A Formal Security Model of the Infineon SLE 88 Smart Card Memory Management

David von Oheimb[1], Georg Walter[2], and Volkmar Lotz[1]

[1] Siemens AG, Corporate Technology, D-81730 Munich
{David.von.Oheimb|Volkmar.Lotz}@siemens.com
[2] Infineon AG, Secure Mobile Solutions, D-81609 Munich
Georg.Walter@infineon.com

Abstract. The Infineon SLE 88 is a smart card processor that offers strong protection mechanisms. One of them is a memory management system, typically used for sandboxing application programs dynamically loaded on the chip. High-level (EAL5+) evaluation of the chip requires a formal security model.
We formally model the memory management system as an Interacting State Machine and prove, using Isabelle/HOL, that the associated security requirements are met. We demonstrate that our approach enables an adequate level of abstraction, which results in an efficient analysis, and points out potential pitfalls like non-injective address translation.

Keywords: Security, formal analysis, smart cards, memory management, Interacting State Machines, Isabelle/HOL.

1 Introduction

Since smart cards are becoming widely spread and are typically used for security-critical applications, smart card vendors face the demand for validating the security functionality of their cards wrt. adequacy and correctness. Third-party evaluation and certification is accepted as the appropriate approach, making it quite an active field. Certification of smart card processor products according to the Common Criteria [CC99] typically refers to the Smartcard IC Platform Protection Profile [AHIP01] and its augmentations like [AHIP02]. Based on these documents, the security target [WN03] for the Infineon SLE 88 smart card chip demands assurance level EAL5 to be achieved, in particular requiring formal reasoning on the requirements level, viz. a formal security model.

Infineon could make use of an extension of the established LKW model [LKW99] which already covers most aspects of security. Yet the SLE 88 offers a new security feature that requires special attention: a sophisticated memory management. For its evaluation we have developed a formal model which we describe in detail in the present article. The upcoming field of multi application smart cards motivate protection of applications from each other. The model shows that this can be effectively achieved with classical hardware based separation of memory areas. This feature may be used in particular within Java Virtual Machine implementations, yielding major progress in the area of dynamically loadable applications for smart cards.

The memory management security model is given in terms of Interacting State Machines (ISMs) introduced in [Ohe02,OL03]. ISMs are state-transition automata that communicate asynchronously on multiple input and output ports and thus can be seen as high-level Input/Output Automata [LT89]. They have turned out to be appropriate for the task of security modeling, for instance the full formalization of the LKW model with the Isabelle theorem prover in [OL02].

Most of related work on high EAL evaluation for smart cards, done e.g. at Trusted Logic and Philips, is unpublished. There are publications dealing with the Java Card runtime system [MT00] and with smart card operating systems in general [SRS+00]. Note that these focus on software, while we focus on hardware.

2 SLE 88 Memory Management

In this section we introduce the virtual memory system of the SLE 88 with their associated security objectives and protection mechanisms.

2.1 Memory Organization

The physical memory of the SLE88 family is handled via 22 bit *physical effective addresses (PEAs)*. Virtual memory is addressed via 32 bit *virtual effective addresses (VEAs)*. The atomic units of the translation from virtual to physical addresses are *pages* of 64 bytes, which results in 6 bit wide *displacements*, i.e., address offsets. Peripheral hardware is memory mapped and thus can be accessed — and protected — in the same way as ordinary memory cells.

Typically, there are several independent software modules of different origin. Therefore, virtual memory is logically divided into 256 *packages* of equal size, such that the *package address (PAD)* makes up the upper 8 bits of the VEA. Packages 0 to 2 are *privileged* because they control security-critical entities. Package 0 contains the *security layer (SL)*, package 1 contains the *platform support layer (PSL)*, also known as *hardware abstraction layer (HAL)*, and package 2 contains the *operating system (OS)*. Of the remaining *regular* packages, those with numbers 3 to 15 are reserved, while those with numbers 16 to 255 are available for (third-party) application software to be uploaded on demand.

2.2 Security Requirements

The security objective relevant here is O.Mem-Access: "Area based Memory Access Control", defined in [WN03, §4.1]:

The TOE must provide the Smartcard Embedded Software with the capability to define restricted access memory areas. The TOE must then enforce the partitioning of such memory areas so that access of software to memory areas is controlled as required, for example, in a multi-application environment.

This means in particular that inter-package access to code and data should be restricted and that the corresponding protection attributes should be controlled by (specially protected) privileged packages only. Detail on the associated *Security Functional Requirements (SFRs)* may be found in [WN03, §5.1.1.2].

2.3 Protection Mechanisms

Virtual memory is associated with *effective access rights (EARs)* determining the read, write, and execute access of packages. Their granularity is 256 bytes, corresponding to the lower 8 bits of the VEAs. Moreover, each physical *page block* of 16 bytes, corresponding to the lower 4 bits of the PEA, is associated with additional security attributes referred to as *block protection field (BPF)*. The only information we will need in the model is a predicate called *PASL* specifying whether a page block should be accessible by SL only.

An EAR is given by a two-letter code where each of the letters may be `W`, `R`, `X`, or `-`, which specify read/write access to data, read-only access to data, executing access to code, and no access, respectively. The first letter refers to access within a package, while the second letter refers to access of one package (the source) to some other package (the target). The only allowed combinations are `WW`, `WR`, `RR`, `W-`, `R-`, and `X-`. Note that the EAR gives an implicit classification of memory sections as code or data. Code can be marked only with `X-`, which indicates that inter-package code fetch is generally prohibited. Regardless of the EAR, privileged packages have free data access to all other packages except SL.

Apart from the restrictions on (linear) code fetch, inter-package control transfer is allowed only if the target holds a special PORT instruction sequence that defines the set of packages allowed to enter.

