by the label *input*(v). In consequence, this formalization requires the validation of *each* input, including input that is *not* passed to system commands. (Apart from that, the two formalizations are equivalent.) This formalization simplifies the correct placement of labels, because one simply annotates *all* execution steps where the program reads input; i.e., one does not need to think about whether input will eventually be passed to a system command.

Validation of data retrieved from databases: The informal description of the secure coding aspect requires the validation of all input before passing it to system commands. In some cases, however, program input might be handled as follows: In a first step, program input is stored in a database. At a later point, possibly in a different execution of the program or even in an execution of a different program, the same data is retrieved from the database and passed to a system command. If this data is not validated, unvalidated input data is passed to a system command. It is therefore sensible to also validate data retrieved from a database. The following formalization includes this additional requirement, where the label *retrieveFromDatabaseForSysCmd*(v) describes execution steps where the result of a database query is stored in the memory location v:

[1] Note that this constitutes a mistake in *applying* the formalization; it is *not* a problem of the formalization.

The labeled transition system $(S, S_0, L, \rightarrow)$ satisfies the selected secure coding aspect with the additional requirement described above, if and only if the following property is satisfied for all $v \in \mathit{Mem}$:

$$(S, S_0, L, \rightarrow) \models \Box\Big((\mathit{inputForSysCmd}(v) \vee \mathit{retrieveFromDatabaseForSysCmd}(v)) \longrightarrow ((\neg \mathit{useInComputation}(v) \wedge \neg \mathit{passToSysCmd}(v)) \,\mathcal{U}\, \mathit{validate}(v))\Big)$$

3.2 Sanitize the Output

Secure Coding Guideline (provided by Siemens [Sie09]):

"Sanitize outgoing data and replace special characters with HTML representation"

"Do not send a http response or other special characters to the client. Sanitize by replacing characters with their HTML representations. The following table shows a selection but not a complete list of dangerous special characters, commonly used in attacks.

Special Character	**Definition**	**HTML-Entity/-Representation**
<	less-than sign	`<`
>	greater-than sign	`>`
&	ampersand	`&`
"	double quotation mark	`&quot;`
'	single quotation mark	`&#39;`
;	semicolon	`&#59;`

Whenever possible sanitize all output data of untrusted sources (i.e. user input) before it is sent to the browser. Avoid embedding user input into client side script code whenever possible. Use existing sanitization functions of frameworks if available."

Selected Secure Coding Aspect:

"Whenever possible sanitize all output data of untrusted sources (i.e. user input) before it is sent to the browser."

Precise formulation of the selected secure coding aspect: Values that contain data from untrusted sources (i.e., user input) or that have been computed based on such data must be sanitized before they are sent to the browser.

Formalization of the selected secure coding aspect:

The labeled transition system $(S, S_0, L, \rightarrow)$ satisfies the selected secure coding aspect, if and only if the following property is satisfied for all $v \in \mathit{Mem}$:

$$(S, S_0, L, \rightarrow) \models \Box\Big(\mathit{writeUntrustedDataTo}(v) \longrightarrow ((\neg \mathit{sendToBrowser}(v))\ \mathcal{U}\ (\mathit{sanitize}(v) \vee \mathit{overwrite}(v)))\Big)$$

The formalization uses four labels that are parametric in an element v of the set *Mem*. The elements of this set correspond to the memory locations used in a program. The label *writeUntrustedDataTo*(v) describes execution steps where the value of the memory location v is changed such that the new value depends on data from an untrusted source and the new value is not the result of a sanitization method. The label *sendToBrowser*(v) describes execution steps that cause the value in v to be sent the browser. The label *sanitize*(v) describes execution steps that correspond to the completion of the sanitization of the value in v. The label *overwrite*(v) describes execution steps in which the value in v is overwritten with a value that does not depend on data from untrusted sources.

The formalization specifies that whenever data depending on untrusted sources is written to a memory location v and that data is not the result of a sanitization method, then the contents of this memory location must not be sent to the browser unless the value in v has been sanitized or overwritten with a value not depending on data from untrusted sources.

Exemplary mapping of labels to execution steps: Table 4 lists the abstract labels used in the formalization and provides an informal description of the execution steps that are described by each label. Moreover, the table illustrates how to map the abstract labels to execution steps by providing at least one example for each label.

Application scenario: As an example we consider the Java Servlet shown in Listing 2. This Servlet provides the possibility to leave messages on a website in a simple fashion. The Servlet stores the messages in the linked list `entries` (compare lines 2 and 7). The user can add a new message by providing the HTTP request parameter named *message* (compare lines 6 to 8). The `PrintWriter` provided by the object `res` of type `HttpServletResponse` is used to send the Servlet's response to the browser. The response is constructed in lines 10 to 17 via calls to the method `out.write` generating HTML code for all entries contained in the list `entries`. Before being passed to the `PrintWriter`, the entries are sanitized by replacing special characters with their HTML representation (compare line 13).

In this scenario, the user-provided messages constitute input data that will be used to compute data sent to the browser. Therefore, the assignment in line 7 has to be annotated with

Label	Description	Exemplary execution steps
writeUntrustedDataTo(*v*)	Label indicating that the value of memory location *v* is changed, such that the new value depends on untrusted data (except when storing the result of a sanitization method in *v*)	1. The return value of the method `req.getParameter("userinput")` is assigned to the variable *v*, where `req` references an object of type `HttpServletRequest` of the Java Servlet API 2. The return value of the method `resultset.getString("columnName")` is assigned to the variable *v*, where `resultset` references an object of type `ResultSet` of the JDBC API 3. Assignment of a value depending on untrusted input data to the variable *v*
sendToBrowser(*v*)	Label indicating that the value of the memory location *v* is sent to the browser	Invoking the method `println(v)` of the `PrintWriter`-object associated to a Java Servlet
sanitize(*v*)	Label indicating that the sanitization of the value in memory location *v* has completed	1. Executing the assignment `entry = sanitize(entry)`, where the method `sanitize` sanitizes a String value 2. A method of the object *v* sanitizing the data fields of the object terminates
overwrite(*v*)	Label indicating that a value *not* depending on data from untrusted sources is written into the memory location *v*	A hard-coded constant is assigned to variable *v*

Table 4: Labels used in the formalization of the secure coding aspect

```
1  public class Shoutbox extends HttpServlet {
2    private List<String> entries = new LinkedList<String>();
3
4    public void doGet(HttpServletRequest req, HttpServletResponse res)
5                                  throws ServletException, IOException {
6      if (req.getParameter("message") != null) {
7        entries.add(req.getParameter("message"));
8      }
9      PrintWriter out = res.getWriter();
10     out.write("<html><head><title>shoutbox</title></head><body>");
11     for (int i=0; i<entries.size(); i++) {
12       String entry = entries.elementAt(i);
13       entry = HtmlUtil.htmlEscape(entry);
14       out.write(entry);
15       out.write("<hr/>");
16     }
17     out.write("</body></html>");
18   }
19 }
```

Listing 2: Java Servlet using sanitization

the label *writeUntrustedDataTo*(entries). In consequence, the assignment to the variable `entry` in line 12 has to be annotated with the label *writeUntrustedDataTo*(*entry*). The execution step corresponding to the method call `out.write(entry)` has to be annotated with the label *sendToBrowser*(*entry*). The execution step corresponding to the return from the method `HtmlUtil.htmlEscape(entry)` has to be annotated with the label *sanitize*(*entry*). Note that the assignment to the variable `entry` in line 13 need not be annotated with the label *writeUntrustedDataTo*(*entry*), as it assigns the result of a sanitization to the variable `entry`.

The Servlet satisfies the formalized secure coding aspect as all data depending on untrusted sources is sanitized before being sent to the browser.

