Formal Security Policy Models for Smart Card Evaluations

Gerd Beuster
Fachhochschule Wedel
Feldstraße 143
22880 Wedel, Germany
gb@fh-wedel.de

Karin Greimel
NXP Semiconductors Austria GmbH
Mikronweg 1
A-8101 Gratkorn, Austria
karin.greimel@nxp.com

ABSTRACT

For high security ICs, a security evaluation by an independent institution is of great importance to strengthen the confidence in the security of the product. Common Criteria (CC) is a widely used evaluation method for security products. In many countries, CC evaluations are required by law for certain IT products. For high assurance, CC requires a formal model of the implemented security policies. We show how such a formal security policy model based on temporal logic and model checking can be developed for the real world evaluation of a Security IC. We argue that temporal logics and model checking is suitable for the formal requirements of a CC Evaluation Assurance Level 6 evaluation, because models and security requirements can be developed by anybody with moderate knowledge of formal methods. Additionally, proofs (or refutations) are generated automatically.

Categories and Subject Descriptors

B.7 [**Integrated Circuits**]: Miscellaneous; F.4.1 [**Theory of Computation**]: Mathematical Logic and Formal Languages—*Temporal logic*

Keywords

Common Criteria, EAL6, Security Policy Model, NuSMV, Model Checking

1. INTRODUCTION

The Common Criteria for Information Technology Security Evaluation (CC) ([2]) are an international standard for the evaluation of computer security products. Common Criteria provides different *Evaluation Assurance Levels*, ranging from EAL1 to EAL7. Starting with EAL6, Common Criteria requires a *Formal Security Policy Model* (FSPM) and a proof that insecure states are unreachable. We developed such an FSPM and the proofs for a Security IC. We decided to model the IC as a finite state machine and let a model checker give the proof.

In general, a model checker takes a model and a formal specification. It returns true if the model satisfies the specification and false otherwise. Compared to other methods, like interactive theorem proving, the advantage of our approach is that model checking is fully automatic and constructive refutations are provided in case proofs fail.

For our FSPM, we use the model checker NuSMV [3]. In this paper, we show how parts of the security policies given in the Security IC Platform Protection Profile [4] can be formalized for model checking. While developed for a concrete product evaluation, we consider our formal definitions of security policies suitable for generalization to other products and product types.

Related Work

Most of the publications describing a FSPM for CC use theorem provers. For example, a FSPM was developed and proven for the INTEGRITY-178B real-time operating system using the ACL2 theorem prover [7]. The main policy proven for INTEGRITY-178B was a policy about separating operating system kernels called "GWV policy" [5], which is very similar to the Access Control Policy in this evaluation. The ACL2 theorem prover has also been used in the evaluation of the Rockwell Collins' AAMP7G microcontroller to show that it satisfies the GWV kernel separation criteria [6]. Infineon verified the SAPM of their SLE 88 [8] with the theorem prover Isabelle.

The NuSMV model checker, which is used in this evaluation, has been used in a number of projects to proof security and safety properties, for example in formal modeling the FCS 5000 flight control system and the ADGS-2100 Adaptive Display and Guidance System, Operational Flight Program (OFP) [7].

2. FORMAL MODEL OF THE IC

This section describes the model of the IC. The model is based on *Modes*. Specific access rights to IC components are associated with every mode, like access to certain memory areas or coprocessors. At its core, it models these modes by a state variable *CPU*. The range of this variable is shown in Table 1. In User Mode, the IC controls access to memories through the Memory Management Unit (MMU). The MMU translates virtual to physical addresses.

Access is controlled in two ways. First, memory is split into two parts. One part is available in Firmware Mode. The other is available in System Mode and User Mode. Sec-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'12 March 25-29, 2012, Riva del Garda, Italy.
Copyright 2011 ACM 978-1-4503-0857-1/12/03 ...$10.00.

System State	Description
SystemM	Mode for execution of application programs. In this mode, all resources available to application programs are accessible.
UserM	Restrictive mode for execution of application programs. Accessibility of resources is configurable in System Mode. Memory access is moderated by the MMU.
FirmwM	Mode for emulating other Security Smart Card products. The memory used in this mode is completely separated from the memory available in System Mode and User Mode.

