A Formal Security Model for Microprocessor Hardware

Volkmar Lotz[1], Volker Kessler[1], and Georg Walter[2]

[1] Siemens AG, Corporate Technology, D-81730 Munich,
{Volkmar.Lotz,Volker.Kessler}@mchp.siemens.de
[2] Siemens AG, D-81541 Munich

Abstract. The paper introduces a formal security model for a microprocessor hardware system. The model has been developed as part of the evaluation process of the processor product according to ITSEC assurance level E4. Novel aspects of the model are the need for defining integrity and confidentiality objectives on the hardware level without the operating system or application specification and security policy being given, and the utilisation of an abstract function and data space. The security model consists of a system model given as a state transition automaton on infinite structures, and the formalisation of security objectives by means of properties of automaton behaviours. Validity of the security properties is proved. The paper compares the model with published ones and summarises the lessons learned throughout the modelling process.
Key words: security, hardware, formal security models.

1 Introduction

When evaluating a system according to ITSEC's [8] quality assurance level E4 and beyond, the provision of a formal model of the underlying security policy is an integral part of the assessment procedure. The formal model is an "abstract statement of the important principles of security that the TOE [target of evaluation] will enforce" [8, p.33]. In general, it includes a formal specification of the security objectives and a formal system model that emphasises the security enforcing functions either by explicitly modelling them or by implicitly describing them in terms of the effects that they are expected to achieve. The goal of formalising these points is to review and get a deeper understanding of the security principles, to be able to analyse formally the validity and consistency of the security objectives, and to provide confidence that the system satisfies its security requirements by showing that the system model is adequate with respect to the security enforcing functions of the system.

In this paper we present a formal security model for microprocessor hardware that has been developed in the context of an E4 evaluation of the processor chip, and that shows some novel aspects compared to existing formal security models [10]. The most unusual feature of the work lies in the fact that the formal model has been defined for a hardware system where neither the operating system nor

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. I, LNCS 1708, pp. 718–737, 1999.

the intended applications are known exactly at the time the hardware is designed, except for the fact that they will run in security sensitive environments. In particular, the security policy of the operating system as well as the applications is not yet defined, thus forcing the model to express security objectives by referring to an abstract application or operating system function space that is expected to satisfy its own security policy.

Since there is a wide range of possible applications, the hardware does not enforce a particular security policy, but rather provides basic security services on a physical and logical layer, like encryption, that can be utilised by operating systems and applications in order to maintain their security requirements. As a consequence, the security objectives of the hardware are stated in abstract terms, e.g. "legitimate access" or "undesired modifications". The above mentioned reference to higher order concepts, e.g. definition of function space and application of operating system functions, allows for an appropriate formalisation of these abstract terms.

In the following, we give a short introduction to the application domain and the hardware system to be modelled. Then, our security model is introduced in detail: we describe the system model which is based on a variant of state transition automata over infinite structures, show the formalisation of the security objectives, and give a summary of the proofs showing that the abstract system provably meets the security objectives. Though we could not refer to or adapt an already existing formal security model, our model shows some similarities concerning modelling and proof principles to published models, which will be discussed in detail.

Finally, the paper presents some of the lessons learned with respect to the pragmatic aspects of formal security modelling within an evaluation process according to E4. We are convinced that, even when not striving for certification, formal security modelling is useful. However, there are some key factors that have to be considered if the modelling work is to be successful in terms of applicability, expressiveness and, last but not least, efficiency. We will investigate these topics at the end of the paper. Furthermore, we examine whether our particular model can be used as a basis for a whole class of models for processor applications.

For reasons of exposition, the paper shows a simplified version of the original model that was submitted to the evaluation and certification process. However, the original model differs only in technical details, thus the results that have been derived can be equally applied.

2 Security Target

The target of evaluation is a security processing system with a CPU (including an encryption unit), RAM memory, ROM memory and EEPROM memory, which holds the stored data even if the power is turned off (see Fig. **??**). The most important security objective is to preserve the security of information stored in the memory components. In detail, this means

- The data stored in any of the memory components shall be protected against unauthorised access or modification.
- The security relevant functions stored in any of the memory components shall be protected against unauthorised access or modification.
- As a consequence, it must not be possible to execute any hardware test routine without authorisation.

These objectives are achieved by implementing a set of security enforcing functions which mainly perform the following two tasks:

- The system passes several phases during its lifetime. The entry to the phases is controlled by a phase management, which checks different flags and gives a specified level of authorisation.
- Additionally, encryption of the memory components is implemented in hardware. Encryption utilises several keys and key sources with a chip specific random number among them.

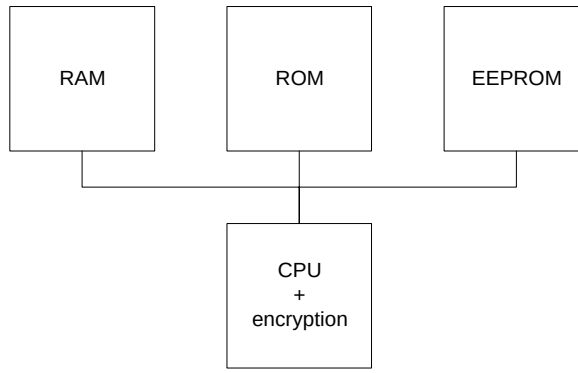

Fig. 1. Processor architecture

Though the ITSEC allow defining a security model in terms of references to existing and published models, for our application we could not make use of this option. On one hand this is due to the fact that the well-known models like Bell-LaPadula etc. refer to operating system or higher levels, where operations like reading, writing, or granting access rights form a suitable level of abstraction. For the processor hardware we do not know which kind of functionality the operating system will offer. Thus it cannot be anticipated that the system models of the given security models are adequate. We only know that there will be some functionality offered by an operating system or an application. We have to assume that executing this functionality is legitimate in any case, since operating system and application are considered to be responsible for enforcing their respective security policy.

