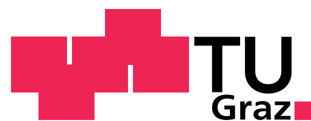
Dipl.-Ing. Johannes Loinig Bakk.techn.

Security and Performance Verification of Secure Embedded Systems

---

Dissertation

vorgelegt an der
Technischen Universität Graz



zur Erlangung des akademischen Grades
Doktor der Technischen Wissenschaften
(Dr.techn.)

durchgeführt am Institut für Technische Informatik
Technische Universität Graz
Vorstand: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Reinhold Weiß

Graz, im Oktober 2012

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am ....

(Unterschrift)

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....

date (signature)

Kurzfassung

Moderne Eingebettete Systeme erfordern für gewöhnlich ein hohes Maß an Informationssicherheit. Unglücklicherweise ist diese in einem System weder sichtbar noch messbar. Sie muss als Gesamtkonzept im System und auch im Entwicklungsprozess berücksichtigt werden. Sorgfältige Systemverifikation ist notwendig um sicherzustellen, dass alle Sicherheitsanforderungen implementiert wurden. Solche Verifikationen auf Systemebene sind bei der Komplexität heutiger Systemen sehr aufwändig und deswegen besonders teuer.

Die Berücksichtigung von Sicherheitsanforderungen während eines Hardware/Software Codesign Prozesses hat einen großen Vorteil: Die Anforderungen können von Beginn des Entwicklungsprozesses an berücksichtigt werden und somit fortlaufend mit dem Rest des Systems bezüglich Kriterien wie der Ausführungsgeschwindigkeit optimiert werden. Allerdings ist darauf zu achten, dass Optimierungen Verletzungen von Sicherheitsanforderungen zufolge haben können. Deswegen ist eine fortlaufende und automatisierte Sicherheitsverifikation in einem iterativen Codesign-Prozess unumgänglich.

Diese Arbeit beschreibt einen innovativen und iterativen Entwicklungsprozess. Dieser erlaubt die fortlaufende Optimierung und Verifikation des Systems. Der Entwicklungsprozess basiert auf einem Systemmodell welches iterativ verfeinert und optimiert wird. Dies wird wiederholt bis alle Systemanforderungen erfüllt sind. Jede Iteration beinhaltet dabei ihre eigene Verifikation aller Sicherheitsanforderungen. Dies stellt sicher dass Verletzungen von Sicherheitsanforderungen unverzüglich bemerkt und behoben werden.

Der wissenschaftliche Beitrag dieser Dissertation ist ein innovativer Hardware/Software Codesign-Ansatz inklusive der Verifikation von Sicherheitsaspekten basierend auf einer systemweiten Simulation. Die nahtlose Integration von (informationstechnischen) Sicherheitsanforderungen in das System sowie auch in den Entwicklungsprozess erlaubt eine regelmäßige Optimierung und automatisierte Verifikation des Systems. Das Konzept beruht auf ergänzende Informationen (sogenannte Meta-Informationen) die im Model bzw. in der Implementierung des Systems eingebettet sind. Diese Informationen können zur Laufzeit oder während einer Simulation ausgewertet werden und helfen das System zu Analysieren, zu Optimieren und auch zu Verifizieren. Unter anderem wurde eine solche Verifikation im Zuge dieser Dissertation mittels Formalen Methoden durchgeführt.

Abstract

Modern embedded systems very often need to provide a certain level of information security. Unfortunately the security of a system is not very visible or measurable. Security is not only system wide concept; it has also to be integrated into the development process. Careful security verification has to be done to evaluate if the system fulfills all security requirements. Such verification is a system wide task and for systems with modern complexity typically a very time consuming and expensive task.

Adding security aspects during hardware/software codesign has the great benefit that they can be considered from the beginning of the development process. They can be continuously optimized with the remaining system. However, this raises the question how to avoid an unrecognized violation of security requirements along optimizations. Consequently security verification has to be integrated into the iterative refinement loops of the codesign process. Accordingly verification has to be an automated task.

This thesis explains a novel iterative development process that allows continuous system optimization, security verification, and functional verification. It is based on a system model that is iteratively refined and optimized until all functional requirements are met. Each iteration contains a simulation based security verification. This ensures that any appearing security violation is immediately recognized.

The novel approach in this thesis is based on additional information (so called meta-information) in the model or implementation of the system. This meta-information can be evaluated during runtime or system simulation time and is used to analyze, optimize, and verify the system. For security verification the meta-information represents functional security requirements. During simulation this data is evaluated to determine the dynamic dependencies of system modules and security requirements. Based on that a security verification is performed e.g., by using a formal tool like a model checker.

Thus, the main contribution of this thesis is a novel hardware/software codesign methodology which allows simulation based security verification. The seamless integration of security requirements in the development flow allows continuous optimization and verification of the system. This is combined with a model checking approach to gain from the benefits of formal verification while staying applicable for industrial development of secure embedded systems.

Acknowledgements

First, I would like to thank my supervisor Prof. Dr. Reinhold Weiß for his helpful advice and mentoring throughout my dissertation research and writing processes at the Institute for Technical Informatics at the Graz University of Technology. I would also like to thank Dr. Christian Steger for his technical and organizational support during the runtime of the project and beyond. In addition, a great thanks to all of my colleagues at the institute, and above all, a sincere thanks to Silvia Reiter who patiently helped me with most of the organizational formalities at the university.

I would like to thank Andreas Mühlberger at NXP Semiconductors Austria, who along with Dr. Christian Steger made this project possible. My very special thanks goes to Ernst Haselsteiner who was always a great mentor and even in difficult times always found some time to help me with my work at NXP and at the university. Also, many thanks to all the colleagues at NXP in Gratkorn and in Hamburg who never failed to provide a helping hand when I needed one.

Last, but not least, I would like to thank my family. I would heartily like to thank my wife Brigitte and my daughter Anna for their patience when I had to work instead of spending time with them. I would also like to thank my parents, my grand parents, and my parents-in-law for their tremendous support in the last years.

Lannach, October 2012 Johannes Loinig

Extended Summary

Modern embedded systems provide services for communication and information exchange, entertainment, education, and much more. Almost all of these use cases come with one key requirement that has shown to be extremely important in recent years: embedded systems need to provide a high level of information security. The main reason for this lies in the nature of embedded systems. They are placed (embedded) wherever they are needed, which makes them easily accessible not only for users but also for attackers.

Information security is a moving target. Attack methodologies and technologies are constantly improving continuously. Early smart card attack methods, for example, strategically interrupted the power supply in order to cause system inconsistencies. Smart cards (which are typical secure embedded systems) need to provide a so called anti-tearing mechanism to counteract such an attack. Now, only few years later attackers use much more sophisticated ways such as power glitches and laser attacks, to attack secure systems. Developers of secure embedded systems have to take under consideration that ways to counteract attacks might have to change even before development is finished. Consequently, the development process of secure embedded systems has to be flexible enough to react effectively to constantly changing security requirements.

Unfortunately, security is not a very visible or measurable property. In order to verify the security level of a system, it has to be carefully evaluated. Specifically, it has to be evaluated wether the level of security is capable of countering all potential attacks. Security is a system wide concept. It has to be considered in every part of the entire system and from the very beginning of the development process. Accordingly, security verification is a system wide task.

Hardware/software codesign proved to be highly applicable for optimization of systems on an architectural level. Taking information security into account during hardware/software codesign allows development of secure and highly optimized systems. However, it also raises some questions. How can developers make sure that the security level is not compromised by mistake during optimization? How could a developer recognize such a case early enough in the development process? This thesis aims to discuss possible solutions for these problems.

This dissertation is the result of the *HiPerSec* project, funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under contract FFG 816464. Project partners include the Institute for Technical Informatics, Graz University of Technology, and NXP Semiconductors Gratkorn, Austria. HiPerSec was started in order to investigate new hardware/software codesign methodologies that provide sufficient information for performance and security evaluations of secure embedded systems.

Common security verification methodologies are either performed manually or based on formal models. Manual verification is typically used in industrial environments, while formal verification is mainly used in research studies. The reason for this discrepancy is that formal methods are not yet applicable for systems with industrial complexity because of their immense computational effort. Furthermore, manual verification is a rather documentation centric approach. During verification, documentation is first checked against the specification. Then, the implementation has to be verified against the documentation. On the contrary and as shown in Figure 1, the methodology discussed in this thesis proposes an implementation centric approach[1]. Verification is embedded in an iterative hardware/software codesign process. It becomes automated, it is performed on system level and is considering all necessary system components, and it is also independent of the abstraction levels used during development.

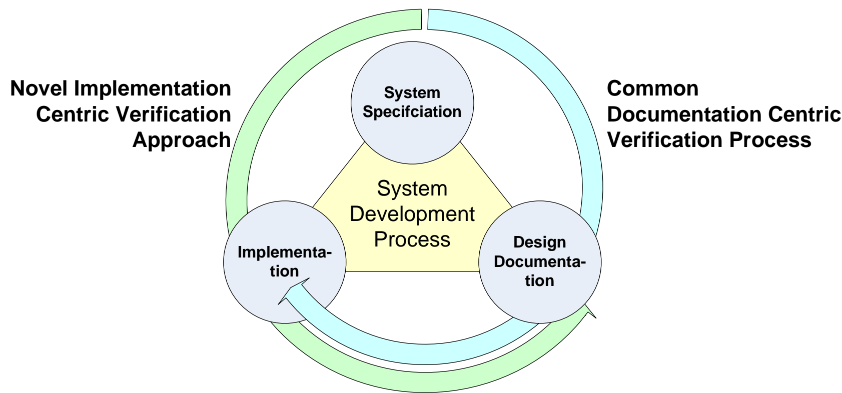


Figure 1: Common Documentation Centric Verification Approach vs. Novel Implementation Centric Verification Approach. Common documentation centric verification approaches are based on documents which describe the mapping between system specification and system implementation. Conversely, in an implementation centric approach the implementation itself is verified automatically against the specification.

This thesis explains a novel iterative development process that allows continuous system optimization, security verification, and functional verification shown in Figure 2. Functional and security specifications are used to create a system model. The model is then refined and verified iteratively. This process consists of (a) modeling the functional parts of the system as well as the security architecture, (b) a model based simulation to gather system parameters, (c) an analysis and verification of the simulation results, and (d) refining and optimizing the system. Iterations are repeated until the system meets all functional and security requirements. Each iteration includes a simulation based security verification[2], which ensures that the previous refinement or optimization step did not cause any violation of security requirements. Analysis showed that smart card hardware systems

[1] J. Loinig, C. Steger, R. Weiss, and E. Haselsteiner, Towards formal system-level verification of security requirements during hardware/software codesign, in SOC Conference (SOCC), 2010 IEEE International, pp. 388-391, Sept. 2010.

[2] J. Loinig, C. Steger, R. Weiss, and E. Haselsteiner, Idea: Simulation Based Security Requirement Verification for Transaction Level Models, in Engineering Secure Software and Systems (ESSoS), vol. 6542 of Lecture Notes in Computer Science, pp. 264–271, Springer Berlin / Heidelberg, 2011.

include dozen of applied security requirements where 70 to 90 percent of them could be verified by demonstrated simulation based and automated verification approaches. This implies a significant reduction of effort during the development process of a system.

Furthermore, simulation based functional verification[3] allows detailed functional verification of system-states that are difficult to reproduce in final implementations. A simulation based verification of anti-tearing mechanisms of smart cards shows an improvement of 20% and more in verification performance.

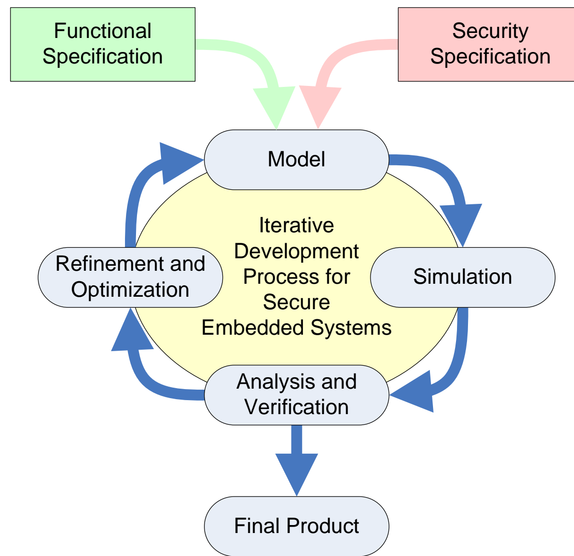


Figure 2: Overview of Iterative Development for Secure Embedded Systems. Functional and security specifications are implemented in a system model. This model is simulated, analyzed, verified, refined, and optimized in an iterative process until all requirements are met.

The novel approach in this thesis is based on evaluating meta-information during system simulation. Meta-information is machine readable textual data added to the source code of system modules. This additional data does not influence the model's functional behavior, but it can be evaluated by the executing or simulating environment. We used meta-information to analyze, optimize, and verify systems. A meta-information based analysis of a smart card operating system showed[4] that 60% of the implemented software functions have to be security verified. Furthermore, it identified software functions which are important for system verification. This knowledge makes system verification more efficient and lowers the risk of overlooking some functions. In addition, the analysis revealed that 50% of these functions do not need all the security mechanisms every time they are

[3] J. Loinig, C. Steger, R. Weiss, and E. Haselsteiner, Fast simulation based testing of anti-tearing mechanisms for small embedded systems, in Test Symposium (ETS), 2010 15th IEEE European, p. 242, May 2010.

[4] J. Loinig, C. Steger, R. Weiss, and E. Haselsteiner, Identification and Verification of Security Relevant Functions in Embedded Systems Based on Source Code Annotations and Assertions, in Information Security Theory and Practices. Security and Privacy of Pervasive Systems and Smart Devices, pp. 316-323, Springer, 2010.

used. This implies great potential for performance optimizations.

A meta-information based system optimization[5] of an anti-tearing mechanism showed a reduction of costly (time and power) write-operations to non-volatile memory. 25% to 70% of write-operations in embedded systems can be prevented when appropriate meta-information is evaluated during runtime of the system. Research[6] has shown that this can improve the execution time of smart card payment applications by more than 17%.

As shown in Figure 3, meta-information derived from the system's security specification is used to represent functional security requirements for security verification. Meta-information is then added to its respective system model module. This information is not dependent on how a requirement is implemented, consequently allowing enough flexibility for a codesign process. Specifically, when a functional module description is altered, meta-information can remain unchanged. A complete model simulation is performed to understand the dynamic dependencies of all the system modules. The functional module descriptions define this dynamic behavior. In order to perform security verification, any dynamic dependencies of present security requirements need to be resolved by evaluation of mata-information during a simulation.

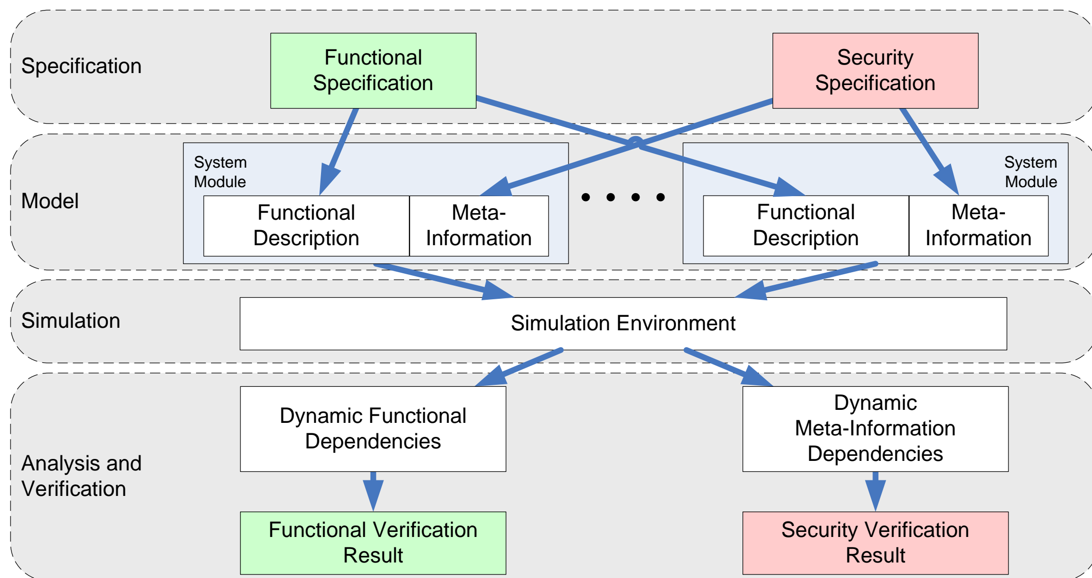


Figure 3: Simulation Based Meta-Information Evaluation. While the functional specification is modeled by functional descriptions, the security specification is represented by additional information (meta-information). Together the functional description and the meta-information are evaluated during the simulation of the system model to verify the functional behavior as well as the correct implementation of the security specification.

[5] J. Loinig, P. Glatz, C. Steger, and R. Weiss, Performance Improvement and Energy Saving Based on Increasing Locality of Persistent Data in Embedded Systems, in Systems (ICONS), 2010 Fifth International Conference on, pp. 175-180, April 2010.

[6] J. Loinig, C. Steger, R.Weiss, and E. Haselsteiner, Java Card Performance Optimization of Secure Transaction Atomicity Based on Increasing the Class Field Locality, in Secure Software Integration and Reliability Improvement. SSIRI 2009. Third IEEE International Conference on, pp. 342-347, July 2009.

In addition the verification approach presented in this thesis includes a new way of approaching for formal verification of modeled security architecture[7]. Security specification is not only used to add meta-information to the system model, as previousely described, but it is also used to define a set of formal rules also known as a formal security policy. After simulation of a system, simulation output which contains any dynamic dependencies of the security requirements, is verified against such a security policy. This is done by using a formal model checker as depicted in Figure 4. Automated simulation-based formal verification of a sample smart card, which includes hardware and software components and different abstraction levels, takes less than 90 seconds when performed on a conventional personal computer with a security policy that includes multiple interdependent security requirements. This constitutes acceptable verification performance for an iterative development process, such as described in this dissertation.

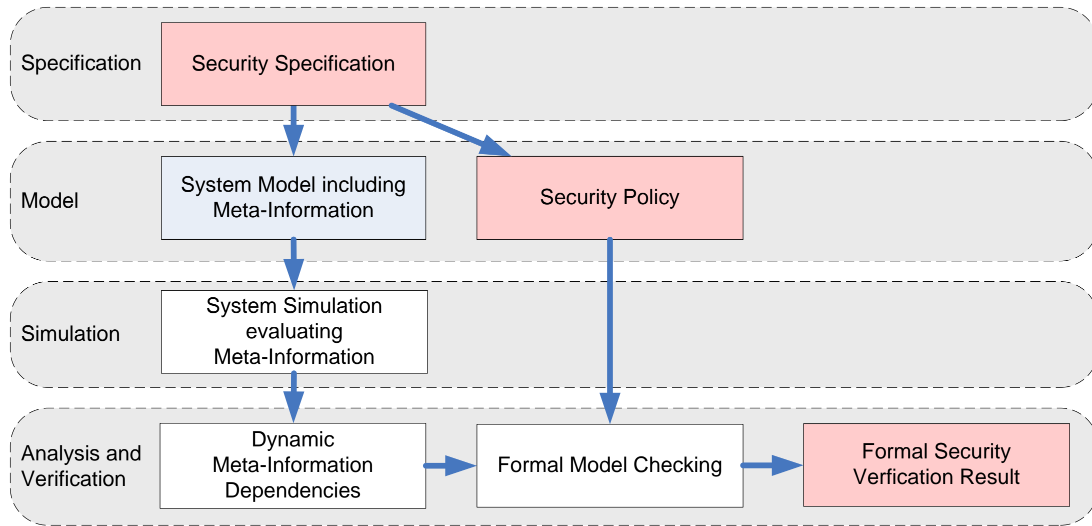


Figure 4: Formal Simulation Based Security Verification Concept. The security specification is formalized in a security policy. The simulation results of the system model are verified by model checking against this formal policy. This allows proving that the requirements in the security specification are correctly applied in the system model.

Overall, the main contribution of this thesis to the research of secure embedded systems is a novel hardware/software codesign methodology which allows simulation based security verification. Seamless integration of security requirements into the development flow allows continuous optimization and verification of any given system. Because of continuous verification, optimization can be done without risk to violate security requirements. Additionally, another key contribution of this thesis is the concept of simulation based formal verification approach, which allows fast formal verification of security requirements in system models independently from the abstraction level of a model. The main purpose of this approach is to make formal approaches applicable for industrial development of secure embedded systems.