Table 1: TOE Modes Description (excerpt)

ond, memory can be segmented into smaller areas and access rights (readable, writable or executable) can be defined for these segments.

The MMU Table stores memory access rights. Note that the MMU Table itself is stored in memory. Therefore it is possible to store an MMU Table in memory writable in User Mode. In this case a User Mode process may manipulate the MMU Table, possibly circumventing restrictions.

Modeling Memory Segmentation: MMU Tables

In order to abstract from the implementation details and the complexity of multiple tables and processes at arbitrary memory addresses, the following model is used: We explicitly model two user processes (process 0 and process 1) and one memory segment only. We also do not use explicit memory addresses, but Boolean variables (*mmuTableInSeg[0]* and *mmuTableInSeg[1]*) indicating that the process' MMU table is in the modeled segment or somewhere else. Also, details of the MMU table data structure are not modeled. Only the access rights are given by the variables *MMUtable[0]* and *MMUtable[1]* which can be subsets of {r,w,x}. The values represent read, write, and execute access, respectively.

Modeling Memory Partitioning: Firmware Firewall

The mechanism separating Firmware Mode from System and User Mode is called "Firmware Firewall". The Firmware Firewall is modeled similar to the MMU. A state variable *FMcanAccessSegment* indicates if Firmware Mode processes have access to the modeled memory segment.

Modeling Operations

State transitions are triggered by *operations*. Operations can be triggered by software running on the IC (e.g. memory write operations), external events (e.g. an attacker manipulating the IC), or internal events. The operations used in the formal security policies given in Section 3 are shown in Table 2.

3. SECURITY POLICIES

The TOE is evaluated according to the Security IC Platform Protection Profile [4]. This Protection Profile defines a number of Security Function Policies (SFP). Due to space limitations, only the the *Access Control Policy* is shown here. All theorems are given in the NuSMV specification language [1]. The following theorems define the *Access Control Policy*.

Operation	Description
OpProc0SegmentWrite, OpProc1SegmentWrite	User Mode process 0/1 writes to modeled memory segment.
OpProc0SegmentAccess, OpProc1SegmentAccess	User Mode process 0/1 accesses modeled memory segment.
OpSMmemoryAccess	System Mode process accesses modeled memory segment.

Table 2: Operations (excerpt)

All User Mode memory accesses are moderated by the MMU. In User Mode, memory access is possible only when the MMU table allows it (MMUtable[0] $\neq$ *none*). Note that here we do not distinguish between read, write, and execute accesses. (Process 1 case is similar)

INVARSPEC
((CPU = UserM) $\wedge$ OpProc0SegmentAccess) $\rightarrow$
(MMUtable[0] $\neq$ *none*)

In the following formulas, the property that memory content does not change is represented by the statement

$$\text{mmuTableInSeg[0]} \wedge (\text{MMUtable[0]} = next(\text{MMUtable[0]}))$$

While this statement explicitly talks about changes in the MMUtable of process 0, it can be interpreted as any change in the memory segments accessible by process 0. (Process 1 case is similar.)

The MMU Table of a process may only change if the memory segment where the MMU table resides is writable by a User Mode process. Note that we do not model changing an MMU Table in System Mode, because the Access Control Policy requires isolation of User Mode processes from each other only.

INVARSPEC
((CPU = UserM) $\wedge$ ($next$(CPU) = UserM)$\wedge$
($\neg$mmuTableInSeg[0]) $\wedge$ ($\neg$mmuTableInSeg[1])) $\rightarrow$
((MMUtable[0] = $next$(MMUtable[0]))$\wedge$
(MMUtable[1] = $next$(MMUtable[1])))

In User Mode, the MMU table of a process does not change unless it is writable by one of the user processes. (Process 1 case is similar.)

INVARSPEC
((CPU = UserM) $\wedge$ ($next$(CPU) = UserM)$\wedge$
mmuTableInSeg[0]$\wedge$
(MMUtable[0] $\notin \{w, rw, wx, rwx\}$)$\wedge$
(MMUtable[1] $\notin \{w, rw, wx, rwx\}$)) $\rightarrow$
(MMUtable[0] = $next$(MMUtable[0]))

User Mode Processes are isolated from each other, i.e. one process cannot write memory assigned to another process

unless explicitly permitted.