On the other hand, among the most important modelling principles that were to be followed in our work we had the requirement of keeping the model as simple and abstract as possible without losing meaningfulness. This was motivated by the need for discussing the model with non-experts in formal methods, in particular the processor development staff, and to accelerate the evaluation process. As a consequence, we intended to define the model without using modelling aids, i.e. formal concepts that occur in the model in order to explicate its properties, but that do not have an immediate correspondence within the real system. Thus, for example, formal models relying on access control concepts including the definition of rights and the utilisation of some access control data structure could not be appropriately applied. For further comparison to existing models see Sect. 5.1.

The formal security policy model of the processor hardware consists of two parts: a system model describing the processor's behaviour on an abstract level by means of a state transition automaton with input and output, and a set of security objective specifications given as properties on automata behaviours. Thus we are able to prove the validity of the security properties with respect to the system model. Interpreting the system model in terms of the real processor then allows one to conclude with some evidence that the processor indeed meets its security objectives as required by ITSEC E4 assessment criteria.

3 System Model

In this chapter, we present the system model part of the security model in detail. We formally define the processor as a state transition system and explain why it is an appropriate abstraction of the original processor hardware. Though the explanation is informal, it is an important part of the process, since it defines the modelling relationship and is the basis for interpreting the results of formal security analysis.

3.1 General System Model

We assume the reader to be familiar with tuples, functions and (finite or infinite) sequences. We will use common notation including $\Pi_i(s)$ for tuple projection, $dom(f)$ and $rng(f)$ for function domain and range, A^* and A^∞ for finite and infinite sequences over a set A, $A^\omega = A^* \cup A^\infty$ for arbitrary sequences, and $._i$, $\circ$, $\sharp$, and $\sqsubseteq$ for selection, concatenation, length and the prefix relation on sequences, respectively.

The system model itself is defined as a variant of state transition automata over infinite structures [16].

Definition 1. *An* automaton M *is a tuple of five components:*

$$M = (State, In, Out, S_0, \delta)$$

- *State denotes a set of states,*
- *In denotes the set of possible inputs,*
- *Out denotes the set of possible outputs,*
- $S_0 \subset State$ *is the set of initial states,*
- $\delta \in State \times In \to \mathcal{P}(State \times Out)$ *denotes the state transition function.*

Inputs cannot be predicted, thus we require that each input is enabled at each point of time. As a consequence, the state transition function is total by definition. Note that an automaton as defined above behaves nondeterministically, which has been introduced to capture underspecification with respect to the system's behaviour.

The semantics of a thus defined automaton M is defined as the set of possible behaviours, where a behaviour is characterised by a sequence of states, inputs, and outputs.

Definition 2. *A* behaviour *of* $M = (State, In, Out, S_0, \delta)$ *is given by a triple*

$$b = (s, in, out) \quad \in State^{\omega} \times In^{\omega} \times Out^{\omega} \ ,$$

where $s_0 \in S_0$ *and* $(s_i, out_{i-1}) \in \delta(s_{i-1}, in_{i-1})$ *for all* $i \in \mathcal{N}, 1 \leq i \leq \sharp b$.
The set of possible behaviours of M *is defined by:*

$$\|M\| = \{b \mid b \text{ is a behaviour of } M\} \ .$$

The decision to use this kind of automaton even though it is non-standard was driven by the need to keep the model simple and abstract as well as to be able to model and prove the relevant properties. In particular, we do not suffer from including infinite sequences, since the security objectives can be modeled by safety properties [4] allowing for inductive proofs, but we are thus able to exclude termination issues from the model. Additionally, we need an automaton with output in order to specify that an adversary gets some information as the result of successfully performing an attack.

3.2 State Space

The processor system's state space is given by the life cycle phase, the data objects and the function objects. The life cycle phase is included in the state specification because of its importance with respect to authorisation: the processor phase reflects security relevant environment properties and thus allows to deduce information about subjects requesting an operation and admissibility of requests. Data and function objects differ by functions being executable, i.e. the function identifier may occur as parameter to an *exec* operation (see Sect. 3.3). Due to the nature of the security objectives and interface and due to the fact that on the processor level executable functions cannot be further specified we distinguish between object identifiers and an object evaluation function, with the range of the evaluation function not being further specified. We partition the set of object identifiers with respect to the following definition.

Definition 3. *We define the following set of phase and object identifiers:*

$$Ph, O, F, F_{Test}, F_{Test0}, F_{Test1}, F_{Sec}, F_{nSec}, D, D_{Int}, D_{Sec}, D_{nInt}, D_{nSec}$$

- $Ph = \{0, 1, 2, \infty\}$ *denotes the set of identifiers of phases during the system's life cycle.*
- O *denotes the set of all objects consisting of two disjoint subsets* $O = F \cup D$.
- F *is the set of function identifiers. In particular, it contains the subsets* $F_{Test}, F_{Test0}, F_{Test1}, F_{Sec}, F_{nSec}$*:* F_{Test0} *and* F_{Test1} *denote the test functions used throughout phase 0 and 1, resp., with* $F_{Test} := F_{Test0} \cup F_{Test1}$ *being their disjoint union.* $F_{Sec} \subseteq F$ *is the subset containing the security critical function's identifiers. Thus,* $F_{nSec} = F \setminus F_{Sec}$ *denotes the set of identifiers of functions which are not security relevant.* $F_{Test} \subset F_{Sec}$ *holds.*
- D *is the set of data object identifiers. We both have* $D_{Int} \subset D$*, the set of identifiers of data that are to be integrity protected, and* $D_{Sec} \subset D_{Int}$ *the set of identifiers of data objects for which integrity as well as confidentiality is to be protected. We have* $Key \in D_{Sec}$ *(the encryption master key) and* $SN \in D_{Int} \setminus D_{Sec}$ *(the serial number).* $D_{nInt} = D \setminus D_{Int}$ *denotes the set identifiers of data objects that are not security relevant, and* $D_{nSec} = D \setminus D_{Sec}$ *those of non-confidential objects.*

Note that compared to Sect. 2 we have extended the phase specification by ∞ which denotes a phase corresponding to the system being in an error state. Also note that for functions denoted by elements of F we only have one category of security relevance, whereas data objects are distinguished with respect to integrity and confidentiality protection needs. This is an immediate consequence of the system structure and the security objectives. In particular, there is the processor's serial number for which only integrity is required.