[7]J. Loinig, C. Steger, R. Weiss, and E. Haselsteiner, Security Verification on Mixed Level System Models based on Event Calculus Model Checking, Journal of Systems Architecture (under review), Elsevier B.V

Contents

Table of Contents x

1 Introduction to Security and Performance Verification of Secure Systems **1**
1.1 Motivation . . . 1
1.2 A Definition of Information Security and System Security . . . 2
1.3 Problem Statement . . . 4
1.4 The HiPerSec Project . . . 4
1.5 Contribution and Significance . . . 5
1.6 Overview of this Thesis . . . 5

2 State of The Art Development and Verification Methodologies **6**
2.1 Secure Embedded Systems . . . 6
2.2 Smart Cards and Java Cards . . . 7
2.3 Common Criteria for Security Evaluation . . . 8
2.4 Hardware Software Codesign . . . 9
2.5 System Level Verification . . . 10
2.6 Event Calculus . . . 11
2.7 Summary . . . 12

3 Novel Codesign Methodology for Verification of Secure Embedded Systems **13**
3.1 Methodology Overview . . . 13
3.2 System Security Specification . . . 14
3.3 Secure System Model . . . 18
3.4 Simulation Based Verification . . . 20
3.5 System Optimizations . . . 22

4 Methodology Evaluation and Case Studies **24**
4.1 Common Basics for the Case Studies . . . 24
4.2 Simulation Based Functional Verification . . . 25
4.2.1 Java Card Anti-Tearing Mechanism . . . 26
4.2.2 Case Study 1: Verification of an Anti-Tearing Mechanism . . . 27
4.3 Evaluation of the Concept of Meta-Information . . . 27
4.3.1 Case Study 2: System Optimization based on Meta-Information . . . 28
4.3.2 Case Study 3: Identification of Security Relevant System Components . . . 29
4.4 Simulation based Security Verification . . . 30
4.4.1 Case Study 4: Simulation based System Security Verification . . . 31
4.4.2 Case Study 5: A (Semi-) Formal Event Calculus based Model Checking Approach . . . 32
4.5 Evaluation Summary . . . 37

5 Conclusion and Future Work **39**
5.1 Conclusion . . . 39
5.2 Future Work . . . 40

6 Publications **42**
6.1 Security Verification on Mixed Level System Models based on Event Calculus Model Checking . . . 43
6.2 Identification and Verification of Security Relevant Functions in Embedded Systems Based on Source Code Annotations and Assertions . . . 58
6.3 Idea: Simulation Based Security Requirement Verification for Transaction Level Models . . . 66
6.4 Towards Formal System-Level Verification of Security Requirements during Hardware/Software Codesign . . . 74
6.5 Performance Improvement and Energy Saving based on Increasing Locality of Persistent Data in Embedded Systems . . . 78
6.6 Java Card Performance Optimization of Secure Transaction Atomicity Based on Increasing the Class Field Locality . . . 84
6.7 Fast simulation based testing of anti-tearing mechanisms for small embedded systems 90

References **97**

A Additional Information **101**
A.1 Axioms of the Event Calculus Domain Description . . . 101
A.2 Call Graph of the Java Card Application JavaPurse . . . 102
A.3 Event Calculus Symbols of Security Requirements . . . 103

List of Tables

4.1 Number of Security Functional Requirements in Smart Card Hardware Platform Security Targets of different Vendors 31
4.2 Security Policy Rules 33
4.3 Verification Times with split Trace File 37

A.1 Event Calculus Domain Descriptions of Security Verifications 102
A.2 Symbols and Explanations of Functional Security Requirements 104

List of Figures

1 Common Documentation Centric Verification Approach vs. the Novel Implementation Centric Verification Approach . . . v
2 Overview of Iterative Development for Secure Embedded Systems . . . vi
3 Simulation Based Meta-Information Evaluation . . . vii
4 Formal Simulation Based Security Verification Concept . . . viii

3.1 Overview of the Novel Codesign Methodology . . . 14
3.2 Extraction of Rules from a Security Specification . . . 15
3.3 Relationships of SFR Dependency Predicates: (a) requires/implements and (b) fulfills/expects . . . 16
3.4 Abstraction Levels of the Novel Codesign Methodology . . . 18
3.5 Simulation Based Verification Approach with Meta-Information . . . 21
3.6 Overview of the (semi-) formal Simulation based Security Verification Approach . . . 22

4.1 Java Card Overview . . . 25
4.2 The Tearing Problem . . . 26
4.3 Performance of Functional Verification of an Anti-Tearing Mechanism with and without Test Vector Compaction . . . 27
4.4 Performance Gain for an Anti-Tearing Mechanism with Meta-Information based Data Allocation . . . 28
4.5 Performance Optimization Results of using Meta-Information with an Anti-Tearing Mechanism . . . 29
4.6 Results of Meta-Information based Analysis . . . 30
4.7 Evaluation of Meta-Information in SystemC during Simulation . . . 32
4.8 Security Requirement Traces of BAC Authentication in a High Level Model . . . 34
4.9 Complete Security Requirement Trace of BAC Authentication in a High Level Model 35
4.10 Shortened Security Verification Output with f2lp . . . 35
4.11 Smart Card Models under Refinement and Verification . . . 36
4.12 Verification Performance . . . 36

A.1 Call Graph of JavaPurse . . . 103

List of Abbreviations

APDU	Application Protocol Data Unit
API	Application Programming Interface
BAC	Basic Access Control
BP	Backup Phase
CC	Common Criteria
EAL	(Common Criteria) Evaluation Assurance Level
EC	Event Calculus
EEPROM	Electrically Erasable Programmable Read-Only Memory
ES	Embedded System
F(L)M	Functional (Level) Model
FPGA	Field Programmable Gate Array
FR	Functional Requirement
GC	Garbage Collector
JCVM	Java Card Virtual Machine
LPI	Logical Page Identifier
MCIF	Model-to-Checker Interface
NVL	New Value Logging
OS	Operating System
OVL	Old Value Logging
PPI	Physical Page Identifier
PSL	Property Specification Language
RFID	Radio Frequency Identification
RP	Restore Phase
SA	Security Annotation
SAF(L)M	Security Annotated Functional (Level) Model
SC	Security Constraint
SDL	Spatial Data Locality
SF(L)M	Secure Functional (Level) Model
SFR	(Common Criteria) Security Functional Requirement
ST	(Common Criteria) Security Target
ST(L)M	Secure Transaction (Level) Model
SIL	Secure Implementation Level
TB	Transaction Buffer
TDL	Temporal Data Locality
TLM	Transaction Level Model
TM	Transaction Mechanism
TOE	(Common Criteria) Target of Evaluation
TV	Test Vector
UML	Unified Modeling Language
VM	Virtual Machine

Chapter 1

Introduction to Security and Performance Verification of Secure Systems

This chapter introduces the topic of security and performance verification of secure systems. More precisely, in all case study examples of this work Java Cards as typical representatives of secure systems are verified. It also shows the necessity for new design and verification methodologies to be developed for future secure systems. To arrive at a common understanding of the term *secure system*, and in order to provide a contextual background for this work, Section 1.2 of this thesis defines the terms *information security* and *system security*.

Furthermore, this thesis is a summary of the *HiPerSec* project and its results. *HiPerSec* is presented in more detail in Section 1.4. The underlying problem statement for this research project is given in Section 1.3. The contribution and significance of this thesis are explained in Section 1.5. Finally, Section 1.6 gives a detailed overview of all other chapters in this work.

1.1 Motivation

Taking a through look at today's embedded systems, it became apparent that a clear definition of such systems does not exist. Technological advancements such as the Internet and mobile communication cause almost every system to be somehow embedded in a broader system. Even very pure functions such as mobile multimedia systems (like MP3 players), are equipped with sensors and interfaces in order to remain connected with the reset of the world. Perhaps that is why they became so popular and have finally even reached the end-user marked.

Embedded systems are used on a very broad spectrum of use cases. For example, they are used for mobile communication, information exchange, education, entertainment, and much more. These use cases come with one common feature: a high number of items sold. The high profit potentials made them interesting for the industry, but it also made them very interesting for attackers. The more people use a certain system, the more profitable it can be to exploit a certain vulnerability of that system. This becomes even more relevant

when a system, in the hands of an end user or an attacker, obviously gives them more desirable or undesirable control over the system.

Practice has shown that if it is profitable to exploit vulnerabilities (in some cases even if it is not profitable), people will try to do so, and in some cases they will succeed. Consequently the number of attacks, their quality, and their efficiency is continuously growing in correspondence with the number of existing embedded systems, use cases, and users. This requires the industry, and also the research community, to work not only on more sophisticated countermeasures against attacks but also on better methodologies to develop and evaluate systems with respect to security and other functional and non functional properties.

This work explains and discusses such a novel methodology for security and performance verification of embedded systems. Systems which are not considered in this work are PC-like or server based systems for example ones providing distributed computing, information hosting or processing, web applications, and so on. None of the fundamentals of this work are restricted to embedded systems. Thus, they could be applied to more complex systems as well. However, additional aspects which are not directly essential for the security of a certain embedded system (i.e. protocol security) have to be considered as well. Even though it has not been shown that these aspects can't be handled by the methodology explained in this work, additional case studies would be needed to prove its applicability.

1.2 A Definition of Information Security and System Security

Briefly looking up the terms *information security* and *system security* shows that there is no explicit definition for them. Taking into account all the different use cases of systems that need security, it quickly becomes obvious why there can't be such a definition. Precisely, different use cases mean different potential attack scenarios. This further means different countermeasures are necessary, but not only do the countermeasures define if a system is sufficiently secure, other aspects have to be considered as well. The following sections of this thesis discuss and define the meaning of security in the context of our research.

Attacks, Countermeasures, and Risks

Confidentiality, integrity, and availability are typically used to explain information security. Confidentiality means that information is protected from being read and understood by unwanted entities. Integrity protects data against undetected changes. Availability ensures that a system can be used whenever needed. Even though there are well defined, and much more detailed, explanations for these properties, they have a very theoretic nature and are not universally applicable for different systems. The following example should explain this argument.

Considering an MP3 player, a user would most probably not really care if a person is able to unintentionally read the data on the device (confidentiality issue). Even unwanted changes to the data (integrity issue), although surely unpleasant for the user, would probably be acceptable for them if the device was cheap enough. As long as data availability is not significantly reduced by an attack, the customer would probabbly consider the product as secure enough. Considering a banking card, the situation obviously becomes completely different, even though from a technical point of view the underlying technology of both

devices is not that different. The price of the device cannot be the reason for the different meaning of security, as most banks charge less money for a banking card as than the cost of a typical MP3 player. From this simplistic, but plausible example, we can easily derive the first part of a definition for system security:

Definition 1. *If a system is secure or not, depends on potential risks that come with the system's use cases. Specifically, security mechanisms implemented in the system's hardware and software have to effectively reduce the risk caused by potential attacks to an acceptable degree*

The Life Time Cycle of Secure Systems

Industry, researchers, evaluation laboratories, and attackers are continuously investigating how secure systems can be attacked. This includes developing new attacks, making them more efficient, and making them easier and respectively cheaper to apply. This has to be taken into account when countermeasures are developed and when the life time cycle of a system is being defined. Again, this is explained by a short example.

Around 1998, first attacks based on disturbances in a processor were successfully executed [1, Chapter 8.2]. First, this was done by using very intense flash lights, but soon very expensive laser cutters had to be used to overcome the shields and sensors of a secure micro controller. The attacks had a very theoretic nature as only a few organizations world wide had the necessary equipment and know-how to perform such attacks. However, laser technology improved significantly and become comparatively cheap in recent years, mainly due to the use of diode lasers. Today, 13 years later, every organization that seriously works on secure embedded systems owns a laser attack setup that can automatically scan a chip's surface for vulnerable points. Multiple laser attacks are easily possible, even if performed at different places on the chip, and at different time points.

One may argue that 13 years of investigation of better attacks and laser technology is not a short time. However, we have to look at this relatively and take into account the life cylce of secure systems. Banking cards are valid for a few years, credit cards even longer, and electronic government documents, such as passports can have a life cycle of a decade and longer.

In sum, security is a fast moving target which, has to be ahead of it's times. In other words, it is not sufficient if a countermeasure is only able to protect against an attack today. The countermeasure should be able to protect the system during it's complete life cycle.

Definition 2. *A system is secure if its countermeasures can be considered to be adequate for the full life time of a given system.*

Of course it is possible that new attacks, methods, or technologies get invented that brake a security mechanism. This makes it even more important for the industry to have methodologies in place that allow appropriate and fast reaction to such cases.

Verifiable Security

While it is rather easy to verify if a system is able to functionally cope with a use case, this does not hold for most of the security mechanisms. Typical security mechanisms are

invisible during a regular operation. Their strengths and weaknesses only show up when the system is under attack. Performing an attack for verification reasons is like applying a test vector to a device under test. Like testing cannot show that there is no bug, applying certain attacks cannot show that there are no vulnerabilities. Instead, the development and verification methodology has to be able to give evidence that all requirements are fulfilled to provide the necessary countermeasures.

Definition 3. *A system can only be considered as secure if its countermeasures are successfully verified against all known attacks with respect to the system's use cases.*

Security verifications are done by internal stakeholders, typically within the development or quality assurance team, and external stakeholders. Latter ones can be customers, laboratories, researchers, and other evaluation organizations. For high secure applications security verifications can be explicitly required by customers or governmental regulations.

1.3 Problem Statement

It is known that a system cannot be made secure by taking certain system components into account only [2] while ignoring others. Instead, security has to be a system wide concept that is considered from the very first beginning of the design and development phases.

Hardware/software codesign has shown to be well applicable for the development of devices which have to be optimized [3] for system parameters such as execution performance, chip size, or power consumption. Regarding such parameters security mechanisms can be optimized like other functionalities of a system but when doing so an additional risk of decreasing the security has to be expected.

To solve this, security verification has to find its way to the hardware/software codesign concept. This has to be achieved without restricting it to certain components, abstraction levels, or implementation layers. Furthermore, it should not significantly reduce flexibility or agility of the underlying hardware/software codesign idea.

1.4 The HiPerSec Project

This thesis is the summarized result of the *HiPerSec* project. HiPerSec was funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under contract FFG 816464. Project partners are the Institute for Technical Informatics at the Graz University of Technology and NXP Semiconductors Gratkorn Austria. The project funding mode is a dissertation grant (German: Dissertationsstipendium).

Project Goals

The project partners already had experience with hardware/software codesign methodologies and secure embedded systems. They recognized a flaw for designing, modeling, implementation, and verification of security mechanisms in existing development methodologies. The HiPerSec project was started to investigate in future methodologies without this deficiency. The main research goal in the accepted FFG project proposal was formulated as:

> [...] Develop a Hardware/Software codesign methodology that is able to provide sufficient information for a design space evaluation with respect to security and performance. [...]

This was split in following sub goals:

- A hardware/software codesign approach should be used to allow concurrent development of hardware and software of secure embedded systems - especially smart cards and Java Cards.
- The codesign methodology should allow early architectural optimizations and the verification of
  - system performance and
  - system security.

1.5 Contribution and Significance

This thesis comprises following major contributions:

1. The thesis describes a concept of a hardware/software codesign methodology that allows simulation based verification of system parameters. The main novelty is the embedded verification of security requirements in all design phases and abstraction levels [4, 5]. This allows optimization of security mechanisms in a system wide approach without the risk of unwanted reduction of the system's security level.
2. To do so, meta-information is included into system models and implementations. Such meta-information is shown to be suitable for system optimization [6, 7] and system verification [7–9].
3. Furthermore, a novel verification approach is discussed that combines the hardware/software codesign methodology with a formal Event Calculus model checking approach [4, 5, 10]. This is used to continuously perform formal and semi-formal security verification during the iterative hardware/software codesign process.

1.6 Overview of this Thesis

The remainder of this thesis is organized as follows: Chapter 2 describes related work and the state of the art of secure embedded systems, security evaluation, hardware/software codesign, and system level verification. Chapter 3 explains the proposed novel codesign methodology for secure embedded systems. This starts with a methodology overview, continues how to specify secure systems, and describes system models and their simulation and verification. The chapter is closed by summarizing optimization strategies. Chapter 4 explains how the novel codesign methodology was evaluated. This includes simulation based functional verification, security verification, and formal verification. Chapter 5 concludes this thesis and provides an outlook on possible future work. Finally, Chapter 6 lists the publications related to this dissertation.

Chapter 2

State of The Art Development and Verification Methodologies

2.1 Secure Embedded Systems

Security has to be a system wide concept. According to [2], making a system secure is a design problem that needs an integrated approach in order to be solved. It has to be handled on all abstraction levels such as the protocol level, algorithm level, architecture level, and the implementation levels below. In clear text, security is much more than just adding certain security mechanisms like cryptographic processors or virus scanners. One has just to think about secret cryptographic keys stored in an embedded system and used by a cryptographic processor. Even if the cryptographic processor is designed and developed in a perfect way, this is of no use if the keys are handled in an insecure way. Obviously, the security concept which may be based on the secret keys is affected by system wide components that are not directly related to the cryptographic processors. Several examples are the memory (like ROM or EEPROM where the keys are stored) and parts of the software which copy the keys from ROM to the cryptographic processor.

In addition, security has to be embedded deeply in the development process and considered as early as possible in the design life cycle [11]. Very often, security experts are the only people in the development team that understand all necessary security requirements. However, the system cannot be developed only by security experts who might only do the system architecture. In fact, the implementation of the system has to make the security concept to become real. This is typically done by a number of hardware and software developers who often do not know all the security background. The design and development process has to close the gap between the secure design done by security experts and its implementation in order to overcome the risk that the security concept is damaged by wrong implementation decisions or mistakes.

Different approaches to handle the development of a secure embedded system were investigated. The *Common Criteria* (CC) [12–14] defines the de-facto process for security evaluation. However, it does not define in detail how the development or verification of the system has to happen in order to guarantee the necessary system security. It seems that there is a lack of clear relationship between the CC process and the system development approach [15]. The CC will be discussed in more detail in Section 2.3. Security verification

methodologies are summarized in Section 2.5.

Especially when thinking about embedded systems, where resources like a chip's footprint or the available power or energy are strongly limited, optimizations are essential in making a system comply with the requirements of its use cases. It has to be ensured that undesirable tradeoffs between security and other constraints do not happen [11]. The authors of [2] propose a codesign of domains instead of a codesign of implementations where security is one of the domains (and e.g., graphics and networking are others). Various types of security are discussed and described as elements of a security pyramid: user authentication, confidentiality, privacy, and much more. However, [2] also states the hardware/software codesign problem to find a valid mapping between the implementation of the system and these elements.

One of the smallest secure embedded systems are smart cards, used in high secure applications like banking and e-government. The next section summarizes some related work around smart card development.

2.2 Smart Cards and Java Cards

Smart cards are pocket sized computers. Around the 1970s the idea of placing integrated circuits on a plastic card to provide processing capabilities was born [1, 1.1 The History of Smart Cards]. Since that time, on the one hand, a lot of improvements have been done. Today's smart cards are equipped with contact and contactless interfaces, non-volatile memory, cryptographic co-processors, operating systems like *MULTOS* [16] and *Java Card* [17], and much more. On the other hand, mainly caused by their very small footprint, most commonly used smart cards are still based on an old 8-bit 8051 architecture with some few kBytes of RAM and some few hundred kBytes of non-volatile memory (typically EEPROM or FLASH). Obviously this significantly limits the computational power of these systems.

While MULTOS [16] is a native operating system where the smart card applications are programmed with a native programming language like C or assembler and executed directly on the processor on the smart card, Java Cards provide a Java Runtime Environment including a Java Virtual Machine. Java Card applications (so called applets or cardlets) are programmed in the object oriented high level language Java and compiled to an intermediate representation (e.g., bytecodes). Thus, applets are executable on any smart card, fulfilling the Java Card specification - independently from the underlying hardware or its vendor. The Java Runtime Environment is an additional software layer providing functional features like an API and many security improvements [18] like strong typing of variables (handled during compile time of the applications) and checks of access rights to data stored on the card (executed during the runtime of the applet).

Schneier states in [19] that the main problem of secure embedded systems like smart cards is that the person who is carrying the card is typically not the owner of the data on the card. In other words, the card holder is not the card or data owner. For example, the card holder can be the legal holder of a credit card, or it can be a fraud who is trying to misuse a stolen card. Consequently, the card holder must not be able to access data on the card without restrictions. Notice, that in detail much more entities are involved in the smart card business: the card holder, the card terminal, the data owner, the card issuer,

the card and terminal manufacturer, software manufacturer... The security system on the smart card must ensure that only the card or data owner is under control of the card. In addition, Schneier discusses the technical differences between smart card systems and other typical computers [19]. The functionality of a smart card system is split unusually, as the card is unable to interact with the world (it has no outside peripherals).

Attacks, security mechanisms, and standards are evolving very rapidly [11]. Even though smart cards are very small systems, their complexity is far too high to prove complete implementations. In order to have a methodology for security evaluation, the Common Criteria (CC) were developed. The next section gives a summary of the idea behind a CC evaluation. Formal methods to prove the system are discussed later in Section 2.5.

2.3 Common Criteria for Security Evaluation

Common Criteria [14] is *the* de-facto standard for security evaluation. This thesis does not include a full documentation of the CC process. Instead, it describes the basic concept of Security Functional Requirements (SFRs) which is in common with the methodology discussed in this dissertation. Details of the complete CC evaluation process can be looked up in [12] and [14].

The CC evaluation is a manual and very time consuming task. Based on documentation it is first evaluated to see whether Security Mechanisms are able to encounter threats against the system successfully. Then, it has to be evaluated whether these Security Mechanisms are implemented correctly.

Security Mechanisms are defined by Security Functional Requirements. A set of SFRs is given in the CC standard but this set can be extended if necessary. The CC Security Target (ST) documents the security architecture of a secure product, the so called Target of Evaluation (TOE), by explaining (among other things) the SFRs provided by the implementation.

In addition, the CC standard defines seven Evaluation Assurance Levels (EALs). They contain developer action elements and evaluator action elements. These actions have to be performed (during the development respectively during the evaluation) in order to gain a CC certificate for the TOE. EAL 1 is the minimum level of assurance. For a verification point of view, it is required to perform functional testing of the TOE. EAL 6, in contrast, requires a semi-formal verification and EAL 7 (the maximum level of assurance) a formal verification. However, the CC specification does not state how the (semi-) formal verification has to be done.

The authors of [15] state that the CC process needs a better integration into the engineering processes. CC documents do not support requirement engineering very well. In other words, the documents are good for evaluating a well defined system (which is the purpose of the CC), but they are not very useful in early development phases like the specification phase. However, the security requirements obviously need to be an essential part of this phase when developing a secure system.

The authors of [20] also state a lack of methodological support for development processes in the CC standard. To overcome this, the authors propose a CC based security requirement engineering process which supports the systematic treatment of security requirements in the software development life cycle. The methodology is based on an ex-

tended version of UML (UMLSec), which is able to represent security related information. This UML based representation is used to model the system and to discuss conclusions with experts. A manual requirement inspection phase is used to evaluate the results, e.g., CC documents.

Additionally the authors of [20] state that the security problem is solved when a Security Requirement Rationale Document was written and successfully evaluated - which can be done by their proposed methodology. Contrary to that, in this thesis, the security problem is considered as solved when additionally the implementation was shown to fulfill the specified security architecture. To achieve this, the security verification has to be embedded in a complete development process. As in this work hardware and software development is considered the security verification was embedded in a Hardware/Software Codesign process. HW/SW codesign is discussed in the next section.

2.4 Hardware Software Codesign

In a HW/SW codesign process hardware and software is developed in parallel. This allows optimizations like shorter development cycles, less expense, maximized processing power and component reuse [3]. Key phases of a HW/SW codesign process are (1) specification, (2) modeling, (3) simulation and verification, (4) modification and refinements, (5) model mapping, and (6) implementation and prototyping. Hardware and software are modeled and simulated together in order to gain detailed information about the system's behavior. According to the results, the system components are refined to achieve optimized results. In very early development phases it is even not defined which system modules will be implemented in hardware or in software. This decision is typically done as late as possible during the mapping phase.

In order to use a codesign approach, a system level model that provides on one hand side sufficient flexibility for refinement and mapping and provides on the other hand side enough details to estimate system parameters has to be used. A Transaction Level Model (TLM) as described in [21] can be used to do so. In a TLM the system's communication is separated from the system's computational tasks. Such models can be implemented on different abstraction levels like the specification level, component-assembly level, the bus-arbitration level, or the bus functional level. Different abstraction levels allow different accuracy of estimations, e.g., of the computation time or communication time. The authors of [22] have shown that TLMs can be even accurate enough to perform energy estimations of different bus architectures.

As already mentioned, [2] states that other domains like security shell be considered in a codesign process as well. In [23] this was done to use HW/SW codesign to develop an FPGA based secure hardware component. This hardware allows fast decryption of executable software code. An appropriate compiler automatically generates encrypted and unencrypted code depending on the required security level of the code. With this mechanism, the hardware is able to recognize tampered executables. This is a good example for the optimization potentials of security aspects in codesign approaches but it does not yet consider co-verification of security aspects. Security verification has to be done on a system level and is thus well placed in HW/SW codesign as well as optimization activities. System level verification is explained in the next section.

2.5 System Level Verification

System level verification can be performed on system level models [21]. This can help identifying design problems in early development stages and thus reduces cost and development time. In [24], functional verification of a system is explained by usage of transactors. These transactors are modules that connect the system modules during simulation. In this work, the transactors (which are commonly used to fit incompatible module interfaces) are able to perform checks to the outputs of the connected system modules. Thus, internal states of the system can be easily verified during system simulation. The authors of [25] propose the usage of the Property Specification Language (PSL) to define the properties to be checked during the simulation. PSL is a language for meta-information that can be added to the system model and evaluated by the simulation environment. Such meta-information is also called an annotation. The authors of [26] use such annotations to verify power requirements during system level simulation. The annotations describe the power consumption of all modules in their different power states. Use cases are simulated in order to compute the associated system's power consumption.

