Choosing a Formalism for Secure Coding: FSM vs. LTL

Technical Report TUD-CS-2013-0180

June 2013

Markus Aderhold
Alexander Gebhardt
Heiko Mantel

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Modeling and Analysis of
Information Systems

1 Introduction

Many secure coding guidelines give advice on how to avoid certain code patterns that may lead to vulnerabilities [Sea08, LMS+11]. Since the informal description of patterns as well as the given advices differ in their level of abstraction and precision, the process of recognizing the pattern and following the corresponding advice is usually not straightforward and thus error-prone.

In order to get reference points for secure coding that are more precise than informal descriptions of guidelines, formal descriptions in Linear Temporal Logic (LTL) were proposed in [ACMS10]. Formalizations in LTL specify the required causal and temporal relations between program actions that are relevant to secure coding guidelines. For the analysis of C code, finite state machines (FSM) were used in [CDW04, TYHD09] as a formalism to capture temporal safety properties.

In this report, we consider the choice of an appropriate formalism for secure coding. On the one hand, the chosen formalism shall be sufficiently expressive so that it can capture the property that is described by a given secure coding guideline. On the other hand, the formalism shall be easy to understand so that formalized secure coding guidelines reach a large audience (e.g., software auditors, software developers, and developers of analysis tools). For instance, formulas in LTL can express both safety properties and liveness properties, while FSMs as used in [CDW04, TYHD09] only capture safety properties. However, FSMs have the advantage of a convenient graphical notation that visualizes the respective safety properties.

The purpose of this report is to provide some help in choosing a formalism for secure coding. Specifically, we present formalizations of five secure coding guidelines from [ACMS10] as FSMs and contrast them with the existing formalizations in LTL. Furthermore, we present formalizations of four secure coding guidelines from [Sea08] as FSMs and in LTL. All of these formalizations serve as examples in order to identify advantages and disadvantages of the formalisms in the context of secure coding.

In Section 2, we define FSMs as a formalism for specifying secure coding guidelines. Section 3 presents formalizations of secure coding guidelines from [ACMS10] as FSMs. In Section 4, we describe formalizations of secure coding guidelines from [Sea08] as FSMs and in LTL. Section 5 discusses some advantages and disadvantages of the formalisms based on the exemplary formalizations.

2 Formalism

This section describes our system model, the use of nondeterministic finite state machines (FSMs) to specify temporal safety properties, and a graphical notation for FSMs.

2.1 Modeling the Execution of Programs

As in [ACMS10], we describe the possible program executions via a so-called *Labeled Transition System*. *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*). The following definitions coincide with those from [ACMS10]:

Definition 2.1 (Labeled transition systems)**.** *A* labeled transition system *(LTS) 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* $\rightarrow \subseteq S \times (\mathscr{P}(L) \setminus \{\emptyset\}) \times S$ *(where* $\mathscr{P}(L)$ *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, but it can also be annotated with more than one label (one concrete label and zero or more abstract labels).

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

Definition 2.2 (Execution sequences)**.** *An* execution sequence *of the labeled transition system* $(S, S_0, \rightarrow, L)$ *is a pair* $\sigma = \big((s_i)_{i\in\mathbb{N}}, (L_i)_{i\in\mathbb{N}}\big)$ *of infinite sequences, written as*

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

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

1. *For each* $i \in \{0, 1, \dots\}$*,*
   - *(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* $L_i' \in \mathscr{P}(L) \setminus \{\emptyset\}$ *and* $s_i' \in S$ *such that* $(s_i, L_i', s_i') \in \rightarrow$*.*
2. *For each* $i \in \{0, 1, \dots\}$*, if* $L_i = \emptyset$*, then* $L_j = \emptyset$ *for each* $j > i$*.*

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).

2.2 Specification of Temporal Safety Properties with FSMs

Before using nondeterministic finite state machines (FSMs) to specify temporal safety properties, we briefly recapitulate the standard definition of nondeterministic finite state machines [HU79]:

Definition 2.3 (FSM)**.** *A* nondeterministic finite state machine *(FSM) is a 5-tuple* $M = (Q, \Sigma, \delta, q_0, F)$ *where* Q *is a finite set of states,* Σ *is a finite set of input symbols,* $\delta : Q \times \Sigma \to \mathscr{P}(Q)$ *is a so-called transition function,* $q_0 \in Q$ *is the initial state, and* $F \subseteq Q$ *the set of final states.*

The transition function δ *is extended to sets of states and strings (i.e., finite sequences) of input symbols with* ϵ *being the empty string by:*

$$\begin{aligned} \widehat{\delta} &: \mathscr{P}(Q) \times \Sigma^* \to \mathscr{P}(Q) \\ \widehat{\delta}(Q', \epsilon) &:= Q' \\ \widehat{\delta}(Q', wa) &:= \{q \in Q \mid \exists q' \in \widehat{\delta}(Q', w).\ q \in \delta(q', a)\} \end{aligned}$$

The set $\mathscr{L}(M)$ *of strings* $w \in \Sigma^*$ *that are* accepted *by* M *is defined as*

$$\mathscr{L}(M) := \left\{w \in \Sigma^* \mid \widehat{\delta}(\{q_0\}, w) \cap F \neq \emptyset\right\}.$$

In addition to the standard definition, we require the transition function δ to specify at least one possible successor state $q' \in Q$ for each $q \in Q$ and for each $s \in \Sigma$:

$$\forall q \in Q.\ \forall s \in \Sigma.\ \exists q' \in Q.\ q' \in \delta(q, s)$$

In order to specify a temporal safety property with an FSM, we will use the labels from the labeled transition system that describes the possible program executions as input symbols of the FSM: $\Sigma := L$. The set F of final states will represent the states where the temporal safety property has been violated, so F can be considered as the set of error states.

Intuitively, a labeled transition system satisfies the temporal safety property given by an FSM if it is impossible to reach an error state with any execution sequence.

Definition 2.4 (Satisfaction of temporal safety properties)**.** *Let* $\sigma = s_0 \xrightarrow{L_0} s_1 \xrightarrow{L_1} \ldots$ *be an execution sequence of some labeled transition system* $(S, S_0, L, \to)$*. Furthermore, let* $M = (Q, \Sigma, \delta, q_0, F)$ *an FSM such that* $\Sigma = L$*. We say that the execution sequence* σ satisfies *the temporal safety property given by* M *if and only if* $\epsilon \notin \mathscr{L}(M)$ *and for each* $n \in \mathbb{N}$ *and for each* $w = w_0 \ldots w_n \in \Sigma^*$*: If* $w_i \in L_i$ *for each* $i \in \{0, \ldots, n\}$*, then* $w \notin \mathscr{L}(M)$*.*

2.3 Graphical Notation

In this section, we explain the relation between a graphical notation of an FSM with the elements from the tuple $(Q, \Sigma, \delta, q_0, F)$.

As in the exemplary FSM depicted in Figure 2.1, we represent states with nodes in the graphical notation (depicted as circles). The nodes are labeled with elements from Q. For each state in Q, there exists one node in the graph. All and only the final states are double-circled. Directed edges between circles, denoted by lines, are labeled with $s()$ for elements $s \in \Sigma$ and represent transitions from δ: Each edge (with one exception), starts at some node and ends with arrowhead at either the same or at another node. For each edge in the graph labeled with $s()$, starting at a node labeled with q, and ending at a node labeled with q', it holds that $\delta(q, s) = q'$. An edge without a label and without a start node is directed at the initial node.

In the following, we introduce three abbreviations to the graphical notation concerning the edges.

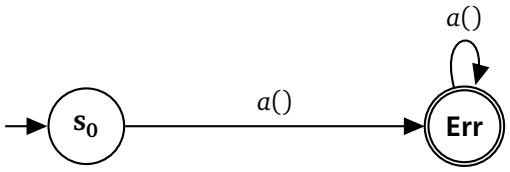


Figure 2.1: Standard notation

Implicit self-loops

The first abbreviation is the omission of edges for transitions that retain the state, i.e., *self-loops*. Omitting self-loops is a common technique for minimizing the graphical representation of an FSM. Formally, if $\delta(q,s) = \{q\}$, then the self-loop on state q for input symbol s is omitted in the graphical notation. Conversely, if for some state q and some input symbol s there is no outgoing edge from q labeled with s, then $\delta(q,s) = \{q\}$. For example, in Figure 2.2 all self-loops of the FSM in Figure 2.1 are made implicit.