Revisiting the formalization from Section 3.1: There is a certain similarity between the validation of input before passing it to system commands and the sanitization of output before sending it to the browser. In both cases, data needs to be checked (i.e., validated or sanitized, respectively) before relaying it. By simply replacing the labels in the formalization of input validation from Section 3.1 (see page 12), we get the following formalization of output sanitization, where *inputForSysCmd*(v) is replaced by *untrustedInputForBrowserOutput*(v), *passToSysCmd*(v) is replaced by *sendToBrowser*(v), and *validate*(v) is replaced by *sanitize*(v):

> The labeled transition system $(S, S_0, L, \rightarrow)$ satisfies the selected secure coding aspect, if and only if the following property is satisfied for all $v \in \mathit{Mem}$:
>
> $$(S, S_0, L, \rightarrow) \models \Box\Big(\mathit{untrustedInputForBrowserOutput}(v) \longrightarrow ((\neg \mathit{useInComputation}(v) \ \wedge \ \neg \mathit{sendToBrowser}(v)) \ \mathcal{U} \ \mathit{sanitize}(v))\Big)$$

The label *untrustedInputForBrowserOutput*(v) describes execution steps where input from an untrusted source is written to memory location v that will be sent to the browser or used to com-

pute values that will be sent to the browser. Similarly to Section 3.1, label *useInComputation*(v) describes execution steps where the value of v is used in computations. The resulting formalization specifies that whenever input data from untrusted sources is stored in a memory location v such that this value will be sent to the browser or used to compute values that will be sent to the browser, then the contents of v may neither be used in computations nor be sent to the browser before being sanitized.

This is a sensible course of action as well, because the final browser output will only depend on data that has already been sanitized. Hence the above formalization could inspire an additional secure coding guideline, which requires the sanitization of data *before* it is used in the computation of the actual browser output. Thus, the computation may already assume that it operates on *sanitized* data. The original secure coding aspect (as formalized on page 18) in addition ensures that the final output is sanitized as well.

Recommendations for making the secure coding guideline more precise: The formalization of the secure coding guideline reveals that the informal guideline should be made more precise. In particular, the meaning of "whenever possible" should be elaborated on so that the formalization can reflect this relaxation. In order to clarify the informal guideline, it seems promising to collect and investigate code examples of output sanitization.

3.3 Secure the Internal Flow

Secure Coding Guideline (provided by Siemens [Sie09]):

"Use prepared statements for database queries"

"Prepared statements are a feature of a database to prepare instructions to the database without supplying the parameters. Instead of the parameters placeholders are used. The parameters provided later on are checked for validity first before being processed. Therefore, prepared statements prevent SQL injections effectively and shall always be used for security as well as performance reasons.

Make sure that user input is only bound to parameters in the prepared statement. It must not affect the logic of the query.

Stored procedures, parameterized queries
When utilizing stored procedures or prepared queries, make sure to use prepared statements also in the definition of the procedures or queries.

Object-Relational Mappers
Object-Relational Mappers store objects in a relational database, which appears to the program as a object oriented database. When applying Object-Relational Mappers make sure that they use prepared statements for all queries to the database."

Selected Secure Coding Aspect:

"Make sure that user input is only bound to parameters in the prepared statement. It must not affect the logic of the query."

Precise formulation of the selected secure coding aspect: User input or data that has been derived from user input must never be used in the construction of Prepared Statements. Such data may only be bound to parameters in Prepared Statements.

Formalization of the selected secure coding aspect:

The labeled transition system $(S, S_0, L, \rightarrow)$ satisfies the selected secure coding aspect, if and only if the following property is satisfied for all $v \in Mem$:

$$(S, S_0, L, \rightarrow) \models \Box\Big(writeInputTo(v) \longrightarrow ((\neg constructPreparedStatementWith(v)) \,\mathcal{U}\, overwrite(v))\Big)$$

This formalization uses three labels that are parametric in an element v of the set *Mem*. The elements of this set correspond to the memory locations used in a program. The label *writeInputTo*(v) describes execution steps where the value of the memory location v is changed such that the new value depends on user input. The label *constructPreparedStatementWith*(v) describes execution steps where the value of the memory location v is used as a parameter for a method that constructs a Prepared Statement (i.e., the formula ¬*constructPreparedStatementWith*(v) states that no Prepared Statement is constructed using the value in memory location v). The label *overwrite*(v) describes execution steps that overwrite the memory location v with data that does not depend on user input.

The formula specifies that whenever values depending on user input are written into a memory location v, the value of this memory location must not be used in the construction of Prepared Statements unless it is overwritten with a value that does not depend on user input.

Exemplary mapping of labels to execution steps: Table 5 lists the abstract labels used in the formalization and provides an informal description of the execution steps that are described by each label. Moreover, the table illustrates how to map the abstract labels to execution steps by providing at least one example for each label.

Application scenario: As examples we consider the three Java methods `insertUser1`, `insertUser2`, and `insertUser3` shown in Listing 3. Method `insertUser1` initially reads a username from the command line (compare line 3). Then it creates a Prepared Statement with one parameter for the username (lines 4 and 5), sets this parameter to the previously read username (line 6), and executes the Prepared Statement (line 7). In addition to a username, method `insertUser2` reads a group ID from the command line (compare line 13), and then prepares a statement where this group ID is used as a part of the table into which the username is inserted (lines 14 and 15). Afterwards, it performs the same actions as the method `insertUser1`. The method `insertUser3` differs from the method `insertUser2`, as it does not use the String `groupID` in the creation of the Prepared Statement, but rather adds an additional parameter for the table name to the Prepared Statement (compare lines 24 and 25). Before executing the Prepared Statement, this additional parameter is set to the value of the variable `groupID` (line 26).

Label	Description	Exemplary execution steps
writeInputTo(v)	Label indicating that the value of memory location v is changed and now depends on input data	1. Assigning the value `("SELECT * FROM "+input)` to variable v, where variable `input` contains user input 2. Assigning the return value of the Java method `BufferedReader.readln()` to v, where the `BufferedReader`-class provides input from the user console 3. Assigning the return value of the Java method `req.getParameter("input")` to variable v, where *req* references an object of type `HttpServletRequest`
constructPreparedStatementWith(v)	Label indicating that memory location v is used as parameter for a method constructing a Prepared Statement	Invoking the method `conn.prepareStatement("SELECT * FROM "+v)`, where `conn` references an object of type `Connection` of the JDBC API If the expression used as parameter for the method `prepareStatement` does not contain any variables, this method call does not obtain this label
overwrite(v)	Label indicating that the value of the memory location v is changed and now does not depend on input data	1. A constant value is assigned to variable v 2. Variable v is overwritten by data that does not depend on user input

Table 5: Labels used in the formalization of the secure coding aspect

```
1  public void insertUser1() throws IOException, SQLException {
2    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
3    String username = in.readLine();
4    PreparedStatement stmt = con.prepareStatement(
5        "INSERT INTO `users` (`name`) VALUES (?)");
6    stmt.setString(1, username);
7    stmt.execute();
8  }
9
10 public void insertUser2() throws IOException, SQLException {
11   BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
12   String username = in.readLine();
13   String groupID = in.readLine();
14   PreparedStatement stmt = con.prepareStatement(
15       "INSERT INTO `group_" + groupID + "` (`name`) VALUES (?)");
16   stmt.setString(1, username);
17   stmt.execute();
18 }
19
20 public void insertUser3() throws IOException, SQLException {
21   BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
22   String username = in.readLine();
23   String groupID = "group_" + in.readLine();
24   PreparedStatement stmt = con.prepareStatement(
25       "INSERT INTO ? (`name`) VALUES (?)");
26   stmt.setString(1, groupID);
27   stmt.setString(2, username);
28   stmt.execute();
29 }
```

Listing 3: User input and Prepared Statements

In these methods, the assignments to the variable `username` in lines 3, 12, and 22 and to the variable `groupID` in lines 13 and 23 have to be annotated with the labels *writeInputTo*(*username*) and *writeInputTo*(*groupID*), respectively, as these values are user-provided. The method call to `Connection#prepareStatement` in the method `insertUser2` (compare line 14) has to be annotated with the label *constructPreparedStatementWith*(*groupID*). Note that the calls of the method `Connection#prepareStatement` in lines 4 and 24 do not obtain a *constructPrepared-StatementWith*-label, as the parameters reference constant String values and do not depend on any program variables.

