Specification-based Test Generation for Security-Critical Systems Using Mutations *

Guido Wimmel and Jan Jürjens

Department of Computer Science, Munich University of Technology
Arcisstrasse 21, D-80290 München, Germany
phone: +49-89-28928362 fax: +49-89-28925310
wimmel|juerjens@in.tum.de

Abstract. In specification-based testing, test sequences are generated from an abstract system specification to provide confidence in the correctness of an implementation. For security-critical systems, finding tests likely to detect possible vulnerabilities is particularly difficult, as they usually involve subtle and complex execution scenarios and the consideration of domain-specific concepts such as cryptography and random numbers.
We present research aiming to generate test sequences for transaction systems from a formal security model supported by the CASE tool AutoFocus. The test sequences are determined with respect to the system's required security properties, using mutations of the system specification and attack scenarios. To be able to apply them to an existing implementation, the abstract test sequences are concretized.

Keywords. Test Case Generation, E-Commerce Systems, Security-Critical Systems, Formal Methods, Test Specification, Validation and Analysis, Test Tools, Computer-aided Software Engineering (CASE), AutoFocus.

1 Introduction

Security aspects are playing an increasingly important role in the development of distributed systems. However, developing security-critical systems is very difficult. Although undetected vulnerabilities can cause enormous damage, often security-related requirements are only formulated imprecisely and considered in the development in an ad-hoc manner. Controlling the quality of such applications is a hard problem, and in many cases vulnerabilities are found after the system has been put into operation.

Formal models can contribute to achieving high confidence in a system's security. A large number of modelling and verification approaches has been proposed (see [GSG99] for an overview), ranging from specifications in general languages such as CSP or TLA to protocol-specific formalisms such as CASPER (a protocol definition language that can be translated to CSP) or BAN (a logic modelling the actions and beliefs of the parties during execution of the protocol).

Motivated by this, our work presented in [WW01,JW01a] aims to integrate security aspects into systems development, based on formal models reflecting the security requirements and threat scenarios. Important concepts are tool support and use of description techniques

* This work was partially supported by the German Ministry of Economics within the FairPay project.

understandable for non-expert software engineers. The formal models can later be used for certification purposes (e.g. based on the Common Criteria for Security Evaluation), leading to a high assurance with respect to the application's quality.

However, in general the actual implementation is much more complex. To allow proofs of security properties, abstraction techniques are used: in models of cryptographic transactions, messages, keys and random numbers are usually represented by abstract data entities which can be arguments to abstract operations such as encryption or hashing, and part of the actual messages exchanged may have been left out. Besides, as the security model is usually developed independently of the implementation (mostly, after the implementation, though this is not desirable), it cannot be concluded from the correctness of a security model that the implementation is secure.

Confidence in the correctness of an implementation can be gained by extensive testing. Testing for security holes is usually restricted to penetration testing (a so-called "tiger-team" of experts manually tries to break the system or tools such as SATAN are used to search for known vulnerabilities). This approach is not satisfactory as it depends largely on the skill of the employed tiger team or the knowledge encoded into the tool, which does not consider application-specific security requirements.

In this paper, we show how to complement this approach by generating test sequences from a security specification. The aim is to find those test sequences that are most likely to detect possible vulnerabilities. For this purpose, we adapt methods from classical specification-based testing to the application domain of security-critical systems. Specifically, we include domain-specific concepts such as cryptography, knowledge or access to secrets, and threat scenarios. Test sequences likely to detect vulnerabilities are computed using mutations of the specification that lead to violation of the security requirements.

Besides, we show how to translate the abstract test sequences derived from the security model to concrete test sequences that can be applied to an existing implementation. The approach is demonstrated at the example of a part of the Common Electronic Purse Specifications (CEPS), a proposed global standard for purse cards.

Providing a possibility to perform security testing within a general CASE tool is an important feature of our work, since it lowers the threshold for considering security in the development process at all. Also, specification-based testing may be a promising way of introducing formal methods into the industrial development context, as it has been argued before (for security-critical systems, e.g. in [FBGL94]). The work presented here builds on experience for example from using specification-based test sequences in firewall testing [JW01b].

This paper is organized as follows. In Section 2 we introduce the tool AutoFocus and show how to specify security-critical systems using an extension of AutoFocus models. In Section 3 we describe our approach of generating security related test sequences from such specifications and their concretization to implementation test sequences. Our case study CEPS is presented in Section 4. We end with references to related work (Section 5) and conclude in Section 6.

2 Security Models in AutoFocus

To model and test security-critical systems, we use the tool AutoFocus. AutoFocus is a CASE tool for graphically specifying distributed systems. It is based on the formal method Focus [BS01], and its models have a simple, formally defined semantics. AutoFocus offers

standard, easy-to-use description techniques for an end-user who does not necessarily need to be a formal methods expert, as well as state-of-the-art techniques for validation and verification. It features simulation, code generation, test sequence generation and formal verification of the modelled systems.

Systems are specified in AutoFocus using static and dynamic views, which are conceptually similar to those offered in UML-RT. To be able to model security-critical systems, we included security aspects into the AutoFocus description techniques. In this section, we briefly explain the extensions together with an abstract syntax for the relevant subset of AutoFocus (excluding hierarchy). More motivation and examples on security modelling are given in a less formal way in [WW01,JW01a].

System Structure Diagrams and Attack Scenarios AutoFocus System Structure Diagrams (SSDs) are similar to UML component diagrams and describe the structure and interfaces of a system. In the SSD view, a system consists of a number of communicating components, which have input and output ports for sending and receiving messages. The ports are connected via directed channels.