INVARSPEC
((CPU = UserM) ∧ (*next*(CPU) = UserM)∧
mmuTableInSeg[0]∧
(MMUtable[1] ∉ {*w*, *rw*, *wx*, *rwx*})) →
((MMUtable[0] = *next*(MMUtable[0]))∨
OpProc0SegmentWrite)

The second part of the security properties of the Access Control Policy define the separation between memory assigned to Firmware Mode and memory assigned to the other modes.

When a memory segment is accessible in User Mode, it is not accessible in Firmware Mode.

INVARSPEC
((OpProc0SegmentAccess) ∨ (OpProc1SegmentAccess)) →
¬FMcanAccessSegment

When a memory segment is accessible in System Mode, it is not accessible in Firmware Mode.

INVARSPEC
OpSMmemoryAccess → ¬FMcanAccessSegment

When a memory segment is accessible in Firmware Mode, it is not accessible in System Mode.

INVARSPEC
FMcanAccessSegment → ¬OpSMmemoryAccess

When a memory segment is accessible in Firmware Mode, it is not accessible in User Mode.

INVARSPEC
FMcanAccessSegment →
¬((OpProc0SegmentAccess)∨
(OpProc1SegmentAccess))

Default values should be as restrictive as possible. For User Mode MMU tables, that means that User Mode processes should have no access rights at all after a reset.

INVARSPEC
(CPU = ResetM) →
((*next*(MMUtable[0]) = *none*)∧
(*next*(MMUtable[1]) = *none*))

4. MODEL CHECKING RESULTS

NuSMV [3] has been used for model checking. In the development of the formal model and preliminary runs of the model checker, nearly all proof obligations could be proven. The proof obligations that failed initially fell into two categories: Some of them identified attack paths that are available before deployment of the TOE only. These attack paths depend on certain activities while the TOE is in Test Mode. In practice, these attacks are not relevant, because in the deployment phase, Test Mode is no longer available. The second class of spurious proof fails was due to simultaneous access of MMU configuration parameters. Since these conditions can occur only when two processes with the same access rights interfere with each other, and since no privilege escalation may result from these racing conditions, the security policy is not violated in these cases. Therefore no redesign of the TOE was necessary, even in the cases where proofs failed initially. In these cases additional requirements for the TOE environment (for example, the TOE must be completely configured before leaving the factory) suffice to satisfy all security requirements. Additional proof obligations not directly related to security functionality lead to new insights into the working of the TOE and helped to improve the TOE documentation.

5. CONCLUSIONS

We showed that the Security IC can not reach a state that is not secure, using a model checker. The model checker proved that the formal model of the IC implements the security policies.

Developing a Security Policy Model requires consistent, unambiguous, and complete documentation. Thus, it does not only add assurance in terms of mathematical proof but also in enforcing accurate documentation.

Beside providing an FSPM for the Common Criteria evaluation of a concrete product, our approach can be generalized for formal models of other products and product types.

6. REFERENCES

[1] R. Cavada, A. Cimatti, C. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev. *NuSMV 2.5 User Manual*.

[2] *Common Criteria for Information Technology Security Evaluation Version 3.1 Revision 3*, July 2009.

[3] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. Int. Conf. on Computer-Aided Verification (CAV 2002)*, 2002.

[4] European Smart Card Industry Association (Eurosmart). *Security IC Platform Protection Profile Version 1.0*, June 2007.

[5] D. Greve, M. Wilding, R. Richards, and W.M. Vanfleet. Formalizing security policies for dynamic and distributed systems. In *Systems and Software Technology Conference (SSTC 2005)*, 2005.

[6] David S. Hardin. A robust machine code proof framework for highly secure applications. In *Proc. of the 2006 ACL2 Workshop*, 2006.

[7] Steven Miller. Will this be formal? *Theorem Proving in Higher Order Logics*, pages 6–11, 2008.

[8] David von Oheimb, Georg Walter, and Volkmar Lotz. A Formal Security Model of the Infineon SLE 88 Smart Card Memory Managment. In Einar Snekkenes and Dieter Gollmann, editors, *ESORICS*, volume 2808 of *Lecture Notes in Computer Science*, pages 217–234. Springer, 2003.