The method `insertUser1` satisfies the formalization, as no user input is used to construct the Prepared Statement. In contrast, the method `insertUser2` does not satisfy the formalization, as the group ID provided as user input is used in the construction of the Prepared Statement.The method `insertUser3` circumvents the problem of the method `insertUser2` by adding an additional parameter to the Prepared Statement. In consequence, all user input is only bound to parameters of the Prepared Statement, and the method `insertUser3` satisfies the formalization.

3.4 Secure the Login and Authentication Procedures

Secure Coding Guideline (provided by Siemens [Sie09]):

"Limit login attempts, transmit passwords encrypted and store hash or encrypted only"

"During the login process, realize the following requirements:

(a) **Limit login attempts to 5 tries per account.**
After 5 failed login attempts lock the account for at least 10 min. If this happens again, lock the account for longer periods of time. Do not use a sleep or delay function (after the unsuccessful login attempts) to "lock" the account, this will provide DoS vulnerability.

(b) **Print only a generic error message.**
An example of generic message is: "User name and password combination not found." Do not give to the user any further information, for instance whether he entered a correct user name or not.

For the transmission and storage of passwords, the following rules are obligatory:

(c) **Transmit passwords using an encrypted channel.**
Recommendation: SSL/TLS, [...], or IPsec in transport mode (end-to-end encryption).

(d) **Whenever possible, store only a salted hash of the password and compare the hashes.**
The salted hash is created by applying a hash function to the password and a fixed (secret) value called salt. The salt prevents attackers from easily building a list of hash values of guessed passwords. The salt must be stored encrypted.

(e) **If necessary to know the password, store it in encrypted form.**
In very seldom cases, the application requires knowledge of the clear text passwords, for instance when the application uses the passwords to authenticate itself. In those cases, document the reason for this need and store the passwords in encrypted form [...]

(f) **Clear passwords from memory immediately after use**, to avoid password exposure in core and dump files.

(g) **Do not hard-code passwords (or keys) in source code** [...]"

Selected Secure Coding Aspect:

"(a) Limit login attempts to 5 tries per account. After 5 failed login attempts lock the account for at least 10 min. If this happens again, lock the account for longer periods of time. Do not use a sleep or delay method (after the unsuccessful login attempts) to "lock" the account, this will provide DoS vulnerability."

Precise formulation of the selected secure coding aspect: When five failed login attempts have been performed for an account, then that account must be locked for at least ten minutes. Whenever another five failed login attempts have been performed for the same account, it must be locked for a longer period of time than the last time it had been locked. When counting the failed login attempts, do not consider those login attempts that fail just because the account is currently locked.

Formalization of the selected secure coding aspect:

The labeled transition system $(S, S_0, L, \rightarrow)$ satisfies the selected secure coding aspect, if and only if the following property is satisfied for all $a \in \mathit{Accounts}$:

$$(S, S_0, L, \rightarrow) \models P(a, 5, 10) \ \wedge \ \Box\Big(\forall k.\ (\mathit{lock}(a,k) \longrightarrow \exists k' > k.\ P(a,5,k'))\Big),$$

$$\text{where } P(a,0,k) = \Big(\exists k' \geq k.\ \mathit{lock}(a,k')\Big) \text{ and}$$

$$P(a,n,k) = \Big((\neg \mathit{fail}(a))\ \mathcal{U}\ \bigcirc P(a,n-1,k)\Big) \text{ for } n > 0.$$

The formalization uses two labels, $\mathit{fail}(a)$ and $\mathit{lock}(a,k)$. The label $\mathit{fail}(a)$ is parametric in an element a of the set *Accounts*, whose elements correspond to the accounts within a system. It describes execution steps that correspond to a failed login attempt for the account a except for failures caused by the account being currently locked. The label $\mathit{lock}(a,k)$ is parametric in an element a of the set *Accounts*, and in a number k. This label describes execution steps that correspond to locking the account a for k minutes.

The formula $P(a,n,k)$ specifies that after n failed login attempts for the account a the account must be locked for at least k minutes. For $n = 0$, this is expressed by specifying that the account is locked for k' minutes with $k' \geq k$. For $n > 0$, this is specified by requiring that either no more failed login attempts for that account occur, or that after the first failed login attempt the formula $P(a, n-1, k)$ must be satisfied, i.e., after $n-1$ failed login attempts the account must be locked for at least k minutes. The formalization can hence be read as follows: After the first five failed login attempts an account must be locked for at least ten minutes, and whenever the account is locked for k minutes, it must be locked for more than k minutes after the next five failed login attempts.

Exemplary mapping of labels to execution steps: Table 6 lists the abstract labels used in the formalization and provides an informal description of the execution steps that are described by each label. Moreover, the table illustrates how to map the abstract labels to execution steps by providing at least one example for each label.

Application scenario: Consider the Java class providing an authentication mechanism that is shown in Listing 4. The class provides password-based authentication via the method `login`. We list reasons why this class does not satisfy the secure coding aspect after describing its functioning and which execution steps should be annotated with the labels used in the formalization.

Label	Description	Exemplary execution steps
$fail(a)$	Label indicating a failed login attempt for the account a that is not caused by the account being locked	A login method called for the account a throws an exception (e.g., the method `LoginContext.login` of the Java API *JAAS* throws a `LoginException`)
$lock(a, k)$	Label indicating that the account a is being locked for k minutes. Sleep or delay methods must not be annotated with this label	A method provided by the authorization API for locking accounts is called with a parameter corresponding to a locking duration of k minutes.

Table 6: Labels used in the formalization of the secure coding aspect

In the case of a successful login, the method returns a Session object for the authenticated user, otherwise it throws an exception. In line 5, a User object for the provided username is obtained from the user database. The first check in line 6 determines if the user account is locked. If the account is locked, an exception is thrown in line 7. The second check in line 9 determines if the maximal number of failed login attempts for the account is exceeded. If this is the case, the account is locked for ten minutes (compare line 10) and an exception is thrown. The final check in line 13 determines if the provided password is correct. If it is incorrect, the number of failed login attempts for the account is incremented (compare line 14). If it is correct, the number of failed attempts is reset (line 17) and the method returns a Session object (line 18).

In this scenario, the set *Accounts* corresponds to the set of Java String values which are valid usernames. All execution steps corresponding to method calls of `User#lock` have to be annotated with the label $lock(a, k)$, where a is the username associated with the object the method is called on and k is the parameter of the method call. The label $fail(a)$ has to be assigned to execution steps corresponding to throwing a `LoginException`, where a is the username of the authentication attempt. Note that login attempts that fail just because the account is locked must not be annotated with the label $fail(a)$.