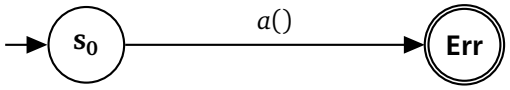


Figure 2.2: Notation that omits implicit self-loops

Parameterized labels

The second abbreviation is the use of *parameterized labels*. A parameterized label can be instantiated by replacing all parameter positions in the label with actual values. An instance of a parameterized label can either be an abstract or a concrete label, while neither abstract nor concrete labels can be parameterized. An FSM using parameterized labels is called a *parametric FSM* and is depicted as in Figure 2.3.

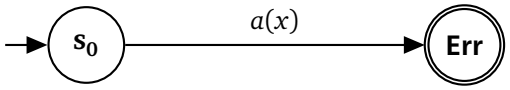


Figure 2.3: Labels parametric in variable x

Given a mapping that associates each parameter x with a value v_x, an instance of a parametric FSM is obtained from the parametric FSM by replacing each occurrence of a parameter x with the corresponding value v_x. For example, the FSM in Figure 2.4 is an instance of the parametric FSM in Figure 2.3. A temporal safety property expressed by a parametric FSM is satisfied if and only if for each mapping from parameters to values the property is satisfied by the corresponding instance of the FSM.

Grouped labels

The third and last abbreviation are *grouped labels*. Where multiple edges e_1 to e_m in the graph have the same start node q and end node q', these edges can be grouped by using a single edge. This edge starts at node q, ends at node q', and is labeled with a comma separated list of all and only the labels

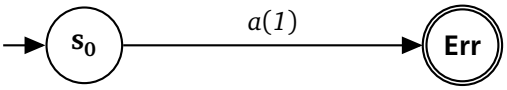


Figure 2.4: Instantiated labels

from the edges e_1 to e_n. For instance, the FSM in Figure 2.5 differs from the FSM in Figure 2.2 in that there is an additional edge from s_0 to Err labeled with $b()$.

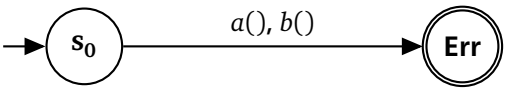


Figure 2.5: Grouped labels

3 Secure Coding Guidelines for Java

This section presents formalizations for five secure coding guidelines as FSMs. For each guideline, we select the same secure coding aspect from the guideline as in [ACMS10].

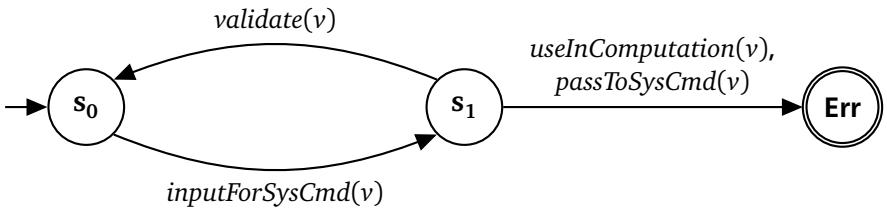
3.1 Validate User Input

The validation of user input shall help to prevent that user input influences the behavior of a program in an unintended manner. The following aspect addresses the validation in the security critical context of system commands.

Selected secure coding aspect (as in [ACMS10, p. 12]):
"Validate every input thoroughly before passing it to system commands."

Precise formulation of the selected secure coding aspect (as in [ACMS10, p. 12]):
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 $(S, S_0, \rightarrow, L)$ satisfies the selected secure coding aspect, if and only if it satisfies the temporal safety property given by the FSM in Figure 3.1 for each $v \in Mem$.



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

Figure 3.1: Validate User Input

The FSM uses the same labels as the LTL formula from [ACMS10, p. 12 f], which is given for comparison purposes below the FSM in Figure 3.1. The labels are parametric in an element v from the set *Mem* that denotes locations in the main memory. A description of all labels used in this formalization is given in Table 3.1.

The FSM represents the selected aspect as follows: It has an initial state s_0, where no input has been processed yet. As soon as input for a system command is written into memory location v, a transition is made into state s_1, where a validation of v is required before its use in a computation or before it is passed to a system command. A validation makes v safe to be processed, which is modeled by a transition into the initial state. Using v in a computation or passing v to a system command is allowed in the initial state unless further input for system commands is written into v. Whenever v is used in a

Label	Description
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
useInComputation(v)	Label indicating that memory location v is used in a computation, except for computations that are performed to validate v
passToSysCmd(v)	Label indicating that the value in memory location v is passed as a parameter to a system command
validate(v)	Label indicating that the value of memory location v is validated successfully

Table 3.1: Labels used in the formalization of the secure coding aspect (as in [ACMS10])

computation or passed to system command before validation, a transition is made into a state *Err*, which models a violation of the secure coding aspect.

3.2 Sanitize the Output

Whereas the validation of user input prevents certain attacks against the program or host, the sanitization of output shall help to prevent malicious users from attacking benign users via malformed browser output, for example.

Selected secure coding aspect (as in [ACMS10, p. 17 f]):

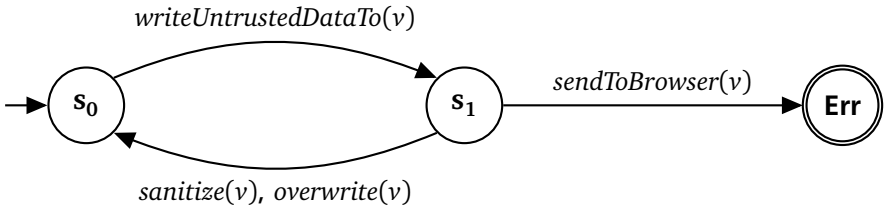
"Whenever possible sanitize all output data of untrusted sources (i.e., user input) before it is sent to the browser."

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 $(S, S_0, \rightarrow, L)$ satisfies the selected secure coding aspect, if and only if it satisfies the temporal safety property given by the FSM in Figure 3.2 for each $v \in Mem$.



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

Figure 3.2: Sanitize the Output

Label	Description
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)
sendToBrowser(v)	Label indicating that the value of the memory location v is sent to the browser
sanitize(v)	Label indicating that the sanitization of the value in memory location v has completed
overwrite(v)	Label indicating that a value not depending on data from untrusted sources is written into the memory location v

Table 3.2: Labels used in the formalization of the secure coding aspect (as in [ACMS10])

The FSM uses the same labels as the LTL formula from [ACMS10, p. 17 f] which is given for comparison purposes below the FSM. The labels are parametric in an element v from the set *Mem*, that denotes locations in the main memory. A description of the labels is given in Table 3.2.

The structure of the FSM in Figure 3.2 is similar to the FSM in Figure 3.1. In the initial state s_0, the content of a memory location v is assumed to be safe to be sent to a browser. As soon as untrusted data is written into v, a transition is made into an intermediate state s_1, where v must not be sent anymore to a browser. If the content of v is sanitized, i.e., modified such that it fits the syntactical security requirements of the browser, or overwritten with trusted data in this state, the FSM transits back into its initial state. Sending the untrusted content of v in the intermediate state to a browser is regarded as a violation of the secure coding aspect and thus leads into the state *Err*.

3.3 Secure the Internal Flow

SQL injection attacks exploit the construction of SQL queries with user data that is not being validated or sanitized. Stored procedures shall prevent SQL injection attacks, if used properly; assigning the untrusted input to parameters in a stored procedure or prepared statement does not affect the logic of the query directly unless the procedure or statement itself parses its parameters as SQL. The guideline "Secure the internal Flow" focuses on the direct effects.