Definition 4. Val *denotes a set of values which are not specified in detail and represent information that is contained in or can be derived from objects. In particular, we have* $\perp \in Val$ *modelling an "empty" information value. For a set of identifiers* $X \subset O$ *we have* Val_X *denoting the set of possible values of objects identified by elements of* X.

For example, $Val_{F_{Sec}}$ is the set of security relevant functions. Those security critical functions that are active, i.e. executable, at a given point of time, are given by a subset of $Val_{F_{Sec}}$.

For a given point within the life cycle of the automaton, the set of available data and functions (*data state*) is completely determined by the set of accessible object identifiers and the actual values of these objects. We model this by functions mapping identifiers to elements of Val, with the function domains determining the active objects. Since we have to distinguish data and functions during analysis, we define a function for each of them, leading to a complete data state description consisting of two mappings.

Let $f \in F$ be a function identifier, and $val \in F \to Val$ an appropriate evaluation. If $val(f) = \perp$, i.e. $val(f)$ is undefined, we interpret this as the identifier f

not being accessible in the given state, which leads to f not being executable in this state. Deletion of a function can be defined by redefining val. If $val(f) = g$ where $g \neq \perp$, a call of f leads to the execution of the operation g. The modelling principle corresponds to viewing the data state as a memory component, which is appropriate with respect to the processor hardware (see Fig. **??**) and its security objectives.

Definition 5. *A* state s *is given by a triple:*

$$s \in Ph \times (F \to Val) \times (D \to Val) \ .$$

We thus have the set of states $State$ *given by :*

$$State = Ph \times (F \to Val) \times (D \to Val) \ .$$

For a given state $s \in State$ *we have:*

- $ph(s) = \Pi_1(s) \in Ph$ *the component denoting the current phase,*
- $data(s) = (\Pi_2(s), \Pi_3(s))$ *the data state component*[1]*,*
- $val_s^F = \Pi_2(s)$ *function state component*[2] *and* $fct(s) = dom(\Pi_2(s))$ *the set of available function identifiers,*
- $val_s^D = \Pi_3(s)$ *the data object state component and* $dt(s) = dom(\Pi_3(s))$ *the set of available data object identifiers.*
- *the function* $val_s : O \to Val$ *defined by:*
  $val_s(o) := val_s^F(o)$ *if* $o \in F$ *,*
  $val_s^D(o)$ *if* $o \in D$ *.*

Since F and D are disjoint, val_s is well defined. We do not require the evaluation function to remain invariant throughout a particular behaviour of the automaton. Thus the model is able to capture attacks resulting from modification of functions, in particular testing software. We abbreviate an error state, which is given if the processor is in phase ∞ by e.

Among the security critical functions, the test functions possibly enabling a phase transition, play a particular role emphasised in Sect. 2. For each of them we define a predicate denoting test results.

Definition 6. *For each* $f \in F_{Test}$ *we define a predicate* $Test_f(val_s)$*. If a test predicate evaluates to true, the test is considered to be successfully passed.*

3.3 Environment Interface

The processor interacts with its environment by means of input and output. We interpret inputs as requests for the execution of operations which can be accepted or denied by the processor. Requests are assigned to a subject, with the processor manufacturer Pmf being particularly distinguished. Including subject identifiers is an abstraction from the real processor system, where the subject identity is derived from the results of an authentication mechanism.

[1] Note that the notion data state also refers to functions.
[2] Note that in this case $val_s^F(f)$ itself is a function.

Definition 7. *Let Sb be a set of* subject identifiers *including $Pmf \in Sb$. A* subject *$sb \in Sb$ is able to request three types of operations, two of them modelling ordinary processor operation and one modelling attack situations:*

- *$exec(sb, f)$ where $f \in F$ models the function call of f;*
- *$load(sb, o, v)$ where $o \in F \cup D$ and $v \in Val$ describes the request for modifying or adding an object o with value v;*
- *$spy(o)$ where $o \in O$ models an eavesdropping action, which is given if a subject tries to force the processor to reveal information without executing a function in F. A spy action is typically given by physical attacks on the processor hardware. Note that the definition of spy is independent of subjects, since it models an attack that should be prohibited.*
- *We also define an input selector function subject by $subject(exec(sb, f)) = sb$ and $subject(load(sb, o, v)) = sb$ where $sb \in Sb, f \in F, o \in F \cup D$ and $v \in Val$.*

Our model of the influence of operation execution, i.e. state transitions of the automaton, on the environment is abstract as well: an output value does not only model the data being output by the processor, but is assumed to include all the information that can be derived from the data. We implicitly assume a partial order on the set of output values modelling information containment, thus we can consider outputs to be elements of $Val \cup \{ok, no\}$. The special values *ok* and *no* describe outputs stating acceptance or denial of requests, or the occurrence of errors.

Definition 8. *Let $f \in F$ be a function identifier, $s \in State$ a state, and $out : Val \to Val$, $change : Val \to (F \to Val) \times (D \to Val)$ suitable selection functions, then*

- *$out(val_s^F(f)(data(s))) \in Val$ denotes the output resulting from execution of f, and*
- *$change(val_s^F(f)(data(s))) \in (F \to Val) \times (D \to Val)$ denotes the state change resulting from execution of f.*

Note that in the above definition, the term $val_s^F(f)(data(s))$ denotes the application of the current evaluation $val_s^F(f)$ of f to the current data state $data(s)$, thus specifying the results of the execution of f.

Our model of executable functions and their effects does not include function parameters. This simplifies the model significantly without restricting its expressiveness. Simply assume that, like in an assembler language, parameters are loaded to data objects and functions immediately address data objects.

From the security modelling and analysis viewpoint, our decision to explicitly model data space and function execution might be considered to provide irrelevant detail compared to a Bell-LaPadula-like approach referring to object access only. However, since our model has been part of an industrial-level IT-SEC evaluation, we were forced to keep the balance between the different system

development tasks related to the formal security model: performing security analysis, serving as the top-level security specification for the evaluation target, and proving the correspondence of the model and the system implementation.Adding some more detail to the security model significantly reduced the complexity of correspondence arguments.