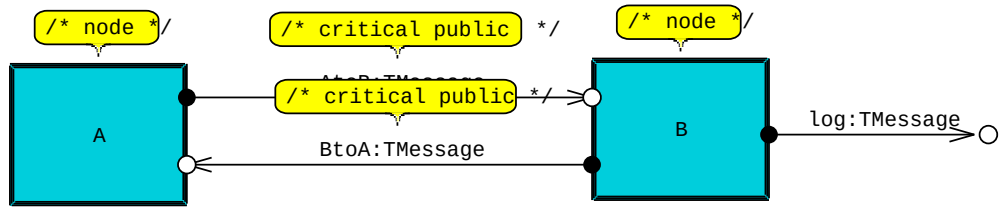
In the abstract syntax, an SSD is a triple $(\mathsf{Comp}, \mathsf{Ports}, \mathsf{Channels})$, where ports are associated to components by $\mathsf{comp}(p) \in \mathsf{Comp}$, and channels are associated to source and destination ports by $\mathsf{sourceP}(c) \in \mathsf{Ports}$ and $\mathsf{destP}(c) \in \mathsf{Ports}$. A component c can have local variables $v \in \mathbf{lVar}$ (where $\mathbf{lVar}$ is a set of variable names) such that $\mathsf{comp}(v) = c$.

For security-critical systems, we have to assume the presence of an attacker trying to exploit security holes. The capabilities of the attacker are defined using threat scenarios incorporated into the specification, which should result from a risk analysis. For this purpose, we define a set $\mathsf{SecAttr} = \{\mathsf{critical}, \mathsf{public}, \mathsf{replace}, \mathsf{node}\}$ of security attributes, and a mapping $\mathsf{secattrSSD} : (\mathsf{Comp} \cup \mathsf{Channels}) \to \mathcal{P}(\mathsf{SecAttr})$ associating a set of security attributes to any component or channel (here $\mathcal{P}(X)$ is the set of subsets of X). Based on the security attributes, the relevant threat scenarios for the system can be generated automatically by AutoFocus.

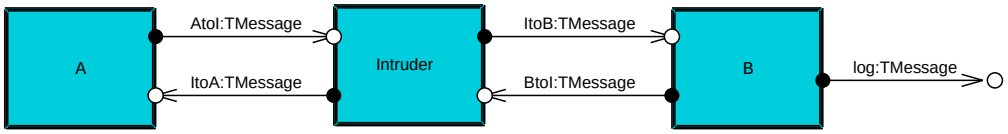
The meaning of the security attributes is as follows:

- If $\mathsf{critical} \in \mathsf{secattrSSD}(c)$ for a component or a channel c, security-critical information is processed in the component or transmitted via the channel. This is used as an indication that security verification and testing must focus on these parts of the model.
- If $\mathsf{public} \in \mathsf{secattrSSD}(ch)$, then ch is a channel whose messages can be accessed and manipulated by the attacker. The corresponding threat scenario is obtained by replacing ch by a set of additional channels such that the communication on ch passes through a component "Intruder".
- If $\mathsf{public} \in \mathsf{secattrSSD}(c)$ for a component c, c can be replaced by an internal attacker having access to all secrets contained in c (e.g., the attacker can access and manipulate the program)
- If $\mathsf{replace} \in \mathsf{secattrSSD}(c)$ for a component c, c can be replaced by an attacker not knowing the secrets of c (e.g., the attacker tries to simulate the behaviour of c without having access to it)
- If $\mathsf{node} \in \mathsf{secattrSSD}(c)$ for a component c, then c is an encapsulated component, to whose internals an attacker has no access.

Fig. 1 (a),(b) shows an SSD with security attributes and the corresponding threat scenario automatically generated by AutoFocus. Note that the behaviour of the Intruder



(a)



(b)

Fig. 1. SSD with Security Attributes and Threat Scenario

component must also be specified, by an AutoFocus State Transition Diagram or a constraint logic program modelling its possible actions (e.g., forwarding, storing, inhibiting and faking messages).

Data Types and Cryptography AutoFocus offers hierarchical data types that are defined using the functional language Quest [PS99], which is similar to Haskell [Tho99]. For models of security-critical transactions, data types for keys, messages and cryptographic operations are provided, as follows:

```
data TKey = EmptyKey | K1 | K2 | ... | Kn;
data TMessage = Empty | Msg1(...) | ... | Msgn(...)
  | Encr(getEncKey:TKey, getEncMsg:TMessage)
  | Mac(getMacKey:TKey,getMacMsg: TMessage)
  | Hash(getHashMsg:TMessage) | Key(TKey);
fun verifyMac(k,Mac(k1,msg),msg1) = ((k == k1) && (msg == msg1))
 |  verifyMac(x,y,z) = False;
```

{K1,...,Kn} in the definition of type `TKey` is the set of key names, and the `Msg1`, ..., `Msgn` stand for possible messages types used in the protocol (for example, `Init(getInitM:Int)` for an initialization message m taking one integer parameter (the amount) that can be accessed via the expression $\mathsf{getInitM}(m)$. `Encr`, `Mac` and `Hash` represent encryption, computation of a message authentication code (MAC) and cryptographic hashing, and `verifyMac` is a function definition representing MAC verification.

For each component c it is specified by $\mathsf{knows}(c)$ which set of keys c currently "knows" (initially, this is determined by the threat scenario) and by $\mathsf{access}(c)$ which set of keys it is allowed to know. If one denotes by $\mathsf{uses}(c)$ the set of keys syntactically contained in the behavioural specification of c, then $\mathsf{uses}(c) \subseteq \mathsf{knows}(c) \subseteq \mathsf{access}(c)$ is a necessary consistency condition for a secure system specification that can be checked automatically by AutoFocus. Key ownership and access is tied to the components because in the AutoFocus SSDs (e.g. in the CEPS model) they represent the concrete instances of the system entities. In dynamic models, this mapping can be realized using a role concept.

Behavioural Specification - State Transition Diagrams Traditionally, techniques similar to message sequence charts are used to specify security protocols. However, to specify complex security-critical transaction systems with many possible interactions, such as CEPS, we consider automata more suitable, as (1) their semantics is better understood, so the behaviour can be modelled more precisely, and (2) the *complete* behaviour of a participant of the protocol can be specified — explicitely including security checks on the incoming messages and failure behaviour, which would otherwise lead to a multitude of possible communication scenarios.

Automata are represented in AUTOFOCUS by State Transition Diagrams (STDs), which are similar to a simplified fragment of Harel's statecharts. Formally, an STD is a pair $(\mathsf{States}, \mathsf{Transitions})$. Additionally, we fix a set **iVar** of transition-local input variables and denote with **Exp** the set of expressions in the Quest functional language, which can in particular contain the input and component variables.