Annotations can also be used to increase security as explained in [27]. A Java Card software developer adds Java annotations to the source code if the application. These annotations are evaluated by the virtual machine to detect fault attacks. As this is executed during runtime of the system, this method comes with a performance penalty. In [28], security annotations are used to trace security requirements through all software development phases. The annotations are available at the final software code but are used to check if the security properties defined in the requirement phase are still valid in the verification phase. The authors of [28] explain this as a light weight alternative to formal methods.

Formal methods allow proving certain properties to verify the system. Obviously to prove the right behavior of the system would be the ultimate verification method. Unfortunately, formal methods are very computational intense. Thus, today's system complexity is too high to prove complete implementations [11]. To use formal methods very abstract system models are used. In [29] a design specification on highest abstraction level is used together with linear temporal logic to prove selected system constraints. Also on specification level but with respect to security constraints the authors of [30] used the Z-notation and a theorem prover to verify CC criteria. To do so, the authors modeled states and operations. Similarly in [31], a state transition model was used to represent and formally verify the secure hardware design of a microprocessor.

Instead of theorem provers, model checking is used in [32], [33], and many more publications. In model checking, it is shown that two models are formally equivalent with respect to certain properties. Typically, one of the models represents the system; the other one represents the specification. Thus, it can be proved if the system model fulfills the specification. In [32], a verification of a security policy was done. In [33], a SystemC model was converted into an abstract formal model in order to finally execute the model checking and prove the SystemC model's correctness.

Besides formal methods, other ways to perform system level security verification exist. In [34], simulated fault injection is used to verify the impact of stuck-at faults, optical fault induction attacks, and power glitches. All faults are injected to memory or bus components of the system. Similarly, the authors of [35] propose (simulated) model based testing where the test vectors are automatically generated form an UMLSec model. The authors of [36]

explain automated static security code checks by searching for a predefined set of patterns and rules in the source code of the system. One difficulty of this approach is that the code checker has to take compilation, synthesis, and optimization techniques of other tools into account. While local analysis of small functions can be relatively easy, global analysis of system wide patterns (potentially spread over hardware and software components) is much more challenging.

The Event Calculus [37, 38] is an action formalism that can be used for security verification as well. In this thesis the Event Calculus is used to perform a model checking based system wide security verification. Thus, it is explained in the next section in more detail.

2.6 Event Calculus

The *Event Calculus* and *Commonsense Reasoning* (based on the Event Calculus) are explained in [37] and [38]. The Event Calculus (EC) is an action formalism based on a first order predicate calculus. The main idea of the EC is to setup a solid theoretical basis for actions and their temporal effects. The EC is able to represent phenomena like actions with direct, indirect, and non-deterministic effects. It can be used to formalize compound and concurrent actions as well as continuous change.

The EC can be used for three different reasoning tasks. A (1) deductive task is used when "what happens when" and "what actions do" are given while "what's true when" is required. In other words, the outcome of a known set of actions is searched. An (2) abductive task searches for "what happens when" while "what's true when" and "what happens when" are given. This is equivalent to searching for actions that lead to a certain result. An (3) inductive task takes "what's true when" and "what happens when" and results in "what actions do". This means searching for general rules that describe a temporal behavior.

The EC is based on actions (or events), fluents (whose values are changing over time), and time points. It includes a set of axioms for following predicates. Notice that there are more predicates described in [37, 38] which are not used in this thesis.

- Initiates(α, β, τ): the event α at time point τ causes the fluent β to hold.
- Terminates(α, β, τ): the event α at time point τ causes the fluent β to cease.
- Happens(α, τ): the event α happens at time point τ.
- HoldsAt(β, τ): the fluent β holds at time point τ.

EC is strongly based on handling non-effects and non-occurrences. This means that (1), unless explicitly defined, actions do not have effects and (2), unless explicitly defined, actions do not occur. These are fundamental rules of the EC and solved by so called *circumscription* or *completion*. The *decreasoner* [39] tool and the *f2lp* [40] tool provide commonsense reasoning based on EC.

The author of [41] used EC for formal analysis of security requirements. The author performed threat analysis based on an abductive commonsense reasoning task. A system's security requirements and their dependencies were modeled by EC axioms. Using the decreasoner performing an abductive task results in sets of actions (if existing) which illustrate unwanted vulnerable effects of the system.

The authors of [42] use the EC and abduction to find solutions how security requirements can be satisfied. From the set of potential solutions (reported by the decreasoner tool) the developer identifies (technically) unrealistic solutions. The developer refines the EC model by adding further information. This reduces the design space for the system under investigation. The EC axioms to be used for abduction are generated automatically from of a meta-model used for requirement engineering.

Similar approaches without using the EC exist. In [43], a framework for representing and reasoning of abstract security properties is discussed. The described approach allows the user to define own properties on a very high abstraction level like "knows", "send", and "receive". Similar to the EC, objects and axioms are defined and a theorem is to be proofed. To perform the formal verification the *Coq* interactive proof assistant is used. The proof has to be developed by the user and is verified by Coq. Thus, contrary to using the EC, the verification is a manual task.

2.7 Summary

Security has to be a system wide concept of embedded systems. Huge technical differences exist between secure embedded systems and "normal" embedded systems. Developing a secure embedded system requires security aspects to be integrated in

- the design,
- the development process, and the
- system verification methodology.

The design has to ensure that security requirements are fulfilled accordingly. To efficiently manage rapidly evolving attack methodologies and techniques, the development process has to be able to react appropriately to changing security requirements. Finally, the system verification must be able to reveal if the implementation actually makes the security concept real.

The Common Criteria process defines the de-facto process for security evaluation of information systems. However, it appears that this documentation based process needs to be better integrated into development methodologies. Security has entered the codesign methodology as its own design domain and as cross-domain verification target. Today's security verification is done on high abstraction levels (e.g., the requirement level). Abstract system models without clear relation to the system's implementation are often verified by formal methods. However, there still exists a gap in security verification of system models and their implementations.

The goal of this thesis is to close this gap by

- seamless integration of functional security requirements in a hardware/software codesign process.
- This includes automated verification and detection of unwanted security violations
- in all development phases and abstraction levels
- while performing system wide optimization.

Chapter 3

Novel Codesign Methodology for Verification of Secure Embedded Systems

As mentioned in the problem statement in Section 1.3, security verification must be included in the a design and development process. This was done in the methodology discussed in this thesis. The present chapter summarizes how security verification for embedded systems was included in an existing codesign methodology. Section 3.1 gives a high level view on the methodology discussed in this thesis. Subsequent sections explain several methodology aspects. The detailed descriptions are done in separate publications appended in Chapter 6.

3.1 Methodology Overview

The main flow of the discussed methodology is shown in Figure 3.1. It can be split into four phases: specification, modeling, simulation, and verification. The first two phases are described in separate subsequent sections. Simulation and verification are combined into one subsection. Figure 3.1 includes the chapter numbers of the corresponding publications in 6. Reprints of the related publications are given in this chapter.

A system's behavior is defined by its specification. As shown in Figure 3.1, for this thesis this is split into a functional specification and a security specification. While functional specifications are not the focus of this thesis, security specifications are because they are one outstanding initial point when developing a secure embedded system. The split into two separate specifications was done because this seemed natural for secure embedded systems: while the functional specification is directly defined by use cases, the security specification is derived from a security analysis (as for example described in the Common Criteria process in Section 2.3). Section 3.2 gives more details about security specifications.

Both specifications are used in an iterative process to develop the system. Development starts with a pure functional high level model; iterations are used to verify, refine, and optimize the system until it is in a state where it can be used for production. When refining a system's module, its abstraction level is changed to a more detailed level. The most detailed level is the implementation level which can be used for production of the system.

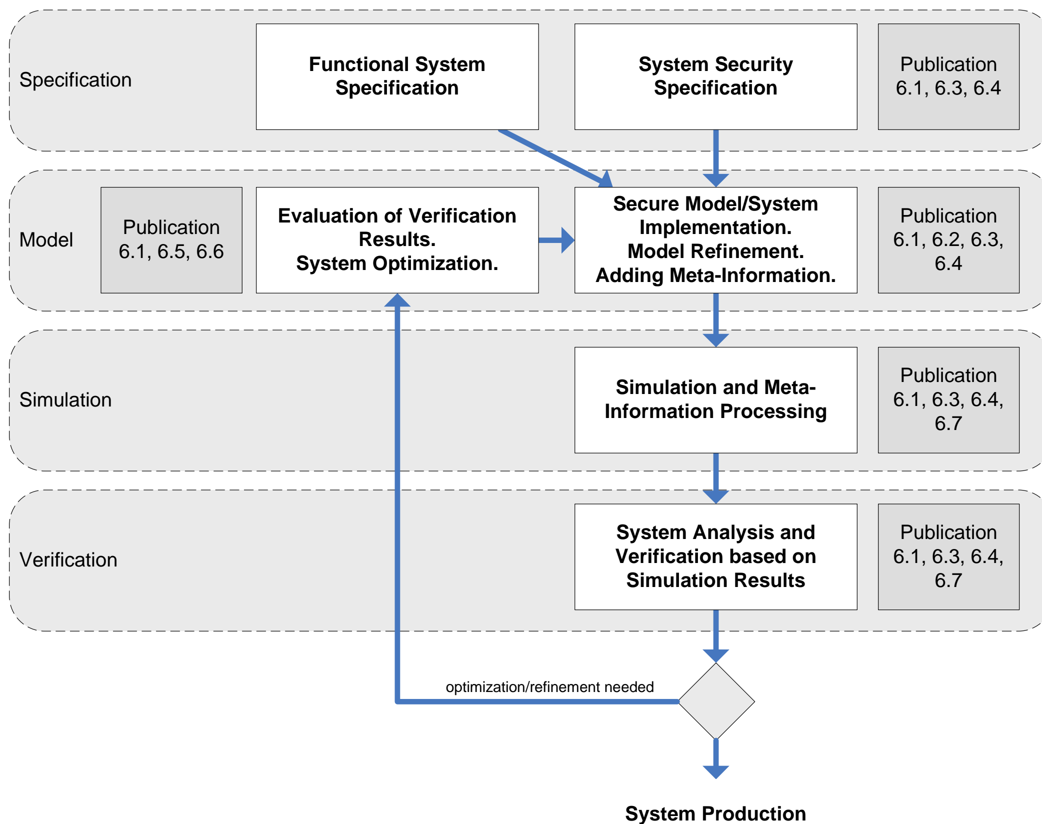


Figure 3.1: Overview of the Novel Codesign Methodology

The model includes a functional simulatable description and additional meta-information are applied to system modules. Such meta-information can be processed during (1) compilation time, (2) simulation time, (3) or execution time. The methodology does not define syntax or semantic for used meta-information, as this depends on the development environment as well as on verification and optimization goals. Section 3.3 explains modeling, meta-information, refinement, and abstraction levels. Section 3.5 describes system optimization approaches.

Both functional verification and security verification are based on a simulation approach. The system's model is simulated to verify its parameters. These parameters are deducted from the functional model itself and from the examination of included meta-information. If the resulting system parameters do not match the expected requirements, a refinement iteration respectively an optimization iteration must be done. Simulation based system verification is explained in Section 3.4.

3.2 System Security Specification

A security specification can be given in different manners. It can be done in a purely formal way by a model, in a textual description, or by usage of tools such as DOORS[1].

[1] http://www-01.ibm.com/software/awdtools/doors/

However, a security verification methodology has to be independent from the modality and the content of the security specification. The one which was used for this thesis is given in parts of the Common Criteria Security Target (ST) covering functional aspects of the system. However, the methodology is not restricted to using an ST. The ST was already explained in Section 2.3.

A security specification consists of security requirements. For an ST, this is a set of Security Functional Requirements (SFRs). These requirements have to be fulfilled by the system in order to provide the security mechanisms that were chosen to provide countermeasures against identified attacks. Consequently, some system modules rely on such requirements, others provide these requirements.

Section 6.3 describes how an ST can be used to formulate such dependencies in a manner that they can be used as meta-information for security verification. The concept of meta-information is describedin detail in the subsequent Section 3.3. At this point, it is only important to understand that meta-information is a textual description of parameters embedded in source code of a model or a system's implementation. The concept is also shown in Figure 3.2.

Common Criteria defines that an ST has to include definitions and dependencies of security mechanisms and SFRs. Security specifications which follow this idea of an ST can be used (1) to extract a list of SFRs which have to be used in a system and (2) to extract security verification rules to evaluate the SFR dependencies. The security requirements can be formulated as meta-information and applied to system modules in the corresponding source files. Together with the verification rules, a simulation based security verification can be performed, as described later.

Notice that SFRs are abstract definitions. Like the ST, they are independent from their certain implementation. Thus, using meta-information instead of concrete implementations is a valid approach.

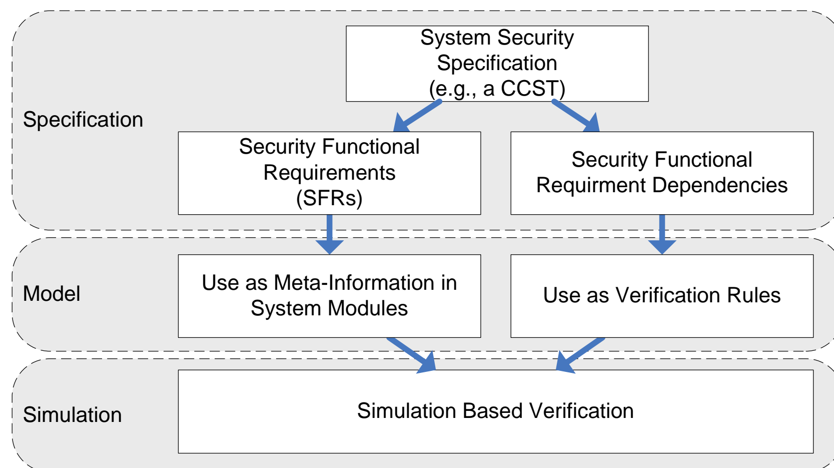


Figure 3.2: Extraction of Rules from a Security Specification

SFR Dependencies

SFR dependencies are mapped to a *requires/implements* scenario where one system module requires an SFR and another module provides the SFR's implementation. Consequently, system modules are annotated with appropriate meta-information holding these SFR dependency predicates *requires* and *implements* for a certain SFR. Obviously, such a rule is only fulfilled if modules that require an SFR access modules that provide the same SFR. As explained in Section 3.4 this temporal access information is generated during the system simulation.

Section 6.4 extends the idea of *requires* and *implements* by more predicates. The essential ones are *fulfills*, and *expects*. As *requires* and *implements*, *fulfills* and *expects* build a logical pair. However, for latter ones, the access direction is reversed: a module annotated with *expects* has to be accessed by a module that is annotated with *fulfills* while latter module implements the SFR. The relationships of these predicates are drafted in Figure 3.3.

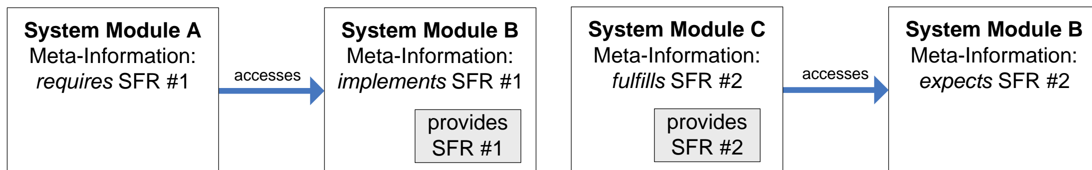


Figure 3.3: Relationships of SFR Dependency Predicates: (a) requires/implements and (b) fulfills/expects

Using Formal Security Specifications

In Section 6.1 SFR dependency rules are formalized to a formal security policy. Such a policy can be used by formal approaches to show that certain system properties are valid. Section 3.4 explains how this can be used for a (semi-) formal simulation based security verification approach.

The underlying formal mechanism used is the Event Calculus. As explained, the Event Calculus is a formal description of temporal actions and reactions. It is based on abstract properties respectively predicates instead of concrete implementation artifacts. Thus, the verification process is independent from the implementation or abstraction level. This allows using the Event Calculus in an iterative refinement approach as described in this thesis.

The Event Calculus defines generic events, generic properties (so called fluents) and modules. As it does not define any concrete relation between these components, a domain description had to be defined. Section 6.1 describes a domain that defines dependencies between SFRs without usage of certain SFRs and system modules. This is done in a similar but generic and formal way as described before with SFR dependency rules. Thus, it is possible to use the domain description for security verification of any arbitrary system - independent from the use cases or its security architecture.

Axioms 3.1 and 3.2 describe the most important rules in the domain description. *initiates* and *terminates* are defined by the Event Calculus. M can be any system module

and R stands for an SFR. T represents a concrete but abstract time point. $requestsR$ and $satisfy$ are events, while $unsatisfied$ is a fluent.

Axiom 3.1 states that if a module M requests an SFR R it immediately becomes unsatisfied (because it requires an SFR but does not implement it). As defined in Axiom 3.2 a $satisfy$ event has to happen in order to terminate the unsatisfying system state. Obviously, any unsatisfying system state must be resolved in a secure system. This is the underlying verification principle in the formal verification approach described in Section 6.1.

$$initiates(requestsR(M, R), unsatisfied(M, R), T). \tag{3.1}$$

$$terminates(satisfy(M, R), unsatisfied(M, R), T). \tag{3.2}$$

Axioms 3.3 and 3.4 describe the necessary circumstances for the $satisfy$ event. The basic meaning is straight forward but the axioms are a bit tricky to read. Axiom 3.3 can be read as: Module M_1 has to be unsatisfied with SFR R_1. M_1 has to be connected to a second module M_2 while this module has to provide the SFR R_1 which is needed by M_1. In addition, M_2 must not be unsatisfied with any other SFR. If all these fluents hold at a time point T, then the $satisfy$ event happens at T.

$$\begin{aligned}
&(\,holdsAt(unsatisfied(M_1, R_1), T)\ \&\\
&\ holdsAt(connected(M_1, M_2), T)\ \ \&\\
&\ holdsAt(provides(M_2, R_1), T)\ \ \&\\
&\ -?[R_2] : (holdsAt(unsatisfied(M_2, R_2), T))\\
&) \rightarrow happens(satisfy(M_1, R_1), T).
\end{aligned} \tag{3.3}$$

Axiom 3.3 does not explicitly forbid other scenarios also to lead to a $satisfy$ event. This is done in Axiom 3.4. Together, both axioms build a bi-implication which is split into two separate implications. Axiom 3.4 can be read as: if $satisfy$ happens on module M_1 with SFR R_1, there must exist a second module M_2 which is connected to M_1, which provides R_1, and which is not unsatisfied with respect to any other SFR.

$$\begin{aligned}
&happens(satisfy(M_1, R_1), T) \rightarrow (\\
&\ holdsAt(unsatisfied(M_1, R_1), T)\ \&\\
&\ ?[M_2] : (\\
&\quad holdsAt(connected(M_1, M_2), T)\ \&\\
&\quad holdsAt(provides(M_2, R_1), T)\ \&\\
&\quad -?[R_2] : (holdsAt(unsatisfied(M_2, R_2), T))\\
&\ )\\
&).
\end{aligned} \tag{3.4}$$

The complete domain description of Section6.1 consists of 26 axioms. Appendix A.1 lists a textual version of all of them.

As mentioned, the domain description is independent from the system to be developed. Thus, it is also independent from the security architecture respectively from a security

specification. A concrete formal specification (a security policy) has to be developed (e.g., as explained before from an ST) in order to verify the system. Such a security policy consists of Event Calculus axioms. In contrast to the domain description, the security policy refers to concrete SFRs to describe their dependencies. Axiom 3.5 gives one example to understand the concept. It states that if the module M_1 enables the SFR r_master_key, it requires another module to provide the SFR $r_derived_key$ or the SFR $r_key_storage$. Explanations for the SFRs and a more comprehensive security policy are given in the case study in Section 4.4.2.

$$\begin{aligned} & happens(enables(M_1, r_master_key), T) \rightarrow (\\ & \; happens(requestsR(M_1, r_derived_key), T) \mid \\ & \; happens(requestsR(M_1, r_key_storage), T) \\ &). \end{aligned} \tag{3.5}$$

3.3 Secure System Model

As mentioned, the development process discussed in this thesis is an iterative model based approach. The starting point is a purely functional model. This model is continuously refined until the modules are implemented in a level of detail which can be used for production. While doing so, the system model passes different abstraction levels which all have their own intent and necessity. The abstraction levels with section numbers of the related publications are depicted in Figure 3.4. The Security Annotated Functional Level Model (SAFLM) and the Secure Functional Level Model (SFLM) are novel in the development process described in this work.

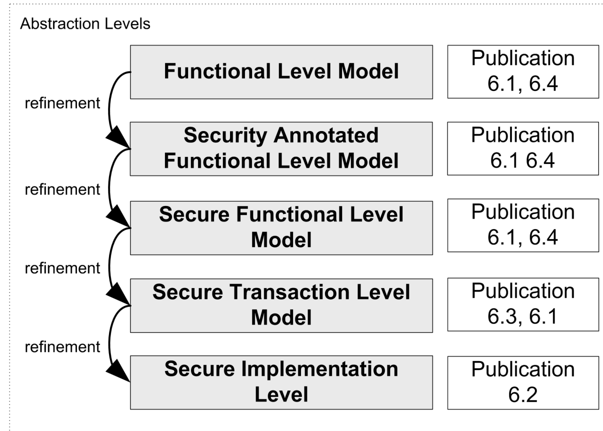


Figure 3.4: Abstraction Levels of the Novel Codesign Methodology

Abstraction Levels

As described in Section 6.4 modeling starts with a pure Functional Level Model (FLM). This model does not provide any security mechanisms. It is used only for functional verification of the system's defined use cases.

However, it was shown that security concepts should be included in the earliest stages of a system's development. Thus, meta-information concerning the system's security specification is added to the FLM in order to get the SAFLM. Such meta-information does not influence the functional behavior of the model, as it is a pure textual description (e.g., of SFRs). But, this meta-information can be evaluated by the simulation environment as described in Section 3.4. Having the functional model and the security related meta-information allows analyzing the model with regard to where security mechanisms have to be included in the system.

As a next step the security mechanisms are modeled as well. This results in the SFLM (also described in Section 6.4). The SFLM is the starting point for all optimization work, because the model includes all security relevant and non security relevant aspects. Subsequent refinement iterations can be done similarly to common hardware/software codesign processes, as these processes do not need to distinguish between security and non security mechanisms.

Because the meta-information is still available, the complete design space exploration can be done without risk of security violations caused by incorrect optimizations. With every iteration, a simulation based verification ensures a consistent security architecture. As described in Section 6.3, the next detailed abstraction layer in the Secure Transaction Level Model (STLM) where hardware/software partitioning takes place.

Further refinement is done in order to get the system's final implementation (Secure Implementation Level, SIL). This is not an abstract model level anymore as modules are represented by concrete implementations. As the meta-information is maintained and refinement as well during the iterations, it can be used to verify the final implementation as described in Section 6.2. The meta-information does not need to stay in the final product if it was used for verification purposes only. Thus, it has no negative impact on the product cost (e.g., due to bigger code size) and cannot reveal any information to potential attackers. However, as shown in Section 3.5 such meta-information can be used for optimization purposes during runtime as well.

Vertical Transactors

In practice, the abstraction levels are not that strictly separated as described before. Instead, the change from one level over to the next one is a continuous process, again performed as particular iterations. As each iteration is finalized with a system verification, the risk of recognizing errors too late is reduced. In addition, if every design decision is handled by its own iteration, the resulting consequences (e.g, with respect to performance) can be immediately detected and corrected if necessary.