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." [ACMS10, p. 22 f]

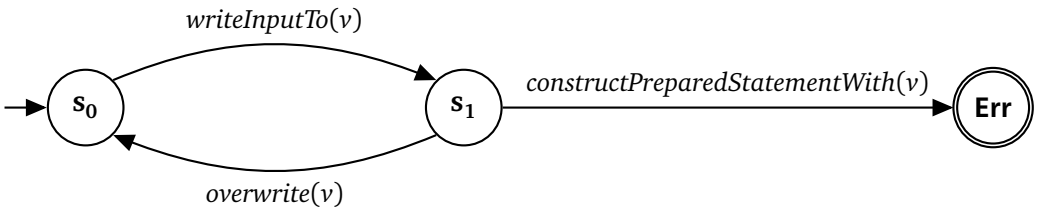
Precise formulation of the selected secure coding aspect (as in [ACMS10, p. 22 f]):

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 $(S, S_0, \rightarrow, L)$ satisfies the selected secure coding aspect, if and only if it satisfies the temporal safety property given by the FSM in Figure 3.3 for each $v \in Mem$.

The FSM uses the same labels as the LTL formula from [ACMS10, p. 22 f], which is given for comparison purposes below the FSM. The labels are parametric in an element v from the set *Mem* that denotes locations in the main memory. A description of the labels is given in Table 3.3.



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

Figure 3.3: Secure the Internal Flow

Label	Description
writeInputTo(v)	Label indicating that the value of memory location v is changed and now depends on input data
constructPreparedStatementWith(v)	Label indicating that memory location v is used as parameter for a method constructing a Prepared Statement
overwrite(v)	Label indicating that the value of the memory location v is changed and now does not depend on input data

Table 3.3: Labels used in the formalization of the secure coding aspect (as in [ACMS10])

Again, the depicted FSM in Figure 3.3 is similar to the preceding ones. In the initial state s_0, memory location v contains non-input data. As input is written into v, a transition is made into state s_1, where the content of v is not safe to be used in the construction of a prepared statement. Overwriting the content of v with non-input data leads the FSM back into its initial state, where a statement may be constructed using v. If a prepared statement is constructed in state s_1, however, the state *Err* is reached, which constitutes a violation of the guideline.

3.4 Secure the Login and Authentication Procedures

Almost all modern password policies limit the number of failed login attempts to a system in order to prevent brute force attacks or guessing of passwords. The following guideline aims to enforce such a policy while paying attention to the problem of denial of service attacks enabled by the policy itself.

Selected secure coding aspect (as in [ACMS10, p. 27 f]):

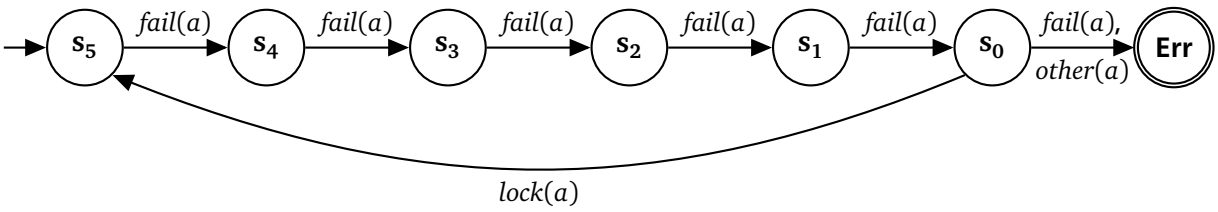
"Limit login attempts to 5 tries per account. After 5 failed login attempts lock the account for at least 10 min. If this happens again, lock the account for longer periods of time. Do not use a sleep or delay method (after the unsuccessful login attempts) to "lock" the account, this will provide DoS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

Precise formulation of the selected secure coding aspect (as in [ACMS10, p. 27 f]):

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 $(S, S_0, \rightarrow, L)$ satisfies the selected secure coding aspect, if and only if it satisfies the temporal safety property given by the FSM in Figure 3.4 for each $a \in Accounts$.



$$(S, S_0, L, \rightarrow) \models P(a,5) \wedge \Box(lock(a) \longrightarrow \bigcirc P(a,5))$$

$$\begin{aligned} &\text{where} \quad P(a,0) = lock(a) \\ &\quad \text{and} \quad P(a,n) = ((\neg fail(a)) \; \mathcal{U} \; \bigcirc P(a,n-1)) \quad \text{for } n > 0 \end{aligned}$$

Figure 3.4: Secure the Login and Authentication Procedures

The labels for both the FSM and the LTL formula are parametric in an element a from the set *Accounts* denoting accounts in the system. A description of the labels is given in Table 3.4.

In the initial state s_5 of the FSM, five login attempts are left for account a. Whenever a login attempt for account a fails, the number of left login attempts for this account is decreased by one, which is modeled by a transition into a state with an index decreased by one, thus equaling the new number of left login attempts. After five failed login attempts, the state s_0 is reached, where any action not locking the account immediately leads into the accepting state denoting a violation of the secure coding guideline. If in s_0 account a is being locked, the number of login attempts is reset to five and the internal lock time is increased for the next round.

Note that the graphical notation of the FSM omits the implicit self-loops for actions $lock(a)$ and $other(a)$ in states $s_1, \ldots, s_5$; e.g., locking the account in any of these states is permitted and would leave the number of remaining login attempts unchanged. In state s_0, however, the transition with label $other(a)$ into state *Err* ensures that there is no implicit self-loop for action $other(a)$; thus the next execution step must lock the account in order to satisfy the secure coding guideline.

The LTL formula $P(a,n)$ specifies that after n failed login attempts for account a, a will be locked. For $n = 0$ this is expressed by requiring the event $lock(a)$ to occur. For $n > 0$ either no login attempt for a fails anymore or a login attempt for a fails and the amount of left login attempts is decreased by one, which is specified by $P(a, n-1)$. The right part of the conjunction in the first line of the formula ensures that whenever an account a is locked after 5 failed login attempts, the number of left login attempts is reset to five. From the definition of the label $lock(a)$, $P(a,n)$ also specifies that the lock time for account a is increased with every execution of the steps corresponding to the label.

Comparing these two formalizations, we observe that the LTL formalization succinctly describes the decreasing number of login attempts via the recursive definition. The FSM is relatively large and less

Label	Description
$fail(a)$	Label indicating a failed login attempt for account a that is not caused by the account being locked.
$lock(a)$	Label indicating that the account a is being locked for some time associated with a, which initially is 10 minutes and is increased with every further lock event. Sleep or delay methods must not be annotated with this label.
$other(a)$	Label indicating an action that does not lock the account a and that is no failed login attempt for account a

Table 3.4: Labels used in the formalization of the secure coding aspect

succinct, but still manageable due to the obvious regularity in the transitions from s_5 to s_0 via s_4, s_3, s_2, and s_1.

Omitting the self-loops for states $s_1, \ldots, s_5$ reduces the size of the graphical representation of the FSM, but it is important to keep the existence of implicit self-loops in mind. For example, in state s_0 no self-loop is desirable, because the secure coding aspect requires the account to be locked after the fifth failed login attempt. Thus it is important to specify the transitions labeled with $fail(a)$ and $other(a)$ from state s_0 to Err. We suspect that the convention of leaving self-loops implicit (though customary as also in [CDW04, TYHD09], for example), might lead to errors in FSM formalizations, because one might forget to keep the existence of self-loops in mind.

Difference to the original formalization:

The original formalization in [ACMS10, p. 27 f] includes the lock time constraint in the form of an additional parameter k in a label $lock(a, k)$. The introduction of this parameter enables reasoning about the lock time, e.g., that it is actually increased with every lock or that it initially is ten minutes for an account. However the corresponding LTL formula for the aspect makes use of first order constructs in terms of quantification over parameters of the propositional formulae, which to our knowledge is unsupported yet by available tools for program analysis. For readability reasons we left the parameter k out of our formalization. An additional dedicated FSM using labels for initialization and increment of the value of k could be specified to reason about the time constraints.