Each transition $t \in \mathsf{Transitions}$ is associated with

- source and target state $\mathsf{source}(t)$,$\mathsf{target}(t) \in \mathsf{States}$,
- $\mathsf{pre}(t) \in \mathbf{Exp}$, where t is a boolean expression, the *precondition* for firing t,
- $\mathsf{inp}(t)$, a finite sequence of pairs (p, x), the *input expressions*, where $p \in \mathsf{Ports}$ and $x \in \mathbf{iVar}$ [1],
- $\mathsf{outp}(t)$, a finite sequence of pairs (p, d), the *output expressions*, where $p \in \mathsf{Ports}$ is a port and $d \in \mathbf{Exp}$ an expression output to p when t is fired, and
- $\mathsf{post}(t)$, a finite sequence of pairs (v, d) where $v \in \mathbf{lVar}$ and $d \in \mathbf{Exp}$, the *postcondition* that assigns d to v when t is fired.

We assume that for each component C in an SSD, there is an STD D that defines its behaviour. In the concrete syntax of STDs, the transitions are annotated with "$\mathsf{pre}(t) : \mathsf{inp}(t) : \mathsf{outp}(t) : \mathsf{post}(t)$", where $\mathsf{inp}(t)$ is denoted as "$inp_1?x_1; inp_2?x_2; ...$", $\mathsf{outp}(t)$ as "$out_1!term_1; out_2!term_2; \ldots$" and $\mathsf{post}(t)$ as "$lvar_1 = term_1; lvar_2 = term_2; \ldots$".

As for the SSDs, we associate security attributes to the states and transitions of an STD by defining a mapping $\mathsf{secattrSTD} : \mathsf{States} \cup \mathsf{Transitions} \to \mathcal{P}(\mathsf{SecAttr})$:

- If $\mathsf{critical} \in \mathsf{secattrSTD}(tr)$, where $tr \in \mathsf{Transitions}$, tr is considered security-critical. A faulty implementation of tr can lead to violations of security properties, making such transitions the focus of test sequence generation.
- If $\mathsf{critical} \in \mathsf{secattrSTD}(s)$, where $s \in \mathsf{States}$, then computations where state s is reached are considered security-critical.

The other security attributes have no meaning for states and transitions. In particular, we do not mark states or transitions with public, assuming that an intruder can either manipulate the complete behaviour of a component or not manipulate it at all.

Extended Event Traces Extended Event Traces (EETs) in AUTOFOCUS represent system runs, similarly to message sequence charts [ITU96]. We will use EETs to represent test sequences showing the communication behaviour for a test scenario.

[1] In general, input expressions can also use pattern matching, defined as an abbreviation for an extended precondition and a substitution for the pattern variables.

Security Requirements In addition to the threat scenario, the security requirements have to be stated. Security requirements will be formulated as first order predicates over execution sequences. Alternatively, formulas in temporal logic can be used, as they can be converted to such predicates. Formally, an execution sequence σ is a valuation of the ports, local variables and components for each execution step: $\sigma : (\mathsf{Ports} \cup \mathbf{lVar} \cup \mathsf{Comp}) \times \mathbb{N} \to \mathbf{Exp}$. For a component c, $\sigma(c,t)$ evaluates to its current control state. With $\sigma(x)$, we denote the restriction of σ to the sequence of valuations of x. Along the lines of [WLPS00], the semantics of an AutoFocus system model can be given as a predicate Ψ, such that $\Psi(\sigma) = \mathit{true}$ if and only if σ is a valid execution sequence of the system.

Security requirements are often defined on complex messages making use of encryption. Similarly to [Pau98], we define $\mathsf{parts} : \mathbf{Exp} \to \mathcal{P}(\mathbf{Exp})$ to be set of all subexpressions that can be obtained by decomposing an expression into its parts; $\mathsf{synth} : \mathcal{P}(\mathbf{Exp}) \to \mathcal{P}(\mathbf{Exp})$ to map a set M of messages to the set of messages given by arbitrarily combining the messages in M; and $\mathsf{analz} : \mathbf{Exp} \times \mathsf{Comp} \to \mathcal{P}(\mathbf{Exp})$, such that $\mathsf{analz}(e,c)$ is the subset of $\mathsf{parts}(e)$ that can be obtained (possibly using decryption) with knowledge of the keys in $\mathsf{knows}(c)$. Then, for example, $\forall \sigma : \Psi(\sigma) \Rightarrow \Phi_1(\sigma)$, where
$\Phi_1(\sigma) := \neg \exists k : k \notin \mathsf{access}(\mathsf{Intruder}) \wedge k \in \cup_i \mathsf{analz}(\sigma(A.out, i), \mathsf{Intruder})$,
specifies that the intruder should not be able to obtain a key k he must not have access to via A's output port $A.out$.

There are different types of security requirements having a similar structure (e.g. requirements on key secrecy or secure logging). AutoFocus supports specification patterns, which can be used for this purpose. Building a library of security requirement patterns on different levels of abstraction is subject of current work. Note that for the test generation, we assume that (as in our example) sufficiently detailed information is given so that the security requirements, which on an abstract level are often non-functional, can be formulated as functional requirements in the model. Obtaining and relating security requirements between different abstraction levels (where new vulnerabilities can be introduced) is an important issue, but out of scope of this paper. Preservation of security properties by refinement has, for example, been considered in [Jür01].

3 Generating Test Sequences for Vulnerabilities

In specification-based testing (see e.g. [DBG01] as a more recent example of the many approaches documented in the literature), test sequences are generated from a specification and used to verify the implementation. To test an implementation for vulnerabilities, we compute test sequences from the security model covering possible violations of the security requirements.

In terms of the above specification framework, a test sequence is a projection of an execution sequence σ to the ports, thus describing the input/output behaviour. Test scenarios are given by test case specifications, formulated as predicates $\Phi_i(\sigma)$. Test sequences for Φ_i then are valid executions fulfilling the Φ_i and thus can be computed as solutions to $\Psi \wedge \Phi_i$. In AutoFocus, a constraint solver is used for this purpose [LP00]. To ensure termination of the search (provided the expressions on the transitions terminate), σ must be limited by a maximum length.