3 Formalism and Tools

For modeling (and partially verifying) the SLE 88 memory management, we take the ISM [OL02] approach. This means that we formally define and analyze its security model as an Interacting State Machine (ISM) [Ohe02,OL03] within the theorem prover Isabelle/HOL [Pau94].

Interacting State Machines (ISMs) are automata whose state transitions may involve multiple input and output simultaneously on any number of ports. For the SLE 88 security model we need only a single basic ISM with one input port accepting machine (micro-)instructions and one output port issuing the reaction of the memory management system. What we do make use of is single state transitions as well as transition sequences which result from system runs.

Isabelle is a generic theorem prover that has been instantiated to many logics, in particular the very practical *Higher-Order Logic (HOL)*. HOL [PNW+] is a predicate logic based on the simply-typed λ-calculus and thus in a sense combines logical and functional programming. Proofs are conducted primarily in an interactive fashion assisted by automatic and semi-automatic tactics.

4 System Model

In order to provide an comprehensive instructive presentation of the formal model, we reproduce all the definitions, lemmas and theorems (essentially leaving out proof scripts and a few other parts needed for technical reasons only),

augmented with comments, just as they appear in the Isabelle theory sources[3]. By employing the automatic LaTeX typesetting facility of Isabelle, we achieve on the one hand maximal accuracy of the presentation, retaining the mathematical rigor and the "flavor"[4] of the machine-checked specifications, and on the other hand good readability by using standard logical notation as far as possible and interspersing textual explanations and motivation.

A very important design principle is to keep a high level of abstraction, which improves readability and simplifies the proofs. Therefore, we model only those features that are strictly relevant for security, abstracting away unnecessary detail caused e.g. by efficiency optimizations. For the same reason we often use a modeling technique called underspecification, i.e. for part of the logical types and constants we do not give full definitions but only declarations of their names.

4.1 Overview

Following the standard approach to security analysis, we provide a system model describing the abstract behavior of the memory management and formalize the security objectives as properties of the system model. The state-based ISM approach fits well with modeling both the page table and the physical memory as state components of the system, mapping virtual addresses to physical addresses and physical addresses to values. Our specification of (both virtual and physical) addresses represents the structure of the memory organization as described in §2.1. Values are only of interest in case of PORT instructions; we leave other values unspecified. The model further includes mappings describing the assignments of BPFs to page blocks and EARs to sections of the virtual address space.

To complete the system model, state transitions represent the different kinds of memory access that may occur. For each of them there is a corresponding input message for the ISM triggering a transition. Each transition produces an output stating whether the access is granted or denied. In case of denial, we have different output messages representing the different traps or alarms. The computation of the output refers to our formalization of the protection rules stated in §2.3. A transition may also result in modifications of state components, for instance, write access to the main memory or page table updates.

4.2 Addressing

First we have to define several aspects of the SLE 88 address space introduced in §2.1. These include the type of package addresses, *PAD*, defined as the disjoint sum of privileged and regular PADs, where we enumerate all three possibilities for privileged packages but do not specify the actual range of regular PADs:

datatype *pri_PAD = SL | PSL | OS* — package addresses 0 - 2
typedecl *reg_PAD* — package addresses 3 - 255
datatype *PAD = Pri pri_PAD | Reg reg_PAD* — package addresses, priv. or regular

[3] Isabelle/HOL adopts the notational standards of functional programming, writing for instance (multi-argument) function application as *f x y z* instead of $f(x, y, z)$.
[4] For example, the order of definitions is strictly bottom-up.

Next, we define a predicate distinguishing privileged from regular packages.

```
consts    is_Pri :: "PAD ⇒ bool"
primrec "is_Pri (Pri p) = True"
        "is_Pri (Reg r) = False"
```

While PADs form the upper (i.e., most significant) part of virtual addresses, displacements *DP* form the lower sections used for addressing individual bytes of memory within a page. We need to split them further because there are four page blocks within a page that are associated with their own BPFs. Note that despite the names that contain numbers giving bit positions, we do not actually specify the concrete ranges of the types declared but just state that *DP* is the Cartesian product of the two other types:

```
typedecl DP_lo      — 4-bit offset within page block (with same BPF)
typedecl DP_hi      — 2-bit page block address within page
types    DP         — 6-bit displacement within VEAs and PEAs
                    = "DP_hi × DP_lo"
```

A virtual effective address consists of the package address, a middle part that we call *VEA_mid*, and the displacement. We have to further split the middle part because only the upper 16 bits of it are used to determine the EAR associated with the address. We also define the type *VP* of virtual page pointers which will be mapped to physical page pointers.

```
typedecl VEA_mid_lo — 2-bit part of VEA_mid with identical EARs
typedecl VEA_mid_hi — 16-bit part of VEA_mid with different EARs
types    VEA_mid    — 18-bit middle part of VEA
                    = "VEA_mid_hi × VEA_mid_lo"
         VEA_dEAR   — 24-bit upper part of VEA determining EARs
                    = "PAD × VEA_mid_hi"
         VEA        — 32-bit virtual effective address
                    = "PAD × VEA_mid × DP"
         VP         — 26-bit virtual page pointer
                    = "PAD × VEA_mid"
```

Physical page pointers *PP* are combined with displacements to form physical effective addresses. The part of PEAs determining the BPF is called *PEA_dBPF*.

```
typedecl PP         — 16-bit physical page pointer
types    PEA_dBPF   — 18-bit page block address determining the BPF
                    = "PP × DP_hi"
         PEA        — 22-bit physical effective address
                    = "PP × DP"
```

We define an auxiliary function *PAD* extracting the package information from any address containing a PAD as its uppermost part, simply by projecting on this first part of the tuple: `PAD (pad,x) = pad`