Possible modification of the original aspect:

Locking an account after a fixed amount of failed login attempts, no matter how many attempts succeeded in between, might be a suitable behavior for some scenarios; e.g., for authentication attempts with electronic cash cards. In some other application scenarios, however, it might be desirable to let a successful login reset the number of left failed login attempts; for example, this concerns scenarios where authentication credentials are frequently mixed up. Such a scenario is not covered by the current formulation of the aspect. In the following part, the corresponding idea from [ACMS10] is adapted and applied to the FSM in Figure 3.4.

Precise formulation of the modified secure coding aspect (as in [ACMS10, p. 27 f]):

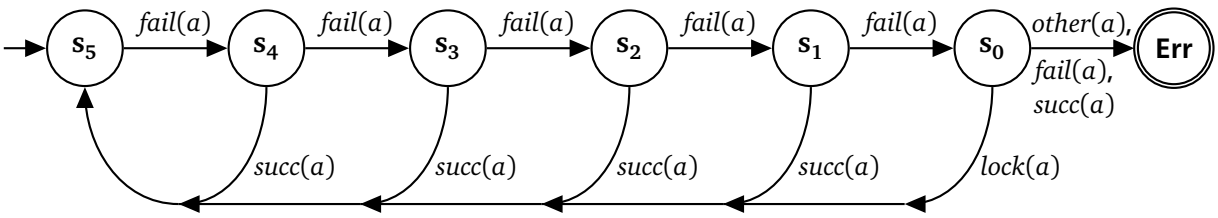
When five failed login attempts have been performed for an account and no successful login attempt has been performed for that account between those failed attempts, then that account must be locked for at least ten minutes. Whenever five failed login attempts occur again without an intermediate successful 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.

Label	Description
$fail(a)$	Label indicating a failed login attempt for account a that is not caused by the account being locked.
$lock(a)$	Label indicating that the account a is being locked for some time associated with a, which is 10 minutes for the first lock and is increased with every further lock. Sleep or delay methods must not be annotated with this label.
$succ(a)$	Label indicating a successful login attempt for account a. The method annotated with this label shall reset the lock time for account a to 10 minutes.
$other(a)$	Label indicating an action that does not lock the account a and is neither a failed nor a successful login attempt for account a.

Table 3.5: Labels used in the formalization of the secure coding aspect

Formalization of the modified secure coding aspect:

The labeled transition system $(S, S_0, \rightarrow, L)$ satisfies the modified secure coding aspect, if and only if it satisfies the temporal safety property given by the FSM in Figure 3.5 for each $a \in Accounts$.



$$(S, S_0, L, \rightarrow) \models P'(a,5) \wedge \Box((lock(a) \vee succ(a)) \longrightarrow \bigcirc P'(a,5))$$

where $P'(a,0) = lock(a)$
and $P'(a,n) = ((\neg fail(a)) \; \mathcal{U} \; ((\bigcirc P'(a,n-1)) \vee succ(a)))$ for $n > 0$

Figure 3.5: Secure the Login and Authentication Procedures

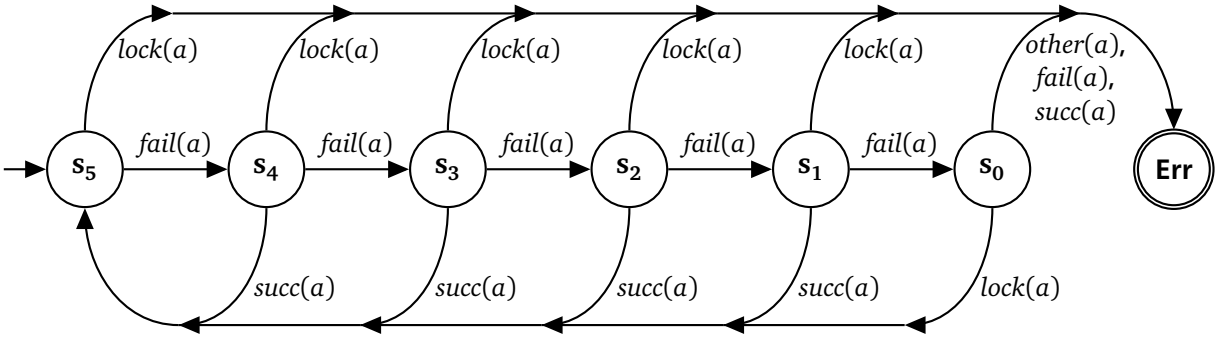
In the formalization of the modified secure coding aspect, we introduce the label $succ(a)$ that describes the execution steps corresponding to a successful login attempt. All labels utilized in this formalization and their description are given in Table 3.5.

Like in Figure 3.4, for the FSM in Figure 3.5 a sequence of five failed login attempts will be counted down and leads from state s_5 to state s_0. In state s_0 locking the account will reset the number of left login tries to 5 and lead back to state s_5. Successful logins in states s_5 to s_1 now reset the number of login tries for an account, which is modeled by transitions into the initial state s_5 labeled with $succ(a)$. As before, if any execution step not associated with locking account a is made in state s_0, a transition is made into the final state *Err* and a violation of the secure coding guideline is detected.

Further modification of the original aspect:

While the original secure coding aspect emphasizes the importance of early locking and increasing lock times to limit the success of password guessing or brute force attacks on the authentication, it does not specify when not to lock. A system that locks an account after each failed or successful login satisfies the original secure coding aspect, as does a system that keeps all accounts locked continuously. Both these systems are secure regarding their login policy, but their usability is questionable.

In order to improve the secure coding aspect in this respect, we add the following requirement to the precise formulation: "Do not lock account a before five login attempts for a failed since its creation or since the last lock." The corresponding formalization in Figure 3.6 introduces five new edges into the FSM: From each of the states s_5 to s_1 an edge labeled with $lock(a)$ now leads into the error state in order to forbid early locking.



$$(S, S_0, L, \rightarrow) \models P''(a,5) \wedge \Box((lock(a) \vee succ(a)) \longrightarrow \bigcirc P''(a,5))$$

$$\text{where } P''(a,0) = lock(a) \text{ and}$$
$$P''(a,n) = (\neg(fail(a) \vee lock(a))) \; \mathcal{U} \; ((fail(a) \wedge \bigcirc P''(a,n-1)) \vee succ(a)) \text{ for } n > 0$$

Figure 3.6: Secure the Login and Authentication Procedures

3.5 Maintain Session Control

This guideline addresses the common problem of impersonation through session hijacking in communications between servers and clients.

Selected secure coding aspect (as in [ACMS10, p. 32 f]):

"All user-related session IDs must be immediately invalidated 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 (as in [ACMS10, p. 32 f]):

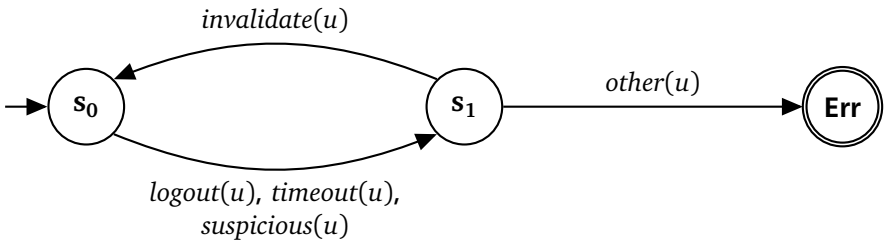
Session IDs must be invalidated immediately under the following circumstances:

- Whenever the user logs out, the session IDs related to that user must be immediately invalidated.

- Whenever 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 $(S, S_0, \rightarrow, L)$ satisfies the selected secure coding aspect, if and only if it satisfies the temporal safety property given by the FSM in Figure 3.7 for each $u \in \mathit{UserIDs}$.



$$(S, S_0, L, \rightarrow) \models \Box((\mathit{logout}(u) \vee \mathit{timeout}(u) \vee \mathit{suspicious}(u)) \longrightarrow \bigcirc \mathit{invalidate}(u))$$

Figure 3.7: Maintain Session Control

The FSM uses the same labels as the LTL formula from [ACMS10, p. 32 f], which is given for comparison purposes below the FSM. The labels are parametric in an element u from the set *UserIDs* that denotes user IDs in a system. A description of the labels is given in Table 3.6.