This class does not satisfy the formalization for several reasons: For instance, accounts are always locked for ten minutes, although the time that an account is locked should be incremented after each sequence of five failed login attempts. Another reason is that accounts are not locked directly after the fifth failed login attempt. Furthermore, the counter for failed login attempts is reset after a successful login. While the first two reasons are obvious violations of the guideline, the last "problem" of the program (i.e., resetting the counter for failed login attempts after a successful login attempt) looks like program behavior that should be accepted by the secure coding guideline. We address this point in the following recommendation.

```
1  class Authenticator {
2    private UserDB users;
3    ...
4    public Session login(String username, String password) throws Exception {
5      User user = users.queryUser(username);
6      if (user.isLocked()) {
7        throw new AccountLockedException();
8      }
9      if (user.getFailAttempts() > 5) {
10       user.lock(10);
11       throw new AccountLockedException();
12     }
13     if (!user.checkPassword(password)) {
14       user.incFailAttempts();
15       throw new LoginException();
16     }
17     user.resetFailAttempts();
18     return new Session(user);
19   }
20 }
```

Listing 4: Validate input before passing it to system commands

Recommendations for making the secure coding guideline more precise:

1. **Resetting the number of unsuccessful login requests.** In the above formulation of the secure coding guideline, accounts are locked after five failed login attempts, even if a successful login attempt for that account occurs before the fifth failed attempt. This appears too strict, when one considers that even login attempts by authorized users may fail, e.g., due to typing errors. We therefore recommend the following more precise formulation:

> *When five failed login attempts have been performed for an account and no successful login attempt has been performed for that account between those failed attempts, then that account must be locked for at least ten minutes. Whenever five failed login attempts occur again without an intermediate successful login attempt for that account, the account must be locked for a longer period of time than the last time it had been locked. When counting the failed login attempts, do not consider those login attempts that fail just because the account is currently locked.*

The formalization of this more precise formulation is as follows:

The labeled transition system $(S, S_0, L, \rightarrow)$ satisfies the more precise variant of the selected secure coding aspect, if and only if the following property is satisfied for all $a \in \mathit{Accounts}$:

$$(S, S_0, L, \rightarrow) \models P'(a, 5, 10) \ \wedge \ \Box(\mathit{succ}(a) \longrightarrow P'(a, 5, 10)) \wedge \Box\Big(\forall k.\ (\mathit{lock}(a, k) \longrightarrow \exists k' > k.\ P'(a, 5, k'))\Big),$$

$$\text{where } P'(a, 0, k) = \Big(\exists k' \geq k.\ \mathit{lock}(a, k')\Big) \text{ and}$$

$$P'(a, n, k) = \Big((\neg \mathit{fail}(a))\ \mathcal{U}\ \Big((\bigcirc P'(a, n-1, k)) \vee \mathit{succ}(a)\Big)\Big) \text{ for } n > 0.$$

Here, a successful login request (described by the label $succ(a)$) must reset the number of possible login requests to five and the minimal time for which the account needs to be locked to ten minutes.

2. **Unlocking locked accounts.** In the current formulation of the secure coding guideline it is not intended that locked accounts are unlocked, e.g., by an administrator. Since locking the account for numerous times may result in locking times much larger than ten minutes, this would allow "denial of service"-attacks against a user. We therefore recommend the following more precise formulation that supports resetting accounts:

> *When five failed login attempts have been performed for an account, then that account must be locked for at least ten minutes. Whenever another five failed login attempts have been performed for the same account, the account must be locked for a longer period of time than the last time it had been locked. When counting the failed login attempts, do not consider those login attempts that fail just because the account is currently locked.*
>
> *If an account is unlocked, reset the required minimal locking time back to ten minutes.*

The formalization of this more precise formulation is as follows:

The labeled transition system $(S, S_0, L, \rightarrow)$ satisfies the more precise variant of the selected secure coding aspect, if and only if the following property is satisfied for all $a \in Accounts$:

$$(S, S_0, L, \rightarrow) \models P''(a, 5, 10) \ \wedge \ \Box(reset(a) \longrightarrow P''(a, 5, 10)) \wedge \Box\Big(\forall k.\ (lock(a, k) \longrightarrow \exists k' > k.\ P''(a, 5, k'))\Big),$$

$$\text{where } P''(a, 0, k) = \Big(\exists k' \geq k.\ lock(a, k')\Big) \text{ and}$$

$$P''(a, n, k) = \Big((\neg fail(a))\ \mathcal{U}\ \Big((\bigcirc P''(a, n-1, k)) \vee reset(a)\Big)\Big) \text{ for } n > 0.$$

Execution steps that correspond to the unlocking of the account a must be annotated with the label $reset(a)$. Whenever a reset occurs, the number of possible unsuccessful login requests must be reset to five and the minimal time that the account needs to be locked must be reset to ten minutes.

3. The two previous recommendations can be combined as follows:

> *When five failed login attempts have been performed for an account and no successful login attempt has been performed for that account between those failed attempts, then that account must be locked for at least ten minutes. Whenever five failed login attempts occur again without an intermediate successful login attempt for that account, the account must be locked for a longer period of time than the last*

time it had been locked. When counting the failed login attempts, do not consider those login attempts that fail just because the account is currently locked.

If an account is unlocked, reset the required minimal locking time back to ten minutes.

It is straightforward to obtain a formalization for this more precise formulation by combining the previous two specifications in one single LTL specification:

The labeled transition system $(S, S_0, L, \rightarrow)$ satisfies the combination of both proposed more precise variants of the selected secure coding aspect, if and only if the following property is satisfied for all $a \in \mathit{Accounts}$:

$$(S, S_0, L, \rightarrow) \models P'''(a, 5, 10) \ \wedge \ \Box((\mathit{reset}(a) \ \vee \ \mathit{succ}(a)) \longrightarrow P'''(a, 5, 10)) \wedge \Box\Big(\forall k.\ (\mathit{lock}(a, k) \longrightarrow \exists k' > k.\ P'''(a, 5, k'))\Big),$$

$$\text{where } P'''(a, 0, k) = \Big(\exists k' \geq k.\ \mathit{lock}(a, k')\Big) \text{ and}$$

$$P'''(a, n, k) = \Big((\neg \mathit{fail}(a))\ \mathcal{U}\ \Big((\bigcirc P'''(a, n-1, k)) \vee \mathit{reset}(a) \ \vee \ \mathit{succ}(a)\Big)\Big) \text{ for } n > 0.$$

3.5 Maintain Session Control

Secure Coding Guideline (provided by Siemens [Sie09]):

"Invalidate session after logout, timeout or suspicious user-activity and destroy all related data on server side"

"After the **logout** all user-related session IDs must be immediately **invalidate** the session on the server side in the following situations:

- Logout
- Timeout
- Occurrence of a condition that indicates that the user is misbehaving (e.g. trying to access information by suspicious input, or trying to enter a script, etc)

The Logout procedure must obey the following rules:

- Logout link/button needs to be visible on every page
- Logout procedure must start immediately after clicking logout button/link or the corresponding confirmation dialogue to logout.

Timeout good practice:

- Limit session time to the minimum still acceptable value, from user experience perspective

Invalidation procedure:

- Delete all information in session object after logout
- Save only necessary persistent data in databases, files, etc.
- Do not reuse session objects
- Invalidate on the server side"

Selected Secure Coding Aspect:

"After the logout all user-related session IDs must be immediately invalidate the session on the server side in the following situations:

- Logout
- Timeout
- Occurrence of a condition that indicates that the user is misbehaving (e.g., trying to access information by suspicious input, or trying to enter a script, etc)"

Precise formulation of the selected secure coding aspect: Session IDs must be invalidated immediately under the following circumstances:

- Whenever the user logs out, the session IDs related to that user must be immediately invalidated.
- If the session of a user ID has timed out, the session IDs related to that user ID must be immediately invalidated.
- If a condition indicates that a user with a certain user ID misbehaves, the session IDs related to that user ID must be immediately invalidated.