3.4 State Transition Function and Model Definition

Definition 9. *The formal system model is given by an automaton*

$$M = (State, In, Out, S_0, \delta) \ .$$

- $State = Ph \times (F \to Val) \times (D \to Val)$,
- $In = \{exec(sb, f) | sb \in Sb,\ f \in F\} \cup \{load(sb, o, v) | sb \in Sb,\ o \in F \cup D,\ v \in Val\} \cup \{spy(o) |\ o \in O\}$,
- $Out = Val \cup \{ok, no\}$,
- $S_0 \subset State$ *where for all* $s_0 \in S_0$ *we have* $ph(s_0) = 0$ *and* $F_{Test} \subseteq fct(s_0)$.
- $\delta \in State \times In \to \mathcal{P}(State \times Out)$ *is defined by the state transition rules below.*

It remains to define the state transition rules. For reasons of exposition, the following rule definitions in general only give those state components explicitly that are changed if a state transition occurs. In any case, all the components that are not explicitly mentioned remain invariant. Following definition 1, the state transition function is total. However, organisational and technical security measures ensure that only reasonable inputs will occur in processor applications. For example, operating system functionality is checked, and abuse of application functionality does not disclose security relevant information because of encryption. We therefore simply assume that only reasonable inputs occur. To keep things clean, we do not encode this assumption immediately in the system model, but rather state it as a property of the input sequences that may occur. In Sect. 4.1 the model is thus extended by a set of assumptions explicitly defining the notion "reasonable input". The processor's security mechanisms ensure validity of these assumptions.

We start with the definition of the state transition rules for *exec* inputs that apply to the automaton in phase 0, the construction phase. Let $s \in \{0\} \times (F \to Val) \times (D \to Val)$ be a state in this phase of the processor's life cycle

(R0.0) $\delta(s, exec(sb, f)) = (s', \text{“}ok\text{”})$ if $sb = Pmf$,
where $ph(s') = 1$, $fct(s') = fct(s) \setminus F_{Test0}$, $f \in fct(s) \cap F_{Test0}$
$val^F_{s'} = val^F_s|_{fct(s) \setminus F_{Test0}}$ and $Test_f(val_s)$

(R0.1) $\delta(s, exec(sb, f)) = (s', \text{“}ok\text{”})$ if $sb = Pmf$,
where $ph(s') = 2$, $fct(s') = fct(s) \setminus F_{Test}$, $f \in fct(s) \cap F_{Test1}$
$val^F_{s'} = val^F_s|_{fct(s)\setminus F_{Test}}$ and $Test_f(val_s)$

(R0.2) $\delta(s, exec(sb, f)) = (e, \text{“}no\text{”})$ if $sb = Pmf$,
with arbitrary $data(e)$ $f \in fct(s) \cap F_{Test}$
and $\neg Test_f(val_s)$.

(R0.3) $\delta(s, exec(sb, f)) = (s', out(val^F_s(f)(data(s))))$ if $sb = Pmf$,
where $ph(s') = 0$, $f \in fct(s)\setminus F_{Test}$
$data(s') = change(val^F_s(f)(data(s)))$

(R0.4) $\delta(s, exec(sb, f)) = (s, \text{“}no\text{”})$ if $sb \neq Pmf$
or $f \notin fct(s)$

(R0.0) models the fact that only the hardware manufacturer is allowed to execute tests. The required test function must a member of the set of active, i.e. executable, functions. If the test succeeds, phase 1 will be entered. Note that when entering phase 1 the test functionality according to phase 0 will be deleted, i.e. removed from the set of active functions. The model does not exclude that the test is separated in different tasks that are to be executed in sequence. The essential thing is that after finishing the last part the next phase is entered. The execution of previous parts of the test is modelled by (R0.3).

(R0.1) describes the case in which a test function intended for phase 1 is inadvertently executed in phase 0. Then, the processor moves immediately to phase 2, without being able to load application functionality (cf. the rules for *load* below). Though leading to an unusable system, this state transition does not contribute to a security violation. If a test fails, the system will enter an error state as described by rule (R0.2).

In phase 0, active functions may be executed only by the processor manufacturer. This does not change the phase. Additional requirements on functions executable in phase 0, e.g. that they should not modify any of the currently available functionality, are expressed as predicates over elements of $rng(val^F_{s_0})$ and required in Sect. 4.1.

Altogether, in phase 0 only the processor manufacturer's requests for function execution are accepted by the system, all other requests will be denied as being described by (R0.4). Obviously, requests for the execution of functions that are currently not executable are denied as well, independent of the requestor.

The transition rules for *exec* inputs in phase 1, the upload phase, are defined completely analogous to those referring to phase 0. Let $s \in \{1\} \times (F \to Val) \times (D \to Val)$ be a state in the second phase of the processor's life cycle.

(R1.1) $\delta(s, exec(sb, f)) = (s', \text{“}ok\text{”})$ if $sb = Pmf$,
where $ph(s') = 2$, $fct(s') = fct(s) \setminus F_{Test1}$, $f \in fct(s) \cap F_{Test1}$
$val^F_{s'} = val^F_s|_{fct(s)\setminus F_{Test1}}$ and $Test_f(val_s)$

(R1.2) $\delta(s, exec(sb, f)) = (e, \text{“}no\text{”})$ if $sb = Pmf$,
with arbitrary $data(e)$ $f \in fct(s) \cap F_{Test1}$
and $\neg Test_f(val_s)$

(R1.3) $\delta(s, exec(sb, f)) = (s', out(val^F_s(f)(data(s))))$ if $sb = Pmf$,
where $ph(s') = 1$, $f \in fct(s)\setminus F_{Test1}$
$data(s') = change(val^F_s(f)(data(s)))$

(R1.4) $\delta(s, exec(sb, f)) = (s, \text{“}no\text{”})$ if $sb \neq Pmf$
or $f \notin fct(s)$