This refinement concept makes it necessary for system modules modeled or implemented on different abstraction levels to interact. Obviously, this is nothing which will be included in the final system for production. Thus, this should not be handled by the common system module interfaces. Instead, Section 6.1 describes novel *Vertical Transactors* which are used to provide a tool for interfacing between different abstraction levels. Unlike common transactors, vertical transactors do not connect incompatible system modules but provide a "short cut" between modules on incompatible abstraction levels. As a matter of course, when the system is finished for production, all system modules are on one and the same abstraction level (the SIL) and vertical transactions are no longer needed.

3.4 Simulation Based Verification

Simulation based verification has many benefits for secure embedded system development. In contrast to a verification on a hardware platform (e.g., with emulator boards or FPGAs), a simulation provides better observability and controllability. Both are very important and critical when developing secure systems, because the final system should neither provide one nor the other to a potential attacker. Thus, the system is developed in a way which reduces observability and controllability.

A simulation provides both aspects without adding potential back doors to a system meant to be secure as the system stays unchanged. Section 6.7 explains how this can be used to verify security mechanisms which are otherwise difficult to test. Simulated fault injection provides test vectors that would be difficult or even impossible to apply. Furthermore, higher controllability allows applying the test vectors directly to the simulation environment. This allows a better verification performance.

Applying and Evaluation of Meta-Information in System Models

On its own the simulation environment cannot extract enough information from the system module descriptions to perform an analysis for a useful verification or optimization. Additional information has to be placed in the source code of the system modules by the developer. This meta-information can be evaluated during compile time, simulation time, or run time of the system. The semantics are not pre-defined by the underlying modeling language. This makes meta-information so useful for verification and optimization reasons (optimization explained in Section 3.5) because the meta-information can represent any parameters related to the system under development.

Simulation based security verification with meta-information added to system modules is described in Section 6.4 and shown in Figure 3.5. Meta-information about required and implemented security requirements (as described before in Section 3.2 is added to system modules. During simulation, the simulation environment is used to extract the dynamic behavior of the system model. This is done based on the functional descriptions of the system modules. This dynamic behavior is verified against the functional specification but also together with the meta-information against the security specification. Section 6.4 shows how this is done for the SAFLM and the SFLM.

Section 6.3 applies this concept to STLMs containing hardware and software components. In such a case the simulation environment needs to extract call graphs for the software components and data flow graphs for the hardware components. Based on these graphs, the security verification rules as explained before can be evaluated.

(Semi-) Formal Simulation Based Verification

Section 6.1 explains a formal way of a security verification. One can argue that it is in fact a semi-formal verification, as some of the verified parameters are generated during the system's simulation. Thus, the verification result still depends on the choice of useful input data during the simulation.

Simulation based verification has one big advantage over purely formal methods. Typically, purely formal methods are either based on highly abstracted models without any incorporation to any real implementation, or restricted to very small parts of a system

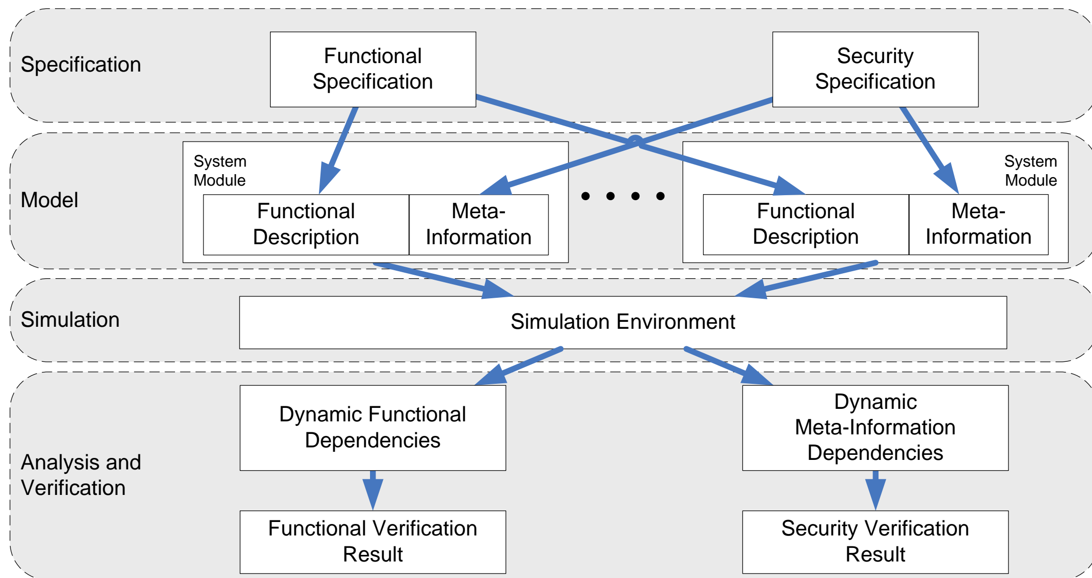


Figure 3.5: Simulation Based Verification Approach with Meta-Information

(due to the verification complexity). Thus, purely formal methods are not applicable for iterative system wide concepts as proposed in this thesis. In contrast, a simulation based approach is. Even if a pure formal approach would theoretically give a more precise and exhaustive verification, one can argue that a fast and flexible but semi-formal approach is more helpful during system development today. Once pure formal methodologies have improved significantly, this might change.

Figure 3.6 shows an overview of the security verification approach of Section 6.1. The security specification is formulated as a formal security policy using Event Calculus axioms. Additionally, meta-information related to SFRs is added to the system modules. During simulation, the simulation environment generates trace files with respect to module and SFR activities. These trace files are formally model checked against the security policy. It id immrfiszrlx recognized if any use case causes a behavior out of the security specification.

In addition, the formal security policy is used in a formal model finding process. To do so, the Event Calculus is used in an abductive task, while model verification is a deductive task. This verifies the consistency of the security policy itself. If no model that matches the security policy can be found, it is impossible for any system to fulfill the policy. Consequently, the security policy has to be revised.

Furthermore the concept in Section 6.1 includes adding platform constraints to the verification process. To keep the illustration simple, this is not shown in Figure 3.6. Platform constraints narrow the design space because they describe certain limitations of a chosen hardware platform. Such constraints include Event Calculus axioms that explicitly forbid relying on hardware features which do not exist on the selected platform.

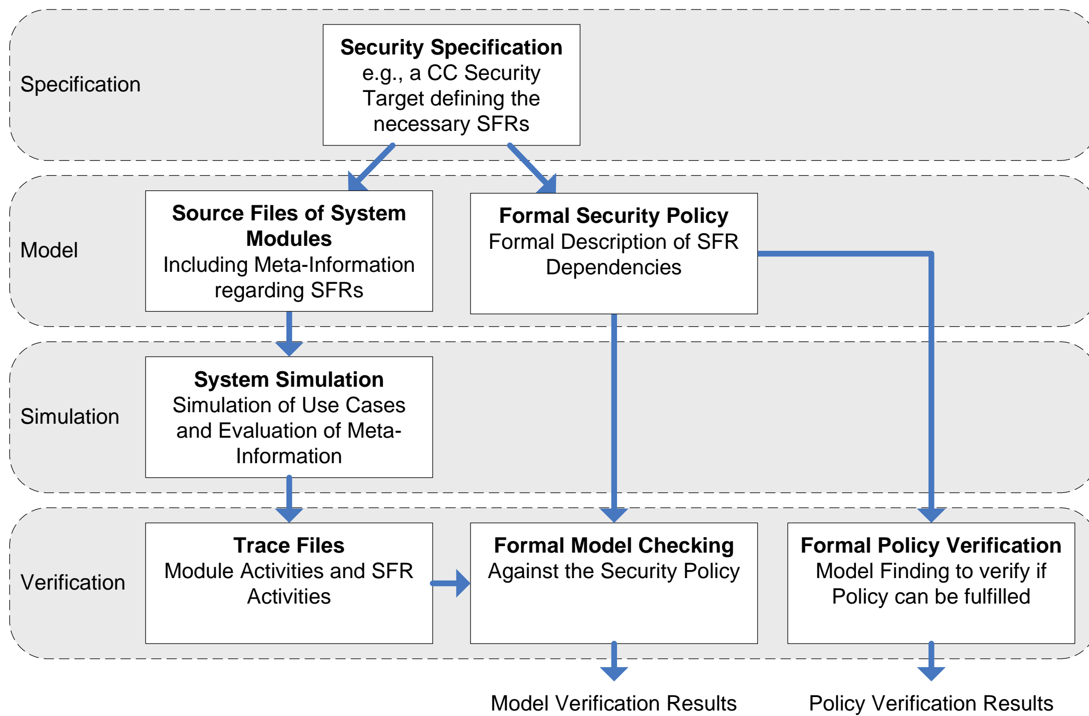


Figure 3.6: Overview of the (semi-) formal Simulation based Security Verification Approach

3.5 System Optimizations

Section 6.6 explains one successful cross abstraction level optimization for secure embedded systems. Software properties that result from the most abstract software representation (the application) are applied together with low level hardware platform properties to archive an execution performance optimization: meta-information describing which data in memory is frequently used is added to the according source files. This can be done by the software developer, as he or she is typically able to estimate the memory access behavior of the application under development very well. In contrast to the security verification described before, this meta-information is evaluated by the runtime environment (instead of the simulation environment). With respect to the meta-information, the runtime environment allocates the memory at locations where the hardware platform supports faster memory accesses.

Notice that the same optimization could be done without meta-information if the runtime environment is able to provide the estimations done by the developer. However, for small embedded systems this can be a not achievable task due to resource restrictions.

The concept explained in Section 6.5 automatically generates meta-information to perform an execution performance optimization similar to the one described above. In contrast, a static analysis of software components is used to do so. Based on this analysis the meta-information causes the runtime environment to increase the temporal locality of multiple data fields. This is based on grouping data fields which are used temporally closely together very often.

Using meta-information during the system's runtime for optimizations has a big benefit:

the evaluating system can react adequately to such meta-information. One optimization strategy can perform very well on one hardware platform but does not necessarily work as well on other platforms. Adding a static optimization would reduce the re-usability and flexibility of the optimized modules. In contrast to that, meta-information can easily be ignored or even removed whenever needed, as the meta-information has no impact to the functional description of the module.

The mentioned optimizations are just examples to show that cross abstraction level optimization can work by using meta-information. The methodology discussed in this thesis does not limit any optimization approaches or targets. Using multiple meta-information in system modules does not cause them to influence each other. Thus, the concept of using meta-information for verification and optimization goals is well applicable for development of secure embedded systems.

Chapter 4

Methodology Evaluation and Case Studies

The novel codesign methodology discussed in this thesis could be split into three different concepts: (1) simulation based functional verification, (2) usage of meta-information, and (3) simulation based security verification on different abstraction levels. These concepts are evaluated in different case studies summarized in this chapter.

First, Section 4.2 describes the functional verification of an anti-tearing mechanism (described in more detail in Section 4.2.1. Section 4.3 evaluates the concept of meta-information in a codesign process. Finally, Section 4.4 summarizes case studies for simulation based security verification approaches. Before stepping into the methodology evaluation details, Section 4.1 explains some basics which all the case studies have in common.

4.1 Common Basics for the Case Studies

All case studies are based on smart card systems. Smart cards are typical secure embedded systems. Their integrated design allows them to be used for high secure use cases such as banking and e-government. Originally intended to be pocket sized plastic cards, today they are also used as secure elements embedded in more sophisticated systems such as smart phones. The secure use cases, the small chip size, and the power constraints (smart cards are very often powered externally by an electromagnetic field) of smart cards - parameters which are typically rather contradicting - make smart card development a good application area for optimization and security verification methodologies. Thus, it seemed obvious to select such systems for the evaluation of the novel codesign methodology in this thesis.

To give an idea of the computational capabilities of smart cards the most important hardware parameters should be mentioned here. Typically, smart cards are still based on an 8-bit or 16-bit micro controller with some few kByte of RAM and some few hundred kByte of non-volatile memory (e.g., EEPROM). Due to size limitations, most smart cards do not have their own power supply. They are powered externally by the smart card reader. This can be done by a contact interface or by a contactless interface (inductive coupling) which results in strict power consumption limitations. To support cryptographic algorithms with an appropriate execution performance one or multiple co-processors are added to the system. More details about smart card systems can be found in [1].

Nevertheless, smart card use cases have very strict limits for maximal execution times. One has just to think of a queue of people at the subway entrance or passport check point to understand why. In addition, extra computational overhead is added by security mechanisms. The Java Card [17], for example is a smart card system which provides a full version of a software virtual machine. Interpreting bytecodes instead of direct execution of native instructions enhances the security level, but obviously also increases the computational effort. Nevertheless, Java Cards are more popular than ever before. Figure 4.1 gives an overview of a Java Card. Applications, implemented in a subset of the Java Language, are separated by a firewall and executed on a virtual machine. The Java Card Runtime Environment provides an API tailored for smart card use cases. A native operating system provides the low level software layers which, among other things, interacts with the smart card hardware (e.g, interfaces).

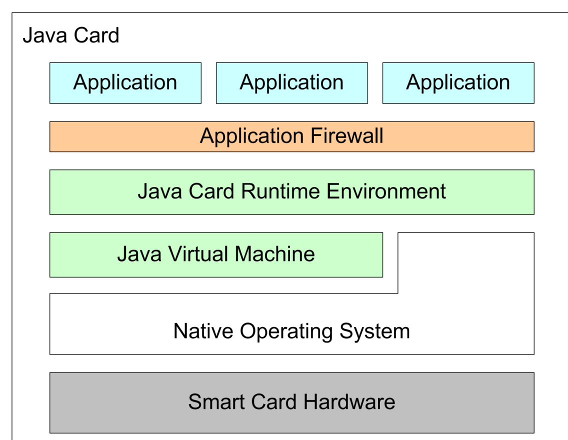


Figure 4.1: Java Card Overview

Typically, security features are spread allover the entire Java Card system. Some of them are implemented in applications; others are defined by the Java Card specification and implemented in the Java Card Runtime Environment or the native operating system, and again others are implemented on the hardware layer. As already mentioned, all of them have to be seen together as a system-wide security concept instead of single punctual secure solutions.

4.2 Simulation Based Functional Verification

First, let's clarify the difference between functional verification and a security verification which is evaluated later in the thesis. Functional verification gives information if functional features behave in their defined way. A functional feature is mainly defined by its input, its output, and possibly a resulting internal state change. Thus, a functional verification can be done by applying an action (or in other words a test vectors) to the system and checking its reaction.

In contrast to that, security mechanisms cannot necessarily be defined by their resulting output - respectively, it could be defined in a way which is difficult or important to verify. This should be explained by an example: one typical security mechanism is to provide

reliable random numbers. Such random numbers are used in cryptographic algorithms. Thus, they must not be predictable or manipulable by an attacker. Smart cards typically provide a hardware random number generator which performs statistical tests and is physically protected (e.g., against laser attacks). Obviously a functional verification of these parameters is hardly possible.

Section 4.2.2 summarizes the functional verification of the anti-tearing mechanism of a Java Card. This mechanism is explained beforehand in Section 4.2.1. The case study in Section 4.2.2 shows that a functional verification on the basis of a system simulation is not only possible but also can be highly efficient.

4.2.1 Java Card Anti-Tearing Mechanism

The Java Card anti-tearing mechanism is one of the most important security mechanisms of Java Cards. As mentioned, smart cards (and Java Cards as well) do not have their own power supply. Instead, they have to rely on an external powering by the smart card reader. If the smart card reader switches off the power supply or the card is torn out of the reader during a write operation into the non-volatile memory on the smart card, this obviously immediately aborts the write operation. As the write operation was not finished, the data in the memory can be inconsistent. Anti-tearing provides a mechanism to write into non-volatile memory without the risk of data inconsistencies caused by unexpected power loss [18].

Figure 4.2 depicts such a scenario. A smart card changes from system state A to system state B because of a write operation to non-volatile memory. Notice, that this is a persistent system state that remains in case of a reset. Another write operation is aborted. Consequently, the card stays in an inconsistent system state.

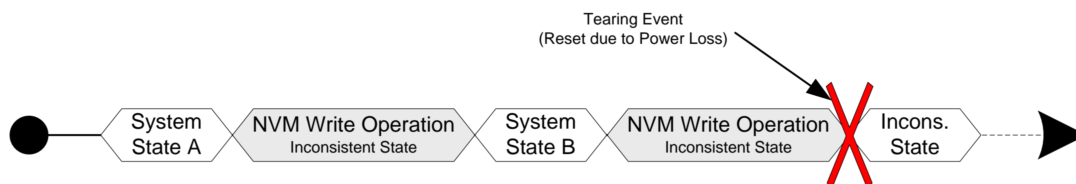


Figure 4.2: The Tearing Problem

Oestreicher explains two different approaches for anti-tearing mechanisms in [44]: *Old Value Logging* makes a backup of data in non-volatile memory before it is going to be overwritten. *New Value Logging* stores the new data into a buffer before the old data is overwritten. If a tearing event happens, the card is able to restore a consistent version of the data at the next power up phase. Each tearing-save write operation to non-volatile memory is split into several particular write operations. Each of these write operations change the internal state of the Java Card. Thus, each of these write operations have to be verified carefully if there is a risk that the card stays in an inconsistent state after a tearing event.

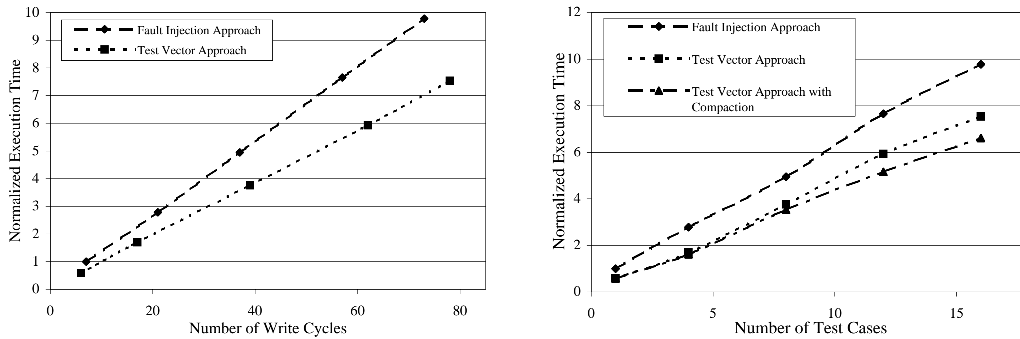
4.2.2 Case Study 1: Verification of an Anti-Tearing Mechanism

Section 6.7 explains a simulation based verification approach of an anti-tearing mechanism described in Section 4.2.1. The verification is based on simulated fault injection. The simulated fault is a tearing event that is injected for every write operation to non-volatile memory during execution of use cases. The verification methodology is tailored to a functional verification of arbitrary anti-tearing mechanisms.

The verification is split into two phases. In the first phase, a test case (a piece of software which is using the anti-tearing mechanism in a certain way) is executed without fault injection. This phase is used to collect memory access data (for non-volatile write operations). This data is used to generate test vectors which are finally applied in a loop one after another in the second phase.

As explained in Section 6.7 and shown in Figure 4.3(a), a significant verification performance gain can be archived in comparison to a common fault injection approach. The time values are normalized: the shortest verification time is set to 1. 16 test cases were executed, which resulted in 70 write operations in total and a performance gain of 20% in comparison to a common fault injection.

Furthermore, the approach allows compaction of test vectors which reduces the number of write operations. Figure 4.3(b) shows additional performance gain of 17% for the test cases chosen in Section 6.7.



(a) Verification Performance without Test Vector Compaction

(b) Verification Performance with Test Vector Compaction

Figure 4.3: Performance of Functional Verification of an Anti-Tearing Mechanism with and without Test Vector Compaction

4.3 Evaluation of the Concept of Meta-Information

As explained, the usage of meta-information is a key concept of the codesign methodology discussed in this thesis. The big advantage of using meta-information, is that it is independent from the annotated functional module as well as independent from the modeling or programming language. In this thesis, meta-information is used for system optimization and system analysis. The first one is explained in Section 4.3.1, the latter one is summarized in Section 4.3.2. Together, these sections should demonstrate the applicability of

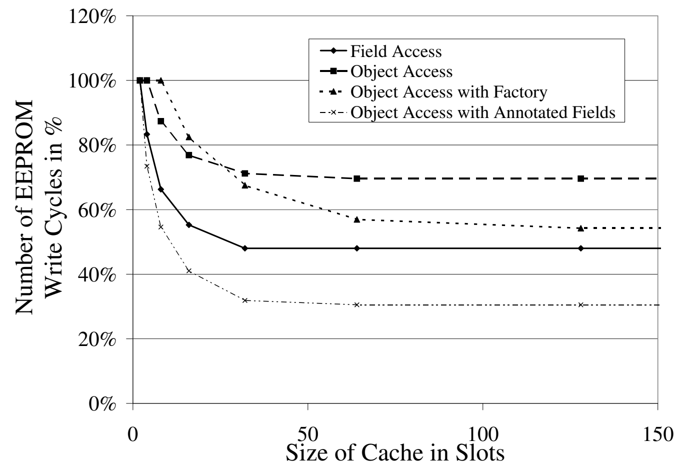


Figure 4.4: Performance Gain for an Anti-Tearing Mechanism with Meta-Information based Data Allocation

meta-information in a codesign methodology.

4.3.1 Case Study 2: System Optimization based on Meta-Information

Section 6.6 and Section 6.5 describe system optimizations performed on the basis of meta-information. Again, the anti-tearing mechanism explained in Section 4.2.1 was selected for the use cases. As mentioned, anti-tearing splits up single non-volatile write operations which have to be tearing save to several particular write operations. Such write operations are very time consuming. This makes the anti-tearing mechanism one of the most performance consuming security mechanisms. The optimization approach in this case study targets the reduction of costly write operations.

Section 6.6 explains how frequently used data can be marked with meta-information and consequently allocated on one EEPROM page. EEPROM is usually organized in pages where the page is written in one write operation - independent from how many bytes of a page have changed. Thus, allocating data field frequently used in one use case on one dedicated EEPROM page reduces the write operations during anti-tearing.

The resulting performance gain is shown in Figure 4.4. The optimization uses a cache. Different cache sizes up to 150 cache slots (each slot has some few bytes depending on the implementation) were evaluated. Using the cache without meta-information resulted in a performance gain of 16% to 50%. With meta-information the performance gain was increased by 26% to 70%.

Section 6.5 extends the optimization of Section 6.6 by grouping frequently used data fields. This is done on the basis of a static analysis of the Java Card application. Data fields are grouped in a way so that during regular use cases the necessary write operations to EEPROM are minimized.

Equation 4.1 shows the number of necessary write operations for an anti-tearing mechanism based on the Old Value Logging approach. n is the number of elements (data fields)

to be protected from tearing. s^{TB} is the size of the data to be written in the tearing buffer, while s^{NEW} is the size of the new data to be written. p is the EEPROM page size.