Formalization of the selected secure coding aspect:

The labeled transition system $(S, S_0, L, \rightarrow)$ satisfies the selected secure coding aspect if and only if the following property is satisfied for all $u \in \mathit{UserIDs}$:

$$\begin{aligned}(S, S_0, L, \rightarrow) \models\ & \Box\Big(\mathit{logout}(u) \longrightarrow \bigcirc \mathit{invalidate}(u)\Big) \wedge \\ & \Box\Big(\mathit{timeout}(u) \longrightarrow \bigcirc \mathit{invalidate}(u)\Big) \wedge \\ & \Box\Big(\mathit{suspicious}(u) \longrightarrow \bigcirc \mathit{invalidate}(u)\Big)\end{aligned}$$

The formalization uses four labels that are parametric in an element of the set *UserIDs*. The elements of this set correspond to the user IDs in a system. The labels $\mathit{logout}(u)$, $\mathit{timeout}(u)$, and $\mathit{suspicious}(u)$ describe execution steps that correspond to the logout of u, to a timeout for u, or to the detection of suspicious behavior of u, respectively. The label $\mathit{invalidate}(u)$ describes execution steps that invalidate the session IDs of u.

Hence the formalization specifies that whenever a user with ID u logs out or the session of the user with ID u times out or suspicious behavior of the user with ID u is detected, the session IDs related to the user ID u are invalidated immediately, i.e., in the next execution step.

Exemplary mapping of labels to execution steps: Table 7 lists the abstract labels used in the formalization and provides an informal description of the execution steps that are described by each label. Moreover, the table illustrates how to map the abstract labels to execution steps by providing at least one example for each label.

Application scenario: The application scenario shown in listing 5 contains a simple Java Servlet that is called when a user presses a logout button. The first command of the method `doGet` obtains a reference to the session object corresponding to the request and invalidates the session of that request via the corresponding method of the session object. Afterwards, a log entry is generated.

The invocation of the `doGet` method has to be annotated with the label $\mathit{logout}(u)$, where u is the user ID the Servlet is executed for. The corresponding invocation of the method `invalidate` has to be annotated with the label $\mathit{invalidate}(u)$.

Label	Description	Exemplary execution steps
logout(u)	Label indicating that the user with ID u has logged out	Invoking the `doGet` or `doPost` method of a Java Servlet that is called by user ID u signalling that the user wishes to log out
timeout(u)	Label indicating that the session of user ID u has timed out	A method used as a timeout-listener is invoked
suspicious(u)	Label indicating that suspicious behavior of the user with ID u has been detected	An input validation method returns the value `false`, because the user with ID u has provided an input containing JavaScript code
invalidate(u)	Label indicating that all session IDs for user ID u are being invalidated	1. Invoking the method `HttpSession.invalidate()` of the Java Servlet API for the `HttpSession`-object associated with user ID u 2. Invoking the method `SecurityContextHolder.getContext().setAuthentication` of the Spring Security Framework with parameter `null`

Table 7: Labels used in the formalization of the secure coding aspect

```
1  public class LogoutServlet extends HttpServlet {
2    public LogoutServlet () {super();}
3
4    public void doGet(HttpServletRequest req, HttpServletResponse resp) {
5        req.getSession().invalidate();
6        logger.log("User logs out: "+req.getRemoteUser());
7        returnLogoutPage(resp);
8    }
9    ...
10 }
```

Listing 5: Invalidating session IDs

Recommendation for making the secure coding guideline more precise: The secure coding guideline states that session IDs must be invalidated *immediately* when the user logs out. However, it is unclear how *immediate* the invalidation should and can occur. Firstly, even in the above application scenario the first action is a call of the method `getSession`, and not of the method `invalidate`. Secondly, if resources bound to a session (like, e.g., database connections) must be closed before invalidating the session ID, this conflicts with the requirement to immediately invalidate the session ID. We propose to use a formulation that is more explicit than requiring the "immediate" invalidation of session IDs. We choose an approach that replaces "immediately" with the requirement that session IDs are invalidated within the method that handles the logout, the timeout, or the detection of suspicious behavior, respectively. The corresponding formulation is as follows:

Provide specific methods for handling the logout of a user, the timeout of a session, and the detection of suspicious behavior of a user. These methods shall be called when a user logs out, when a session times out, or when suspicious user behavior is detected, respectively. Within these methods, the session IDs of the corresponding user ID must be invalidated.

This formulation is made precise with the following formalization, where $logoutDone(u)$ describes execution steps corresponding to the return from the method handling the logout procedure, $timeoutDone(u)$ describes execution steps corresponding to the return from the method handling a timeout, and $suspiciousDone(u)$ describes execution steps that correspond to the return from methods handling occurrences of suspicious user behavior:

The labeled transition system $(S, S_0, L, \rightarrow)$ satisfies the more precise variant of the selected secure coding aspect if and only if the following property is satisfied for all $u \in UserIDs$:

$$
\begin{aligned}
(S, S_0, L, \rightarrow) \models \\
&\Box\Big(logout(u) \longrightarrow \big(\Diamond invalidate(u) \ \wedge \ ((\neg logoutDone(u)) \ \mathcal{U} \ invalidate(u))\big)\Big) \wedge \\
&\Box\Big(timeout(u) \longrightarrow \big(\Diamond invalidate(u) \ \wedge \ ((\neg timeoutDone(u)) \ \mathcal{U} \ invalidate(u))\big)\Big) \wedge \\
&\Box\Big(suspicious(u) \longrightarrow \big(\Diamond invalidate(u) \ \wedge \ ((\neg suspiciousDone(u)) \ \mathcal{U} \ invalidate(u))\big)\Big)
\end{aligned}
$$

The formalization specifies that if a logout, a timeout, or suspicious behavior occurs, then the session IDs must eventually be invalidated, and the invalidation must occur before returning from the method that handles the corresponding event.

3.6 Enforce a Strict Authorization Model

Secure Coding Guideline (provided by Siemens [Sie09]):

"For every request, check and enforce user permission for the resource considering given parameters and current context"

"Check on each request for a resource whether the user has all required permissions: Do not link users directly to permissions, instead link users to roles and roles to permissions. Instead of roles, it is possible to link users to *groups* and/or to *attributes*, but the concept of roles should be preferred, because it is simpler, and more extensively used.

To access a specific resource, often not only a role is relevant, but also a further *role parameter* or *user attribute*.

Some generic permissions may be granted to all users within a certain role, independent of the parameter or attribute. But to access a specific information or resource the parameters or attributes may be relevant. In those cases, be sure to check these values too."

Selected Secure Coding Aspect:

"Check on each request for a resource whether the user has all required permissions: Do not link users directly to permissions, instead link users to roles and roles to permissions."

Precise formulation of the selected secure coding aspect: A resource may only be requested for a user after having performed a *successful* check that the user is assigned a role that has sufficient permissions to use the resource. For each subsequent request, an additional successful check must be performed.

Formalization of the selected secure coding aspect:

The labeled transition system $(S, S_0, L, \rightarrow)$ satisfies the selected secure coding aspect if and only if the following property is satisfied for all $u \in \mathit{UserIDs}$ and all $r \in \mathit{Resources}$:

$$(S, S_0, L, \rightarrow) \models P(u,r) \wedge \Box\Big(\mathit{request}(u,r) \longrightarrow \bigcirc P(u,r)\Big)$$

$$\text{where } P(u,r) = \Big((\neg \mathit{request}(u,r))\ \mathcal{U}\ \mathit{checkOK}(u,r)\Big)$$

The formalization uses two labels that are both parametric in u and r, the first being an element of the set *UserIDs*, the second being an element of the set *Resources*. The elements of the set

Label	Description	Exemplary execution steps
request(u, r)	Label indicating that the user with ID u requests resource r	1. A method called by the user with user ID u that creates a reference to a file with fully qualified name r in the form of an object of type `InputStream` 2. The user with ID u invokes the method `findByPrimaryKey` of an Entity Bean's home interface that returns a pointer to a corresponding remote interface r
checkOK(u, r)	Label indicating that a successful check has been performed whether user ID u is assigned a role that has sufficient permissions to use the resource r	1. A method that checks whether the calling user has sufficient permissions to read a given file returns a value indicating that the check was successful 2. Successful return from a method that queries an LDAP-server whether a user has a certain role and then checks whether that role is allowed to access data represented by a Java Entity Bean

Table 8: Labels used in the formalization of the secure coding aspect

UserIDs correspond to the users in a system, and the elements of the set *Resources* correspond to the resources that can be requested by those users. The label *request*(u, r) describes execution steps that start a request of the resource r for the user u, and the label *checkOK*(u, r) describes execution steps that correspond to a successful check whether user u is assigned a role that has sufficient permissions for r.