To define the state transition rules for *exec* in phase 2, the operational phase, let $s \in \{2\} \times (F \to Val) \times (D \to Val)$. We have

(R2.1) $\delta(s, exec(sb, f)) = (s', out(val^F_s(f)(data(s))))$ if $f \in fct(s)$
where $ph(s') = 2$,
$data(s') = change(val^F_s(f)(data(s)))$

(R2.2) $\delta(s, exec(sb, f)) = (s, \text{“}no\text{”})$ if $f \notin fct(s)$

From the rules above it follows that in phase 2 everyone may execute the functionality that is available on the processor, regardless of the subject's role. However, the rules defined for phase 0 and 1 guarantee that there are no phase 0 and 1 test functions available in phase 2. The rules for phase 2 do not restrict the operating system or application designer in enforcing their own security policy. The means that implementing such a security policy must be part of the application functionality itself and may impose additional requirements on the data object or function state components. The detailed investigation of an application's security policy is, however, beyond the scope of a processor security model.

To complete the transition rules for *exec* we have to consider the error state. Let $e \in \{\infty\} \times (F \to Val) \times (D \to Val)$ be an error state.

(R3.1) $\delta(e, exec(sb, f_{SN})) = (e, SN)$ if $f_{SN} \in fct(e)$

(R3.2) $\delta(e, exec(sb, f)) = (e, \text{“}no\text{”})$ if $f \neq f_{SN}$
or $f_{SN} \notin fct(e)$

In the error state the only executable function is the one revealing the processor's serial number. Any other function calls result in an error message and do not change the state.

Next, we define state transition rules for *load*. For all $s \in Ph \times (F \to Val) \times (D \to Val)$, $sb \in Sb$, $o \in O$, $v \in Val$ we have

(R4.1) $\delta(s, load(sb, o, v)) = (s', \text{“}ok\text{”})$ if $ph(s) = 1$,
where $ph(s') = 1$, $sb = Pmf$,
$fct(s') = fct(s) \cup \{o\}$, $o \in F_{nSec}$
$val^F_{s'} = val^F_s|_{F \setminus \{o\}} \cup \{o \mapsto v\}$

(R4.2) $\delta(s, load(sb, o, v)) = (s, \text{“}no\text{”})$ otherwise

An upload of new functionality is only allowed in phase 1 which is considered to be the phase where the operating system and the application is brought onto the processor hardware. By definition uploadable functions are not security relevant. This emphasises that functions that are security critical on the application or operating system level are not treated as being security relevant within the processor's security model, since they do not refer to the security objectives of the processor itself. Apart from this, rule (R4.1) does not impose further restrictions on the values of o and v. If there exist additional restrictions that, for instance, address the semantics of uploaded functions to avoid undesired side effects, they are formalised by model assumptions as presented in Sect. 4.1. Besides those defined by (R4.1) no other upload attempts are allowed throughout the processor's life cycle, which is formalised by (R4.2).

Finally, the transition rules for *spy* inputs remain to be defined. Recall that our model of the *spy* operation mainly addresses attacks on the physical level whereas (even illegitimate) utilisation of available functions in order to achieve knowledge about objects is covered by the *exec* rules. Typically misuse of executable functions is subject of an operating system or application security policy and does not apply to the processor model. See Sect. 4 for our formalisation of the notion “legitimate access”. Let $s \in Ph \times (F \to Val) \times (D \to Val)$ and $o \in O$.

(R5.1) $\delta(s, spy(o)) = (s, val_s(o))$ if $o \in F_{nSec} \cup D_{nSec}$ and $s \neq e$

(R5.2) $\delta(s, spy(o)) \in \{(e, val_s(o)), (e, \perp), (s, \perp)\}$ if $o \in F_{Sec} \cup D_{Sec}$ and $s \neq e$

(R5.3) $\delta(e, spy(o)) = (e, \perp)$

(R5.1) states that objects not being security relevant may be disclosed anyway without changing the system's state. (R5.2) describes the processor's reaction to *spy* attacks on security critical objects by three cases that may occur non-deterministically. $(e, val_{P_1}(o))$ as successor state models the case of “destructive reading”: since we have to consider even attacks on physical level, e.g. inspecting the silicon with a microscope, we cannot rule out that attacks may reveal information even about security critical objects. Thus, we at least require

that if an attack is successful the processor hardware will be "destroyed", i.e. cannot regularly be used further on. This is modelled by moving to the error state, with rules (R3.1) and (R3.2) formalising what is meant by "cannot regularly be used".

Depending on the kind of attack, we also may enter the case that after attacking a security critical object the processor is not being destroyed. If that occurs, no information is allowed to be disclosed, modelled by $(s, \perp)$ being the successor state. The even stronger case with $(e, \perp)$ as the successor state is included for completeness.

(R5.3) formalises the effect of *spy* attacks if the processor is already in the error state. Since the error state describes destruction of the processor as result of security enforcing functions operating on the physical, or hardware, level, we may reasonably assume that any further attack will require highly extensive resources, and thus is improbable to successfully occur. The security model gives an idealised description by defining that *spy* operations in the error state do not reveal any information regardless of the object being addressed.

The *spy* rules explicitly describe the effects of possible attacks in terms of state change and output. A more abstract modelling alternative would have been to add model assumptions as logical axioms, just like the environment assumptions stated in Sect. 4.1. We chose the above model with respect to subsequent processor design and the informal correspondence arguments, the latter being required by ITSEC E4. In particular, attacks on the physical level are among the most important attacks to be considered during processor design, and a number of security measures have been implemented to counter such attacks.

4 Formalising Security Requirements

4.1 Model Assumptions

In order to exclude pathological cases with respect to environment behaviour, we have to add some assumptions on function objects that are initially available or uploaded during a behaviour of the system. For instance, we have to exclude the case where a hostile operating system or application offers functionality executable by everyone that modifies security relevant functions, or where an application is inadvertently used to store the processor's design description.