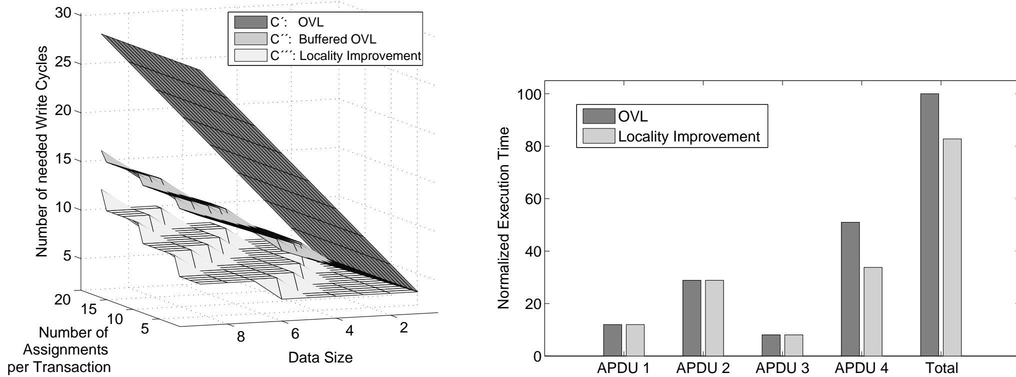
$$C_{OVL} = \sum_{i}^{n} \left\lceil \frac{s_i^{TB}}{p} \right\rceil + n + \sum_{i}^{n} \left\lceil \frac{s_i^{NEW}}{p} \right\rceil + 1 \quad (4.1)$$

Using the optimization in Section 6.5, the most time consuming write operations to EEPROM could be reduced to the number calculated in Equation 4.2. f_b gives the number of write operations to the transaction buffer (the intermediate buffer needed for the anti-tearing mechanism). As one optimization, this number is reduced by usage of a cache. A second optimization reduces the sum in the remainder of Equation 4.1 to a value typically 1. A more detailed explanation is given in Section 6.5.

$$C'''_{OVL} = f_b(n, s^{TB}) + \left\lceil n \frac{s^{NEW}}{p} \right\rceil + 1 \quad (4.2)$$

Figure 4.5 summarizes the achieved performance optimization results. Figure 4.5(a) depicts Equation 4.1 and Equation 4.2 comparing different numbers of variable assignments within one transaction (which have to be protected against tearing) and different amount of data to be protected.

Figure 4.5(b) shows the reduction of normalized execution time for a Java Card application (called JavaPurse) for different APDUs (commands sent to the application). The total execution time of the Java Card application was reduced by 17%.



(a) Performance Comparison with Respect to Equations in Section 6.5

(b) Reduction of time consuming Write Operations in a Java Card Application

Figure 4.5: Performance Optimization Results of using Meta-Information with an Anti-Tearing Mechanism

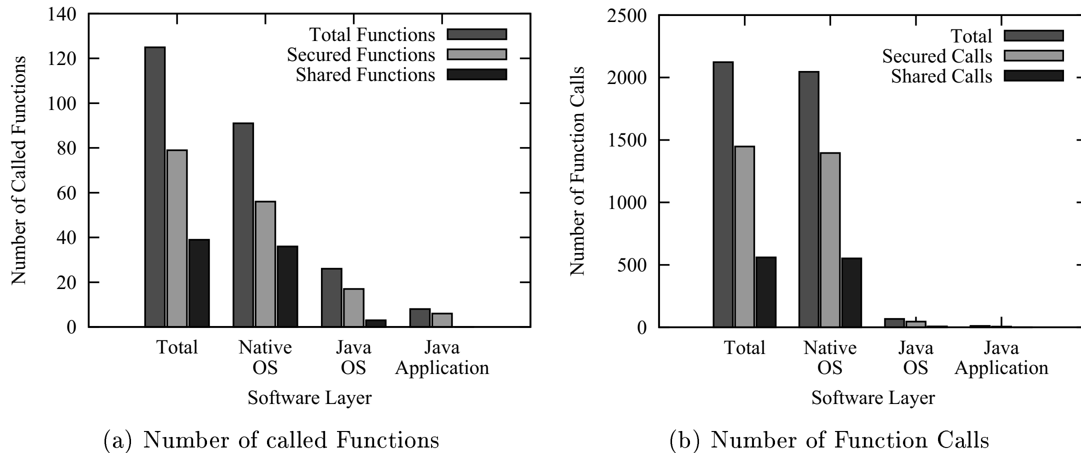
4.3.2 Case Study 3: Identification of Security Relevant System Components

Section 6.2 describes a meta-information based analysis of a Java Card operating system. Not all software functions in an operating system need to have the same security level

- some of them are security related, others are not. The main differentiator, as explained before, is that security related functions need to provide functional security requirements.

To analyze a real Java Card operating system, security related functions in a Java Card application were annotated with meta-information. This meta-information indicates that the annotated function has to be executed in a security sensitive context. Consequently, all called sub functions are also security related. Figure A.1 in the appendix of this work shows a call graph of the analyzed application. This figure should indicate how complex manual analyzing of such call graphs could become. The methodology discussed in Section 6.2 does the same in an automated way based on simulation of use cases. b The result of the meta-information based analysis in Section 6.2 is shown in Figure 4.6. Figure 4.6(a) shows that 63% of all called functions were executed in a security sensitive context while 49% of these functions are also used in a non-secure context. The method in Section 6.2 easily identified the 63% of function calls that have to be considered for a security review. Furthermore, the method indicated that some functions are used in a security relevant and a non-security relevant context. Latter ones could be implemented twice for the security related and the non-security related context. Depending how often these functions are used, this opens a potential performance optimization.

Similarly, Figure 4.6(b) shows the executed function calls (instead of the number of called functions). 68% of all function calls were executed in a secure context while 39% of these function calls were, again, also used when no certain security requirement was required.



(a) Number of called Functions

(b) Number of Function Calls

Figure 4.6: Results of Meta-Information based Analysis

In sum, the results in Section 6.2 show that meta-information evaluated during simulation time of a system is highly applicable for analysis of a system. The analysis can be performed for both security reasons and optimization goals.

4.4 Simulation based Security Verification

An analysis in Section 6.3 motivates the necessity of automated security verification. Different Common Criteria Security Targets were examined with respect to the number of

Security Functional Requirements in hardware platforms for smart cards of different vendors. The number of Security Functional Requirements can be seen as an indication how complex and time consuming a security verification can be, since every applied security requirement has to be verified.

Table 4.1 summarizes the results more precisely explained in Section 6.3: 15 to 21 Security Functional Requirements are totally applied 26 to 60 times (they are usually used more often than once in a system) in the selected smart card micro processors. Notice, that these numbers only include the hardware part of a system. Considering the software components as well the resulting verification effort justifies the need for automated verification methodologies. In this thesis, the automation is done by simulation based approach.

	STMicrosystems	**NXP**	**Samsung**	**Infineon**	**Fujitsu**
# SFRs	15	16	18	21	19
# applied SFRs	40	60	26	35	26

Table 4.1: Number of Security Functional Requirements in Smart Card Hardware Platform Security Targets of different Vendors

The main benefit of simulation based methodologies is that the verification is independent from the manner of module implementations or their abstraction levels. Furthermore, the verification is easy and efficient to apply. Successive sections summarize case studies about the applicability of simulation based verification. First, in Section 4.4.1, the simulation based approach for security verification is evaluated. After that, in Section 4.4.2, results of a method which extends the concept by a formal Event Calculus model checking approach are summarized.

4.4.1 Case Study 4: Simulation based System Security Verification

Sections 6.1, 6.4, and 6.3 use SystemC[1] as a modeling language. In all these case studies a smart card was modeled and a simulation based verification was applied.

Section 6.4 covers the functional abstraction levels. Because SystemC is based on C++ it does not provide any predefined construct for meta-information. Pre-processor based macros were implemented to overcome this drawback. These macros are applied in system modules and compiled with the entire system to function calls to a *model-to-checker* interface. These function calls are executed during the simulation time and report the necessary data to the checker interface as shown in Figure 4.7. The interface is connected to one or more arbitrary checker instances. Depending on the checker instance (and the associated meta-information), different checks can be performed to execute a simulation based verification.

First, experiments with pre-processor based meta-information are done in Section 6.4. Section 6.3 evaluates the same principle on a transaction level model consisting of hardware and software components. Section 6.1 applies the methodology on a mixed abstraction level model including purely functional modules, hardware modules, and software modules. Vertical transactors are used to connect functional components of a Java Card application

[1] http://www.systemc.org

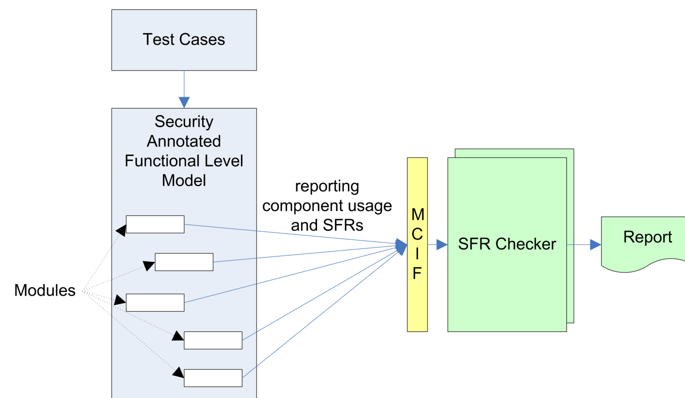


Figure 4.7: Evaluation of Meta-Information in SystemC during Simulation

to low level software implementations executed on the simulated processor. Different execution performance optimizations were done in case studies in Section 6.3 and Section 6.1 while performing simulation based security verification successfully.

Summarizing the results in mentioned sections, applying the meta-information has shown to be easy to use and flexible enough for an iterative development process. Furthermore, the simulation time was not reduced significantly by the processing of the meta-information. However, the verification quality strongly depends on the quality of the checker instance which is independent from the simulation environment. Successive sections explain how the verification quality could be increased by using formal methods in the checker instance.

4.4.2 Case Study 5: A (Semi-) Formal Event Calculus based Model Checking Approach

Section 6.1 describes a simulation based formal approach for security verification embedded in an iterative hardware/software codesign process as described before. The formalism is based on the Event Calculus also discussed before. A mixed abstraction level model of a smart card, including an application and software components of the smart card operating system, is continuously refined to evaluate some execution performance optimizations. At the same time, a simulation based model checking ensures that the formal security policy is not violated.

The Security Policy

A textual version of the security policy used in this case study is shown in Table 4.2. The policy does not yet contain all rules for a complete smart card system. Instead, 8 axioms which are needed to security verify a BAC authentication (Basic Access Control, typically used for machine readable documents such as e-passports) [45] were formulated. Together with the domain description explained in Section 3.4 and listed in Appendix A.1, the smart card model was verified with the f2lp[2] Event Calculus tool.

[2] http://reasoning.eas.asu.edu/f2lp/

Axiom Nr.	Description
1	A key storage is either protected by blinding or keys are stored encrypted. A key storage holds the secret keys. To protect the keys against side channel attacks the keys must be blinded or encrypted. This ensures that leaked information is useless for an attacker.
2	If keys are encrypted, a master key is needed. Encrypting keys requires a system wide secret key. This is the master key.
3	A master key can be derived or stored in a key store. A master key is a secret key. Thus, it has to be protected against unwanted access by an attacker. Derived keys are not stored directly in persistent memory but calculated when they are needed. A key store can also be used but notice the circular dependency.
4, 5, and 6	Whenever a key is used it has to be read, used with a cryptographic operation, and deleted. Usage of a key consists of three phases: (1) read the key from where it was stored and store it temporally at a location where it can be used by the crypto- subsystem. This can be skipped when the crypto-subsystem supports in-place operations for keys. (2) use the key with a cryptographic operation. (3) delete the temporary stored key. This rule is split in three separate axioms.
7	A secure read operation for keys must provide a key store, a key buffer, and a secure bus or a secure copy routine. A secure read has to be performed from a secure storage for keys - the key store. The key must not be stored temporary at any arbitrary location. A dedicated crypto-buffer has to be in place to handle keys securely. The copy routine itself must not leak information through side channels. Either this is done by hardware countermeasures (e.g, a secure bus) or by a secure copy routine in software.
8	A secure copy routine requires reliable random numbers. A secure copy routine requires reliable random numbers to hide information about the secret data to be copied. These random numbers need to be created and tested in a special way so that an attacker cannot predict or manipulate them and thus reduce the security of the copy routine.

Table 4.2: Security Policy Rules

The main goal of the security policy in Table 4.2 is to protect secret keys in the smart card system. During the BAC authentication, secret keys are used which are stored in a key store. The key store is part of the operating system. It handles the storage, encryption, and decryption of secret keys. The BAC application requests a key at the key store and copies it to a buffer for cryptographic operations. After that, the cryptographic operation can be performed. Finally, the key is deleted in the buffer by overwriting it with random numbers. Notice, that this only describes the key handling of the BAC authentication. This is repeated several times during one authentication process. Details about the authentication are described in [45].

The Simulation Output

Figure 4.8 shows a sequence of traces of security requirements when simulating the BAC authentication in an abstract high level model. As seen by the x-axis, no real timing information is available, as the model's components do not model timing behavior. Instead,

a sequence of activations and deactivations of security requirements can be seen. The y-axis shows different acronyms of security requirements which are explained in the Appendix A.3 in this document. In time point (1) a key usage starts. Two secret keys are read from the key storage in (2) and (8) and stored in a temporary buffer. The copy operation is performed by a secure copy routine which provides countermeasures against side channel attacks. This is done in (3) and (9). The mentioned countermeasure requires reliable random numbers which are fetched in (4) and (10). After the cryptographic operations in (5) and (11) the key is deleted from the temporary buffer in (6) and (12). To do so, random numbers are used to overwrite the key. The random numbers are generated in (7) and (13). Please notice that not all requirements needed to fulfill the policy and shown in the figure are explicitly mentioned in this explanation.

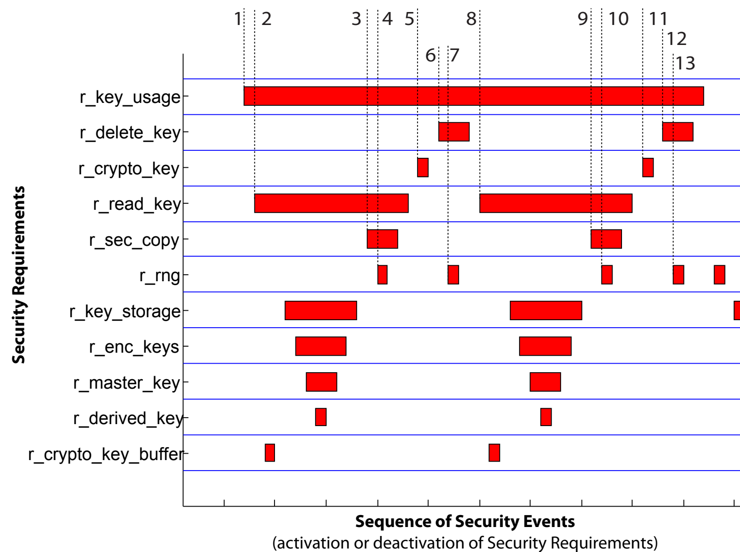


Figure 4.8: Security Requirement Traces of BAC Authentication in a High Level Model

The Security Verification

Using the complete trace of the BAC authentication shown in Figure 4.9 the security verification was split in five steps indicated as (I), (II), (III), (IV), and (V) in the figure. Phase (I) is the decryption of the incoming APDU command. Phase (II) is the preparation of the session key generation. As seen, key objects in the key store are used but no cryptographic operations are done. Phases (III) and (IV) are the creation of the session keys. Finally, Phase (V) is the encryption of the APDU response. Splitting up the verification in smaller portions increases the verification performance as explained later.

A demonstration of the verification can be seen in Figure 4.10. As can be seen, the f2lp tool is used with a couple of other tools. Please consult the manual of the f2lp tool for details how to use it. The verification is performed with the domain description `domain.fo`, the security policy `policy.fo`, the platform constraints `cons.fo`, the traces (e.g., `trace_I.fo` for phase (I)), and the Event Calculus description `DEC.lp` which comes with the f2lp tool. The f2lp tool returns with `Answer: 1` which indicates that a solution was found. This

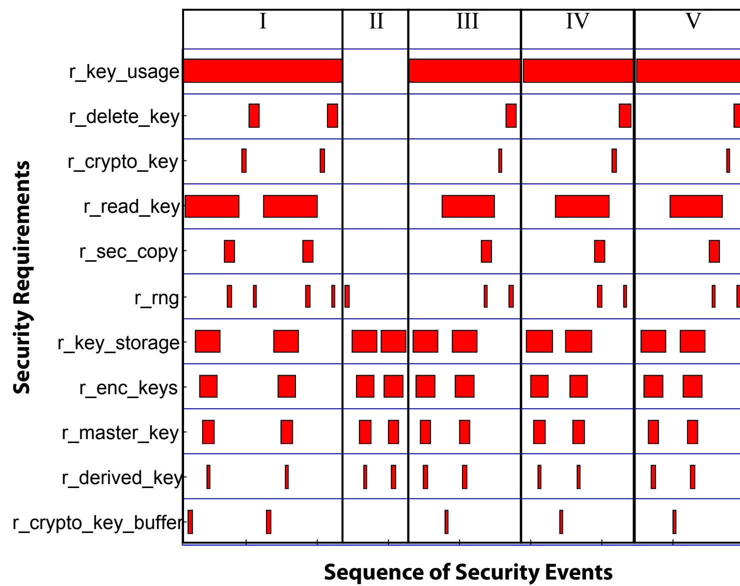


Figure 4.9: Complete Security Requirement Trace of BAC Authentication in a High Level Model

means that the security verification with respect to `policy.fo` was successful. In addition the tool returns the found solution. Notice that due to space limitations, the output in Figure 4.10 is shortened, indicated by the ....

```
$ ./f2lp.exe ../domain.fo ../policy.fo ../cons.fo ../trace_I.fo ./DEC.lp  |
sed 's/_NV_/NV_/' | ./gringo.exe -c maxstep=49 | ./claspD-1.1.exe -n 1
Warning: trajectory/4 is never defined.
Warning: antiTrajectory/4 is never defined.
Warning: releases/3 is never defined.
claspD version 1.1. Reading...done
Answer: 1+
happens(enables(epassport,r_key_usage),0) happens(enables(copykeytocryptobuffer,r_read_key),2)
...
```

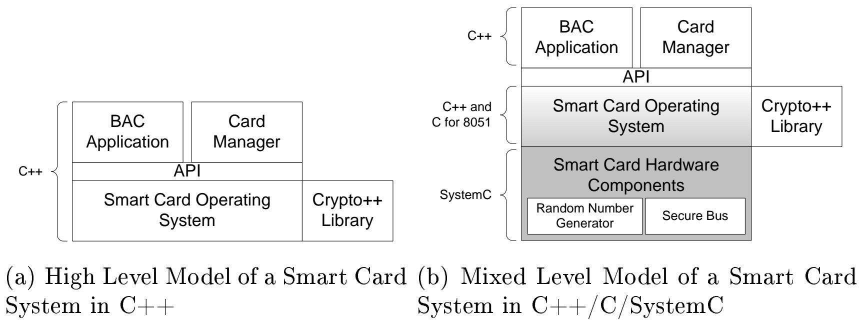
Figure 4.10: Shortened Security Verification Output with f2lp

System Optimizations

Several system optimizations were done in order to perform a design space exploration in the case study described in Section 6.1. The starting point was the already described high level module where all system components were implemented in C++ or provided by external libraries (library for cryptographic operations, Crypto++[3]. This model was refined partly by adding a transaction level model of an 8051 micro processor. In addition, some of the operating system functions were ported C for the 8051 processor. This allowed an early estimation of execution performance critical components of the system.

[3] `http://www.cryptopp.com`

During the design space exploration, a hardware random number generator and a secure bus were added to the transaction level model. Figure 4.11(a) shows the starting point of the refinement iterations - the high level model. Figure 4.11(b) shows the mixed level model where operating system components were ported to C for 8051 and new hardware components were added. Section 6.1 explains the optimizations and their impact to the execution performance results in detail.



(a) High Level Model of a Smart Card System in C++ (b) Mixed Level Model of a Smart Card System in C++/C/SystemC

Figure 4.11: Smart Card Models under Refinement and Verification

During all refinement and optimization iterations, a security verification as described above was performed while the security policy was constant. This demonstrates how flexible the security verification and the security policy are.

Verification Performance

The performance of the security verification using the f2lp tool has shown to be strongly dependant on the number of trace entries to be verified. Figure 4.12 shows measurement results and an extrapolated curve. As seen, it makes sense to split up security reports with many trace entries into smaller parts as the verification time per trace entry increases with the total number of entries.

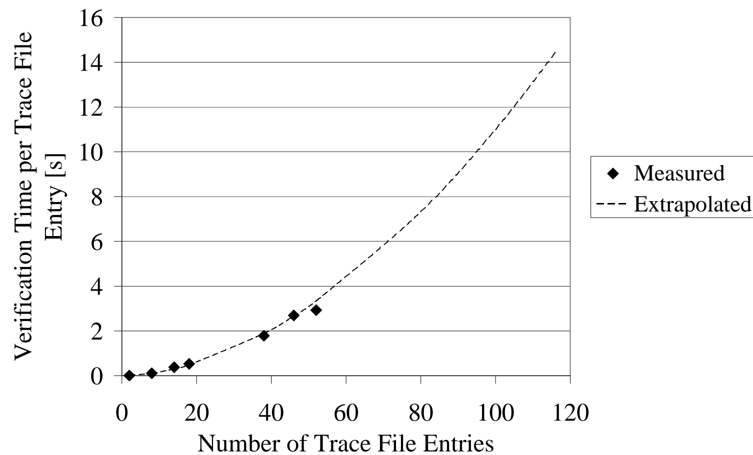


Figure 4.12: Verification Performance

The security report from the BAC authentication to be verified contains 116 trace entries. According to the extrapolated curve in Figure 4.12, a verification of 116 traces would lead to a verification time per trace of approximately 14.4 seconds. Multiplied by 116 log entries, this would result in a verification time of almost half an hour! Splitting up the security reports in as small portions as possible as shown in Table 4.3, the same verification can be done in less than 87 seconds. This is a increase of 95% without the negligible time for splitting up traces.

Number of Traces	**Verification Time**	**Comment**
38	67,9s	
14	5,3s	3 times
8	0,9s	3 times
2	$< 0{,}1$	6 times
Total: 116 traces		Total: < 87s

Table 4.3: Verification Times with split Trace File

Overall, it has to be mentioned that the computational performance and the memory consumption of the f2lp showed some room for improvement. It has to be mentioned that the tool was implemented as a proof of concept Event Calculus prover and was not meant to be used with models of the magnitude similar as in this work. However, the results in Section 6.1 show that a (semi-) formal security verification based on the Event Calculus can be done efficient enough to be applicable for flexible iterative development processes such as hardware/software codesign.

4.5 Evaluation Summary

To evaluate the novel design flow, with respect to security and performance verification, it was split into three main concepts: simulation based functional verification, usage of meta-information (for optimization and verification), and simulation based security verification.

Simulation based functional verification was shown to be highly effective for security mechanisms. This was demonstrated on the anti-tearing mechanism - one of the most important security mechanisms of Java Cards. Controllability and observability are essential when verifying security mechanisms but difficult to achieve with final products or prototypes. Simulating systems, however, demonstrated to increase controllability and observability to execute automated functional verification.

Meta-information can be used for system optimization and verification as well. With meta-information additional information is added to system modules. Such information is easy to apply by developers and can be evaluated during the system's runtime or simulation time depending on its dedicated purpose. During runtime, meta-information has shown to increase efficiency of memory management; during simulation time it was used to automatically partition a system in security critical and non security critical modules. As meta-information is independent from the modules abstraction level and has no impact to their functional behavior it is highly applicable for flexible and iterative design processes.