4.3 Effective Access Rights

We enumerate all allowed EARs as defined in §2.3 and relate them with the access that they grant by functions for intra-package and inter-package access.

```
datatype EAR = WW | WR | RR | Wn ("W-") | Rn ("R-") | Code ("X-")
datatype access_mode = Read | Write | Execute
consts  — access modes for read/write operations
  RWX_own   :: "EAR ⇒ access_mode set"
  RWX_other :: "EAR ⇒ access_mode set"
primrec  — intra-package access
 "RWX_own   WW   = {Read, Write}"
 "RWX_own   WR   = {Read, Write}"
 "RWX_own   RR   = {Read}"
 "RWX_own   W-   = {Read, Write}"
 "RWX_own   R-   = {Read}"
 "RWX_own   X-   = {Execute}"
primrec  — inter-package access
 "RWX_other WW   = {Read, Write}"
 "RWX_other WR   = {Read}"
 "RWX_other RR   = {Read}"
 "RWX_other W-   = {}"
 "RWX_other R-   = {}"
 "RWX_other X-   = {}"
```

4.4 State

Our abstract model of the SLE 88 memory management state consists of three aspects that are crucial for the security analysis:

- the physical memory contents (where the only sort of value we are interested in is a PORT instruction sequence with its associated set of packages) and the PASL predicate associated with page blocks
- the essentials of the page table entries, i.e. page mapping and EARs – there is no need for us to model complex structures like translation lookaside buffers and multi-level page tables required merely for optimization
- the package information contained in the current program counter and in the return address stack

For simplicity, we model PORT instructions as atomic values. We define them as one of the alternatives in a (free) datatype, which implies that they can be distinguished from all other instructions. This is adequate because the SLE 88 instruction layout ensures that PORT instructions are uniquely determined.

```
typedecl value'
datatype value = PORT "PAD set"  — specifying the packages permitted to enter
               | Other_value value'
```

The abstract state itself is defined as a record. Each of the field names induces a corresponding selector function whose first argument is a value, typically called *s*, of type *state*.

```
record state =
— abstraction of physical memory:
  memory    :: "PEA       ⇒ value"  — including peripherals
  BPF_PASL  :: "PEA_dBPF ⇒ bool"   — BPF stating SL-only access to page blocks
— abstraction of page table (package descriptions and translation lookaside buffers):
  PT_map    :: "VP        ⇝ PP"     — page mapping, relative to packages
  PT_EAR    :: "VEA_dEAR ⇒ EAR"     — EARs for 256-byte sections
— abstraction of execution state:
  curr_PAD  :: "PAD"                — currently executing package
  stack     :: "PAD list"           — package part of return addresses
consts s0 :: state  — the initial state
```

The state components *BPF_PASL*, *PT_map*, and *PT_EAR* each define a mapping only for the relevant sections of physical and virtual addresses, which helps to avoid redundancies in particular for update operations. Yet it is often convenient to perform the lookup operation with a full PEA or VEA, respectively. The auxiliary functions *BP_PASL*, *PEA*, and *EAR*, respectively, provide these liftings.

4.5 Assumed Initial State Properties

The security target [WN03] requires that all EARs should be initialized with a reasonable value. Since the exact value is immaterial for our analysis, we apply the standard technique, viz. to declare a constant giving the default EAR of memory sections without actually defining its value.

```
consts default_EAR :: "EAR"  — underspecified
```

The functional specification requires that only the PSL package may call the SL package, which restricts the sets of packages within PORT instructions of SL. We specify this for the initial state *s0* with the following axiom:

```
axioms  — checks by PORT instructions of SL
 init_PORT_SL: "PEA s0 (Pri SL, la) = Some pa ⟹
   memory s0 pa = PORT PADs ⟹ PADs ⊆ {Pri SL,Pri PSL}"
```

The axiom can be read as follows. For any VEA that belongs to SL (i.e., has the form *(Pri SL, la)* for some *la*), if in the initial state it is mapped to any PEA *pa* and a PORT instruction is stored at that address, then the associated set *PADs* of allowed packages may contain only SL and PSL.

A further important requirement is that the BPFs are reasonably set: for any physical pointer *pp* and page block address, PASL should be true iff the page block is owned by SL, i.e. *pp* is associated with some VEA belonging to SL:

```
axioms init_BPF_PASL:
  "BPF_PASL s0 (pp,pb) = (∃ la. PT_map s0 (Pri SL,la) = Some pp)"
axioms init_PT_EAR: "PT_EAR s0 ea = default_EAR"
```

It is evident that the processor should start executing in the SL package with an empty return stack. Though we do not actually need these properties in our proofs, we state them for symmetry:

axioms

```
  init_PAD:   "curr_PAD s0 = Pri SL"  — unused
  init_stack: "stack    s0 = []"      — unused
```

4.6 Aliasing via Page Table

By the construction of the page table mapping, there is the possibility that the mapping is non-injective, i.e., that multiple VEAs refer to the same PEA. The MMU device driver in the PSL package avoids such aliasing, but the page table may be manipulated directly by privileged packages in order to meet extraordinary needs for inter-package sharing. Our model is general enough to handle also such forms of aliasing. Naturally, in such cases the guarantees that can be made are weaker. In particular, conflicting EARs may arise, for example if a certain memory page is mapped for two different packages where one package sets the EAR such that all others should not be able to write to that memory page, while the other package claims to have write access by setting its EAR accordingly. We have identified a predicate on the page table contents that specifies conditions as weak as possible but still guaranteeing inter-package consistency of EARs: if two different packages `p` and `p'` happen to map the same memory page then the EARs associated with that page should be both `WW` or both `RR`.