In the formalization, the formula $P(u, r)$ specifies that the user u must not request the resource r unless it has been successfully checked that u has sufficient permissions for r. The complete formalization hence specifies that a request from user u for the resource r must not occur before the corresponding successful check for u and r, and that after each subsequent request an additional successful check must precede the next request.

Exemplary mapping of labels to execution steps: Table 8 lists the abstract labels used in the formalization and provides an informal description of the execution steps that are described by each label. Moreover, the table illustrates how to map the abstract labels to execution steps by providing at least one example for each label.

Application scenarios: The following two application scenarios illustrate how to apply the formalization for a Java application and for a Java Servlet. The two application scenarios demonstrate that the formalization can be used for quite different kinds of resources and authorization checks.

Scenario 1: Accessing files in Java Consider the Java Program shown in Listing 6. The method `readBudget` receives a username via a String parameter. The method checks whether the corresponding user has read permissions for the file `budget.txt` by invoking the method `checkRead` (compare line 6). If the check fails, the method `readBudget` returns the string `"not supported"` (line 7). If the check succeeds, the first line of the file `budget.txt` is returned (lines 8 and 10).

In this example, execution steps corresponding to the method `checkRead` returning the value `true` have to be annotated with the label *check*(u, r), where u is the username provided as parameter to the method `readBudget` and r represents the file `"budget.txt"`. Execution steps corresponding to the creation of objects providing methods to read the file `budget.txt` (here via the constructor `new FileReader(filename)`) have to be annotated with the label *request*(u, r).

Scenario 2: Using resources in Java Servlets Consider the Java Servlet in Listing 7 that adds billing items to an invoice. If a user has authenticated via HTTP-Authentication, his username is obtained through the invocation of the method `req.getRemoteUser` (compare line 11). The used resources are invoices, their IDs are provided via the request parameter `invoiceId`. Whether a user has sufficient permissions to access an invoice is checked in line 12. Subsequently, the corresponding invoice object is requested and then used in the following loop.

In this scenario, the execution step corresponding to the return from the method `checkPermission` in line 12 has to be annotated with the label *check*(u, r), where u is the username stored in the variable *user* and r is the ID of the invoice. The execution step corresponding to the invocation of the method `getInvoice` in line 18 has to be annotated with the label *request*(u, r), where u and r are as above.

Recommendation for making the secure coding guideline more precise: In the above formalization of the secure coding aspect the permissions of a user for a resource have to be checked before the user makes a request to that resource. In some scenarios where resources are hierarchically structured (consider, e.g., files and directories), it may be sufficient to perform this check for some hierarchically higher resource instead of the resource itself (e.g., for a directory containing a file instead of the file itself). We propose the following more precise formulation supporting hierarchically structured resources:

A resource may only be requested for a user if it has been successfully checked that the user is assigned a role that has all required permissions to request the resource. If suitable, the check can also be performed for another resource containing the requested resource. Such a check is required on each request for a resource.

```
public class Budget {

  public String readBudget (String username) {
    String filename = "budget.txt";

    if (! checkRead(username, filename) )
      return "not supported";
    String s = (new BufferedReader(new FileReader(filename))).readLine();

    return s;
  }
}

public class App {

  public static void main (String args[]) {
    Budget b = new Budget();
    String s = b.readBudget(args[0]);
    System.out.println(s);
  }
}
```

Listing 6: Accessing files in Java

```
public class AddInvoiceItems extends HttpServlet {

  private InvoiceStorage invoices;
  public void setInvoiceStorage(InvoiceStorage invoices) {
    this.invoices = invoices;
  }

  public void doPost(HttpServletRequest req, HttpServletResponse resp) {

    InvoiceId id = new InvoiceId(req.getParameter("invoiceId"));
    String user = req.getRemoteUser();
    try { AccessController.checkPermission(user, id); }
    catch (AccessControlException e) {
      resp.sendError(HttpServletResponse.SC_FORBIDDEN);
      return;
    }

    Invoice invoice = invoices.getInvoice(id);
    int itemCount = Integer.parseInt(req.getParameter("itemCount"));
    for (int i = 0; i < itemCount; i++) {
      String itemDescr = req.getParameter("itemDescr_" + i);
      int itemVal = Integer.parseInt(req.getParameter("itemVal_" + i));
      invoice.addBillingItem(itemDescr, itemVal);
    }

    invoices.commit();

    // Generate HTML response
    ...
  }
}
```

Listing 7: Using resources in Java Servlets

This more precise formulation is formalized as follows:

The labeled transition system $(S, S_0, L, \rightarrow)$ satisfies the selected secure coding aspect for hierarchically structured resources if and only if the following property is satisfied for all $u \in \mathit{UserIDs}$ and $r \in \mathit{Resources}$:

$$(S, S_0, L, \rightarrow) \models P''(u, r) \wedge \Box\Big(\mathit{request}(u, r) \longrightarrow \bigcirc P''(u, r)\Big)$$

$$\text{where } P''(u, r) = \Big((\neg \mathit{request}(u, r))\ \mathcal{U}\ (\mathit{checkOK}(u, r)\ \vee\ \mathit{checkAboveOK}(u, r))\Big)$$

Here, the label $\mathit{checkAboveOK}(u, r)$ describes execution steps that correspond to a successful check whether the user u is assigned a role that has sufficient permissions for a resource that contains r.

3.7 Use Cryptography Properly

Secure Coding Guideline (provided by Siemens [Sie09]):

"Use storage mechanisms for key material provided by framework or operating system"

"Key material is defined as:

- Passwords (DB and OS)
- symmetric keys
- private keys
- public keys
- certificates

If key material has to be stored on your system, never store it in clear text in the file system instead use one of the following mechanisms:

- Use storage mechanisms provided by the framework
- Try using the storage mechanisms of the operating system (e.g., Windows Keystore)
- For embedded systems, consider the usage of secure memory or crypto controller

For the handling of the passwords protecting the keystores, the following rules apply:

(a) No passwords in source code
(b) If possible, use external tokens
(c) If possible, let user enter the password
(d) If stored in configuration files: protect the configuration file appropriately [. . .]"

Selected Secure Coding Aspect:

"If key material has to be stored on your system, never store it in clear text in the filesystem instead use one of the following mechanisms:

- Use storage mechanisms provided by the framework
- Try using storage mechanisms of the operating system (e.g., Windows Keystore)
- For embedded systems, consider the usage of secure memory or crypto controller"

Formalization of the selected secure coding aspect:

The labeled transition system $(S, S_0, L, \rightarrow)$ satisfies the selected secure coding aspect, if and only if the following property is satisfied for all $v \in \mathit{Mem}$:

$$(S, S_0, L, \rightarrow) \models \Box\Big(\mathit{writeKeyDataTo}(v) \longrightarrow \Big((\neg \mathit{store}(v))\ \mathcal{U}\ (\mathit{overwrite}(v)\ \vee\ \mathit{writeEncKeyDataTo}(v))\Big)\Big)$$

This formalization uses four labels that are parametric in an element v of the set *Mem*. The elements of this set correspond to the memory locations used in a program. The label $\mathit{writeKeyDataTo}(v)$ describes execution steps that write clear text key data into the memory location v. The label $\mathit{store}(v)$ describes execution steps that store the contents of v in the file system. The label $\mathit{overwrite}(v)$ describes execution steps where the value of v is overwritten with data that does not depend on key data. The label $\mathit{writeEncKeyDataTo}(v)$ describes execution steps where encrypted key data is written into the memory location v.