$e \in \mathbf{Exp}$	$\epsilon(e)$	
a of type Bool	$\epsilon(a) \cup \neg a \cup \mathbf{true}$, if a of type Bool	missing or wrongly implemented condition
$a == b$	$\{(a == y) \mid y \in \epsilon(b)\} \cup \{(x == b \mid x \in \epsilon(a)\} \cup \mathbf{true}$	faulty check,e.g. for an identity of a party
$a \wedge b$	(analogous to $a == b$, possibly also $(a \vee y), (x \vee b)$ ($\wedge$ replaced by $\vee$),...)	boolean operator replacement
	(similarly for other boolean operators)	
$\mathsf{Encr}(k, a)$	$\{\mathsf{Encr}(k', a) \mid k' \in \mathsf{knows}(c)\} \cup \{\mathsf{Encr}(k, x) \mid x \in \epsilon(a)\}$	key confusion
$\mathsf{Mac}(k, a), \mathsf{Hash}(a)$	(analogous to $\mathsf{Encr}(k, a)$)	
$\mathsf{Key}(k)$	$\{\mathsf{Key}(k') \mid k' \in \mathsf{knows}(c)\}$	
$\mathsf{verifyMac}(a)$	$\{\mathsf{verifyMac}(x) \mid x \in \epsilon(a)\} \cup \mathbf{true}$	faulty MAC verification
$\mathsf{Msg1}(a_1, \ldots, a_n)$	$\{\mathsf{Msg1}(x_1, a_2, \ldots, ..., a_n) \mid x_1 \in \epsilon(a_1)\} \cup \ldots \cup \{\mathsf{Msg1}(a_1, a_2, \ldots, ..., x_n) \mid x_n \in \epsilon(a_n)\} \cup \mathsf{Empty}$	corrupted message
$\mathsf{is\_Msg1}(a)$	$\{is\_Msg1(x) \mid x \in \epsilon(a)\} \cup \mathbf{true}$	missing type check

Fig. 2. Possible mutations

3.1 Vulnerability Coverage Using Mutations

As is is not feasible to exhaustively test every behaviour of a security-critical system, first appropriate test case specifications have to be selected. For security testing, the aim is to cover a large number of possible vulnerabilities.

One can use structural coverage criteria such as state or transition coverage on the models [OXL99] and restrict them to those that are marked "critical", but this has the drawback that it does not take into account the security requirements.

The difficulty with defining coverage criteria related to the security requirements is that they are mostly universal properties. Therefore, a security requirement Φ_i can only be used to verify the model, not the implementation. If a trace fulfilling $\neg\Phi_i$ is found, the model violates the security requirement and must be corrected. Otherwise, Φ_i by itself cannot be used to select relevant traces, as *all* traces satisfy Φ_i.

In this case, mutation testing resp. fault injection techniques [Off95,VM98] prove to be promising approaches. In mutation testing, errors are introduced into a program (leading to a set of mutants), and the quality of a test suite is measured by its ability to distinguish the mutants from the original program (to "kill" the mutants). Fault injection works in a similar way, but is often also used for reliability evaluation (determining if a program tolerates a perturbation of the code or data states).

We introduce errors into the specification of the security-related behaviour, generate the threat scenarios and determine if and how the introduced errors can lead to security violations. The introduced errors can correspond to errors in the implementation or to attacks leading to such errors, e.g. subjecting a smart-card to environmental stress.

In our formal framework, mutations are generated by selecting a transition t of the STD of a component to be tested and applying a mutation function $\epsilon : \mathbf{Exp} \rightarrow \mathcal{P}(\mathbf{Exp})$ either to the precondition $pre(t)$, or to one of the output expressions or postconditions. This leads to a set of mutated transitions t'_j.

The mutation function ϵ can be based on general possible mutations for expressions and operands (e.g. operator or operand replacement) proposed for Ada by Offutt [OVJ96].

For security testing, ϵ should be based on common programming errors likely to lead to vulnerabilities, such as missing plausibility checks or wrong use of identities [AKS96]. In addition, in our model cryptography must be taken into account, leading to mutations corresponding to confusion of keys or secrets or missing or wrongly implemented verification of authentication codes.

Figure 2 shows such mutations, based on the AUTOFOCUS model described in Section 2. The actual mutation function ϵ to be applied depends on the available time and computing power. Only a part of the mutations shown in the table can be selected (e.g. only replacement of boolean (sub-)expressions by **true** corresponding to missing checks), or other mutations (e.g. from [OVJ96]) can be added.

Now, we proceed as follows. For the component c to be tested, we determine the set of mutated STDs (derived by replacing a transition t with t'_j). The transitions t are chosen from the transitions marked critical. We now generate the threat scenarios, but take the mutated version of c instead of the original one. This way, we obtain a mutated system model Ψ'. For each security requirement Φ_i, we then try to compute a system run satisfying $\Psi' \wedge \neg\Phi_i$ using the test sequence generator. If this is successful, it indicates that the mutation of t to t'_j introduces a vulnerability with respect to Φ_i and the trace σ shows how it can be exploited.

The input data to c from all traces σ determined this way gives us a test suite for c, covering possible vulnerabilities with respect to the security properties and the attack scenario. To determine the expected outputs, we use the original specification of c as an oracle.

3.2 Concretization of Abstract Tests

The abstract test sequences computed from the formal security specification still have to be translated to concrete test data (i.e., byte sequences) that can be used to test the actual implementation.

In many cases, concretization can be achieved using straightforward mappings between abstract and concrete test data [DBG01], and executing the test using a test driver that passes the inputs to the component to be tested and verifies if the outputs are as expected. However, testing security critical systems involves additional complications, mainly because of non-determinism, for example arising from randomly generating keys and nonces, and the use of cryptographic primitives:

- In formal specifications, cryptographic primitives are usually modelled symbolically, rather than as sequences of bytes, to make verification feasible (see [AJ01] for a justification of this general approach). The test driver has to map these symbols to sequences of bytes in a consistent way. Conversely, sequences of bytes created and output by the tested component (for example random values such as nonces or session keys) must be stored by the test-driver and used in place of the relevant symbols in the test-values of the remainder of the execution.
- Sometimes, values (such as transaction numbers or time stamps) are abstracted away in formal specifications to simplify verification (and because they are seen to be independent from a security property at hand). These have to be included in the concrete test-data in a consistent way.
- If encryption is used, the test driver must know the corresponding keys and encryption algorithms to be able to compute the encrypted input data and verify encrypted output data.