constdefs

```
  EARs_consistent :: "state ⇒ bool"
 "EARs_consistent s ≡ ∀p p' vea_mid_hi vea_mid_hi' lo lo'.
        PT_map s (p,vea_mid_hi,lo) = PT_map s (p',vea_mid_hi',lo') ⟶
        PT_map s (p,vea_mid_hi,lo) = None ∨ p = p' ∨
        PT_EAR s (p,vea_mid_hi) = WW ∧ PT_EAR s (p',vea_mid_hi') = WW ∨
        PT_EAR s (p,vea_mid_hi) = RR ∧ PT_EAR s (p',vea_mid_hi') = RR"
```

4.7 Interface

We define the SLE 88 memory management system as an Interacting State Machine (ISM) with a rather trivial interface: it has one input port named `In` and one output port named `Out`.

datatype `interface = In | Out`

The messages exchanged with the environment are either instructions given to the system or results sent by the system. The instructions are abstractions of the usual CPU (micro-)instructions where we focus on code fetch (which is the first step of each instruction execution), memory read and write, various forms of branches, and write operations to various special registers including the page

table. The chip may respond with positive or negative acknowledge or various traps (which will be explained where appropriate) in case of denied access.

```
datatype message =
  Code_Fetch  VEA  — is meant to precede each other type of instruction
— read/write operations:
| Read_Mem    VEA
| Write_Mem   VEA value
— control transfer operations:
| Jump VEA
| Call VEA
| Return
| Write_RetAddr   VEA
— operations for setting security attributes and page table entries:
| Write_BPF_PASL  PEA_dBPF  bool
| Write_PT_EAR    VEA_dEAR  EAR
| Write_PT_map    VP        "PP option"
— outcome of operations:
| Ok | No  — access granted or denied without generating a trap
| MPA | MPSF | RLCP | MPBF | PRIV | MCR  — traps
```

4.8 Auxiliary Access Functions

For modeling the access control checks performed when executing access operations, it is beneficial to factor out common behavior and to reduce the complexity of the associated system transitions by defining dedicated auxiliary functions.

The function `mem_access` takes as its arguments the access mode, the current system state `s`, and the virtual address `va` to be accessed. It determines whether the current package, which is the subject (called *source*) of the operation at hand, is allowed to access — in the given mode — the package given by the PAD of `va`, which is the object of the operation (called *target*). In particular, it checks whether

- the virtual address is mapped to some existing PEA `pa` (and otherwise causes a Memory Protection Package Boundary Fault trap)
- the source is privileged and performs a read or write access where the target is some other package except SL [5], or the EAR associated with `va` allows access with the given mode, making the distinction if the access is local or to some other package (and otherwise causes a Memory Protection Access Violation or MPBF trap)
- PASL is true for `pa` iff the target is SL (which checks consistency of the PASL setting), or SL accesses data – for testing purposes – in a page block not belonging to SL where PASL is true (and otherwise causes a Memory Protection Security Field trap).

```
constdefs  — read/write access restrictions to main memory
 mem_access :: "access_mode ⇒ state ⇒ VEA ⇒ message"
```

[5] This operation is typical for e.g. the operating system loading a package

```
"mem_access mode s va ≡ case PEA s va of None ⇒ MPBF | Some pa ⇒
  let source = curr_PAD s; target = PAD va in
  (if is_Pri source ∧ mode≠Execute ∧ source≠target ∧ target≠Pri SL ∨
      mode ∈ (if source=target then RWX_own else RWX_other) (EAR s va)
   then (if ((BP_PASL s pa = (target = Pri SL)) ∨ BP_PASL s pa ∧
                  source = Pri SL ∧ mode ≠ Execute ∧ target ≠ Pri SL)
         then Ok else MPSF)
   else (if mode ≠ Execute then MPA else MPBF))"
```

The function `Call_access` takes as its arguments the current system state `s` and the virtual address `va` to be called. It grants intra-package calls (i.e., the PAD of the target `va` equals the current PAD), and otherwise checks whether

- `va` is mapped to some PEA `pa` (and otherwise causes a Memory Protection Package Boundary Fault trap)
- the value stored at `pa` is a PORT instruction (and otherwise causes a Privileged Instruction trap)
- the PORT instruction allows the current package to enter (and otherwise typically just returns from the call without causing a trap).

constdefs — restrictions for procedure calls

```
  Call_access :: "state ⇒ VEA ⇒ message"
 "Call_access s va ≡ if PAD va = curr_PAD s then Ok else
      case PEA s va of None ⇒ MPBF | Some pa ⇒
     (case memory s pa of PORT PADs ⇒
      if curr_PAD s ∈ PADs then Ok else No | Other_value v ⇒ PRIV)"
```

The function `Write_PT_access` takes as its arguments the current system state `s` and the package to be affected. Writing to the page table is granted only if the current package is privileged. It must be even SL if the target package is SL. Otherwise a Memory Protection Core Register Address trap is generated.

constdefs — restrictions for writing page table information

```
 Write_PT_access :: "state ⇒ PAD ⇒ message"
"Write_PT_access s target ≡ if (target=Pri SL ⟶ curr_PAD s=Pri SL) ∧
                              is_Pri (curr_PAD s) then Ok else MCR"
```

4.9 Transitions

The core of our security model is the definition of the ISM that specifies the overall memory management system of the SLE 88. For each kind of instruction that may be issued (by sending it to the ISM) there is one transition rule. Transitions are atomic and instruction execution is meant to be sequential. The system reacts by outputting a value that indicates granted or denied access, where the latter typically leads to a trap. In our abstract model there is no need to specify trap handling. A couple of the transition rules have preconditions, and most of them have postconditions specifying changes to (part of) the system

state. Since conditional changes to mappings are very common, we define the syntactic abbreviation "`c ? f(x:=y)`" ⇀ "`if c then f(x := y) else f`".