Simulation based security verification has shown to be flexible, abstraction level independent, and efficient for system wide security architectures. This is a great advantage in

comparison to purely formal verification methodologies which can only be performed on very abstract models or small parts of a system. The simulation based security verification was successfully integrated into an iterative hardware/software codesign process without any reduction of flexibility during the design space exploration. Furthermore, it was successfully extended to a (semi-) formal approach by usage of model checking methodologies.

Together, these evaluation results show that the novel concept of security and performance verification discussed in this thesis is applicable for development of secure embedded systems.

Chapter 5

Conclusion and Future Work

This chapter concludes this work by recapitulating the goals and summarizing the concept to fulfill them. Furthermore, it highlights the benefits of the novel approach in this thesis and discusses possible future work in codesign methodologies for secure embedded systems.

5.1 Conclusion

Driven by a gap between security verification of abstract models and security verification of implemented systems, this work focuses on the seamless integration and verification of functional security requirements into a hardware/software codesign methodology. The codesign approach allows system wide optimization by performing design space exploration. This allows best results in the most important parameters of modern embedded systems such as footprint (equivalent to price) and execution performance. Functional security requirements are considered in the development process as soon as possible. This allows taking them into account when performing the system optimizations. This is one of the advantages of the approach described in this work, as security features are known to cause significant performance impact.

To perform the required seamless integration of the security requirements into all development phases and abstraction levels, they are represented as additional information in the system's model respectively in its implementation. This so called meta-information can be automatically evaluated during development time, e.g., for system verification, or at run time e.g., for system optimization. An automated system security verification based on system simulation including evaluation of this meta-information can be performed. The automatism proposed in this work allows doing so in each iteration of a codesign process. This allows recognizing security violations as soon as they happen in the development process. This is one of the major benefits of this approach.

Based on the simulation based evaluation of meta-information representing security requirements, formal system verification can be done. A simulation environment evaluates the meta-information of the system and generates a formal model of dependencies between security requirements. These dependencies are verified against a formal set of rules (a so called security policy) which defines the security architecture of the system. To do so, the

Event Calculus, a formalism to represent temporal actions and their effects, is used. The formal verification is performed by model checking using an Event Calculus reasoning tool.

One great benefit of using meta-information for security verifications is that they are independent of the security requirement's implementations. Thus, meta-information can be applied very early in the design process. They are independent of the implementation domain (e.g., hardware or software). In addition, they can be refined with the remaining system during design space exploration, are easy to apply by system developers without a lot of security or verification background, and can be evaluated automatically by existing tools. Using meta-information in system models and implementations for verification gives sufficient flexibility to be used in iterative codesign methodologies. Additionally, they allow efficient management of rapidly changing security requirements.

The development methodology in this work was evaluated in different experiments and a case study. In the case study, a smart card system was modeled including the hardware, the operating system in software, and a smart card application. The system model was a mixed abstraction layer model which combined different abstraction levels in one model. The hardware was modeled in SystemC, while the operating system and the smart card application were implemented in C for the targeted smart card hardware platform. By adding meta-information representing different security requirements automated security verification across all these abstraction layers was possible. The verification was executed continuously when performing different system optimizations in hardware and software. Overall, the methodology in this work has shown to be highly applicable to be used in industrial development of secure embedded systems even though improvements, especially in the areas of formal verification tools and support are possible.

5.2 Future Work

This work concentrated on development and evaluation of technical concepts for a hardware/software codesign process for secure embedded systems. It was evaluated by experiments and a case study. During this work it was attempted to stay in close contact with an industrial partner to develop concepts which are highly applicable in industrial environments on short term. However, it could not be a target of this dissertation to evaluate the secure codesign methodology in an industrial project. Nevertheless, to definitely answer the question of industrial applicability, especially by meaning of cost and savings, this needs to be done as future work. The codesign methodology has to be implemented for an industrial project and a system with realistic complexity. In any case, the verification costs have to be compared to cost of projects without the approach in this thesis.

The concepts in this work are following the requirements described by the Common Criteria process. Before using this codesign methodology for systems that need a security evaluation according to Common Criteria, the methodology has to be discussed with Common Criteria experts and evaluators. It has to be noted, that even if this development process was not accepted by external evaluators, it will pay off for internal security verification. However, if refinements in the process were needed for acceptance the benefits during de-

velopment and evaluation are bigger.

Within the context and working period of this thesis two follow-up research projects were worked out and proposed to funding partners (Österreichische Forschungsföderungsgesellschaft[1]). Both projects were accepted and are currently running. *CoCoon* is working on novel security concepts for user centric ownership Java Cards. Modern high-end Java Cards will have to allow post issuance installation of Java Card applications. This enables to perform new attacks on these secure embedded systems. CoCoon investigates in novel defensive execution environments and application verification methods to overcome these threats. Hardware and software concepts will be evaluated. Therefore a codesign methodology will be developed. This methodology will be strongly based on the approach in this work. *DAVID* investigates scalable secure software systems for low-end use cases. Smart card platforms with very restricted capabilities are targeted by this research project. On the one hand, this means that software has to be highly efficient and scalable to fit onto such a platform. On the other hand, a certain level of security is required. Thus, development and verification has to be done very carefully, which is the focus of DAVID.

[1] htpp://www.ffg.at

Chapter 6

Publications

Publication 1 (under review): *Security Verification on Mixed Level System Models based on Event Calculus Model Checking*, Preprint submitted to Journal of Systems Architecture, Embedded Software Design, Elsevier B.V

Publication 2: *Identification and Verification of Security Relevant Functions in Embedded Systems Based on Source Code Annotations and Assertions*, Workshop in Information Security Theory and Practices, WISTP '10, Passau, Germany, 12-14 April 2010, Information Security Theory and Practices. Security and Privacy of Pervasive Systems and Smart Devices, Lecture Notes in Computer Science, 2010, Volume 6033/2010, 316-323

Publication 3: *Idea: Simulation Based Security Requirement Verification for Transaction Level Models*, International Symposium on Engineering Secure Software and Systems, ESSOS '11, Madrid, Spain, 9–10 February 2011, Engineering Secure Software and Systems, Lecture Notes in Computer Science, 2011, Volume 6542/2011, 264-271

Publication 4: *Towards Formal System-Level Verification of Security Requirements during Hardware/Software Codesign*, 23rd IEEE International SOC Conference, SOCC '10, Las Vegas, USA, 27–29 September 2010

Publication 5: *Performance Improvement and Energy Saving based on Increasing Locality of Persistent Data in Embedded Systems*, 2010 Fifth International Conference on Systems, ICONS '10, Menuires, France, 11–16 April 2010

Publication 6: *Java Card Performance Optimization of Secure Transaction Atomicity Based on Increasing the Class Field Locality*, Third IEEE International Conference on Secure Software Integration and Reliability Improvement, SSIRI '09, Shanghai, China, 8–12 July 2009

Publication 7a and b: *Fast simulation based testing of anti-tearing mechanisms for small embedded systems*, 2010 15th IEEE European Test Symposium (ETS), ETS '10, Prague, Czech Republic, 24–28 May 2010

Security Verification on Mixed Level System Models based on Event Calculus Model Checking

Johannes Loinig[a], Christian Steger[a], Reinhold Weiss[a], Ernst Haselsteiner[b]

[a]*Graz University of Technology, Institute for Technical Informatics, Graz, Austria*
[b]*NXP Semiconductors Austria, Gratkorn, Austria*

Abstract

Security verification is going to be increasingly important in the development of future embedded systems. The complexity of systems, attack scenarios, and countermeasures are going to force development processes to include formal or semi-formal approaches. In model based development methodologies, such as hardware/software codesign, models contain components implemented on different abstraction levels. A system security verification approach has to be able to include all of these different abstractions.

A predicate based security verification approach, such as discussed in this work, can fulfill these requirements. Event Calculus, a formalism of temporal logic is utilized to define formal security policies which are then used in a model checking approach to verify system security properties.

Keywords:
Security, Event Calculus, Formal Verification, Model Checking, Hardware/Software Codesign, Simulation

1. Introduction

Information security is a major feature of modern systems. Adequate security verification is a crucial element during development of them. This is especially important for embedded systems, which are often difficult, expansive, or even impossible to access for maintenance, particularly when already delivered to the field. Unfortunately, unlike other typical requirements of embedded systems such as performance or power consumption, security is a requirement which is difficult to be expressed by meaningful metrics. Too many completely different aspects, which are spread all over the complete system, would have to be considered. In addition, external factors, which are not directly related to system design or implementation (i.e. how easy it is for an attacker to access the system, or how beneficial it is to hack it) have to be taken into account. Typically, they lead to extensive manual security verification, which is time consuming, expensive, and often needs to be redone when parts of the system are altered due to changes in use or function, or in attack scenarios.

Various research projects investigate different approaches that make the lives of secure embedded system developers easier. Verification methodologies and tools have been developed which allow defining and verifying security properties in models and implementation of systems. Formal verification seems to be very promising both from both the technical and marketing view. First of all, proving the system's security properties helps system architects and developers manage the necessary level of security. Second, proven security of a product is an essential differentiating factor on the highly competitive embedded system market.

As a result of the immense complexity of modern embedded systems, formal verification is typically either done on a very high abstraction level, or on selected parts of the final implementation. This is dependent on a vast amount of functional features mainly provided by software modules, the capability of the underlying hardware components, the ability of potential attackers, and the practicality of recent attacks. Nevertheless, it is obvious that security has to be considered in a very broad perspective, specifically as a system-wide concept applied and verified in all design phases, and on all abstraction levels and implementation layers. Most existing security verification approaches fail to handle the flexibility of the applied development methodology, especially, when an iterative development approach such as hardware/software codesign is used. Nevertheless, flexible codesign approaches are very powerful because they allow the development of systems with optimal in-

teraction between hardware and software components.

The motivation for this research was to develop and evaluate a verification methodology that combines the advantages of formal verification and flexible hardware/software codesign methodologies. The problem statement we want to address is that formal verification aspects must do not have to hinder, but support iterative development processes. Thus, verification must be (1) flexible enough to handle effective design space exploration, (2) simple to apply in order to keep verification efforts and costs minimal, and (3) able to cover security requirements independently on various abstraction levels or implementation layers.

Our methodology accomplishes the aforementioned criteria by combining flexibility of simulating mixed abstraction level models with aspects of formal Event Calculus reasoning. In our approach, Event Calculus is used to define a formal domain description and a security policy. The domain defines properties for a secure embedded system while remaining independent from any system under development. The security policy describes properties of the system's security architecture. Formal reasoning is used as the first step to prove the consistency of a policy. Then, iterative simulation based model checking is executed in order to verify the continuously refined system against the security policy.

The remainder of the article is structured as follows: Section 2 summarizes basics and work related to embedded system development and security verification. Section 3 describes our concept, including model based development, simulation based model checking, and Event Calculus based verification. We applied our approach in a case study described in Section 4 and we discuss the results in Section 5. This includes system optimization with respect to execution speed of a smart card application, the verification's performance, and a summary of our experiences with Event Calculus tools. Finally, in Section 6, we provide a conclusion.

This work is a result of a project which is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology (contract FFG 816464).

2. Basics and Related Work

This work is based on some concepts that have not been directly addressed in recent research work. However, we explain them in this section to provide the reader of this article with better understanding of field of research where this work was done.

2.1. Secure Embedded Systems and Smart Cards

In this work we discuss very small embedded systems (even though the presented concept is not necessarily restricted to small embedded systems) with a main focus on security. A typical example of a small embedded system is a smart card, or more precisely, a smart card micro controller which nowadays comes in different packages. Smart card micro controllers often need to be wrapped in pocket-sized plastic cards (banking, credit card, electronic identification), but other use cases require placement in more powerful devices such as smart phones. Most smart card use cases have two things in common: (1) smart cards appear in items that sell in high numbers, which is interesting for the market, but also generates pressure for keeping prices competitive, and (2) smart cards require a very high security level.

First of all, the type of use defines the required level of security. Moreover, security is a moving target. Laboratories (e.g., security evaluation labs) and attackers continuously work on developing new attack strategies and ways to make already known attacks more powerful and efficient. The longer the development of a system takes, the higher the risk becomes, that new security requirements have to be taken into account as well. This happens because new attacks present themselves or because known, but earlier inapplicable attacks, become threats due to better tools. Until recently, for example, laser attacks have been a purely theoretical risk because of the expensive equipment required. However, nowadays, labs can afford several lasers, which perform fast sequential light attacks in different places on a chip. Development teams and methodologies have to be able to react adequately to the rapidly evolving theoretical and practical attacker community. They need to adapt their systems according to new circumstances and they need to do it fast.

In the semiconductor industry, the product's price is still mainly defined by the size of a chip. This means that new features, including security features, cannot be added at free will. Every design decision has to be deliberated very carefully, even though the embedded security market seeks trade offs between price and system security. A prominent example of that is the Java Card Connected [1] specification, which has been developed for years by the Java Card Forum[1]. Unfortunately, semiconductor manufacturers had to recognize that the smart card market was not willing to pay for the size of the chip which was consequently needed to provide all the

[1] `http://www.javacardforum.org`

new fancy but pricey features defined in the Java Card specification.

To give the reader an idea of the size of smart card micro controllers: they are typically still based on an 8-bit 8051 derivate with some few kByte of RAM and some few hundred kByte of non-volatile memory (EEPROM or Flash). However, one should not underestimate the complexity of such a system. Java Cards, for example, provide a full virtual machine, including: an application firewall, reams of security features implemented in software and hardware, garbage collection, a very comprehensive API for cryptographic algorithms, and lots more. A very extensive and costly security evaluation, (e.g. according to the Common Criteria [2]) is necessary to guarantee a security level that is needed during the product's life cycle, which can last years.

2.2. The Common Criteria Process

Common Criteria [2] is *the* de-facto standard for security evaluation. In this work we do not describe the whole process but only the basic concept of Security Functional Requirements which corresponds with our methodology. For details about the complete evaluation process please consult the book [3] which gives a good explanation of the complete process or the specification in [2].

The Common Criteria process is a rather documentation centric approach. To provide a minimal description, an analysis identifies potential threats, which then have to be countered by *Security Mechanisms* implemented in the system. Behavior of the Security Mechanisms is in turn defined by *Security Functional Requirements*. A set of Security Functional Requirements is given by the Common Criteria standard but can be extended if necessary. The *Security Target* documents the security architecture of a certain system by explaining (among other things) which Security Functional Requirements need to be implemented. A security evaluation typically includes the verification if an implementation and the according documentation in the Security Target correspond. This is a manual and very time consuming task.

Seven *Evaluation Assurance Levels*, or EALs, are defined in the standard. They contain, for example, developer action elements and evaluator action elements that have to be performed to gain a Common Criteria certificate for a product. EAL 1, the minimum level of assurance, requires functional testing, whereas EAL 6 refers to semi-formal verification and EAL 7 to a formally verified design. The Common Criteria process does not specify how or on which abstraction level the (semi-) formal verification has to be done.

In common with the authors of [4] and [5] we argue that the Common Criteria process is missing a clear reference to the development process. Our methodology aims to close the gap between supporting the verification of a system's model and implementation against Security Targets.

2.3. Security Functional Requirements

In our approach we re-use the idea of Security Functional Requirements. As previously described, they are the very basic components of a system's security architecture. Notice however, that Security Functional Requirements do not describe in detail how the requirement has to be implemented in a system. Instead, they only designate certain requirements which have to be fulfilled because other system components rely on them. Considering the early development phase, this is an irreplaceable advantage. In this phase, the shape of final implementation has often not yet been defined. However, it might already be important to identify essential security components, which provide certain Security Functional Requirements.

The analysis in [6] shows that a smart card hardware platform can easily consist of 60 Security Functional Requirements. This already large number includes neither the software layers of the smart card application nor the operating system. Notice that, verification effort typically increases exponentially with the complexity of the system. We understand this as an indicator that future system development will rely on automated security verification tools.

Our approach aims to provide the link between a formal description of how Security Functional Requirements and system modules are interrelated to achieve the necessary system security. To do so, we use meta-information for Security Functional Requirement in the source code of the system's model, as well as in the source code of its implementation.

2.4. Related Work

Although formal verification is not a very new topic in research, it recently received a lot of attention, especially in the area of security verification. One of the main reasons for this attention is unfortunately, also one of the main drawbacks. The continuously growing complexity of modern systems makes it necessary to use comprehensive verification methods. Unfortunately, the complexity of today's systems is already much too high to run a complete formal system verification, which includes all the hardware and software components.

A rather non-technical classification of verification methods splits them into approaches that (1) use very

abstract high level models and (2) approaches that prove certain low level implementation details. The problem we identified is that proving properties of very abstract models does not necessarily increase the security of a real implementation. That's the reason why these approaches are often only used to verify specifications.

Modeling languages such as the Z Notation [7] or UML [5, 8] for high level models, typically used with theorem provers or model checkers [9], are too abstract to be compiled or synthesized into real systems that provide sufficient performance and are cost effective enough to produce. In addition, these languages can not yet be used for typical design space exploration tasks in which the architect or developer tries to optimize essential parameters (like execution performance, power consumption, etc.) of a system. This very often leads to a separate implementation of security and functional models. The security model is typically generated and verified first, whereas the functional model is used to refine the system components afterwards. All changes in the functional model have to be re-integrated into the security model and checked again to be sure that no security violation happened during the refinement process.

Low level verification methods for system level models, hardware, and software are described in [10, 11, 12, 13] and many other publications. Even though, some of them use high level languages like SystemC for hardware, or Java for software, the verification is based on low level representations like netlists or bytecodes. The computational effort is usually extremely high and can even be too high to be executed on today's computers (e.g., as a consequence of the state space explosion). Such low level representations are typically generated by tools such as compilers or synthesis tools. However, this can be very time consuming. Consequently, generating low level representations requires a steady implementation of the modules. In early design phases mock-up modules are often used during design space exploration because the developer might not yet know if or how the final module will be implemented in hardware or software. Implementing these mock-ups in a detailed way so that they can be processed by appropriate tools and so that their low level representations can be verified is an unwanted expense.

Our approach fits somewhere in the middle of these aforementioned processes. Instead of low level details, our approach uses predicates for Security Functional Requirements in a system. Such abstract descriptions are implementation independent and their verification is flexible enough for hardware/software codesign methods.

Using abstract properties or predicates for security verification is already described in [14, 15] and [16]. Event Calculus, a formalism of temporal logic, is often used to define and verify a meta-model of a system. Like above, we argue that there is a missing link between verification of these meta-models and the real systems to be implemented.

A detailed description of the Event Calculus is given in [17, 18] and [19]. Basic principle is that the Event Calculus defines predicates like $Happens(\alpha(\gamma), \tau)$ and $HoldsAt(\beta(\gamma), \tau)$, where α is an event (also called action) that happens at time point τ. β is a property (a so called fluent) that holds at time point τ. Events and fluents are not pre-defined by the Event Calculus. They can be defined in an appropriate domain description and can be used with typed arguments (γ in the examples above). Such predicates are used to define axioms that can be formally checked with tools such as the *decreasoner*[2] or *f2lp*[3].

An example is given in Axiom 1 which states that if *John* wants to walk through *Door*, the door has to be open. It does not yet define if the door is open or not, but additional axioms can describe events that can cause the door to be opened. All axioms combined together can be used in three ways: (a) in an abductive way - to find necessary events (so that John can, for example, walk through the door), (b) in a deductive way - to find the result of events (i.e. if the door is open or closed), (c) in an inductive way - to find rules (John can't walk through the door if it is closed).

$$Happens(walk_through(John, Door), T) \rightarrow HoldsAt(open(Door), T). \quad (1)$$

In [14] and [15] abduction problems are used to find potential abuse in attack scenarios. We use the Event Calculus to formally define a domain of rules for Security Functional Requirements and a policy that describes the relations of Security Functional Requirements. The policy is first verified against the domain in an abduction task in order to find potential solutions. Afterwards, the system model can be checked against the policy by using deduction. We will explain in more detail how the Event Calculus is used in Section 3.3.

3. Event Calculus based Model Checking

Our development process is based on three corner stones: (1) model based development, (2) simulation

[2] `http://decreasoner.sourceforge.net`

[3] `http://reasoning.eas.asu.edu/f2lp/`

based model checking, and (3) Event Calculus based verification. Model based development means that we start with a very abstract pure functional model. This model is continuously refined until it meets all requirements and the system's modules can be ported to a real implementation. This implementation can be seen as refinement of a model as well. Simulation based model checking uses traces generated during simulation of use cases to verify system parameters. In case of our work these parameters are Security Functional Requirements. Event Calculus based verification is used to verify these traces against a formal description of a security policy. Following subsections describe these methods in more detail.

3.1. Model based Development

The basic principle of model based development is a top-down approach and depicted in Figure 1. The starting point is a pure functional high level model. This model is refined iteratively. With every refinement cycle a simulation is used to analyze the system's properties. These properties can be functional properties as well as security properties.

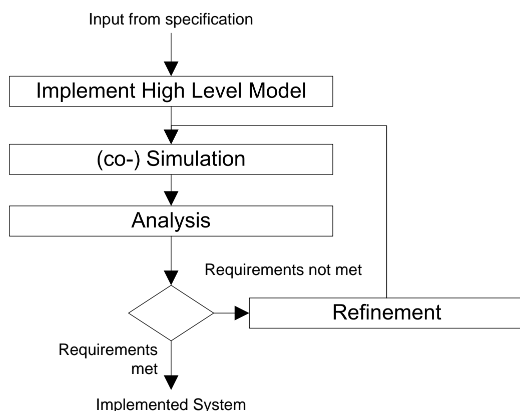


Figure 1: The Iterative Process

As described in [20], the model passes different abstraction levels. It is essential for security concepts to be considered and reflected in a model as soon as possible in the development process. Thus, after completing a pure functional model, the model needs to be extended through security requirement's meta-information. From this point in time security verification can be performed with every refinement cycle (the verification methodology is described in the following subsections). Then, models of security mechanisms and functional components of the system are created. This is necessary, as they are also needed for partitioning, mapping, and optimization procedures of the hardware/software codesign process. Abstraction levels and their purposes are summarized in Figure 2.

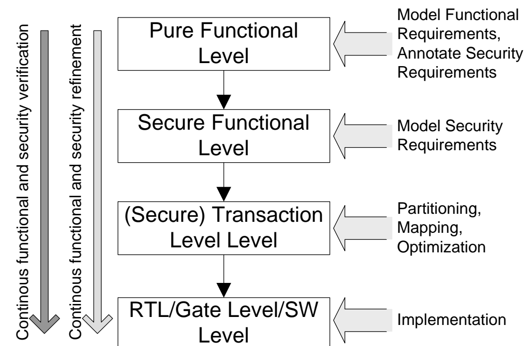


Figure 2: Abstraction Levels

The refinement from one abstraction level to the next one is a continuous task. During design space exploration some modules are more interesting than others. To generate more accurate analysis results these modules have to be refined on a more accurate level. Other modules currently under development, however, are not that interesting and would be very time-consuming to refine as well. Notice, that refinement is often used to "try out" a design decision (this is typical in the design space exploration). Because design decisions can be withdrawn when the analysis shows improper results, refinement should be as effortless as possible. This means that the model based development approach has to be able to handle different modules on different abstraction levels. A model that follows this concept we call a mixed (abstraction) level model. Co-simulation allows simulation of such mixed level models.