– Hash values or message authentication codes contained in the output data can only be verified when the complete data that was hashed is available to the test driver.

We fix a set of transaction variables TransV and define a concretization of abstract messages by mapping each message type M in the data type definitions (see Section 2, e.g. `Init`) to a sequence
concrete$(M) : d_1^M, d_2^M, \ldots, d_{n_M}^M$ of concrete data elements $d_i^M \in \mathbb{Z} \cup \mathsf{TransV} \cup \mathbf{Exp}$. Thus, d_i^M can be

– an integer value (corresponding to a constant sequence of bytes)
– a transaction variable (used to represent transaction data such as timestamps to be stored by the test driver)
– an AutoFocus expression, in which message M can be referenced by "this"

In the last case, the expression is evaluated and the result is again concretized. The transaction variables $v \in \mathsf{TransV}$ are associated with a set values$(v) \subseteq \mathbb{Z}$ of possible values. In addition, each data element has to be assigned a field length, which we omit here for simplification. Keys k are mapped directly to a transaction variable concrete$(k) : d_1^k \in \mathsf{TransV}$. The actual concretization, i.e. the values for the d_i^M and values(v) must be provided by the developer. See Section 4 for an example from the CEPS study.

An algorithm for the corresponding test driver is given in Fig. 4. It uses the algorithms **gen_sequence** and **verify_sequence** (see Fig. 3) to generate concrete test data from abstract output messages, resp. to compare abstract input messages to the test data received. In **verify_sequence**, *first*(s) denotes the prefix of s corresponding to d_i^M, *removefirst*(s) denotes the remaining part of s.

The idea of the algorithms is as follows. Constants in concrete(M) are passed directly to the implementation or compared with the received data. If a transition variable v appears in concrete(M) for an output message, either a new concrete value is chosen for v, or an already chosen value is added to the store *store* of the test driver. When data is received corresponding to v, it is either compared to the value already chosen, or the received value is added to the store. Encrypted messages, hashes and message authentication codes can be computed using the data available on sending. On reception, it is possible that a key or a part of the data to be hashed is unknown to the test driver, so **gen_sequence** would choose its own (most likely, wrong) concrete values thus changing the store. In this case, instead a condition is added to *conditions*. At each later step of the sequence, the test driver checks if those conditions become evaluatable and if they can fail. The processing of messages by the test driver is repeated for each step of the test sequence (sending or receiving a message M_{abstr} to/from port p of the component under test). Note that if more than one transaction is to be tested using a single test sequence, the store must be reset between the transactions. In addition, in some cases a fixed prolog to the test sequences may have to be generated by the test driver.

4 The CEPS Case Study

As an example case study, we examined a part of the Common Electronic Purse Specifications (CEPS). CEPS define requirements for a globally interoperable electronic purse scheme providing accountability and auditability. The specifications outline overall system security, certification and migration. For more detail on CEPS cf. [CEP01].

VAR $store : \mathcal{P}(\mathsf{TransV} \times \mathbb{Z}) = \emptyset$, $conditions : \mathcal{P}(\mathbf{Exp}) = \emptyset$;

algorithm gen_sequence(M_{abstr})
{compute concrete data from abstract message M_{abstr}}
$s \leftarrow \epsilon$
if $M_{abstr} = \mathsf{Encr}(k, x)$ or $M_{abstr} = \mathsf{Mac}(k, x)$ or $M_{abstr} = \mathsf{Hash(x)}$:
    $concr\_msg \leftarrow \mathsf{gen\_sequence}(x)$
    if $M_{abstr} = \mathsf{Encr}(k, x)$ or $M_{abstr} = \mathsf{Mac}(k, x)$: $concr\_key \leftarrow \mathsf{gen\_sequence}(k)$
    apply encryption, mac generation or hashing to $concr\_msg$; append to s
else determine message type M of M_{abstr}; $(d_1^M, \ldots, d_{M_n}^M) \leftarrow \mathsf{concrete}(M)$
    for $i \in \{1 \ldots M_n\}$:
        if $d_i^M \in \mathbb{Z}$: append d_i^M to s
        elseif $d_i^M \in \mathsf{TransV}$:
            if $\exists x : (d_i^M, x) \in store$: append x to s
            else choose $x \in \mathsf{values}(d_i^M)$; append x to s, $store \leftarrow store \cup (d_i^M, x)$
        elseif $d_i^M \in \mathbf{Exp}$: append gen_sequence(evaluate(d_i^M[**this** $\leftarrow M\_abstr$])) to s
return s

algorithm verify_sequence(M_{abstr}, s)
{verify concrete data s w.r.t. abstract message M_{abstr}}
if $M_{abstr} = \mathsf{Encr}(k, x)$:
    if $\exists k : (k, v) \in store$: $s' \leftarrow$ decrypt(v,$first(s)$); verify_sequence(x,s'); $s \leftarrow removefirst(s)$
    else $conditions \leftarrow conditions \cup (d_i^M = s)$
elseif $M_{abstr} = \mathsf{Mac}(k, x)$ or $M_{abstr} = \mathsf{Hash(x)}$:
    if $gen\_sequence(x)$ computable without changing store
        and $\exists k : (k, x) \in store$ (for $M_{abstr} = \mathsf{Mac}(k, x)$):
        $s' \leftarrow gen\_sequence(x)$; compare $hash(s')$ resp. $Mac(v, s')$ to $first(s)$
        $s \leftarrow removefirst(s)$
    else $conditions \leftarrow conditions \cup (d_i^M = s)$; $s \leftarrow removefirst(s)$
else determine message type M of M_{abstr}; $(d_1^M, \ldots, d_{M_n}^M) \leftarrow \mathsf{concrete}(M)$
    for $i \in \{1 \ldots M_n\}$:
        if $d_i^M \in \mathbb{Z}$: compare($first(s)$,d_i^M); $s \leftarrow removefirst(s)$
        elseif $d_i^M \in \mathsf{TransV}$:
            if $\exists x : (d_i^M, x) \in store$: compare($s, x$); $s \leftarrow removefirst(s)$
            else $store \leftarrow store \cup (d_i^M, first(s))$; $s \leftarrow removefirst(s)$
        elseif $d_i^M \in \mathbf{Exp}$: verify_sequence(evaluate(d_i^M[**this** $\leftarrow M\_abstr$]), s)
return **false** if any comparison failed