ism *SLE88_MM =*
  ports *interface*
    inputs *"{In}"*
    outputs *"{Out}"*
  messages *message* — instructions received or indications of success sent
  states data *state* **init** *"s0"* **name** *"s"* — the initial state is *s0*
  transitions
Code_Fetch: — Okay if the PAD of `va` equals the current PAD and has the EAR `X`- and `va` is mapped to some page block where PASL is true iff the current PAD is SL.
  in *In "[Code_Fetch va]"*
  out *Out "[mem_access Execute s va]"*
Read_Mem:
  in *In "[Read_Mem va]"*
  out *Out "[mem_access Read s va]"*
Write_Mem: — Sets the memory cell at address `va` to the value `v` by the value `v` if access is granted. If the target package is SL and PASL is false for the affected page block, it may non-deterministically – as specified using the free variable *belated_MPSF* – write the value even though the access is denied, namely if the MPSF trap is delayed.
  in *In "[Write_Mem va v]"*
  out *Out "[mem_access Write s va]"*
  post *memory := "(mem_access Write s va = Ok ∨*
mem_access Write s va = MPSF ∧ belated_MPSF ∧
PAD va = Pri SL ∧ ¬BP_PASL s (the (PEA s va))) ?
(memory s)(the (PEA s va) := v)"
Jump: — Only intra-package jumps are permitted.
  in *In "[Jump va]"*
  out *Out "[if PAD va = curr_PAD s then Ok else MPA]"*
Call: — If the call is allowed then the current PAD is updated and its old value is pushed on the abstract return stack.
  in *In "[Call va]"*
  out *Out "[Call_access s va]"*
  post *curr_PAD:="if Call_access s va = Ok then PAD va else curr_PAD s",*
stack :="if Call_access s va = Ok then curr_PAD s#stack s else
stack s"
Return: — The first precondition states that the stack is non-empty with top element `r` while the second precondition just gives an abbreviation. A return into SL is not allowed, causing a Return Leave Current Package Mistake trap, otherwise `r` is popped from the stack and becomes the new current PAD.
  pre *"stack s = r#rs", "ok = (r = Pri SL ⟶ curr_PAD s = Pri SL)"*
  in *In "[Return]"*
  out *Out "[if ok then Ok else RLCP]"*
  post *curr_PAD := "if ok then r else curr_PAD s",*
stack := "if ok then rs else stack s"

Write_RetAddr: — Setting the return address, i.e. the stack top, to an address whose PAD is different from the current one is possible only for privileged packages.

pre *"stack s = r#rs", "ok = (PAD va=curr_PAD s ∨ is_Pri (curr_PAD s))"*
in *In "[Write_RetAddr va]"*
out *Out "[if ok then Ok else No]"*
post *stack := "if ok then (PAD va)#rs else stack s"*

Write_BPF_PASL: — Only SL is allowed to change the block protection field.

in *In "[Write_BPF_PASL ba b]"*
out *Out "[if curr_PAD s = Pri SL then Ok else MCR]"*
post *BPF_PASL := "curr_PAD s = Pri SL ? (BPF_PASL s)(ba:=b)"*

Write_PT_EAR:
in *In "[Write_PT_EAR ea e]"*
out *Out "[Write_PT_access s (PAD ea)]"*
post *PT_EAR := "Write_PT_access s (PAD ea) = Ok ? (PT_EAR s)(ea:=e)"*

Write_PT_map:
in *In "[Write_PT_map vp ppo]"*
out *Out "[Write_PT_access s (PAD vp)]"*
post *PT_map := "Write_PT_access s (PAD vp) = Ok ? (PT_map s)(vp:=ppo)"*

Having given all the above definitions, we use them for stating and proving security properties. Many of these require additional assumptions on reasonable behavior of the SL package, which we will give as additional axioms restricting the transitions of the ISM.

5 Security Properties

5.1 Overview

Given the system model in the form of an ISM, we are ready to formalize the security requirements of §2.2 as properties of (sequences of) ISM state transitions. Since the security requirements are formulated on a very high level, expressing the properties and arguing for their completeness has been appropriately done by discussing them with the requirement engineers, taking into account the SLE 88 specifications and the justifications given in the security target, which define details like access modes, EARs, the PASL attribute, and their intended effect.

The main concern of the security requirements is separation of applications, i.e., suitable restriction of inter-package access, which we address by the theorems

- *interpackage_Read_Mem_respects_EAR* and *interpackage_Write_Mem_respects_EAR*, addressing inter-package read/write protection, described in §5.2, and
- *interpackage_transfer_only_via_Call_to_suitable_PORT_or_Return*, addressing inter-package control transfer, described in §5.4.

Another critical issue is the special protection of the SL package because it manages the security attributes, onto which access control is based. By stating the series of theorems given in §5.3 culminating in *only_SL_changes_SL_memory* and *only_SL_reads_SL_memory*, and in §5.4 culminating in *only_PSL_enters_SL*, we have covered all properties implied by the security requirements.

Proving that the theorems hold for the given system model completes the formal security analysis. The proofs show some inherent complexity, for instance by having to consider layered protection mechanisms and effects of aliasing, i.e., non-injective page table mappings. Still, due to adequate modeling and the powerful Isabelle proof system, developing the machine-checked proofs has been a matter of just a few days.

The act of conducting proofs identifies necessary assumptions concerning the initial state and the access control attribute settings for the SL package. In particular, we introduce a notion of consistency of EAR assignments that is useful in case of aliasing.