We state three axioms that describe expectations about environment behaviours that agree with the security objectives. Since axioms address the environment, they are separated from the system model definition. However, this does not imply that the processor's security functions do not contribute to avoiding undesired consequences in case of the assumptions being violated. For example, initially available security critical functionality f can be stored in the processor's read-only memory component, thus even an $exec(sb, f')$ call of a "hostile" function f' violating axioms 1 and 2 below cannot modify f.

Axiom 1 (Integrity with respect to initially available functions)
Initially available functions only change the data object state, but not $fct(P)$ *itself, i.e. for all* $f \in fct(s_0)$ *and for all* $s \in S$ *we have*

$$\Pi_1(change(val_s^F(f)(data(s))) = val_s^F \ .$$

Axiom 2 (Integrity with respect to non-security-relevant functions)
Functions $f \in F_{nSec}$ *that are not security relevant only change objects that are not security relevant, i.e. elements of* $D_{nInt} \cup F_{nSec}$. *For all* $f \in F_{nSec}$ *and for all* $s \in S$ *we have*

$$\Pi_1(change(val_s^F(f)(data(s)))|_{F_{Sec}} = val_s^F|_{F_{Sec}} \ ,$$
$$\Pi_2(change(val_s^F(f)(data(s)))|_{D_{Int}} = val_s^D|_{D_{Int}} \ .$$

Axiom 3 (Confidentiality) *Let* $b = (s, in, out) \in \|M\|$ *be a behaviour of* M *and* $Val_{s_i}^{F_{Sec}} := \{val_{s_j}^F(f) \,|j < i, f \in F_{Sec}\}$, *for* $i < \sharp s$, *the set of security critical functions that have been available in at least one state up to the* $i-th$ *state transition of* b. *For all* $i \in \mathcal{N}$ *where* $ph(s_i) = 2$ *and* $f \in fct(s_i)$ *we have*

$$out(val_{s_i}^F(f)(data(s_i))) \notin Val_{s_i}^{F_{Sec}} \ .$$

Note that Axiom 3 is stated purely for reasons of being able to conduct the security proofs. The processor's security enforcing functions guarantee that non-security-relevant functions cannot access the security critical functionality. However, to prove validity of the security properties, we also have to consider the case where an application function guesses or accidentally reveals security relevant information. This case is unlikely to occur in practice, but it has to be explicitly excluded in the security model, which is done by adding Axiom 3.

4.2 Security Properties

To complete the formal security model we have to formalise the processor security objectives that have been stated in chapter 2. The formalisation refers to the behaviours of the automaton that describes the system model and is given in definition 9.

Requirement SO1. "The hardware must be protected against unauthorised disclosure of security enforcing functionality."

In our model disclosure of information is defined by an appropriate value occurring in the output sequence of a behaviour. Only the processor manufacturer is allowed to know about the security enforcing functions. If any other subject gets to know about security enforcing functions, e.g. by means of an attack, the processor must be destroyed which in our model means that the error state is entered.

FSO 1 Let $b = (s, in, out) \in \|M\|$ and $Val_{s_i}^{F_{Sec}}$ as defined in Axiom 3. For all $j < \sharp b$ it must hold that

$$out_j \in Val_{s_j}^{F_{Sec}} \quad \Rightarrow \quad (subject(in_j) = Pmf \ \vee \ s_{j+1} = e) \ .$$

Requirement SO2. "The hardware must be protected against unauthorised modification of security enforcing functions."

Modification includes the deletion of functions. In terms of the system model the requirement states that in any behaviour the evaluation of the corresponding function object identifier always returns the same value, if the evaluation is defined for the identifier. Due to our model design, this value must correspond to the initial value of the function object. Deletion of security relevant functions which means removing the corresponding identifier from the evaluation domain is allowed in case of phase transitions. We add an additional requirement that deletion of security functions in states other than the error state is only permitted during phase transitions.

FSO 2.1 Let $b = (s, in, out) \in \|M\|$. For all $i \in \mathcal{N}$ where $s_i \neq e$ it must hold that

$$f \in F_{Sec} \cap fct(s_i) \Rightarrow val^F_{s_i}(f) = val^F_{s_0}(f) \ .$$

FSO 2.2 Let $b = (s, in, out) \in \|M\|$. For all $i \in \mathcal{N}$ where $s_{i+1} \neq e$ it must hold that

$$ph(s_i) = ph(s_{i+1}) \Rightarrow fct(s_i)|_{F_{Sec}} \subseteq fct(s_{i+1})|_{F_{Sec}} \ .$$

Requirement SO3. "The information stored in the processor's memory components must be protected against unauthorised access."

We restrict the formal definition of this requirement to the *spy* operations, since the output of available operating system or application functionality obeys the corresponding security policy which is unknown for the time being. Thus we may not exclude that an application does not consider the output of security relevant data to be harmful. Additionally, it does not suffice to solely consider output sequences, since we have to take into account destructive reading. In order to avoid compromising confidentiality, destructive reading may only occur once in a behaviour, after that the error state must be entered.

FSO 3 Let $b = (s, in, out) \in \|M\|$. For all $i \in \mathcal{N}$; $o \in F_{Sec} \cup D_{Sec}$ it must hold that

$$out_i = val_{s_i}(o) \wedge in_i = spy(o) \Rightarrow$$
$$\{s_j | j > i\} = \{e\} \wedge \{out_j | j < i, in_j = spy(o') \text{ with } o' \in F_{Sec} \cup D_{Sec}\} = \{\perp\}.$$

Requirement SO4. "The information stored in the processor's memory components must be protected against unauthorised modification."

Without knowing about the operating system and application security policy, the requirement can only be interpreted in the sense that any modification resulting from the execution of available functions is considered to be authorised. Here, available means initially existing or being uploaded by performing a *load* operation. It is the task of the operating system or application designer to only upload functions that do not violate their security policy. Besides function execution, operations or state transitions leading to data state changes are only allowed to behave in the desired way.