Fig. 3. gen_sequence and verify_sequence

algorithm do_test
for each step (p,M_{abstr}) in test sequence
    if output message: send $gen\_sequence(M_{abstr})$ to p
    else wait for input s on p (fail on timeout)
        if verify_sequence(M_{abstr},s) = **false**: fail
    if $\exists\ c \in conditions$: c evaluatable and $evaluate(c)$ = **false**: fail

Fig. 4. Test Driver Algorithm

We consider a central part of CEPS, the (unlinked, cash-based) load transaction, which allows the cardholder to load electronic value onto a card in exchange for cash at a load device belonging to the load acquirer. The participants involved in the transaction protocol are the customer's card, the load device and the card issuer. The load device contains a Load Security Application Module (LSAM) that is used to store and process data. We concentrate on vulnerabilities arising from possible failures during one protocol execution. Therefore, to increase readability, protocol details that aim to prevent replay attacks (such as transaction date and time, transaction numbers and identifiers) were abstracted away.

4.1 CEPS Security Model

Figure 5 shows the corresponding AutoFocus system structure diagram, containing the security attributes that describe the considered threat scenario. The public tags indicate the channels that can be attacked (all channels between the components, but not the channels used to write log information). The components themselves are assumed to be protected from manipulation, denoted by the security tag node. In the implementation, Card is a tamper-resistant smart card, LSAM a security module, and the Issuer system is out of reach of an intruder.

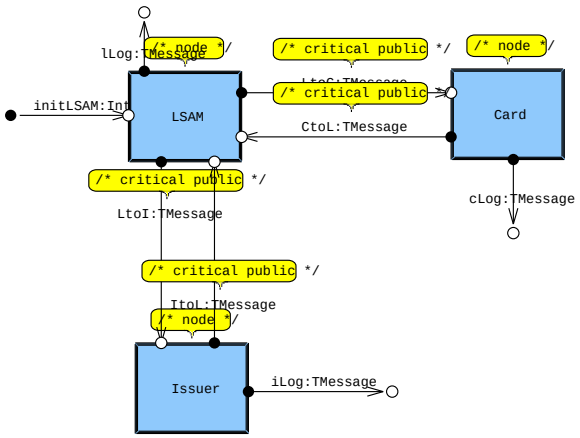


Fig. 5. System structure diagram for Load transaction

Let us concentrate on generating test sequences for the Card component, whose behavioural specification is shown in Fig. 6. Fig. 6 also includes the security annotations, indicating which of the states and transitions were labelled critical. The focus of the model is on the security-related behaviour, so most of the states and transitions correspond to receiving, verifying or creating cryptographic messages or writing secure logs. On the other hand, the part of the STD where the InquireCardInfo message is processed (state Inquiry, to obtain the current balance of the card) is not regarded as security-critical. It is just provided for information purposes and not protected from manipulation.

The CEPS load protocol works roughly as follows: the LSAM sends an initialization message Init(m) to the card with the amount to be loaded. The card's response to the

LSAM contains a message authentication code (MAC) S1, which is forwarded to the issuer together with data protecting its integrity and securing the issuer's decisions. On successful verification of this data, the issuer sends a MAC S2 to the LSAM, indicating his decision to allow the load transation. The LSAM forwards S2 to the card (Credit message), which verifies it, adjusts the balance and replies with a MAC S3. S3 is forwarded to the issuer informing him about the result of the transaction. At the end of a transaction, all parties write entries to their log files and stop in the LoadSucc state, resp. in the LoadFail state if an error was detected.

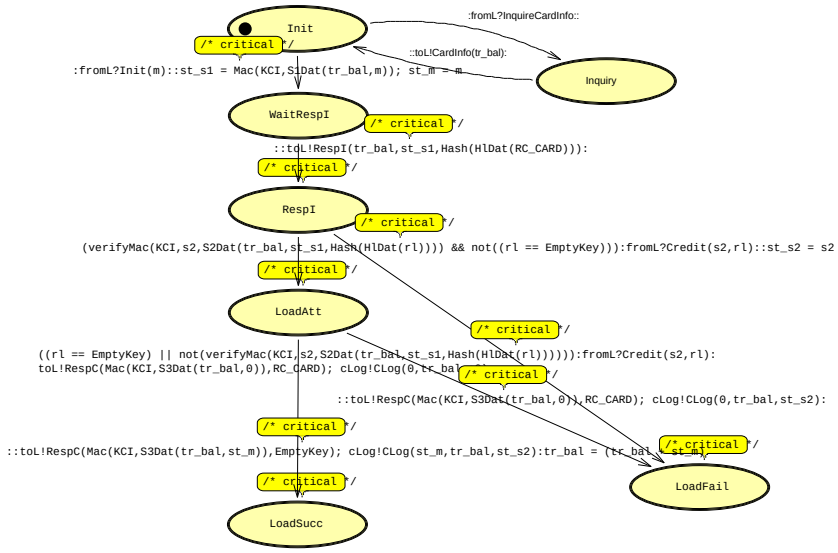


Fig. 6. STD for card, with security annotations

The main security objective of CEPS is resistance to fraud between the participating parties. For instance, if an amount of money is credited to the card in the unlinked load transaction, the load acquirer must owe this amount to the issuer. In case of a failed transaction (e.g. because of a communication problem or attack), funds can only be returned if proof of the failure has been obtained.