5.2 Inter-package Read/Write Protection

Our first two theorems state basic properties of inter-package read and write access. If in any state *s*, a read instruction for some virtual address *va* not belonging to the current package is successful, then this has been done by a privileged package accessing a package other than SL, or read (or read/write) access is granted by the EAR associated with *va*. Note that the access rights are determined at the virtual (not: physical) address level, which opens up the possibility of inconsistencies incurred by aliasing, i.e. different access paths to the same physical memory area. In effect, the accessibility of a memory area is determined by the minimum protection of all related EARs. Only if inter-package consistency of the EARs is ensured, we can guarantee that for any other virtual address *va'* belonging to a different package and mapped to the same physical address, the associated EAR is the same (and cannot be *WR* because this EAR is not symmetric) and thus no unwanted access is possible.

theorem *interpackage_Read_Mem_respects_EAR: "⋀va va'.*
⟦((p,s),c,(p',s')) ∈ Trans; hd (p In) = Read_Mem va; hd (p' Out) = Ok;
PAD va ≠ curr_PAD s⟧ ⟹ is_Pri (curr_PAD s) ∧ PAD va ≠ Pri SL ∨
(EAR s va = WW ∨ EAR s va = WR ∨ EAR s va = RR) ∧
(EARs_consistent s ⟶ PEA s va' = PEA s va ⟶ PAD va ≠ PAD va' ⟶
EAR s va' = EAR s va ∧ EAR s va ≠ WR)"

Some notational remarks are advisable here: in Isabelle formulas, '⋀' is a universal quantifier; multiple premises are bracketed using '⟦' and '⟧' and separated using ';'. The term *hd (p In)* refers to the input and *hd (p' Out)* to the output of the transition *((p,s),c,(p',s'))* which takes the state *s* to *s'*.
The proof of this theorem proceeds by case distinction on the transition rules. The only non-trivial case is the one of *Read_Mem* where we unfold the definitions of *mem_access*, *RWX_other*, and *EARs_consistent* and perform standard predicate-logical reasoning and term rewriting.

The analogous theorem concerning the write instruction is a bit simpler because there are less cases that allow write access:

theorem *interpackage_Write_Mem_respects_EAR: "⋀va va'.*
⟦((p,s),c,(p',s')) ∈ Trans; hd (p In)=Write_Mem va v; hd (p' Out)=Ok;
PAD va ≠ curr_PAD s⟧ ⟹ is_Pri (curr_PAD s) ∧ PAD va ≠ Pri SL ∨

```
EAR s va = WW ∧ (EARs_consistent s ⟶ PEA s va' = PEA s va ⟶
                 PAD va ≠ PAD va' ⟶ EAR s va' = WW)"
```

5.3 Read/Write Protection for SL Memory

The next bunch of lemmas and theorems focus on the protection of the memory areas of the SL package.

Only SL may change the mapping of PEAs belonging to SL. More precisely, for any sequence of transitions *ts* that may result from a system run and any state transition from *s* to *s'* within it, unless the current package is SL, the page table mapping concerning SL is the same for *s* and *s'*. This is a simple consequence of the definition of *Write_PT_access* used in the rule *Write_PT_map*.

theorem *only_SL_changes_PT_map_of_SL:*
```
"⟦ts ∈ TRuns; ((p,s),c,(p',s')) ∈ set ts; curr_PAD s ≠ Pri SL⟧ ⟹
  PT_map s (Pri SL, lvp) = PT_map s' (Pri SL, lvp)"
```

The analogous property holds for the EARs associated with SL memory:

theorem *only_SL_changes_EAR_of_SL:*
```
"⟦ts ∈ TRuns; ((p,s),c,(p',s')) ∈ set ts; curr_PAD s ≠ Pri SL⟧ ⟹
  EAR s (Pri SL, lva) = EAR s' (Pri SL, lva)"
```

Similarly, only privileged packages may change EARs:

theorem *only_Pri_change_EAR:*
```
"⟦ts ∈ TRuns; ((p,s),c,(p',s')) ∈ set ts; ¬is_Pri (curr_PAD s)⟧ ⟹
  EAR s va = EAR s' va"
```

Later we will need an invariant stating that the EARs associated with SL deny any access by other packages. In order to establish this property, we have to assume that the default EAR denies such access as well and that SL sticks to this policy when writing EARs:

axioms *default_EAR_denies_RWX_other:* `"RWX_other default_EAR = {}"`

axioms *Write_PT_EAR_consistent_with_denial_of_RWX_other_for_SL_memory:*
```
"⟦((p,s),c,(p',s')) ∈ Trans; curr_PAD s = Pri SL;
  hd (p In) = Write_PT_EAR (Pri SL, lva) e; hd (p' Out) = Ok⟧ ⟹
  RWX_other e = {}"
```

The necessity of such axioms makes explicit some important assumptions on the initialization of security attributes and the behavior of SL and therefore gives valuable feedback for system software development.

With the help of the two axioms just given and the axiom *init_PT_EAR* given in §4.5, we can prove the invariant easily by induction on the length of transition sequences. In terms of the Isabelle/HOL implementation of ISMs, this invariant can be expressed in the following compact way:

lemma *SL_pages_deny_RWX_other:*
```
"Inv (λs. ∀ lva. RWX_other (EAR s (Pri SL, lva)) = {})"
```

It reads as follows. For any reachable state *s* and any virtual address within the SL package, the associated EARs for other packages is the empty set.
Similar comments apply to the invariant that PASL is true for all memory belonging to SL. It requires the additional assumptions that SL writes the block protection fields and the page table entries for its memory only in a way such that PASL remains true:

axioms *Write_BPF_PASL_consistent_for_SL_memory:*
```
"⟦((p,s),c,(p',s')) ∈ Trans; curr_PAD s = Pri SL;
  hd (p In) = Write_BPF_PASL (pp,dp') b; hd (p' Out) = Ok;
  PT_map s (Pri SL, lvp) = Some pp⟧ ⟹ b = True"
```

axioms *Write_PT_map_consistent_with_BP_PASL_for_SL_memory:*
```
"⟦((p,s),c,(p',s')) ∈ Trans; curr_PAD s = Pri SL;
  hd (p In) = Write_PT_map (Pri SL, lvp) (Some pp); hd (p' Out) = Ok⟧ ⟹
  BP_PASL s (pp,dp)"
```