In the initial state s_0 of the FSM in Figure 3.7, a user u may have valid session IDs and may obtain new ones. Whenever the user logs out or a timeout occurs or suspicious behavior by the user is detected, the state changes to s_1. In this state, all session IDs of user u must be invalidated. The invalidation of user u's session IDs satisfies the requirement of the secure coding aspect, thus a transition is made into state s_0 where the user with ID u may again log in and obtain new session IDs. However, since the invalidation in state s_1 must occur immediately, all other actions taken before a possible invalidation violate the secure coding aspect and lead into the final state *Err*.

Note that the transition labeled with $\mathit{other}(u)$ ensures that there is no implicit self-loop for action $\mathit{other}(u)$ in state s_1 (cf. the analogous construction in Figure 3.4); i.e., the next execution step after a logout, timeout, or detection of suspicious behavior must invalidate the session IDs of user u in order to satisfy the secure coding aspect.

In [ACMS10] it is argued that the requirement of immediate invalidation is too strict, because it is often necessary to perform some operations such as disconnection from bound services or databases before invalidating the session ID. This point will be discussed in the following part with an improved formulation of the aspect.

Improved precise formulation of the selected secure coding aspect (as in [ACMS10, p. 32 f]):

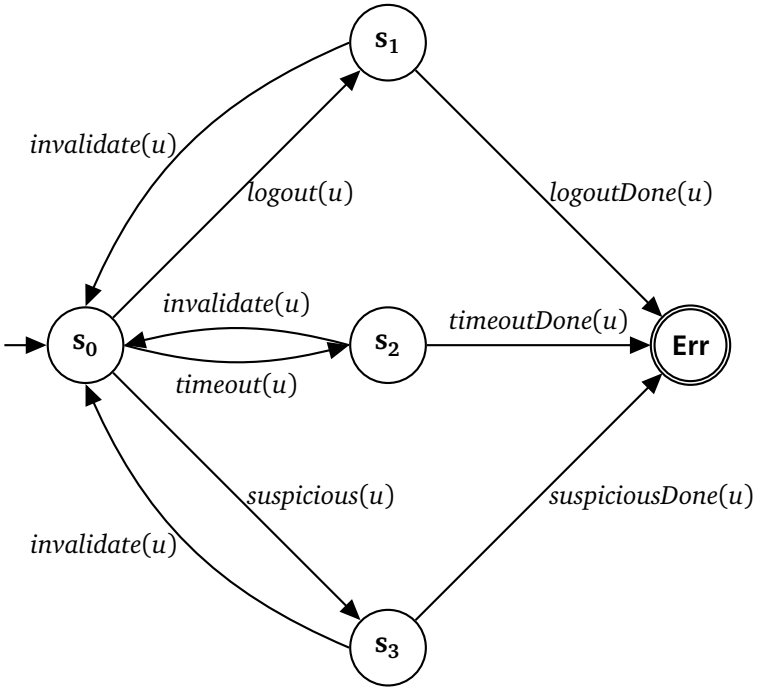
Provide specific methods for handling the logout of a user, the timeout of a session, and the detection of suspicious behavior of a user. These methods shall be called when a user logs out, when a session times out, or when suspicious user behavior is detected, respectively. Within these methods, the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

Label	Description
$logout(u)$	Label indicating that the user with ID u has logged out
$timeout(u)$	Label indicating that the user with ID u has timed out
$suspicious(u)$	Label indicating that suspicious behavior of the user with ID u has been detected
$invalidate(u)$	Label indicating that all session IDs for user ID u are being invalidated
$other(u)$	Label indicating an event that is not related to invalidating session IDs for user ID u

Table 3.6: Labels used in the formalization of the secure coding aspect (as in [ACMS10])

Improved formalization of the selected secure coding aspect:

The labeled transition system $(S, S_0, \rightarrow, L)$ satisfies the selected secure coding aspect, if and only if it satisfies the temporal safety property given by the FSM in Figure 3.8 for each $u \in UserIDs$.



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

Figure 3.8: Maintain Session Control

The invalidation of session IDs shall occur within the method handling logout, timeout and the detection of suspicious behavior, thus it is necessary to introduce additional labels that mark the end of each procedure. A description of the additional labels is given in Table 3.7.

In the initial state s_0 of the FSM in Figure 3.8, an action that requires the session IDs to be invalidated leads into one of the states s_1, s_2, or s_3. Each of these states shows the system during the execution of a logout-, a timeout- or a suspicious behavior handler. If in such a state the session IDs are being

Label	Description
logoutDone(u)	Label indicating that the method handling user u's logout returned from execution
timeoutDone(u)	Label indicating that the method handling user u's timeout returned from execution
suspiciousDone(u)	Label indicating that the method detecting user u's suspicious behavior returned from execution

Table 3.7: Labels used in the formalization of the secure coding aspect

invalidated, a transition is taken back into the initial state where the handler continues its tasks and the secure coding guideline has not been violated for the user in question. If however a handler returns in the states s_1 to s_3, the session IDs have not been invalidated within the handler and the secure coding guideline is violated, leading to the state *Err*.

Difference to the original formalization:

If one of the handlers takes a possibly infinite amount of time to process the detected behavior, our formalization may not detect a violation of the secure coding guideline. The FSM could reside in one of the states s_1 to s_3 for an infinite amount of time, so the state *Err* would never be reached even if the session ID is never invalidated.

Requiring the handlers to terminate is a liveness property, i.e., a property that states that "something good will happen", and cannot be expressed in the FSM formalism. Our formalization focuses on the part of the secure coding aspect that constitutes a safety property: If a handler returns before the concerning session IDs have been invalidated, this will be detected. We leave out the liveness part that states: Whenever a handler is called, it will take only finite time until it returns. The LTL formula in [ACMS10, p. 35] formalizes both the safety and the liveness property by requiring the execution steps associated with the label *invalidate*(u) to be eventually executed, while our LTL formalization does not require this.

4 Secure Coding Guidelines for C

This section presents formalizations for four secure coding guidelines from [Sea08] as FSMs and in LTL.

4.1 CERT C STR31-C

This guideline shall help to prevent certain kinds of buffer overflows in C code.

Secure coding guideline (as in [Sea08, STR31-C]):

"Guarantee that storage for strings has sufficient space for character data and the null terminator."

"Copying data to a buffer that is not large enough to hold that data results in a buffer overflow. While not limited to null-terminated byte strings (NTBS), buffer overflows often occur when manipulating NTBS data. To prevent such errors, limit copies either through truncation or, preferably, ensure that the destination is of sufficient size to hold the character data to be copied and the null-termination character."

Selected secure coding aspect:

"Limit copies either through truncation or, preferably, ensure that the destination is of sufficient size to hold the byte string to be copied and the null-termination character."

Precise formulation of the selected secure coding aspect:

Whenever a string of bytes is to be written into a buffer in memory, either truncate the string or adjust the buffer size such that the string and a null-terminator fit into the buffer.

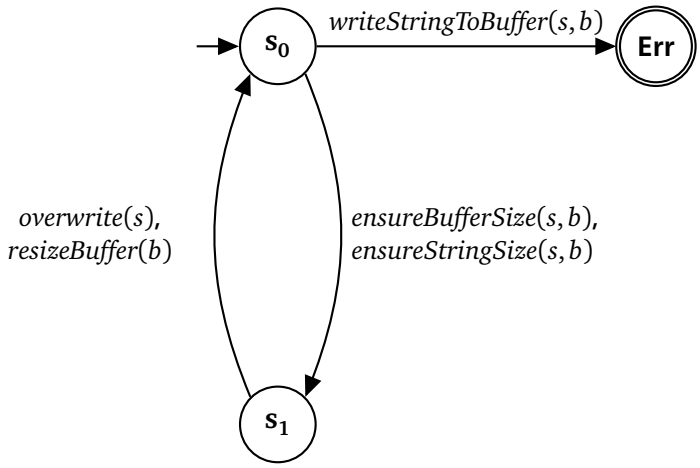
Formalization of the selected secure coding aspect:

The labeled transition system $(S, S_0, \rightarrow, L)$ satisfies the selected secure coding aspect, if and only if it satisfies the temporal safety property given by the FSM in Figure 4.1 for each $s, b \in Mem$.