Since modules use a communication channel appropriate to their own abstraction level, a module on a higher abstraction level cannot directly interact with a lower level one. Functional modules, for example, often use function calls, while the lowest level modules typically use bus accesses or other hardware interfaces. Thus, an appropriate communication channel between abstraction levels is necessary.

Usual transactors (shown in Figure 3) connect modules with incompatible interfaces. The transactor handles data transformation to make communication possible. Transactors are a modeling tool; they are usually not needed in the final product. To handle mixed level models this concept needs to be extended by *vertical transactors*.

Vertical transactors as shown in Figure 4 provide an

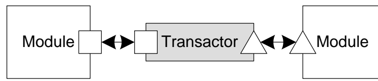


Figure 3: Usual Transactors

interface between modules on different abstraction levels. Such modules do not have an appropriate interface that could be connected through a usual transactor. One example shown later in the case study, is a software module implemented on the functional level in C++ (running directly on the simulation machine). It is connected to a low level software module implemented in C for a smart card processor (and executed on a simulated smart card CPU). These modules are implemented with various tools (e.g., compilers) and executed in different simulation environments. The vertical transactor gives the developer a tool to refine one module to a lower abstraction (e.g., because it is critical for performance and has to be modeled in more detail) while other modules are not critical, and it is sufficient for them to stay more abstract.

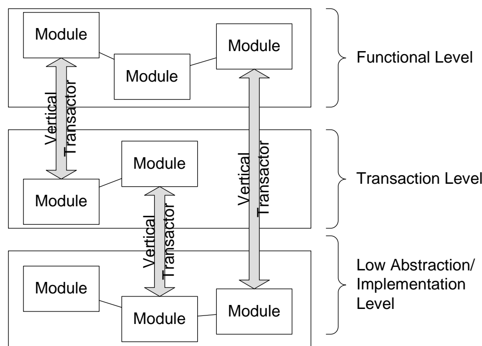


Figure 4: Vertical Transactors

3.2. Simulation based Model Checking

In a mixed level model a detailed verification of the model's source code makes no sense. First, the verification effort is too high. High level modeling languages like SystemC use complex data structures and pointers which are difficult to verify. Lower levels are even worse to formally verify in detail. Concurrent processes, interrupts, and a lot of potential side effects (e.e. caused by global variables, such as special function registers in assembler) make detailed verification impractical today. Second, a detailed verification is not needed during iterations of refinement cycles. We argue that there is no good reason for a detailed but costly formal proof of an abstract module that will never be a part of the final product as we know it. Instead, the verification has to be appropriate for the abstraction level of the modules. As discussed in [6] and [20], we strongly believe an informal, but efficient, simulation based security verification performed during design space exploration is more promising than an expensive formal verification.

In conventional hardware/software codesign processes use cases are applied during the simulation of the model. The model respectively the simulation environment creates a report that is used to analyze the estimated parameters (like the performance) of a system. The analysis results obviously depend on the quality of the use case scenarios, which generate the stimuli for a given simulation.

We extend this methodology by adding a security aspect. This is shown in Figure 5. Security requirements are added to the model of the system as meta-information (so called annotations). This meta-information does not influence functional behavior of the model and it can be used on all abstraction levels. It is evaluated by the simulation environment and used to do a security analysis.

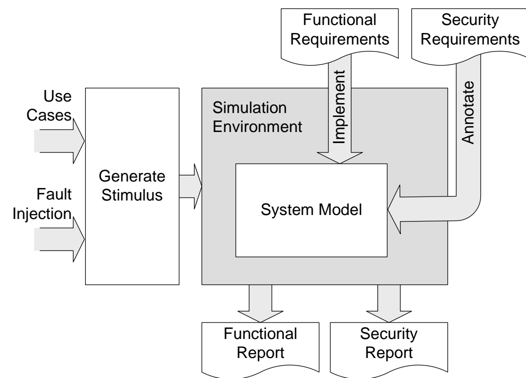


Figure 5: Simulation Based Security Verification

Notice, that security requirements are also implemented during the refinement cycles (this is not shown in Figure 5) as described in the previous section. This, however, does not mean that these requirements are not depicted by meta-information. Instead, the meta-information is refined as well, for example, to indicate that a module provides a certain functional security requirement. This allows for the verification to become more accurate with each refinement cycle.

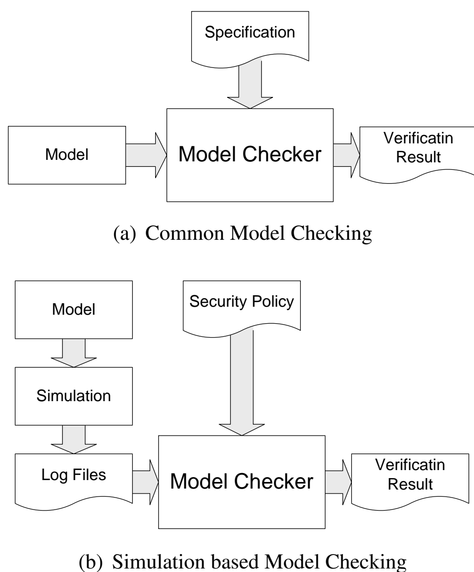
Again, the quality of each verification depends on simulated use cases. Now, these use cases alone are not able to stimulate scenarios where all security requirements are involved. The concept has to be extended

through attack scenarios. Fault injection during simulation provides a convenient way to simulate the system in conditions outside of a normal use case.

To keep it very simple, verifying whether to models are compliant with one another is performed through conventional model checking. One model is a formal description of the behavior as it should be - this is the system specification whereas the second model represents the system to be verified. The model checker checks if the system's model can lead to states that are not defined in the specification. This is shown in Figure 6(a). The problem with this approach is that the model to be verified has to be in a representation which corresponds with the model checker. This could mean that it is either implemented in a certain, very abstract, formal language, or that it is a generated intermediate representation from a concrete implementation (e.g., generated from a hardware netlist).

In case of simulation based model checking the output of a system simulation is used as representative data for the system. In Figure 6(b) log files are checked against a formal security policy. The log files contain traces of security requirements. The verification itself is also done by a formal model checker.

In such a case, only one of two input models of the model checker can be formally defined (the security policy). One can argue that this is a semi-formal approach as some input data for model checking is generated by a simulation, but the verification itself is done formally.



(a) Common Model Checking

(b) Simulation based Model Checking

Figure 6: Common versus Simulation based Model Checking

3.3. Event Calculus based Verification

We use the Event Calculus to formulate rules in the security policy. Using the Event Calculus, temporal interactions of events and properties, or so called fluents, can be formally described. However, the detail meaning of events and fluents is not defined by the Event Calculus itself. In our approach, the fluents represent Security Functional Requirements and events indicate when system modules support and respectively require appropriate security mechanisms. This is defined in our domain description. Security requirements and their interactions are defined in the security policy. The security policy thus describes the abstract security behavior of a system without a relationship to a concrete design or implementation. Platform constraints define the capabilities of a target platform with respect to the security requirements of a policy. This allows a certain refinement of the security policy as platform constraints reduce the design space for the security architecture.

As a first step, the domain description, security policy, and platform constraints (if a target platform has already been selected) can all be verified in a model finding process (in an abductive task). In this process the solver tries to find scenarios of events that match the domain, policy, and the constraints. If a model is indeed found, it describes a potential architecture of modules and related security requirements which fulfill the security objectives on the target platform. If more than one model is found, they provide information about what the target design potentially looks like. If no model is found, a contradiction shows up in the descriptions. In that case, the security policy either is invalid or the it cannot be implemented on the target platform, because the domain description is fixed.

As a second step, the security reports (a trace of security requirements) from mixed level simulations, domain descriptions, security policies, and the platform constraints, are used in a model checking process (a deductive task). If a non-contradicting set of properties results from the events in the security report, the report along with the simulated model fulfill the formally defined security architecture. However, if a contradiction is found, the implementation includes a security violation.

3.3.1. The Domain Description

The domain description of our concept is general enough for any suitable security verification, and can be reused for the development of different systems. It contains 26 axioms in total which all work independently of concrete security requirements as well as system mod-

ules. The main concept of our domain description can be explained with the following axioms:

$$initiates(requestsR(M,R), unsatisfied(M,R), T). \quad (2)$$

Axiom 2 states that when module M requests a requirement R, M becomes unsatisfied with respect to R. Furthermore, *initiates* is defined by the Event Calculus, and *requestsR* (an event) and *unsatsified* (a fluent) are defined by our domain description. The axiom means that the module relies on a security requirement which is not implemented by the module. T stands for the point in time whereas it is not bound by this axiom. This means that the axiom holds true for every point in time.

Obviously, the requirement has to be fulfilled somewhere in the system. Thus, Axiom 3 defines how a module can be satisfied: an event *satisfy* has to happen on the same module M with the same requirement R (*terminates* is defined by the Event Calculus). Again, this axiom holds true for every existing point in time - so whenever *satisfy* happens. However, we have to be more restrictive with respect to points in time where *satisfy* is allowed to happen because we need it to happen when certain conditions are met. Additional axioms 4 and 5 formulate these restrictions. The restrictions can be summarized in the following way: a module becomes satisfied if and only if it is connected to a module that provides the right requirement and is not unsatisfied with respect to another requirement. The reminder of this section explains the necessary Event Calculus axioms to express this rule. The other axioms in our domain description mainly describe necessary events and unwanted side effects; for example events causing modules to become connected and disconnected to other modules. An example side effect is that a module cannot be connected to itself.

$$terminates(satisfy(M,R), unsatisfied(M,R), T). \quad (3)$$

Axiom 4 describes a logical combination of four fluents (*unsatisfied* for M_1, *connected*, *provides*, and *unsatisfied* for M_2 in combination with an existential quantifier). If this logical conjunction holds true, the *satisfy* event happens. In other words: *satisfy* happens when module M_1 is unsatisfied and connected to module M_2, while M_2 provides the same requirement R_1 which caused M_1 to become unsatisfied. In addition, M_2 must not be unsatisfied. Notice, that there must not exist any requirement R_2 (read $-?[R_2]$ as $\nexists R_2$) that makes M_2 to become unsatisfied at the same time point T.

$$\begin{array}{l}(holdsAt(unsatisfied(M_1,R_1),T)\ \& \\ \ holdsAt(connected(M_1,M_2),T)\ \& \\ \ holdsAt(provides(M_2,R_1),T)\ \& \\ \ -?[R_2]:(holdsAt(unsatisfied(M_2,R_2),T)) \\ )\rightarrow happens(satisfy(M_1,R_1),T).\end{array} \quad (4)$$

However, the implication in Axiom 4 does not forbid *satisfy* from happening in different cases (it only states that if the axiom holds true *satisfy* has to happen). We have to define a second rule in Axiom 5, which combined with Axiom 4, builds a bi-implication, because we want that *satisfy only* happens when certain fluents are valid. Axiom 5 can be read as: *satisfy* for M_1 happens if M_1 is unsatisfied with respect to requirement R_1, and if there exists a module M_2 that is connected to M_1 and does provide R_1. Again, M_2 must not be unsatisfied with respect to a different requirement R_2 at the same point in time T.

$$\begin{array}{l}happens(satisfy(M_1,R_1),T)\rightarrow (\\ \ holdsAt(unsatisfied(M_1,R_1),T)\ \& \\ \ ?[M_2]:(\\ \quad holdsAt(connected(M_1,M_2),T)\ \& \\ \quad holdsAt(provides(M_2,R_1),T)\ \& \\ \quad -?[R_2]:(holdsAt(unsatisfied(M_2,R_2),T)) \\ \ ) \\ ).\end{array} \quad (5)$$

3.3.2. The Security Policy

The security policy defines the Security Functional Requirements of a system and how they are interrelated to fulfill its security objective. Informally this is typically documented in a Common Criteria Security Target. A formal description, such as the one in our security policy, has to be used for EAL6 and EAL7.

As we only performed an abstract verification we did not need to describe the detailed properties of the requirements. There are no restrictions for the semantics of axioms in the security policy, so long as they do not contradict the domain description.

Axiom 6 describes an example policy rule for master key handling. Master key handling (*r_master_key*) means that the system uses a master key e.g., for securing confidential data within a system. Such a master key must be either derived (*r_derived_key*) or securely saved in a key store (*r_key_store*). Both requirements are represented in the disjunction, in Axiom 6. Please

notice, that in security policy axioms the requirements are concrete values, while modules and points in time are unbound variables.

$$\begin{aligned}&happens(enables(M_1, r_master_key), T) \rightarrow (\\&\quad happens(requestsR(M_1, r_derived_key), T) \mid\\&\quad happens(requestsR(M_1, r_key_storage), T)\\&).\end{aligned} \tag{6}$$

Axiom 6 can be read as: when any module M_1 provides master key handling at any time point T, it requires either a derived key handling mechanism or a key storage. Notice, that the rule does not state where or how often in the system this has to happen. Neither does it state how this should be implemented. It also does not state that there must only be one module that provides the security requirement. This flexibility is essential for best results during design space exploration.

We do not describe a complete policy in this work, because the security policy is anyway system dependent.

3.3.3. Platform Constraints

Platform constraints allow the reduction of the design space if a certain target platform has already been selected or at least some limiting factors of a platform are known already. These limiting factors are described as Event Calculus axioms. They reduce the number of models that match the formal system description. Primarily, this is useful during the model finding process where the security policy is verified, but it also helps developers keeping platform limitations in mind when doing design space exploration.

One typical platform constraint states that a certain security requirement is provided by one module only. Not doing so allows multiple instances of security mechanisms which has negative impact on the size and price of a system. Axiom 7 is another example. It states that there is no T where any module M provides blinding for keys stored in the system. Blinding is a known countermeasure against side channel attacks. Secret data is masked so that leaked data becomes useless for an attacker.

$$: -holdsAt(provides(M, r_phys_blinded), T). \tag{7}$$

If blinding is not supported automatically (e.g., by the bus), either hardware or software components have to take care of secret key confidentiality. This can be ensured by an axiom in a security policy such as Axiom 8 specifically: keys have to be encrypted (r_key_enc). Similarly, for security policies, as long as there are no contradictions in the domain description, there are no restrictions for the semantics of platform constraints.

$$\begin{aligned}&happens(enables(M_1, r_key_storage), T) \rightarrow (\\&\quad happens(requestsR(M_1, r_phys_blinded), T) \mid\\&\quad happens(requestsR(M_1, r_enc_keys), T)\\&).\end{aligned} \tag{8}$$

4. Case Study

In our case study we developed a model of a smart card system that includes applications, operating system components, and hardware components. As previously described, we started with a pure functional model and refined modules that were important for design space exploration. A security policy was defined and used during modeling and refinement in order to ensure a consistent security architecture.

4.1. The Mixed Abstraction Level Model

Figure 7 shows a snapshot of the model we used for the case study. The model supports different smart card applications that access the operating system features through an application programming interface (API). Two applications were of certain importance for our case study: a card manager application and a Basic Access Control (BAC) [21] application. The card manager application handles the setup of other smart card applications. We used the card manager to install a BAC application. Basic Access Control defines authentication and communication for machine readable documents like e-passports. We used the BAC authentication procedure for our analysis in this case study.

All applications in our case study are written in pure C++. Microsoft Visual Studio 2008 was used to develop the C++ modules. Obviously, C++ is not used to being executed on a smart card platform. The applications in our model were simulated only - they were executed directly by the (simulating) host PC.

The smart card operating system provides basic functionality for memory management (key objects), cryptographic support, and utility functions such as copy routines for arrays. Some operating system components are implemented in C++ (and thus simulated only); others are implemented in C for 8051 and executed on the simulated hardware platform. This allows an early estimation of performance critical operating system functions.

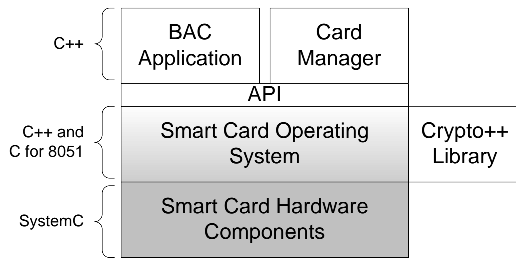


Figure 7: Mixed Level Model for the case study

Keil[4] embedded development tools were used for functions implemented in C.

Vertical transactors interface between the C++ and C components. The transactors are explained in the next section. As indicated in Figure 7 cryptographic support was not yet been refined to the hardware layer. Instead, the Crypto++[5] library was used whenever cryptographic functions were needed.

Smart card hardware was modeled in SystemC[6]. The architecture was based on an 8051 derivate which is typical for smart card systems. The hardware model was a timed transaction level model which provided an instruction set simulator for the 8051 CPU. All SystemC components were developed with Microsoft Visual Studio 2008.

Meta-information was added by using macros in the C/C++ files. Macros were mapping to function calls, which were executed directly by the simulation environment or passed through a special hardware interface from C level to the simulation environment. This hardware interface was used for meta-information only and was not relevant for the final implemented system. Log-files were created for processing by Event Calculus tools for model checking and are described in the following sections of this chapter.

4.2. Vertical Transactors

Vertical transactors were used to pass data between the C++ and the C layer. The C++ layer was directly executed by the simulating host PC while the C functions were executed by the instruction set simulator embedded in the transaction level model, representing the 8051 environment. Thus, when a C function was called in the C++ layer (1) the memory of the PC and the simulated micro controller had to be synchronized, (2) the program counter of the 8051 had to be set to the address of the C function to be executed, and (3) the transaction level model had to be started while the high level abstraction model has to be set to idle until the C function returned. Finally, (4) memory had to be synchronized again in order to reflect memory changes in the C++ layer.

Memory synchronization and program counter handling was done by transactor instances that could be instantiated in the high level model. Process synchronization of high and low level processes was handled by the SystemC simulation environment. A mapping file, generated by Keil development tools during compilation of the C function, contained RAM addresses of data segments and ROM addresses of C functions. The former ones were used for memory synchronization; the latter ones were used to set the program counter of the instruction set simulator, so that it can execute the C functions.

4.3. The Security Policy

The used security policy contained 8 axioms. To achieve better readability, instead of Event Calculus syntax, a textual description is provided in Table 1. As we restricted our case study to BAC authentication processing the security policy only described the key confidentiality of cryptographic keys stored in the smart card system. These keys were used for the authentication process. The policies included rules for key loading, key storage, key usage, and secure destroying of keys with respect to side channel attacks. Among other things, this means that keys must not be stored or transferred within the system without protection against information leakage (e.g., through power profiles or electromagnetic emission).

4.4. Tools and Flow

We used the f2lp Event Calculus solver described in [17]. Using f2lp in an abductive task required *choice rules* for certain events defined in our domain. Axiom 9 shows the choice rule for $accesses$. It allows the model solver to automatically generate $accesses$ events on its own to find a matching model. Choice rules for $accesses$, $frees$, *releasesR*, *enables*, and *disables* were necessary. Notice, that there was no choice rule for $satisfy$ as the occurrence of this event was strictly defined by other axioms (described in 3.3.1). Without choice rules, f2lp assumes that no events happen without an explicit rule in the domain description. Other tools, such as the decreasoner do not need choice rules but generate necessary events automatically. *Circumscription*, if supported by the tool, makes sure that only

[4] `http://www.keil.com`
[5] `http://www.cryptopp.com`
[6] `http://www.systemc.org`

Axiom Nr.	Description
1	A key storage is either protected by blinding or keys are stored encrypted. A key storage holds the secret keys. To protect the keys against side channel attacks the keys must be blinded or encrypted. This ensures that leaked information is useless for an attacker.
2	If keys are encrypted, a master key is needed. Encrypting keys requires a system wide secret key. This is the master key.
3	A master key can be derived or stored in a key store. A master key is a secret key. Thus, it has to be protected against unwanted access by an attacker. Derived keys are not stored directly in persistent memory but calculated when they are needed. A key store can also be used but notice the circular dependency.
4, 5, and 6	Whenever a key is used it has to be read, used with a cryptographic operation, and deleted. Usage of a key consists of three phases: (1) read the key from where it was stored and store it temporally at a location where it can be used by the crypto- subsystem. This can be skipped when the crypto-subsystem supports in-place operations for keys. (2) use the key with a cryptographic operation. (3) delete the temporary stored key. This rule is split in three separate axioms.
7	A secure read operation for keys must provide a key store, a key buffer, and a secure bus or a secure copy routine. A secure read has to be performed from a secure storage for keys - the key store. The key must not be stored temporary at any arbitrary location. A dedicated crypto-buffer has to be in place to handle keys securely. The copy routine itself must not leak information through side channels. Either this is done by hardware countermeasures (e.g, a secure bus) or by a secure copy routine in software.
8	A secure copy routine requires reliable random numbers. A secure copy routine requires reliable random numbers to hide information about the secret data to be copied. These random numbers need to be created and tested in a special way so that an attacker cannot predict or manipulate them and thus reduce the security of the copy routine.

Table 1: Security Policy Rules

necessary events are created in order to match the rules in the model [19].

$$happens(accesses(M1, M2), T). \quad (9)$$

Our SystemC simulation environment generated a security report that could not directly be used with f2lp because SystemC uses a different time base (pico seconds) than f2lp (time points without defined units starting at zero). A tool was implemented, which converted SystemC time stamps to f2lp time points. Such Conversion is a valid procedure, because for the verification the elapsed time between the events is irrelevant as long as the sequence of events does not change.

For the abductive model finding process, f2lp was used with (1) the domain description, (2) the security policy, (3) the platform constraints, (4) the choice rules, and (5) an initial condition that had to be fulfilled. Each of these 5 components were stored in separate textual files to make them easily replaceable. Figure 8 shows a demo call of the f2lp tool. In this demo, `sec_copy_demo.fo` was used which included an initial condition. This initial condition initiated the usage of a secure array copy function. Without such an initial condition only one empty model without any events would be found. Notice, that such an empty model would in fact match the domain description and the security policy as without events no contradictions would be found.

```
$ ./f2lp.exe ../domain.fo ../policy.fo ../cons.fo
../choice.fo ../sec_copy_demo.fo ./DEC.lp |
 sed 's/_NV_/NV_/' | ./gringo.exe -c maxstep=3 |
 ./claspD-1.1.exe -n 1
claspD version 1.1. Reading...done

Models       : 0
Time         : 0.359  (Parsing: 0.359)
```

Figure 8: Demo Run of the f2lp Tools with too short Time Range

As can be seen in Figure 8, the tool had to be used with a set of other tools (sed, gringo, and claspD). For details please consult the documentation of f2lp. The tool was used with a maximum time range of 4 time points (`maxstep=3`) from 0 to 3 for resulting models. This time range was too short: the abductive process was not able to find any models within this range (`Models : 0`).