Together with the axiom *init_BPF_PASL* also given in §4.5, we can prove the invariant in an analogous way.

lemma *SL_memory_has_PASL:*
```
 "Inv (λs. ∀lva pa dp. PEA s (Pri SL, lva) = Some pa ⟶ BP_PASL s pa)"
```

Taking advantage of the two invariance lemmas just given, we prove that only SL can change memory allocated to SL. The proof uses the invariant *SL_memory_has_PASL* concerning PASL three times, where in all these cases there is aliasing in the page table such that the same physical memory area is allocated to both SL and some non-SL package. Thus we can conclude that PASL plays an important role for detecting such (unwanted) aliasing wrt. SL memory.

theorem *only_SL_changes_SL_memory:*
```
"⟦ts ∈ TRuns; ((p,s),c,(p',s')) ∈ set ts; curr_PAD s ≠ Pri SL;
  PEA s (Pri SL, lva) = Some pa⟧ ⟹ memory s pa = memory s' pa"
```

The theorem stating that only SL can read memory allocated to SL requires only the invariant *SL_pages_deny_RWX_other* concerning EARs of SL:

theorem *only_SL_reads_SL_memory:*
```
"⟦ts ∈ TRuns; ((p,s),c,(p',s')) ∈ set ts;
  hd (p In) = Read_Mem (Pri SL, lva);  hd (p' Out) = Ok⟧ ⟹
  curr_PAD s = Pri SL"
```

5.4 Inter-package Control Transfer and PORT Instructions

As can be derived easily from the transition rule for the *Code_Fetch* operation and the definition of *mem_access*, code may be executed only from memory that belongs to the current package and that is marked with the EAR *X-*:

theorem *Code_Fetch_only_local_X:*
```
"⟦((p,s),c,(p',s')) ∈ Trans; hd (p In) = Code_Fetch va; hd (p' Out) = Ok⟧
 ⟹ PAD va = curr_PAD s ∧ EAR s va = X-"
```

Thus, the only form of inter-package code access to be further addressed is transfer of control where the current package changes.

Our next theorem states that the only possibilities for such control transfer is a legal procedure call or return; in more detail: if there is a transition from state *s* to *s'* where the current package changes, then either it has been caused by a call whose target is a virtual address *va* mapped to a physical address *pa* containing a PORT instruction that explicitly allows the calling package to enter, or it has been caused by a return to a package other than SL:

theorem *interpackage_transfer_only_via_Call_to_suitable_PORT_or_Return:*
```
"⟦((p,s),c,(p',s')) ∈ Trans; curr_PAD s' ≠ curr_PAD s⟧ ⟹
  (∃va pa PADs. hd (p In) = Call va ∧ PEA s va = Some pa ∧
                memory s pa = PORT PADs ∧ curr_PAD s ∈ PADs) ∨
  (∃r rs.       hd (p In) = Return ∧ stack s = r#rs ∧ r ≠ Pri SL)"
```

The proof of this theorem is straightforward by case distinction on all instructions available and unfolding the definition of *Call_access*.

Much more involved is the proof of our final theorem stating that only PSL can enter SL: we need an invariant that all PORT instructions contained in memory allocated to SL allow only calls by SL itself and by PSL. This in turn requires two assumptions that SL writes memory allocated to itself and the page table entries for its memory only in a way such that the invariant is maintained:

axioms *Write_Mem_PORT_to_SL_only_SL_PSL:*
```
"⟦((p,s),c,(p',s')) ∈ Trans; curr_PAD s = Pri SL;
  hd (p In) = Write_Mem va (PORT PADs); hd (p' Out) = Ok;
  PEA s va = PEA s (Pri SL, lva)⟧ ⟹ PADs ⊆ {Pri SL, Pri PSL}"
```

axioms *Write_PT_map_pointing_to_PORT_only_SL_PSL:*
```
"⟦((p,s),c,(p',s')) ∈ Trans; curr_PAD s = Pri SL;
  hd (p In) = Write_PT_map (Pri SL, lvp) (Some pp); hd (p' Out) = Ok;
  memory s (pp,dp) = PORT PADs⟧ ⟹ PADs ⊆ {Pri SL, Pri PSL}"
```

Note the essential role of aliasing in the first of these axioms: the instruction intended to write a PORT instruction at virtual address *va* might affect SL memory even if *va* does not belong to SL, namely if there is some other virtual address *(Pri SL, lva)* that happens to be mapped to the same physical address.

Apart from the two axioms, the proof of the invariant requires the axiom *init_PORT_SL* given in §4.5 as well as the theorems *only_SL_changes_PT_map_of_SL* and *only_SL_changes_SL_memory*.

lemma *SL_PORT_SL_PSL:*
```
"Inv (λs. ∀lva pa PADs. PEA s (Pri SL, lva) = Some pa ⟶
          memory s pa = PORT PADs ⟶ PADs ⊆ {Pri SL, Pri PSL})"
```

Exploiting the invariant, the theorem can be proven in just a few steps. It reads as follows: for any sequence of transitions *ts* that may result from a system run and any state transition from *s* to *s'* within it, if SL becomes the current package in *s'*, the current package of the pre-state *s* must have been PSL.

theorem *only_PSL_enters_SL:*
"⟦*ts* ∈ *TRuns; ((p,s),c,(p',s')) ∈ set ts; curr_PAD s* ≠ *Pri SL;*
curr_PAD s' = Pri SL⟧ ⟹ *curr_PAD s = Pri PSL*"

This finishes our abstract formal analysis of the SLE88 memory management.