The selected aspect displays the core property formulated by the guideline. We distinguish two possibilities to limit copies: firstly, ensuring that the string and a null-terminator fit into the fixed size destination buffer (which involves a check and possibly a truncation), and secondly, ensuring that the buffer is large enough for the fixed-size string and a null-terminator (which involves a check and possibly an increment of the buffer size).

The labels for both the FSM and the LTL formula are parametric in elements b and s from the set Mem denoting locations in the main memory. The location s shall point to a null-terminated byte string. A description of the labels is given in Table 4.1.

The FSM in Figure 4.1 formalizes the selected secure coding aspect as follows: In the initial state s_0 two buffers b and s are accessible, where s contains a null-terminated byte string. In this state no checks have been performed and it is unclear if the string in s fits into b. If in this state the string in s is copied into b as described by the label $writeStringToBuffer(s, b)$, the string might be truncated unintentionally or might cause an overflow of buffer b, thus a potential violation of the secure coding guideline STR31-C is signaled with a transition into the state **Err**. Whenever in state s_0 the size of buffer b is checked, the size of the string in s is checked, and the presence of enough space is ensured either by intentional truncation of the string ($ensureStringSize(s, b)$) or by resizing the buffer ($ensureBufferSize(s, b)$), a transition is made into state s_1. In state s_1, copying the string is allowed. If s is overwritten with an arbitrary null-terminated



$$(S, S_0, L, \rightarrow) \models P(s,b) \wedge \square((overwrite(s) \vee resizeBuffer(b)) \longrightarrow P(s,b))$$

$$\text{where } P(s,b) = ((\neg writeStringToBuffer(s,b)) \; \mathcal{U} \; (ensureBufferSize(s,b) \vee ensureStringSize(s,b)))$$

Figure 4.1: STR31-C

byte string or if b is resized, then copying the string is forbidden again, which is formalized by a transition into state s_0.

The LTL formula $P(s,b)$ states that the string in buffer s must not be written into buffer b unless it ensured that b is large enough to hold the string and a null-terminator or that the string to be written is short enough to fit into b including a null-terminator. Thus the whole formula for the aspect states that from the beginning on, no string must be written into a buffer without ensuring that the size of the buffer suffices; the size of the buffer needs to be checked again before writing the string into b whenever the string or the size of the buffer has changed.

Application scenario:

Consider the compliant code example attached to the secure coding guideline in Listing 4.1. The procedure `main` copies the program name from its argument array using the `strcpy()` function. This function copies a string including its null-terminator into a target buffer regardless of the target buffer's size.

Label	Description
ensureBufferSize(s,b)	Label indicating that buffer b's size is compared to byte string s's size and adjusted, if necessary, in order to be large enough to hold s and a null-terminator
ensureStringSize(s,b)	Label indicating that byte string s's size is compared to buffer b's size and that s is trimmed, if necessary, in order to be small enough to fit into b together with a null-terminator
overwrite(s)	Label indicating that s is overwritten, unless this modification of s is done to trim s, cf. *ensureStringSize*(s,b)
resizeBuffer(b)	Label indicating that buffer b is being resized without changing its memory location, unless resizing is done to accommodate string s, cf. *ensureBufferSize*(s,b)
writeStringToBuffer(s,b)	Label indicating that s is written into buffer b

Table 4.1: Labels used in the formalization of the secure coding aspect

```
1  int main(int argc, char *argv[]) {
2    const char* const name = argv[0] ? argv[0] : "";
3    char *prog_name = (char *)malloc(strlen(name) + 1);
4    if (prog_name != NULL) {
5      strcpy(prog_name, name);
6    }
7    else {
8      /* Failed to allocate memory - recover */
9    }
10   /* ... */
11 }
```

Listing 4.1: Code example for STR31-C (as in [Sea08])

The target buffer `*prog_name` is allocated with enough space for the byte string in `argv[0]` and a null-terminator in line 3. When in line 5 `strcpy()` is called, neither a buffer overflow nor an unintentional truncation of the string occurs.

In line 3, the buffer is sized appropriately for the string, which would be associated with the label *ensureBufferSize*(*argv*[*0*],*prog_name*). Line 5 is the copy process and thus associated with the label *writeStringToBuffer*(*argv*[*0*],*prog_name*). Regarding the FSM in Figure 4.1, during lines 1 and 2, the system resides in state s_0. As soon as line 3 is executed, a transition is made into state s_1, and further copying of the string from `argv[0]` to `prog_name` is allowed. Thus the example code complies with the property formalized by the FSM.

4.2 CERT C STR32-C

Strings in C can be regarded as a special case of arrays in memory. The previous guideline focuses on writing beyond the buffer's bounds, whereas STR32-C focuses on reading beyond the bounds of a string, which might cause an error or an information leak.

Secure coding guideline (as in [Sea08, STR32-C]):

"Null-terminate byte strings as required"

"Null-terminated byte strings (NTBS) must contain a null-termination character at or before the address of the last element of the array before they can be safely passed as arguments to standard string-handling functions, such as strcpy() or strlen(). This is because these functions, as well as other string-handling functions defined by C99 [ISO/IEC 9899:1999], depend on the existence of a null-termination character to determine the length of a string. Similarly, NTBS must be null-terminated before iterating on a character array where the termination condition of the loop depends on the existence of a null-termination character within the memory allocated for the string."

Selected secure coding aspect:

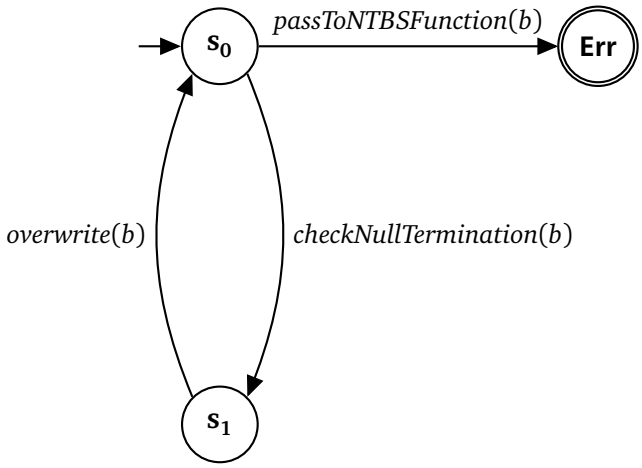
"Null-terminated byte strings (NTBS) must contain a null-termination character at or before the address of the last element of the array before they can be safely passed as arguments to standard string-handling functions, such as `strcpy()` or `strlen()`."

Precise formulation of the selected secure coding aspect:

A reference to a buffer must not be passed to a string-handling function that relies on the content of this buffer to be a null-terminated byte string, unless it is ensured that the content of the buffer is null-terminated within the buffer's bounds.

Formalization of the selected secure coding aspect:

The labeled transition system $(S, S_0, \rightarrow, L)$ satisfies the selected secure coding aspect, if and only if it satisfies the temporal safety property given by the FSM in Figure 4.2 for each $b \in Mem$.



$$(S, S_0, L, \rightarrow) \models P(b) \wedge \square(overwrite(b) \longrightarrow P(b))$$

$$\text{where } P(b) = ((\neg passToNTBSFunction(b)) \mathcal{U} checkNullTermination(b))$$

Figure 4.2: STR32-C

The labels for both the FSM and the LTL formula are parametric in an element b from the set *Mem* denoting a location in the main memory. The buffer b may contain arbitrary character data and is not known to be null-terminated. A description of the labels is given in Table 4.2.

The FSM in Figure 4.2 describes the secure coding aspect formally. In the initial state s_0, a buffer b for byte strings is accessible. When b's content is ensured to be null-terminated, a transition is taken to state s_1, where passing b to a string-handling function is allowed. If the content of b is at least partially overwritten, a transition is taken to state s_0. If b is passed to a string-handling function without ensuring that b is null-terminated beforehand, a transition is taken to state *Err*, because the secure coding guideline is violated in this case.