An example for a security requirement is

$\Phi_{LOG}(\sigma) := \neg\exists m, m', x, y, t : \mathsf{CLog}(m, x, y) \in \sigma(Card.cLog) \wedge$
$\mathsf{LLog}(m', t) \in \sigma(LSAM.lLog) \wedge (m \neq m')$

meaning that the log entries of Card and LSAM must agree for successful transactions, so that with the data of the LSAM the load acquirer can prove that m was credited to the card.

4.2 Test Sequence Generation for CEPS

For simplicity, we demonstrate test sequence generation for the Card component in the CEPS load protocol using a small subset of the possible mutations explained in Section 3.1: $\epsilon(a)$ can be **true** if a is a boolean (sub-)expression in a precondition of a critical transition

Fig. 7. Test Sequence for Load transaction

(first line of Figure 2). This results in altogether 9 mutations of the Card STD, 4 of which lead to test sequences violating security requirements.

As an example, consider the transition from state RespI to LoadAtt in Fig. 6. In this transition, the Card receives the MAC s2 and the random value rl from the LSAM. The precondition consists of two parts: verifying s2 and checking if rl is not empty. Replacing the MAC verification (subterm `verifyMac(...)`) with **`true`** results in a test sequence violating Φ_{LOG}.

Fig. 7 shows the begining and end of this test sequence as an AutoFocus EET (the actual test sequence is 28 steps long). Here, the LSAM is initialized to transfer 1 unit of money. However, when this initialization is passed to the Card, the intruder (who has control over all messages on the public channels) changes the amount to 10. In the model, the intruder signals its activity by sending a cPresent message to the environment. Because of the manipulation, in the Load message passed to the Issuer later, the amounts sent by the issuer and by the card (via the MAC s1) differ, so the issuer aborts the transaction. The MAC s2 passed back via the LSAM to the card should now lead to abortion of the transaction by the card (as s2 contains an empty key instead of rl). However, as s2 is not verified, this is not the case, and the card erroneously increases its balance by 10 units and writes this to the log. Computation of this test sequence took about 300s.

Concretization We demonstrate how to concretise test sequences consisting of abstract messages to byte sequences at the example of the first two messages in the test sequence depicted in Fig. 7. The relevant part of the data type definition reads as follows:

```
data TMessage = ... | Init(getInitM: Int)
                    | RespI(getRiBal:Int, getRiS1:TMessage, getRiHC:TMessage)
                    | S1Dat(getS1Bal:Int, getS1M:Int) | HlDat(getRC: TKey) | ...
```

Message M	concrete(M)
Init	90, 50, 00, 00, 18, 17, $DTHR_{LDA}$, $CURR_{LDA}$, ID_{LACQ}, ID_{LDA}, getInitM(this), 00
RespI	21, ID_{ISS}, ID_{CEP}, $DEXP_{CEP}$, NT_{CEP}, getRiS1(this), getRiHC(this), 00, 90, 00
S1Dat	getS1Bal(this), $BALMAX_{CEP}$, $CURR_{LDA}$, $DEXP_{CEP}$, $DTHR_{LDA}$, ID_{CEP}, ID_{ISS}, ID_{LACQ}, ID_{LDA}
HIDat	ID_{LACQ}, ID_{LDA}, ID_{ISS}, ID_{CEP}, NT_{CEP}, R_{CEP}
...	

$v \in$ TransV	values(v)
$DTHR_{LDA}$	$\{x \in \mathbb{Z} : x$is BCD coded date/time$\}$ (transaction date/time)
ID_{LACQ}	e.g. $\{1234ABCD\}$ (ID load acquirer)
RC_{Card}	$\{x \in \mathbb{Z} : x$ is 16 byte Integer$\}$ (random number of card)
NT_{CEP}	$\{x \in \mathbb{Z} : x$ is 4 byte Integer$\}$ (transaction identifier from card)
KCI	e.g. $\{780A \ldots B6\}$ (16 byte key between Card and Issuer)
...	

Fig. 8. Concretization of abstract CEPS messages

Table 8 shows the concretization mapping for the abstract messages and the values of the transaction variables, taken from the CEPS specification. The test driver translates the first message Init(10) to the card e.g. to the byte sequence

90 50 00 00 18 17 02 04 04 09 01 0C CC 0E 12 34 AB CD 12 34 56 78 0A BC 00 00 00 0A 00

The message type is Init, so the test driver looked up *concrete*(Init) in Table 8. The first bytes (90 50 .. 17) are constants, for $DTHR_{LDA} \in$ TransV (next 5 bytes) a value has been chosen, etc. The underlined part corresponds to the transaction amount 10. As the reply RespI(0, Mac(KCI, S1Dat(0, 10)), Hash(HIDat(RC_CARD))), the test driver expects

21 AB CD EF 12 34 56 78 0A BC 03 04 04 $\underline{NT_{CEP}}$ $\underline{S1}$ $\underline{h}$ 90 00

NT_{CEP}, the transaction identifier assigned by the card, is read from the reply and added to the store for later use, the MAC $S1$ can be verified (as all data that is part of $S1$ and the key is known to the test driver), and the hash h is added to the constraints and will be checked at a later execution step.

5 Related Work

There has been extensive research into specification-based testing, including [DF93,PS97,HNS97]; a complete overview has to be omitted. Here we used [LP00] as it has been built into the tool AutoFocus; one could also have used a different approach.

Some of that work has been applied to safety-critical systems; our focus, however, is to adapt these concepts to the domain of security-cricital systems with its specific characteristics as explained in Section 1 (most prominently, the use of cryptography). To the best of our knowledge, this is the first published work using formally-generated test-sequences for security-critical systems, apart from [JW01b] which concerns testing of firewalls.

Dushina et al. explain concretization in their Genevieve framework [DBG01], but do not address the specific issues we explained in Section 3.2.

In intrusion detection (see e.g. [US01] for a model-based approach), a running system is monitored for attacks. Complementary to the *detection* of security violations in fielded systems, we aim to generate test data to find and remove the possibility of such attacks before system deployment. The AVA approach [VM98] is conceptually similar to the fault-insertion explained in Section 3.1, but the focus is on identifying critical statements rather

than finding test sequences (for which random distributions are used), and it does not consider cryptographic mechanisms.