FSO 4 Let $b = (s, in, out) \in \|M\|$. For all $i \in \mathcal{N}$ it must hold that

$$
\begin{aligned}
& data(s_{i+1}) = data(s_i) \\
\vee\ & \exists sb \in Sb,\ f \in fct(s_i) : in_i = exec(sb, f) \\
& \qquad \wedge\ data(s_{i+1}) = change(val^F_{s_i}(f)(data(s_{i+1}))) \\
\vee\ & ph(s_{i+1}) \neq ph(s_i) \wedge val^D_{s_{i+1}} = val^D_{s_i} \wedge\ val^F_{s_{i+1}}|_{F \setminus F_{Test}} = val^F_{s_i}|_{F \setminus F_{Test}} \\
\vee\ & \exists sb \in Sb,\ f \in F,\ v \in Val : in_i = load(sb, f, v) \\
& \qquad \wedge\ val^F_{s_{i+1}}|_{F \setminus \{f\}} = val^F_{s_i}|_{F \setminus \{f\}} \wedge\ val^D_{s_{i+1}} = val^D_{s_i}\ .
\end{aligned}
$$

Requirement SO5. "It may not occur that test functions are executed in an unauthorised way."

Test functions may only executed by the processor manufacturer. Otherwise, a request for execution will be denied.

FSO 5 Let $b = (s, in, out) \in \|M\|$. For all $i \in \mathcal{N}$, $j = 0, 1$, $sb \in Sb$ it must hold that:

$$
\begin{aligned}
& in_i = exec(sb, f)\ \wedge\ f \in F_{Testj} \quad \Rightarrow \\
& \quad sb = Pmf\ \vee\ (s_{i+1} \in \{s_i, e\} \wedge out_i = \text{“}no\text{”})\ .
\end{aligned}
$$

4.3 Security Proofs

Firstly, it has to be shown that the rules for the state transitions are well-defined, thus providing evidence for model consistency and adequacy. It is easy to check, because the case distinctions in the transition rule definitions are given in a constructive way. Secondly, we prove that the model defined in Definition 9 satisfies the security requirements, i.e. the following theorem holds.

Theorem 1. *Given Axioms 1 to 3, $M = (State, In, Out, S_0, \delta)$ satisfies FSO 1, 2.1, 2.2, 3, 4, 5.*

The proof is carried out either by induction on the length of behaviours (which is admissible since the properties FSO 1, 2.1, 2.2, 3, 4, 5 are safety properties) or by immediately considering the case distinctions of the definitions of the state transition functions. Proof details are left out for reason of space ([11] contains the complete proof). The proof makes also use of auxiliary lemmas like

Lemma 1. *Let $b = (s, in, out) \in \|M\|$. Then it holds for all $i, j \in \mathcal{N}$ with $j \geq i$:*

$$ph(s_j) \geq ph(s_i)\ .$$

i.e. the TOE cannot get into a previous phase, especially there is no exit from the error-state.

5 Discussion

5.1 Classification and Comparison

The "well-known" security models [10] often consider either confidentiality, e.g. Bell-LaPadula [1], Denning [6], Goguen-Meseguer [7], Chinese Wall [9], or integrity, e.g. Biba [3], Clark-Wilson [5]. There are only a few examples like Terry-Wiseman [15] which consider both confidentiality and integrity. The security objectives of our model refer to confidentiality like FSO1 and FSO 3, and to integrity like FSO 2 and FSO 4.

In the area of security modelling there has been a long controversial debate whether the definition of *secure* should refer to states or to state transformations. Bell-LaPadula [1] called a system secure iff each (possible) state is secure. McLean pointed out that "any explication of security based on the notion of a *secure state* must fail ... The concept of a secure system must be explicated as one whose initial state is secure and whose system transform is secure" [12, p. 128]. In our model the security requirements refer to the behaviours $b = (s, in, out) \in State^{\omega} \times In^{\omega} \times Out^{\omega}$ which includes both states and state transformations, e.g., requirement FSO 1 refers to state transformation and FSO 2 refers to states.

The BLP-model introduces an access control matrix as a mechanism to enforce the security requirements. In our model we do not introduce any security mechanisms for two reasons discussed in Sect. 2: level of operation and simplicity by avoiding modelling aids. From the modelling point of view our model description is on a higher level of abstraction and it is thus more flexible.

Note that although our model was designed for a hardware system it could be used for software or hybrid systems as well.

5.2 Practical Aspects and Lessons Learned

The work reported in this paper provides strong evidence that drawing up a formal security model is feasible as well as beneficial with respect to systems operating in a security sensitive environment, even if it is not followed by a formal development process up to some design or implementation level. The complete formal modelling work took about two months, including understanding and discussing the system design and security target, investigating modelling alternatives, discussing the model with the development staff, and supporting the evaluation process. The formal parts made up about ten percent of the whole evaluation and certification effort which was even based on existing development documents. These numbers may serve as an indicator for estimating formal modelling efforts in future evaluation processes.

However, a number of critical factors turned out to influence success and efficiency of formal security modelling. Among them there is the appropriateness of the chosen abstraction level, in particular the decision on implicit or explicit modelling of security enforcing functions. Unfortunately, there is no general methodology that helps to determine which function to explicitly model, and ITSEC leave the decision completely to the model designers. In practice, one should

therefore base the decision on its estimated contribution to understandability, simplicity, and meaningfulness of the model. Take our implicit model of encryption as an example: If we had modelled encryption explicitly, we would have been forced to include additional data types (e.g., keys, random numbers, cryptograms), state components (e.g., key space, encrypted memory) and operations (key generation, key distribution, encryption, decryption) to the system model and an adversary description (including a model of the adversary's "knowledge" like [13] or [14]) to the environment model, thus being able to reduce the proof obligations concerning the *spy* operation to a set of assumptions on sound cryptosystems. We do not consider this model extension as being beneficial compared to our approach of simply stating that *spy* does not reveal useful information to the environment, since increased complexity of the model and proofs only leads to yet another set of assumptions that have to be informally interpreted in terms of the real system. As another example, explicitly modelling the phase concept of processor construction and rollout turned out to be reasonable since security objectives like FSO2.2 immediately refer to the phase model.