The LTL formula $P(b)$ describes that a reference to the buffer b must not be passed to a string-handling function that relies on its parameter to be null-terminated unless it is ensured that the string in b is actually null-terminated within the buffer's bounds. The whole formula for the aspect states that whenever the content of buffer b is at least partially overwritten, its null-termination has to be ensured before b to a string-handling function that relies on the null-termination.

Application scenario:

Consider the compliant code example attached to the secure coding guideline in Listing 4.2. The given code copies the string from buffer `source` into buffer `ntbs` using the function `strcpy()`. In line 1, a string is written into newly allocated memory. Such an assignment ensures the null-termination of the character sequence within the array bounds and thus can be annotated with the label *checkNullTermination*(*source*). In line 4 a check occurs if the allocation proceeded without errors. A comparison between the size of the string in buffer `source` and the size of the target buffer is done using the function `strlen()` to calculate the string-length. The behavior of this call depends on the null-termination property of its argument, i.e., `source` and thus can be labeled with *passToNTBSFunction*(*source*). The call to `strcpy()`

Label	Description
checkNullTermination(b)	Label indicating that the byte string in buffer b is verified to be null-terminated within the buffer bounds
passToNTBSFunction(b)	Label indicating that a reference to buffer b is passed to a string handling function that relies on the buffer to contain a null-terminated byte string
overwrite(b)	Label indicating that the content of buffer b is at least partially overwritten, possibly affecting the null-termination property of the content

Table 4.2: Labels used in the formalization of the secure coding aspect

```
1  char *source = "0123456789abcdef";
2  char ntbs[NTBS_SIZE];
3  /* ... */
4  if (source) {
5    if (strlen(source) < sizeof(ntbs)) {
6      strcpy(ntbs, source);
7    }
8    else {
9      /* handle string too large condition */
10   }
11 }
12 else {
13   /* handle NULL string condition */
14 }
```

Listing 4.2: Code example for STR32-C (as in [Sea08])

in line 6 depends on the null-termination of its second parameter and thus can also be labeled with *passToNTBSFunction*(*source*).

If the FSM is in the initial state s_0 before executing line 1, line 1 then triggers a transition into state s_1. The call to `strlen()` in line 5 as well as the possibly following call to `strcpy()` are allowed. However, the left-out code in line 3 (denoted by `/* ... */`) could possibly contain steps that could be labeled with *overwrite*(*source*), which would then trigger a transition back into state s_0 prior to line 5. In this case, the execution of line 5 would violate the secure coding aspect. We regard this example as conditionally compliant, depending on what is abstracted by the placeholder in line 3.

4.3 CERT C ENV31-C

The guideline ENV31-C addresses a problem specific to pointers in C. The process environment (containing variables and assigned values) is accessible through pointers. Since pointers point to static addresses in memory and on some platforms the process environment is suspect to relocation, these pointers need to be validated before the environment is accessed through them.

Secure coding guideline (as in [Sea08, ENV31-C]):

"Do not rely on an environment pointer following an operation that may invalidate it."

"Some environments provide environment pointers that are valid when `main()` is called, but may be invalided by operations that modify the environment. [...] modifying the environment by any means may cause the environment memory to be reallocated with the result that [certain environment pointers] now reference an incorrect location. "

Selected secure coding aspect:

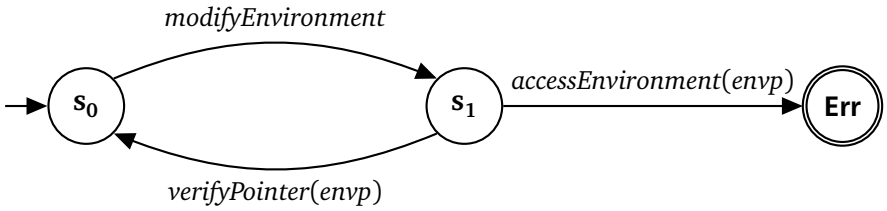
"Do not rely on an environment pointer following an operation that may invalidate it. [...] modifying the environment by any means may cause the environment memory to be reallocated with the result that [certain environment pointers] now reference an incorrect location."

Precise formulation of the selected secure coding aspect:

Whenever the process environment has been modified, verify that a pointer that so far directly referenced the environment still points to the current environment before accessing the environment via the pointer.

Formalization of the selected secure coding aspect:

The labeled transition system $(S, S_0, \rightarrow, L)$ satisfies the selected secure coding aspect, if and only if it satisfies the temporal safety property given by the FSM in Figure 4.3 for each direct environment pointer $envp \in Mem$.



$$(S, S_0, L, \rightarrow) \models \square(modifyEnvironment \longrightarrow (\neg accessEnvironment(envp)\ \mathcal{U}\ verifyPointer(envp)))$$

Figure 4.3: ENV31-C

The labels for both the FSM and the LTL formula are parametric in an element *envp* from the set *Mem* denoting locations in the main memory, where *envp* is a direct pointer to the process environment. A description of the labels is given in Table 4.3.

The FSM in Figure 4.3 formalizes the selected secure coding aspect as follows: When in state s_0 the environment is modified, the FSM changes to state s_1. In state s_1 a direct environment pointer *envp* must be verified to point to the current environment prior to accessing the environment through it. An access to the environment in state s_1 constitutes a violation of the secure coding guideline, thus the state is changed to state *Err*. If in state s_1 the pointer *envp* is verified as described, a transition is taken back into state s_1, where accessing the environment through *envp* is allowed.

The LTL formula states that whenever the environment is modified, it must not be accessed using pointer *envp* unless *envp* is verified to point to the current environment.

Application scenario:

Consider the code example in Listing 4.3. In line 1 a pointer is directed to the current environment via a call to `getenv()`. After checking the pointer in line 2, in line 6 the value of the variable is changed with a call to `setenv()`. Depending on the size of the former value, this call may have caused the environment to be relocated. In line 10 the old environment is accessed using a pointer acquired before the change and thus possibly referencing invalid memory.

According to our definitions of the labels, lines 1 and 10 have to be annotated with the label *accessEnvironment*(*envp*). Line 6 may cause a modification of the environment and thus is labeled with *modifyEnvironment*. The modification in line 6 is done without providing an explicit pointer, so

Label	Description
modifyEnvironment	Label indicating that the environment is modified and thus possibly relocated
accessEnvironment(*envp*)	Label indicating that the environment copy referenced by *envp* is accessed.
verifyPointer(*envp*)	Label indicating that *envp* is determined to point to the current environment.

Table 4.3: Labels used in the formalization of the secure coding aspect

```
1  char* envp = getenv(varname);
2  if(!envp){
3    /* Handle error */
4  }
5  /* ... */
6  if(setenv(varname, "new_value", 1) != 0) {
7    /* Handle error */
8  }
9  if(envp){
10   printf("new value is: %s", envp);
11 }
```

Listing 4.3: Code example for CERT C ENV31-C

line 6 need not be annotated with the label *accessEnvironment*(*envp*). In order to make the code comply to the guideline, the reference could be updated using `getenv(varname)` right before line 9.

4.4 CERT C MEM32-C

During a program run, memory gets allocated and deallocated. Since memory allocation may fail, guideline MEM32-C addresses two common errors in memory allocation: null-pointer dereference and memory-leaks.

Secure coding guideline (as in [Sea08, MEM32-C]):

"Detect and handle memory allocation errors."

"The return values for memory allocation routines indicate the failure or success of the allocation. According to C99, calloc(), malloc(), and realloc() return null pointers if the requested memory allocation fails. [...] Failure to detect and properly handle memory allocation errors can lead to unpredictable and unintended program behavior. As a result, it is necessary to check the final status of memory management routines and handle errors appropriately [...]."