6 Conclusion and Further Work

We presented work on generating test sequences for transaction systems from a formal security model supported by the CASE tool AutoFocus. Going beyond classical specification-based conformance testing, the test sequences are determined with respect to stated security requirements. Using mutations of the system specification and attack scenarios, test sequences are generated that give increased confidence that a system meets the relevant security requirements. We gave results on concretizing abstract test sequences, to be able to apply them to existing implementations. The problem of test-case explosion is handled in so far as only system parts considered as security-critical are tested.

The proposed method seems suitable to be applied to the application domain of security-critical systems, since it allows to find tests likely to detect possible vulnerabilities even in complex execution scenarios. Consideration of domain-specific concepts such as cryptography and random numbers is supported. Given that security aspects are playing an increasingly important role in the development of distributed systems, having a way to do methodological testing of security-critical systems should be a worthwhile goal.

In general, it is unfeasible to verify completely that an implementation faithfully implements its specification. So even given a specification that is proved secure, our approach of ensuring on the implementation level that a system satisfies certain critical security requirements seems to be indispensable.

We explained our approach at the example of the purchase transaction protocol from the Common Electronic Purse Specifications; it is applicable to security-critical systems in general.

Note that our approach only aims to find vulnerabilities that can be detected at the level of abstraction of a given specification (which may however be lowered by refining the specification). Although we had to choose a method of generating test-sequences from formal specifications, the general approach is independent from the specific method, and also from the formal semantics of the used method AutoFocus.

In future work we plan to devise a test case specification language specialized to security-critical systems which allows one to formulate assumptions on the underlying implementation layer of the system and which can be compiled "intelligently" into test cases by applying optimization depending on the test case specification in question.

References

[AJ01] M. Abadi and J. Jürjens. Formal eavesdropping and its computational interpretation. In N. Kobayashi and B.C. Pierce, editors, *Theoretical Aspects of Computer Software (4th International Symposium, TACS '01)*, volume 2215 of *LNCS*, pages 82–94. Springer, 2001.

[AKS96] T. Aslam, I. Krsul, and E. Spafford. Use of A Taxonomy of Security Faults. In *19th National Information Systems Security Conference*, Baltimore, 1996.

[BS01] M. Broy and K. Stolen, editors. *Specification and Development of Interactive Systems.* Springer, 2001.

[CEP01] CEPSCO. Common Electronic Purse Specifications, 2001. Business Requirements vers. 7.0, Functional Requirements vers. 6.3, Technical Specification vers. 2.3, available from http://www.cepsco.com.

[DBG01] J. Dushina, M. Benjamin, and D. Geist. Semi-Formal Test Generation with Genevieve. In *DAC*, 2001.

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *FME '93: Industrial-Strength Formal Methods*, pages 268–284, 1993.

[FBGL94] J. Fitzgerald, T.M. Brookes, M.A. Green, and P.G. Larsen. Formal and informal specifications of a secure system component: first results in a comparative study. In Denvir, Naftalin, and Bertran, editors, *Formal Methods Europe '94: Industrial Benefit of Formal Methods*, pages 35–44. Springer, 1994.

[GSG99] S. Gritzalis, D. Spinellis, and P. Georgiadis. Security protocols over open networks and distributed systems: Formal methods for their analysis, design, and verification. *Computer Communications*, 22(8):695–707, 1999.

[HNS97] S. Helke, T. Neustupny, and T. Santen. Automating Test Case Generation from Z Specifications with Isabelle. In J. Bowen, M. Hinchey, and D. Till, editors, *Proc. ZUM '97: The Z Formal Specification Notation*, volume 1212 of *LNCS*, pages 52–71. Springer, 1997.

[ITU96] ITU. ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). ITU-TS, Geneva, 1996.

[Jür01] Jan Jürjens. Secrecy-preserving refinement. In *Formal Methods Europe*, LNCS. Springer, 2001.

[JW01a] Jan Jürjens and Guido Wimmel. Security modelling for electronic commerce: The Common Electronic Purse Specifications. In *First IFIP conference on e-commerce, e-business, and e-government (I3E)*. Kluwer, 2001.

[JW01b] Jan Jürjens and Guido Wimmel. Specification-based testing of firewalls. In *Andrei Ershov 4th International Conference "Perspectives of System Informatics" (PSI'01)*, LNCS. Springer, 2001.

[LP00] H. Lötzbeyer and A. Pretschner. Testing concurrent reactive systems with constraint logic programming. In *2nd Workshop on Rule-Based Constraint Reasoning and Programming*, Singapore, 2000.

[Off95] J. Offutt. Practical Mutation Testing. In *12th International Conference on Testing Computer Software*, 1995.

[OVJ96] J. Offutt, J. Voas, and J.Payne. Mutation Operators for Ada. Technical Report ISSE-TR-96-09, Information and Software Systems Engineering, George Mason University, 1996.

[OXL99] J. Offutt, Y. Xiong, and S. Liu. Criteria for Generating Specification-based Tests. In *1st IEEE Conference on Engineering of Complex Computer Systems*, 1999.

[Pau98] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.

[PS97] J. Peleska and M. Siegel. Test automation of safety-critical reactive systems. *South African Computer Jounal*, 19:53–77, 1997.

[PS99] J. Philipps and O. Slotosch. The Quest for Correct Systems: Model Checking of Diagrams and Datatypes. In *Asia Pacific Software Engineering Conference 1999*, 1999.

[Tho99] S. Thompson. *Haskell: The Craft of Functional Programming.* Addison-Wesley Longman, 1999.

[US01] P. Uppuluri and R. Sekar. Experiences with Specification-based Intrusion Detection. In *Recent Advances in Intrusion Detection (RAID)*, 2001.

[VM98] J. Voas and G. McGraw. *Software Fault Injection: Inoculating Programs Against Errors.* Wiley, 1998.

[WLPS00] G. Wimmel, H. Lötzbeyer, A. Pretschner, and O. Slotosch. Specification Based Test Sequence Generation with Propositional Logic. *Journal on Software Testing Verification and Reliability*, 10, 2000.

[WW01] G. Wimmel and A. Wißpeintner. Extended description techniques for security engineering. In *IFIP SEC*, 2001.