The formalization states that whenever clear text key data is written into a memory location v, then the contents of this memory location must not be stored in the filesystem unless the value of v is overwritten with data that does not depend on key data or overwritten with encrypted key data.

Exemplary mapping of labels to execution steps: Table 9 lists the abstract labels used in the formalization and provides an informal description of the execution steps that are described by each label. Moreover, the table illustrates how to map the abstract labels to execution steps by providing at least one example for each label.

Application scenario: As an example, consider the program shown in Listing 8. The method `setPassword` obtains a new password from the command line (compare line 8), which is then encrypted. The encrypted version is written to a file via the `println` method of the `BufferedWriter`-object `out` (compare lines 4 and 11).

Here, the execution step corresponding to the assignment from line 8 needs to be annotated with the label $\mathit{writeKeyDataTo}(\mathit{pwd})$, the execution step corresponding to the assignment from line 10 needs to be annotated with the label $\mathit{writeEncKeyDataTo}(\mathit{encodedPwd})$, and the execution step corresponding to the method invocation of `println` in line 11 needs to be annotated with the label $\mathit{store}(\mathit{encodedPwd})$.

This program satisfies the formalized aspect. However, when replacing the code from line 11 with the statement in the comment from line 12, this is no longer true, as `pwd` is not overwritten with encrypted key data or with data that is independent of key data before passing it as a parameter to the method `out.println`.

Label	Description	Exemplary execution steps
writeKeyDataTo(*v*)	Label indicating that clear text key data is written into memory location *v*	1. Assigning a value derived from another variable containing clear text key data to variable *v* 2. Assigning the return value of the method `Keystore.getKey` from the Java Cryptography Architecture to variable *v*
writeEncKeyDataTo(*v*)	Label indicating that encrypted key data is written into memory location *v*, for instance via an encryption, or via a storage mechanism of the framework or the operating system, or via a crypto controller	1. Assigning the value computed by the method `Cipher.doFinal` from the Java Cryptography Architecture for an argument containing clear text key data to variable *v* 2. Copying the value of a memory location containing encrypted key data to memory location *v*
overwrite(*v*)	Label indicating that data which is independent of key data is written into memory location *v*	Assignment of arbitrary, non-key related data to variable *v*
store(*v*)	Label indicating that the data in memory location *v* is stored in the filesystem	Invoking the Java method `Filewriter.write` with argument *v*

Table 9: Labels used in the formalization of the secure coding aspect

```
1  public class PwdExample {
2    static String key = Keystore.getKey();
3    static BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
4    static BufferedWriter out = new BufferedWriter(new FileWriter("pwdstore.txt"));
5
6    public void setPassword () {
7      System.out.println("Enter new password:");
8      String pwd = in.readln();
9
10     String encodedPwd = encrypt(pwd, key);
11     out.println(encodedPwd); // OK
12     // Not OK: out.println(pwd);
13   }
14
15   public static void main (String[] args) {
16     (new PwdExample()).setPassword();
17   }
18 }
```

Listing 8: Changing password

4 Outlook

The precision gained by specifying program properties *formally* is already a significant advantage over informal specifications. Formal specifications clearly identify the execution steps that are relevant for a given property, and often reveal imprecisions and unclarities in the informal descriptions of properties. In addition, formal specifications of secure coding guidelines lay the foundation for the following possibilities (which are outside the scope of this report): dynamically monitoring if a program execution adheres to a secure coding guideline and statically verifying that a program satisfies a secure coding guideline. In this section, we give a brief outlook on these possibilities for the future. Another potential direction for future work is to support the correct application of secure coding guidelines further by providing assistance for the placement of labels. We briefly discuss this possibility at the end of this section.

LTL specifications and runtime monitoring. A *runtime monitor* is a process that runs in parallel to the program that is to be watched. It monitors whether the program adheres to a given specification (e.g., a secure coding guideline).

LTL formulas offer a basis to construct such runtime monitors. We demonstrate this with the example of the formalization for a secure coding guideline focusing on authorization models (see Section 3.6): An aspect of this secure coding guideline states that for each request for a resource it must be checked whether the requesting user has sufficient permissions to access the resource. For user u and resource r, the formalization is given by the following LTL formula:

$$P(u,r) \wedge \square\Big(\mathit{request}(u,r) \longrightarrow \bigcirc P(u,r)\Big),$$
$$\text{where } P(u,r) = \Big((\neg \mathit{request}(u,r))\ \mathcal{U}\ \mathit{checkOK}(u,r)\Big).$$

The LTL specification makes explicit which execution steps must be observed by a runtime monitor for this secure coding guideline: To monitor a program with respect to this secure coding guideline, the runtime monitor must observe all execution steps that correspond to a request for a resource and all execution steps that correspond to a successful check that a user has sufficient permissions to access a resource. Whenever the runtime monitor observes a request without having observed a corresponding successful check earlier, the secure coding guideline is not respected by the program. In this case, the runtime monitor could, for instance, stop the program to inhibit a security violation, or it could simply log the error for auditing purposes.

The construction of a runtime monitor for an LTL formalization of a concrete secure coding guideline provides some challenges. For instance, one has to provide a way to specify the execution steps that the runtime monitor has to observe, e.g., based on program annotations. For these program annotations a well-defined semantics is required so that the program monitor is able to correctly associate the execution steps with the labels from the annotations. Besides, one has to provide an efficient mechanism that allows the runtime monitor to infer whether the LTL formalization is violated by the currently observed execution step. This mechanism could possibly be based on a representation of LTL formulas as finite state automata.

LTL specifications and verification. The goal of *verification* is to prove once and for all that a program satisfies a specification (e.g., a secure coding guideline). The resulting proof

guarantees that *each possible* program execution complies with a specification, whereas a runtime monitor always considers just a *given single* program execution. Consequently, one can view verification as a way of making runtime monitoring redundant, because a runtime monitor never has to intervene in the program execution if the program has been proven to satisfy the given LTL specification.[2]

This extensive security guarantee provided by verification comes at a cost: Program verification is typically much more time-consuming than implementing and running a runtime monitor. To verify a program, either the source code or the compiled code is scrutinized wrt. the specification. In order to verify that a given program satisfies the secure coding guideline discussed above, one needs to show that *on each execution sequence* all resource requests are preceded by a corresponding successful check of the user's permissions to access the resource.

A benefit of formal specifications (such as LTL formulas) is that they can be processed by a computer. Thus so-called *proof assistants* can be used to support the verification process by applying formal methods. For certain programs, *model checking* tools such as NuSMV [CCG+02] or SPIN [Hol03] can automatically check if an LTL formula is satisfied. Furthermore, interactive verification tools such as Isabelle [NPW02], KeY [BHS07], or VSE [AHL+00] help to manage the complexity of the verification task.

Verification can be fruitfully combined with runtime monitoring: A new program can be observed by a runtime monitor to detect accidental security violations during testing. When these security flaws have been fixed so that the program code is stable (in the sense that it is not going to be changed soon), verification of the program can start to ensure that no security flaws have been overlooked in the testing phase. Alternatively, if some parts of a large program are formally verified (for instance, some core API methods), then a runtime monitor can be employed to prevent security violations within the non-verified parts of the program.