Selected secure coding aspect:

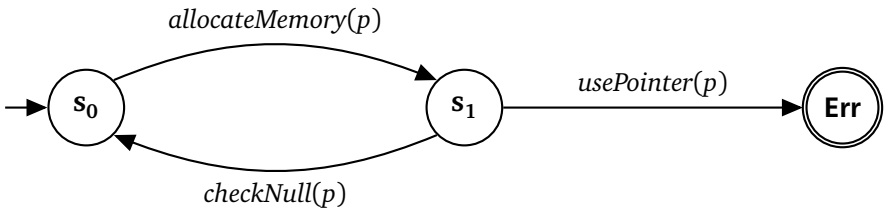
"Detect and handle memory allocation errors from calloc(), malloc(), and realloc()."

Precise formulation of the selected secure coding aspect:

When allocating memory with `calloc()`, `malloc()`, or `realloc()`, ensure that the pointer returned by the functions is not null before dereferencing it.

Formalization of the selected secure coding aspect:

The labeled transition system $(S, S_0, \rightarrow, L)$ satisfies the selected secure coding aspect, if and only if it satisfies the temporal safety property given by the FSM in Figure 4.4 for each pointer $p \in Mem$.



$$(S, S_0, L, \rightarrow) \models \square(allocateMemory(p) \longrightarrow (\neg usePointer(p)\ \mathcal{U}\ checkNull(p)))$$

Figure 4.4: MEM32-C

Label	Description
allocateMemory(p)	Label indicating that a memory allocation function is called and its result is assigned to pointer p
usePointer(p)	Label indicating that the pointer p is used; for example, in an address calculation or by dereferencing it. Null-pointer checks for p must not be annotated with this label.
checkNull(p)	Label indicating that p is checked and found not to be a null-pointer.

Table 4.4: Labels used in the formalization of the secure coding aspect

The labels for both the FSM and the LTL formula in Figure 4.4 are parametric in an element p from the set *Mem* denoting locations in the main memory, where p is a pointer. A description of the labels is given in Table 4.4.

The FSM in Figure 4.4 formalizes the selected secure coding aspect as follows: When in state s_0 a call to a memory allocation function occurs, a transition is made into state s_1. If in this state the returned pointer is used (e.g., dereferenced), a transition is made into the state *Err*, because the secure coding guideline is violated in this case. If in state s_1 a null-check is performed on the pointer finding that the allocation has been successful, a transition back into state s_0 is triggered, where using the pointer is allowed.

The LTL formula states that whenever a memory allocation function is called which can return a null-pointer, the return value has to be checked before use.

Application scenario:

Consider the "Noncompliant Solution (`malloc()`)" attached to the secure coding guideline in Listing 4.4. The function `f()` copies a string provided as a parameter into a newly allocated buffer in memory. In line 2 the size of the target buffer is calculated and the buffer memory is allocated in line 3. Line 3 needs to be labeled with *allocateMemory*(*str*). In line 4, the returned pointer is checked against null, so line 4 needs to be labeled with *checkNull*(*str*). If the allocation has been successful, the control flow will reach line 8, a call to `strcpy()`, to which label *usePointer*(*str*) applies. The same label applies to the call of `free()` in line 10.

If the FSM is in the initial state s_0 when reaching line 1, line 3 will trigger a transition into state s_1. Line 4 is labeled with *checkNull*(*str*) and thus triggers a transition back into state s_0, where the call to `strcpy()` in line 8 as well as the subsequent call to `free()` in line 10 are allowed. Depending on what happens in line 9, this code example can be regarded as complying to the guideline. If in line 9 *str* is assigned the return value of a `realloc()` call, for example, line 10 will be executed in state s_1, which leads to a transition to state *Err*.

```
1  int f(char *input_string) {
2    size_t size = strlen(input_string) + 1;
3    char *str = (char *)malloc(size);
4    if (str == NULL) {
5      /* Handle allocation failure and return error status */
6      return -1;
7    }
8    strcpy(str, input_string);
9    /* ... */
10   free(str);
11   return 0;
12 }
```

Listing 4.4: Code example for CERT C MEM32-C (as in [Sea08])

5 Conclusion

In the preceding sections, we presented formalizations of secure coding guidelines both as finite state machines (FSMs) and in Linear Temporal Logic (LTL). In the first three formalizations (Validate User Input, Sanitize the Output, and Secure the Internal Flow), both the FSMs and the LTL formulas are quite small; in particular, each of the FSMs has only three states. Thus the formalizations should be relatively easy to grasp.

In the fourth formalization (Secure the Login and Authentication Procedures), the FSM has seven states. While we believe that this size is still manageable, it already costs more effort to grasp the FSM compared to the small FSMs from the first three formalizations. In particular, the size of the FSM increases linearly with the number of login attempts that is permitted by the guideline, which is a drawback of this formalization. The corresponding LTL formula is defined recursively with the advantage that this leads to a constant size of the LTL formalization, i.e., the size of the formalization is independent of the number of permitted login attempts. Moreover, the original formalization in LTL from [ACMS10] has the advantage of also including concrete values how long an account shall be locked; these values are missing in our FSM formalization, because the values may increase arbitrarily, which cannot be captured concretely with finitely many states.

In the fifth formalization (Maintain Session Control), the formalization as an FSM only expresses the safety part of the property, while in LTL, it is also possible to express the liveness part (termination of methods). It is debatable whether the restriction to a safety property is significant in this case: For server applications that handle requests from clients, it may be worthwhile to prove termination of all handlers anyway, so formalizing termination for the specific handlers in this guideline might not be necessary.

The formalizations of the guidelines for C are of a similar nature as the first three formalizations for Java: All of them are relatively small (three states in the FSM, and one or two lines for the LTL formulas), and the properties captured by the FSMs are equivalent to the properties captured by the respective LTL formulas.

Our subjective impression is that formalizations as FSMs are generally easier to grasp if there are only few states. This is because the required temporal relation between program actions can be inferred by just following the arrows in the graphical notation. For the LTL formula, knowledge about the temporal operators is required to understand the formula. If the FSM has many states with a regular structure, however, then LTL formulas become at least competitive, because the regular structure can be exhibited by standard techniques from mathematics (e.g., recursion), whereas the FSMs might become unwieldy due to an increasing number of states.

Overall, we did not find strong reasons that would favor one of the formalisms over the other one in the context of secure coding. Subjectively, FSMs seem more convenient if the properties are safety properties and can be expressed with few states. If one chooses FSMs as formalism with the convention of leaving self-loops implicit, we strongly encourage checking *for each state* whether self-loops are justified or whether they need to be excluded for some states. In the latter case, explicit transitions (typically to an error state) need to be added as in our formalizations in Sections 3.4 and 3.5.

In order to make formalizations in FSMs more amenable, our examples suggest that an extended notion of states would be desirable. For instance, in addition to the finite number of states, it would be helpful to maintain additional state information such as the number of left login attempts or the (arbitrarily large) number of time units that an account needs to remain locked. This would help to reduce the number of states in FSMs and it would increase the expressiveness of the formalism.

Acknowledgments

This work was partially funded by the DFG (German Research Foundation) within the Computer Science Action Program in the project FM-SecEng (MA 3326/1-2 and MA 3326/1-3).

Bibliography

[ACMS10] M. Aderhold, J. Cuéllar, H. Mantel, and H. Sudbrock. Exemplary Formalization of Secure Coding Guidelines. Technical report TUD-CS-2010-0060, Technische Universität Darmstadt, Germany, 2010.

[CDW04] Hao Chen, Drew Dean, and David Wagner. Model Checking One Million Lines of C Code. In *NDSS*, 2004.

[HU79] John E. Hopcroft and Jeffry D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[ISO99] ISO. ISO C Standard 1999. Technical report, 1999. ISO/IEC 9899:1999 draft.

[LMS+11] Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland, and David Svoboda. *The CERT Oracle Secure Coding Standard for Java*. Addison-Wesley, 2011.

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

[TYHD09] Syrine Tlili, Xiaochun Yang, Rachid Hadjidj, and Mourad Debbabi. Verification of CERT Secure Coding Rules: Case Studies. In *OTM Conferences (2)*, pages 913–930, 2009.