6 Conclusion

We have introduced a security model for the memory management of the SLE88 smart card processor chip. Memory management contributes to security by providing access control mechanisms on the levels of both virtual and physical memory addresses, allowing to separate applications and privileged SW packages (e.g., the operating system and the security layer SL) as well as applications from each other. Access control is guided by a policy comprising both discretionary (by effective access rights EAR) and mandatory (wrt. SL and privileged packages) rules. Enforcing the policy is non-trivial: the ability to change EARs and address mappings, the interacting levels of protection, aliasing in address translation, inter-package calls, and the peculiarities of SL have to be considered.

The model gives an abstract view of the SLE88 by concentrating on memory access and its protection only, leaving out details of the system and application functionality. Abstraction is achieved by reductions on the data structure and interface design and by underspecification. For instance, many data types used in the model are declared but not actually defined.

Carrying out the formal modeling work turned out to be worthwhile, because it provided new insights and lead to clarification of so far fuzzy concepts. Formal reasoning resulted in a minimal set of requirements on non-injective address mappings that guarantee the maintenance of the security properties. These requirements are given by restrictions on admissible combinations of EAR settings. The derived notion of EAR consistency is the least restrictive one preserving security and offers much more flexibility compared to simply forbidding aliasing. Formal analysis showed that security depends on assumptions on the initial state (e.g., initial EAR and PASL settings) as well as on benign behavior of SL. The assumptions can be interpreted as requirements on configuration upon delivery and on the software development of privileged packages. They clearly indicate the distribution of responsibility between the Target of Evaluation and its environment. Last, but not least, formal arguments lead to a clarification of the role of PASL: the PASL mechanism does not provide additional protection in case of weak EARs for SL, but protects against effects resulting from undesired mapping of both SL and non-SL virtual addresses to the same physical address.

To summarize, results of the modeling and proving process are the identification of relevant assumptions on the system environment and the derivation of new insights in the memory management and its security properties. The cost-benefit ratio is adequate: the whole work required no more than a six weeks effort, largely due to the availability of adequate tool support through Isabelle/HOL and the ISM approach. Thus, the SLE88 memory management security model is an excellent example for the value of formal security modeling in practical industrial-scale applications.

References

AHIP01. Atmel, Hitachi Europe, Infineon Technologies, and Philips Semiconductors. Smartcard IC Platform Protection Profile, Version 1.0, July 2001. http://www.bsi.de/cc/pplist/ssvgpp01.pdf.

AHIP02. Atmel, Hitachi Europe, Infineon Technologies, and Philips Semiconductors. Smartcard Integrated Circuit Platform Augmentations, Version 1.0, March 2002. http://www.bsi.de/cc/pplist/augpp002.pdf.

CC99. Common Criteria for Information Technology Security Evaluation (CC), Version 2.1, 1999. ISO/IEC 15408.

LKW99. Volkmar Lotz, Volker Kessler, and Georg Walter. A Formal Security Model for Microprocessor Hardware. In *Proc. of FM'99 World Congress on Formal Methods*, volume 1708 of *LNCS*, pages 718–737. Springer-Verlag, 1999.

LT89. Nancy Lynch and Mark Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989. http://theory.lcs.mit.edu/tds/papers/Lynch/CWI89.html.

MT00. Stephanie Motre and Corinne Teri. Using B method to formalize the Java Card runtime security policy for a Common Criteria evaluation. In 23rd *National Information Systems Security Conference*, 2000. http://csrc.nist.gov/nissc/2000/proceedings/toc.html.

Ohe02. David von Oheimb. Interacting State Machines: *a stateful approach to proving security*. In Ali Abdallah, Peter Ryan, and Steve Schneider, editors, *Proceedings from the BCS-FACS International Conference on Formal Aspects of Security 2002*, volume 2629 of *LNCS*. Springer-Verlag, 2002. http://ddvo.net/papers/ISMs.html.

OL02. David von Oheimb and Volkmar Lotz. Formal Security Analysis with Interacting State Machines. In Dieter Gollmann, Günter Karjoth, and Michael Waidner, editors, *Proc. of the* 7th *European Symposium on Research in Computer Security (ESORICS)*, volume 2502, pages 212–228. Spinger, 2002. http://ddvo.net/papers/FSA_ISM.html. A more detailed journal version is submitted for publication.

OL03. David von Oheimb and Volkmar Lotz. Generic Interacting State Machines and their instantiation with dynamic features. In *Proc. of the* 5th *International Conference on Formal Engineering Methods (ICFEM)*. Spinger, November 2003. http://ddvo.net/papers/GenISMs.html, to appear.

Pau94. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994. For an up-to-date documentation, see http://isabelle.in.tum.de/.

PNW+. Lawrence C. Paulson, Tobias Nipkow, Markus Wenzel, et al. The Isabelle/HOL library. http://isabelle.in.tum.de/library/HOL/.

SRS+00. G. Schellhorn, W. Reif, A. Schairer, P. Karger, V Austel, and D. Toll. Verification of a formal security model for multiapplicative smart cards. In Frédéric Cuppens, Yves Deswarte, Dieter Gollmann, and Michael Waidner, editors, *Proc. of the* 6th *European Symposium on Research in Computer Security (ESORICS)*, volume 1895. Spinger, 2000.

WN03. Georg Walter and Jürgen Noller, Infineon. SLE88CX720P / m1491 Security Target. http://www.bsi.de/???0215???, Version 1.00, March 2003.