In order to decide upon the appropriate abstraction level and to discuss whether the system is faithfully represented (see [2] for the importance of faithful representation) it turned out to be useful to provide a first version of the formal security model as early as possible within the process and to discuss it with the system development staff. This serves two purposes: on one hand it supports the evolutionary development of the model while maintaining the representation relation and its informal interpretation, on the other hand it supports the review and understanding of the security objectives and the system's security architecture even in the early phases of the development.

Difficulties in modelling occur on a detailed level rather than with respect to general modelling concepts. Typical examples are given by incomplete case distinctions, missing parameters, or wrong handling of exceptional cases. It is therefore of crucial importance to conduct proofs showing that the model is consistent and that the abstract security properties hold with respect to the system model. With merely writing down a model errors will certainly occur. Thus proofs are necessary to provide model consistency, avoid errors in modelling details and to check model adequacy. As a prerequisite for meaningful proofs, a separation between the system model and the definition of the security objectives in the tradition of the Bell-LaPadula model is required.

Altogether, we recommend the construction of a formal security model for reasons going beyond achieving ITSEC's E4 correctness level. In particular it is the deeper and better understanding of the system and its security objectives that makes their formalisation valuable. Actually, the formal model turned out to be the key tool for understanding and assessing the security objectives, particularly by giving a precise meaning to fuzzy notions like "legitimate", "allowed", etc. that still occur in the security target. Additionally, the formalisation explicates the limitations of the security model and even the security target specification. Our particular modelling approach is considered to be beneficial mainly because of its high abstraction level that makes the model easy to under-

stand without losing relevance and its constructive approach to the definition of the state transition function which implies completeness with respect to the abstraction level of the model. The cost-benefit ratio turned out to be adequate.

6 Conclusion

In this paper we have introduced a formal security model for a processor. The model differs from those known in the literature in that it must assume an arbitrary set of functions implemented by an operating system or an application and that it cannot refer to a particular security policy definition. Rather, abstract notions like "legitimate access" are formalised in terms of execution of functions of the actual function space and a particular operation *spy* modelling attacks to the hardware that are performed on a physical level. Besides these novel aspects, the formal approach continues a tradition of well-established modelling principles: separation of formal definitions of system model and security objectives, modelling the system as state transition automaton with input and output, and proving the validity of security objectives by inductive proofs.

The model has been defined as part of an ITSEC evaluation process according to quality assurance level E4. However, formal security modelling has turned out to be valuable even if evaluation is not desired. Formal modelling results in a deeper understanding of the system and its security aspects and is a useful tool in system development. Moreover, our experience is that the additional work required by security modelling can be done with reasonable amount of time and money spent. However, some key factors have to be taken into account, with the selection of an appropriate abstraction level and the inclusion of development staff into the modelling process among them.

The impact of formal security modelling can be increased if one can benefit from a particular modelling effort in other applications as well. We expect that our model can be adequately used as a basic model for processor applications of arbitrary kind including, for instance, software virtual machines: it shows a sufficiently high level of abstraction that, for example, does not restrict it to hardware developments, and it does not assume particular properties of an operating system that is to be loaded onto the processor system. The model assumptions included are satisfied by typical operating systems and thus do not restrict the scope. However, future work has to concentrate on adapting and parameterising the model in order to provide a generic model, or a class of models, respectively, for processor applications.

References

[1] Bell, D.E., Len LaPadula. *Secure Computer Systems: Unified Exposition and Multics Interpretation.* NTIS AD-A023588, MTR 2997, ESD-TR-75-306, MITRE Corporation, Bedford, MA, 1976.

[2] Bell, D.E. *Concerning "Modelling" of Computer Security.* Proc. of the IEEE Symp. on Security and Privacy 1988, 8-13.

[3] Biba, K.J. *Integrity Considerations for Secure Computer Systems.* NTIS AD-A039 324, MTR 3153,ESD-TR-76-372, MITRE Corporation, Bedford, MA, 1977.

[4] Broy,M. *Towards a Logical Basis for Systems Engineering.* Working Material of the 1998 Marktoberdorf Summer School on Calculational System Design 1998. Also to appear in: Broy, M.(ed.) *Calculational System Design.* Springer Verlag, NATO ASI Series F.

[5] Clark, D.C., D.R. Wilson. *Evolution of a Model for Computer Integrity.* Report of the Invitational Workshop on Data Integrity, NIST Publ. 500-168, 1989, Sect. A2, 1-3.

[6] Denning, D.E. *A Lattice Model of Secure Information Flow.* Comm. ACM Vol. 19, No. 5 (1976), 236-243.

[7] Goguen, J.A., J. Meseguer. *Security Policies and Security Models.* Proc. of the IEEE Symp. on Security and Privacy 1982, 11-20.

[8] Commission of the European Communities. *Information Technology Security Evaluation Criteria (ITSEC).* June 1991

[9] Kessler, Volker. *On the Chinese Wall Model.* Proc. of the 2nd European Symposium on Research in Computer Security (ESORICS 92) Toulouse, Springer LNCS 648, 41-54.

[10] Kessler, Volker, und Sibylle Mund. *Sicherheitsmodelle - Baupläne für die Entwicklung sicherer Systeme.* Siemens AG, Zentralabteilung Forschung und Entwicklung, München 1993.

[11] Lotz, Volkmar, Volker Kessler, Georg Walter. *Ein formales Sicherheitsmodell.* Part of the evaluation documentation (internal paper), 1999.

[12] McLean, John. *Reasoning about Security Models.* Proc. of the IEEE Symp. on Security and Privacy 1987, 123-131.

[13] Paulson, L.C. *The inductive approach to verifying cryptographic protocols.* J. Computer Security 6, 1998, 85-128.

[14] Schneider, S. *Security Properties and CSP.* Proc. of the IEEE Symp. on Security and Privacy 1996.

[15] Terry, P., S. Wiseman. *A "New" Security Policy Model.* Proc. of the IEEE Symp. on Security and Privacy 1989, 215-228.

[16] Thomas, Wolfgang. *Automata on Infinite Objects.* in: van Leeuwen, Jan (ed.). Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics. Elsevier, Amsterdam, 1990.