Support for label placement. While it is straightforward to associate execution steps with corresponding concrete labels like, for instance, $a = \mathit{reader.read}()$, there is a potential for making mistakes when associating execution steps with abstract labels. One might, for instance, miss a method invocation corresponding to a call of a system command when associating execution steps with the label $\mathit{passToSysCmd}(v)$ from Section 3.1. For some abstract labels, one also needs to take into account in which context a command is used. For such labels, one might more easily make a mistake such as forgetting to associate an execution step with a label.

As an example, suppose that a developer wishes to check if the Java Servlet shown in Listing 9 complies with the formalized secure coding aspect from Section 3.1, i.e., if the Servlet validates every input before passing it to system commands. Let us begin with the straightforward part of the label placement: The return from method `validFilename` with return value `true` for parameter `filename` in line 7 has to be annotated with the label $\mathit{validate}(\mathit{filename})$. Each usage of the variable `filename` in lines 8, 11, and 12 has to be annotated with the label $\mathit{useInComputation}(\mathit{filename})$. Similarly, each usage of the variable `action` in lines 9 and 10 has to be annotated with the label $\mathit{useInComputation}(\mathit{action})$. The invocation of the function `exec` with

[2] Conversely, runtime monitoring could be viewed as a way of making verification redundant, because if a program is augmented with a runtime monitor that reliably enforces the LTL specification, then the combination of the program with the runtime monitor satisfies the specification by construction.

```
1  public class FileServlet extends HttpServlet {
2
3    public void doGet(HttpServletRequest req, HttpServletResponse res)
4                                throws ServletException, IOException {
5      String filename = req.getParameter("filename");
6      String action = req.getParameter("action");
7      if (!validFilename(filename)) throw new ServletException("Invalid filename");
8      filename = "/userfiles/" + filename;
9      action = String.toLowerCase(action);
10     if (action.equals("delete"))
11       deleteFile(filename);
12     else backupFile(filename);
13   }
14
15   private void deleteFile(String filename) throws IOException {
16     File file = new File(filename);
17     file.delete();
18   }
19
20   private void backupFile(String filename) throws IOException {
21     String cmd = "cp " + filename + " " + filename + ".bak";
22     Runtime.getRuntime().exec(cmd);
23   }
24
25 }
```

Listing 9: Java Servlet using input validation

parameter `cmd` in line 22 has to be annotated with the label *passToSysCmd*(*cmd*).

Now what about the assignments in lines 5 and 6? In line 5, the assignment to the variable `filename` has to be annotated with the label *inputForSysCmd*(*filename*), because this user input will be used in the computation of a parameter for a system command. In contrast, the assignment to the variable `action` in line 6 should *not* be labeled with *inputForSysCmd*(*action*); although user input is written to `action`, this user input will not be used as a parameter of a system command. Obviously, one needs to trace the flow of user input through memory locations to find out if an assignment to a variable v needs to be labeled with *inputForSysCmd*(v) or not. In more complex pieces of code, this may be less apparent and, hence, more error-prone than in this small example Servlet.

In order to avoid errors in the association of execution steps with labels such as *inputForSysCmd*(v), some support is desirable that helps developers in this process. For instance, it might be helpful to give feedback whether label annotations are correct or even to identify label annotations automatically. Giving such support provides some challenges: A first step would be to determine a precise criterion for the correctness of a label annotation. A promising approach for developing such a criterion is the usage of so-called information flow properties. Information flow properties allow to make precise statements about the flow of information in a program, e.g., whether information flows from a variable containing user input to a variable that is passed to a system command. For such a criterion, one could then develop analysis techniques that check whether the label annotations for a given program satisfy the criterion, e.g., whether user input is actually used to compute parameters passed to system commands. Beyond that, one could develop analysis techniques that identify the necessary annotations. For instance, these analysis

techniques could be based on information where input enters a program and which memory locations are passed to system commands.

The Java Servlet from above demonstrates that one needs to consider the effects of commands in the source code thoroughly when applying a secure coding guideline; e.g., one needs to reason about the flow of user input in the example. Of course, these considerations are also necessary when applying the formalized guideline. The explicit association of execution steps with labels provides a basis for checking if all relevant execution steps have been considered correctly for a given guideline. Consequently, automated support for the placement of labels would be a promising future step.

5 Conclusion

In this collaboration between Siemens AG and TU Darmstadt we developed formalizations of individual aspects of seven secure coding guidelines. These guidelines have been kindly provided by Siemens for examination purposes [Sie09]. They are similar to Siemens-internal secure coding guidelines that were developed as recommendations for programmers. The selected guidelines cover a broad spectrum of secure coding domains, which shows that formalizations for such guidelines are feasible.

As the guidelines describe programming practices that support developers in avoiding common security vulnerabilities, the importance of adopting these practices when writing programs is widely acknowledged. Applying these guidelines requires a good understanding of them, which in turn is supported by a precise formulation of the guidelines.

Our formalizations for the secure coding guidelines systematically present the components of the guidelines, i.e., the relevant program actions as well as the required relations between those program actions. The precision gained by the formalization may help developers to identify critical points in the code that are relevant for a given secure coding guideline, and thereby simplifies the correct application of the secure coding guidelines.

Furthermore, the formalization revealed unclarities as well as missing concepts in the secure coding guidelines. This enabled us to give recommendations to improve the formulation of the informal guidelines. These recommendations contribute to reducing the risk that guidelines are misinterpreted and to facilitating that developers can adopt them more consistently.

In summary, the exemplary formalizations for secure coding guidelines provide reference points for secure coding that are more precise than informal descriptions. The formalizations suggest a structured approach to apply them during the development process. We expect that this helps developers to follow these guidelines more easily.

Possible future directions. The precision of the formalized secure coding guidelines creates interesting opportunities for further support of the development of secure programs: With the help of *runtime monitors* based on the formalization, quality assurance can be improved by automatically detecting hazardous situations (i.e., potential violations of the secure coding guidelines) at runtime. Since runtime monitors use label annotations from the programs they monitor, it is important to assign these labels correctly. Therefore quality assurance can be supported even further by developing formal criteria to check the label annotations for correctness. These formal criteria could possibly be checked by an automated tool to simplify the labeling process. An automated verification of the placement of labels could also be beneficial for developers, e.g., during a program inspection. Another potential strand of future work is the development of static *verification* mechanisms that facilitate a more general analysis of programs wrt. secure coding guidelines: Instead of monitoring the program behavior at runtime, an “offline” analysis investigates all conceivable program executions and checks whether a formalized secure coding guideline is satisfied.

The results described in this report show that a formalization of secure coding guidelines is feasible. In order to increase the benefits of secure coding, it would be desirable to formalize further secure coding guidelines.

Acknowledgements

We thank Michael Drescher who contributed numerous illustrative examples to this report.

References

[AHL+00] 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. *International Journal on Software Tools for Technology Transfer, Special Issue on Mechanized Theorem Proving for Technology*, 3(1), 2000.

[BHS07] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An Open Source Tool for Symbolic Model Checking. In *Proceedings of the International Conference on Computer-Aided Verification (CAV-2002)*, volume 2404 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[DE99] S. Drossopoulou and S. Eisenbach. Describing the Semantics of Java and Proving Type Soundness. In *Formal Syntax and Semantics of Java*, pages 41–82, 1999.

[Hol03] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[HR04] M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge University Press, 2004.

[KN06] G. Klein and T. Nipkow. A Machine-checked Model for a Java-like Language, Virtual Machine, and Compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[Sea08] R. C. Seacord. *The Cert C Secure Coding Standard*. Addison-Wesley, SEI Series in Software Engineering, 2008.

[Sie09] Exemplary secure coding guidelines provided by Siemens AG as input. The exemplary guidelines are similar, but not identical to Siemens-internal secure coding guidelines that were developed as recommendations for programmers. 2009.