For Figure 9, the time range was extended by one time point, from 0 to 4. At that time the tool was able to find models. The number of models to return was lim-

```
$ ./f2lp.exe ../sec_copy_demo.fo ../domain.fo
../policy.fo ../choice.fo ../cons.fo ./DEC.lp  |
sed 's/_NV_/NV_/' | ./gringo.exe -c maxstep=4 |
./claspD-1.1.exe -n 1
claspD version 1.1. Reading...done
Answer: 1
happens(enables(module1,r_sec_copy),0)
holdsAt(provides(module1,r_crypto_in_place_key),1)
holdsAt(provides(module1,r_crypto_key),1)
holdsAt(provides(module1,r_derived_key),1)
holdsAt(requires(module1,r_rng),1)
...
happens(enables(module2,r_delete_key),4)

Models      : 1+
Time        : 0.421  (Parsing: 0.421)
```

Figure 9: Demo Run of the f2lp Tools with sufficient big Time Range

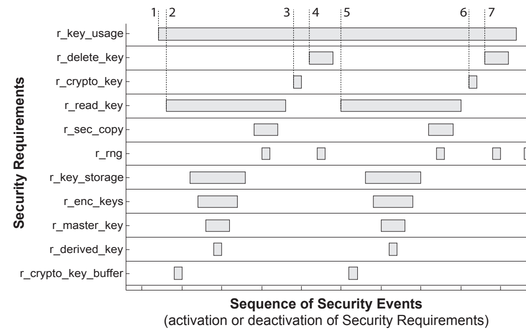


Figure 10: Traces of Security Requirements of the High Level Model

ited to 1 in Figure 9 but the tool reported that other models might exist (`Models : 1+`). Unfortunately, Figure 9 does not show a complete found model; it had to be shortened (indicated by the ...), because of space limitations in this paper.

For deductive model checking f2lp was used with (1) the domain description, (2) the security policy, (3) the platform constraints, and (4) the SystemC security report (with converted time stamps). The tool's call was very similar to that shown in Figure 8 and 9. The tool created a model when the security verification was successful. If there were any security violations with respect to the used policy, no model was produced.

5. Results and Discussion

In the next subsections we discuss the results of our case study. This includes examples of system optimizations which show the potential of our proposed methodology. This section also includes a summary of the achieved verification performance with the implemented Event Calculus approach. Finally, it comments on experiences of using Event Calculus tools.

5.1. System Optimization Results

The starting point for our optimization was a functional high level model which included meta-information about security requirements. This was the first model in our case study that was used for security verification. All modules of the model were implemented in C++. During one BAC authentication we traced the activation and deactivation of security requirements. One section of the traced security requirements is demonstrated in Figure 10. As can be seen on the x-axis of the diagram, there is no real timing information provided by the functional model. Instead, a sequence of activated and deactivated security requirements can be seen.

Marked time point (1) shows the beginning of key usage (*r_key_usage*) in the BAC authentication. Two keys were read (*r_read_key*) in this phase marked with (2) and (5), and preceded cryptographic operations (*r_crypto_key*) in (3) and (6). After that, the key was deleted (*r_delete_key*) in (4) and (7). A secure array copy function (*r_sec_copy*) was used during the read operation of the keys. This copy routine required reliable random numbers (*r_rng*).

During all of the following optimization steps we used such requirement traces to verify them against the security policy. In fact, none of the optimizations performed failed our security verification.

We started refinement by adding a transaction level model for the 8051 micro controller. To utilize these hardware components `secureArrayCopy()` (which provides *r_sec_copy*) was implemented among other functions in C for 8051. According to our security policy we were able to use an appropriate implementation of this function to copy keys without leaking information about the key through side channels. As this function required reliable random numbers (*r_rng*), we added a hardware random number generator to the transaction level model. By reliable random numbers in this context we refer to numbers that were generated by a true random number generator and which were statistically tested. Notice that generation and testing of random numbers is very time consuming.

A corresponding section of the requirement trace of the mixed level model can be seen in Figure 11. Timing information exists in the requirement trace (see the x-axis in Figure 11), since the transaction level model contains a timed instruction set simulator and some timed hardware modules. The most time consuming blocks were *r_sec_copy* at (1) and (3) and *r_delte_key* at (2) and (4). The reason for such high time consumption were

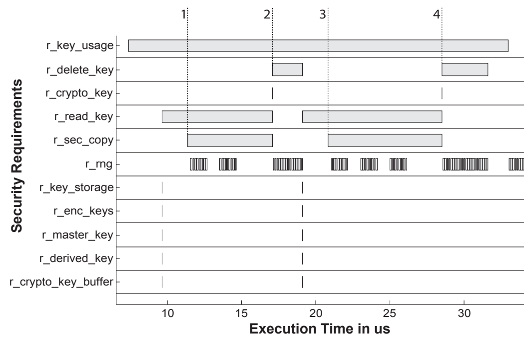


Figure 11: Mixed Level Model

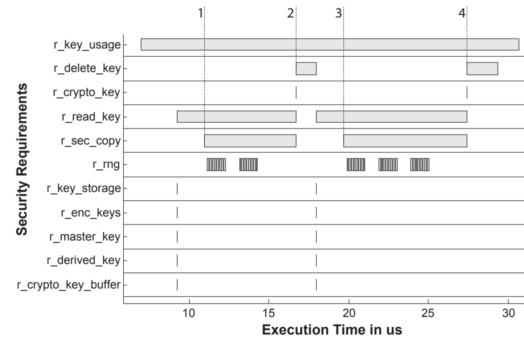


Figure 12: Key Deletion Optimization

the generation and testing of random numbers. Notice, that cryptographic operations did not consume time because they were implemented on a very abstract level. Furthermore, please notice that according to the security policy in Table 1, *r_delete_key* did not require the use of reliable random numbers. However, the implementation overwrote the key with random numbers from the random number generator as this was the only source for random numbers in the modeled system. Thus, *r_rng* appeared in the trace files.

The `secureArrayCopy()` implementation required many reliable random numbers because it is based on random permutation of bytes to be copied. In our first mixed level model 28% of the time required for BAC authentication was consumed by random number generation. Although this is not a very accurate result because too many essential modules had not yet been refined and do consequently not consume time at all. For our case study we decided to increase the system's execution performance by using a more efficient, and less expensive random number generation method.

Consequently, we tried to take advantage of random numbers that do not need to have such a high security level. The time consuming statistical test for such numbers could be skipped. As mentioned, `deleteKey()` is one function that does not need reliable random numbers. Thus, we decided to change the model in a way, that `deleteKey()` cold use untested random numbers.

To do so we refined the hardware random number generator. We implemented a mode for tested, but time consuming generation of random numbers, and another mode for untested, and fast generation of random numbers. This was a crucial design decision. Great care had to be taken, so that random numbers did not get generated in the wrong mode, specifically so that untested random numbers would not be used where tested random numbers were needed. Furthermore, security verification had to show that the random number generator was used in the right mode whenever needed.

However, with our security verification methodology we were able to show that, with respect to the security policy, tested random numbers were used for `secureArrayCopy()`. The requirement trace is shown in Figure 12. The secure copy routine still used tested random numbers in (1) and (3), but the deletion of keys in (2) and (4) did not (*r_rng* does not appear in these cases anymore). This reduced the time consumed by random number generation to 14% and the total authentication time by 9%.

The last system optimization discussed in this work as our attempt to get rid of the time consuming `secureArrayCopy()` function. Similarly to random number generation, we added a mode to the micro controller's bus that enabled hardware blinding for all transmitted data that. When the bus worked in the new added mode, it automatically blinded the transmition of data with a random number. The hardware now provided a countermeasure against side channel attacks, and `secureArrayCopy()` was not needed anymore.

Again, verification against the security policy ensured that the bus was used in the right mode, whenever it was needed. The requirement trace in Figure 13 shows that *r_sec_copy* was not used anymore. Instead, *r_bus_key* was active whenever keys were copied. (*r_read_key*).

Running the bus in the blinding mode caused an additional communication overhead for every bus transaction. Several simulations with an overhead of 1, 5, 10, 20, and 50 clock cycles were performed to evaluate the impact on the system's execution performance. In addition the clock rate was varied between 8 to 48 MHz. The comparison to the previous multi level model (with the hardware random number generator) is shown in Figure 14. For a typical overhead of up to 10 clock cycles, a performance gain of 40% can be achieved when com-

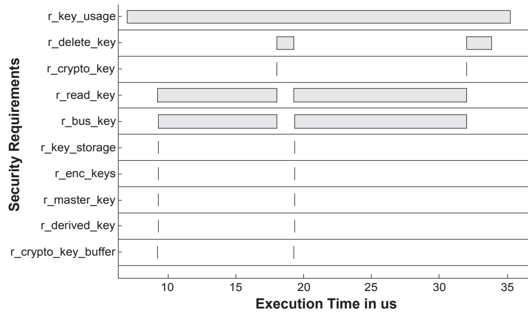


Figure 13: Without Secure Copy

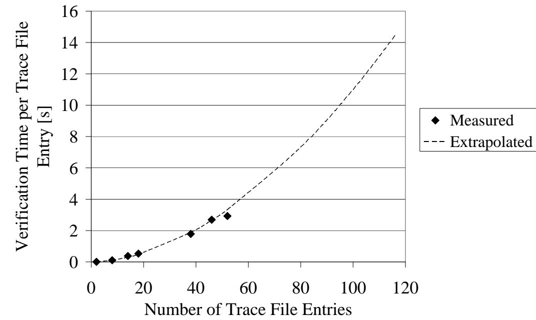


Figure 15: Verification Performance

pared to the performane of `secureArrayCopy()` with a modified random number generator.

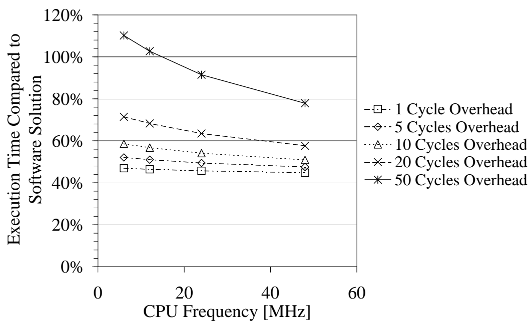


Figure 14: Performance Comparison with previous Model

5.2. Verification Performance

The verification performance of the f2lp tool strongly depends on the number of trace entries in the SystemC security report. We assume that this is caused by longer temporal sequences, which need to be verified, and by the respectively higher number of potential events that can happen. Hence, it is useful to split up the trace files containing security reports into small independent portions, which can be verified separately.

Figure 15 shows some measurement results of the verification performance and the extrapolated curve. The security report from the last multi level model has 116 trace entries. According to the extrapolated curve in Figure 15 this would result in a verification performance of approximately 14,4 seconds per trace entry for a conventional developer PC. This results in almost half an hour for verification of the entire security report.

Table 5.2 shows how we split the trace file to increase the verification performance. The split was done in a very rudimentary way: blocks of traces which are independent from each other were split into separate files. More sophisticated ways to split up the trace files might exist. As shown in Table 5.2 the entire verification was completed in less than 87 seconds. Without the negligible time to split up the trace file, this is a performance gain of almost 95%.

Number of Traces	Verification Time	Comment
38	67,9s	
14	5,3s	3 times
8	0,9s	3 times
2	< 0,1	6 times
Total: 116 traces		Total: < 87s

Table 2: Verification Times with split Trace File

5.3. Event Calculus Tools

Based on our experience with Event Calculus tools we have to say that the user friendliness of those tools leaves room for improvement. The decreasoner showed better performance in the model finding process, especially because it did not require choice rules and used circumscription to generate a minimum set of necessary events. However, f2lp showed a better overall performance and needed less memory. Overall, memory consumption as high for both tools. It strongly depended on the number of involved modules and time points. During our case study we did not manage to verify more than 55 time points at a time. Fortunately, the security reports can easily be split into smaller portions.

Additionally, we have to report very restricted ways to debug axioms. In most cases an error in an axiom lead to a contradiction during verification. In such cases the decreasoner returned partly matching models for debugging purposes, but f2lp did not provide any debugging support. Thus, the main part of the domain description in this work was developed with decreasoner while the security verification was performed with f2lp.

Overall, we have shown that Event Calculus based verification is possible with today's tools. However, it turned out that todays tools need improvement before they can be used in an industrial context or similar complex verification scenarios. Memory consumption, execution performance, and user friendliness should be targeted in future versions of Event Calculus tools.

6. Conclusion

Development of future embedded systems will hardly be done without any security verification. Complexity of these systems has reached a point where both conventional and formal verification techniques have their limitations. Manual verification, as it is done today, will soon become too time consuming and to prone to error. However, existing formal verification methodologies still suffer from computational overhead needed for complete system verification. Thus, an appropriate abstraction is needed; one which is able to cover issues related to design and implementation decisions.

We showed a development and verification methodology that is capable of working on different abstraction levels at the same time. Depending on developers' decisions, modules can be refined sooner or later in the development process. Furthermore, meta-information about security requirements is used to support automated simulation based formal verification. Model development, functional verification, and security verification are fully supported by the discussed novel concepts of vertical transactors and model checking based formal security verification utilizing the Event Calculus.

To explain the practicality of our concept we demonstrated a case study based on a smart card system, which included applications, a smart card operating system, and smart card hardware. Model components were implemented on different layers of abstractions, but verified all together against a formal security policy to argue for the system's over-all security.

Even though existing Event Calculus tools show room for improvement, we believe that a formal predicate based approach such as discussed in this work, can help system architects and developers deliver products with security levels appropriate for future use cases and attack scenarios.

References

[1] Sun Microsystems Inc., Java card platform specification 3.0.1, 2009.

[2] Common Criteria Recognition Arrangement, Common criteria for information technology security evaluation - part 1-3, 2009. Version 3.1 Revision 3 Final.

[3] D. S. Herrmann, Using the Common Criteria for IT Security Evaluation, CRC Press, Inc., Boca Raton, FL, USA, 2002.

[4] F. Keblawi, D. Sullivan, Applying the common criteria in systems engineering, Security Privacy, IEEE 4 (2006) 50–55.

[5] D. Mellado, E. Fernández-Medina, M. Piattini, A common criteria based security requirements engineering process for the development of secure information systems, Comput. Stand. Interfaces 29 (2007) 244–253.

[6] J. Loinig, C. Steger, R. Weiss, E. Haselsteiner, Idea: Simulation based security requirement verification for transaction level models, in: Engineering Secure Software and Systems, volume 6542 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2011, pp. 264–271.

[7] S. Morimoto, S. Shigematsu, Y. Goto, J. Cheng, Formal verification of security specifications with common criteria, in: SAC '07: Proceedings of the 2007 ACM symposium on Applied computing, ACM, New York, NY, USA, 2007, pp. 1506–1512.

[8] S. Motre, C. Teri, Using B method to formalize the Java Card runtime security policy for a Common Criteria evaluation, in: 23rd National Information Systems Security Conference.

[9] V. Lotz, V. Kessler, G. Walter, A formal security model for microprocessor hardware, Software Engineering, IEEE Transactions on 26 (2000) 702–712.

[10] D. Ghindici, G. Grimaud, I. Simplot-Ryl, Y. Liu, I. Traore, Integrated security verification and validation: Case study, in: Local Computer Networks, Proceedings 2006 31st IEEE Conference on, pp. 1000–1007.

[11] R. Drechsler, D. Grosse, Reachability analysis for formal verification of SystemC, in: Proceedings of the Euromicro Symposium on Digital Systems Design, IEEE Computer Society, Washington, DC, USA, 2002.

[12] K. L. Man, Enhancing formal methods for systemc designs, in: Proc of 4th Process Symposium on Embedded Systems.

[13] E. Clarke, D. Kroening, K. Yorav, Behavioral consistency of C and verilog programs using bounded model checking, in: Proceedings of the 40th annual Design Automation Conference, DAC '03, ACM, New York, NY, USA, 2003, pp. 368–371.

[14] N. Amalio, Suspicion-driven formal analysis of security requirements, in: Emerging Security Information, Systems and Technologies, 2009. SECURWARE '09. Third International Conference on, pp. 217–223.

[15] T. T. Tun, Y. Yu, C. Haley, B. Nuseibeh, Model-based argument analysis for evolving security requirements, in: Secure Software Integration and Reliability Improvement (SSIRI), 2010 Fourth International Conference on, pp. 88–97.

[16] A. Mana, G. Pujol, Towards formal specification of abstract security properties, in: Availability, Reliability and Security, 2008. ARES 08. Third International Conference on, pp. 80–87.

[17] J. Lee, R. Palla, Classical logic event calculus as answer set programming, in: Working Notes of the Workshop on Answer Set Programming and Other Computing Paradigms.

[18] M. Shanahan, The event calculus explained, in: Artificial intelligence today, Springer-Verlag, Berlin, Heidelberg, 1999, pp. 409–430.

[19] E. T. Mueller, Commonsense Reasoning, Morgan Kaufmann, 2006.

[20] J. Loinig, C. Steger, R. Weiss, E. Haselsteiner, Towards formal system-level verification of security requirements during hardware/software codesign, in: SOC Conference (SOCC), 2010 IEEE International, pp. 388–391.

[21] T. A. F. Kinneging, PKI for Machine Readable Travel Documents offering ICC Read-Only Access Version 1.1, Technical Report, International Civil Aviation Organization, 2004.

Identification and Verification of Security Relevant Functions in Embedded Systems Based on Source Code Annotations and Assertions

Johannes Loinig[1], Christian Steger[1],
Reinhold Weiss[1], and Ernst Haselsteiner[2]

[1] Institute for Technical Informatics, Graz University of Technology, Graz, Austria
{loinig,steger,rweiss}@tugraz.at
[2] NXP Semiconductors Austria GmbH, Gratkorn, Austria
ernst.haselsteiner@nxp.com

Abstract. Most modern embedded systems include an operating system. Not all functions in the operating systems have to fulfill the same security requirements. In this work we[1] propose a mechanism to identify and maintain functions that have to meet strict security needs. This mechanism is based on annotations representing security constrains and assertions to check these security annotations during the verification phase of the system under development.

1 Introduction

Every modern operating system (OS) is split in hundreds of functions. Not every function has to fulfill the same security requirements. For example, some of them must not leak secrets such as cryptographic keys. Others must be timing invariant for all kinds of inputs. Finally, every OS has many functions that do not need to fulfill special security requirements. Of course they have to work properly and must not open back doors to potential attackers (e.g. by buffer overflows) but actually we do not regard this as a special security constraint. In fact, wee consider this property as normal. In the remainder of this work we call them *security neutral functions*.

Implementing all functions on the highest security level (with respect to all possible security countermeasures) is not feasible and not necessary.The development cost would be too high as implementing secure functions is more time consuming. The performance of a system would be too slow as additional security usually causes a computational overhead. To compensate this faster hardware would be necessary which again increases the costs. Finally, the executable code would increase which is a significant cost factor if the code is masked into the ROM.

If a function has to meet a selected security constraint (SC), e.g. to check program flow integrity, it must be ensured that every subfunction which is called also has to meet the same SC. If not, this may raise a weak point in the chain

[1] This paper is a result of the *HiPerSec* project which is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the contract FFG 816464.

P. Samarati et al. (Eds.): WISTP 2010, LNCS 6033, pp. 316–323, 2010.
© IFIP International Federation for Information Processing 2010

©2010 Springer Verlag. Reprinted, with permission, Lecture Notes in Computer Science, 2010, Volume 6033/2010

of trust representing the call hierarchy of the functions. As the security of the system can only be as strong as the weakest point, the developers must take care that they do not use security neutral functions for operations which are expected to provide security.

The aim of our work is to provide a mechanism to differentiate between functions that have to fulfill security requirements and functions that do not. To do so, we use in-line source code annotations to mark security relevant parts of the source code. Our proposed mechanism allows a design tool to verify these security annotations during the development process and thus helps developers to increase and maintain the security and performance of complex embedded systems.

2 Related Work

Security has to be considered from the beginning of the product life cycle. The implementation of security elements have to be done in a well organized way [1]. Kocher et al. state in [2] that it is a problem that the implementation is very often done by security experts who are the only people in a development team that really understand the security requirements. The reason is that a system's security is deeply rooted in the complete development process and cannot be implemented by covering some few selected points that were identified to have to be secure. A cryptographic function, for example, can be implemented in a perfect way but will be simply useless if the private keys are handled in an insecure way while loading them from memory.

Furthermore, [2] states that a formal verification of programs with realistic complexity is not feasible today. Good engineering practice which covers all software artifacts as security objectives is necessary to develop secure products. Analysis tools and techniques to map security requirements to solutions and explore trade-offs are needed.

In [3] existing static code analysis tools for security checks are summarized. There exist a lot of tools checking for vulnerable constructs, proper usage of types and values, race conditions, and so on. However, the authors state that there is still a deficit for checkers that include relationships between functions.

A model-based planning strategy for security requirements is described in [4]. A high-level model instead of source code is used to verify the formal properties of functional and security requirements.

Source code annotations are declarative information for runtime entities and are supported by several modern programming languages [5] and software frameworks [6]. They can be evaluated by tools during the development process or by the runtime environment during execution time. One example is `@deprecated` in Java which defines that a function should not be used anymore. The Java compiler generates a warning if a deprecated function is used. In this work we use annotations to define which SCs were considered during the implementation of the annotated software functions.

Our proposed mechanism can be used to analyze the final source code of the product. It verifies chains of trust through the whole software of an embedded system across boundaries of functions and software layers. To do so, we use well

©2010 Springer Verlag. Reprinted, with permission, Lecture Notes in Computer Science, 2010, Volume 6033/2010

known and established tools like source code annotations and assertions during the development process.

3 Identification and Verification of Security Relevant Functions

The basis of our proposed concept is that a developer of a security relevant function does not need to think about the security status of called subfunctions. The developer knows which SCs are defined for a function that has to be implemented. He or she can implement the function without risk that used subfunctions will not be able to provide the necessary SCs. Thus, the development process is widely concentrated on the new function which makes development easier, faster, and more secure.

Figure 1 shows the basic workflow of our proposal. Different developers implement functions according to their functional requirements and security requirements. They annotate the newly implemented functions with appropriate SCs. We define the meaning of SCs in more detail in Section 3.2. When the system's modules become integrated a design tool checks the security properties of all used functions, including functions in external libraries. The design tool reports security violations if security properties of functions can not be fulfilled by called subfunctions.

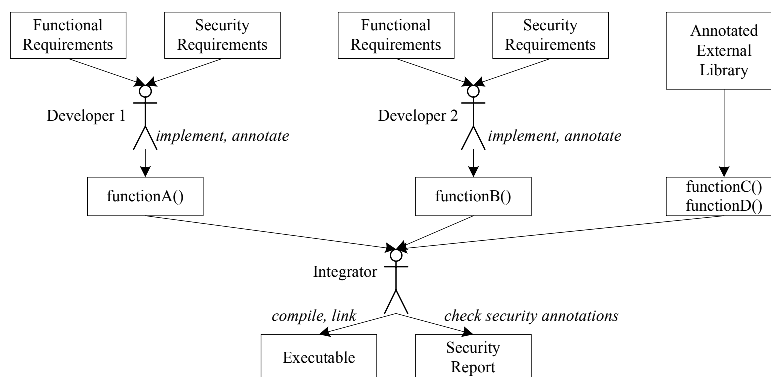


Fig. 1. The workflow of our proposed concept

Furthermore, our proposed concept allows identification of subfunctions that may have a too high security status. Such a function implements SCs that are not required by the calling function. This may be the case intentionally, if the subfunction requires the SC, or by accident if a subfunction was reused. In latter case, the appropriate usage of a function with a lower security status instead, can increase the system's performance without security drawbacks.

©2010 Springer Verlag. Reprinted, with permission, Lecture Notes in Computer Science, 2010, Volume 6033/2010

3.1 Assigning Security Constraints

Every function is annotated with the SC it implements. The same SC is supposed to be provided by called subfunctions. If not, the chain of trust which is implicitly given by the function call hierarchy is broken. This is an indication for a potential security gap in the system. If the subfunction provides a higher security level this is an indicator for potential performance optimizations.

The concept is shown in Figure 2. As can be seen `functionB()` breaks the chain of trust and `functionC()` provides a higher security status which may cause unnecessary performance reduction because the additional but unnecessary security may slow down the function's execution.

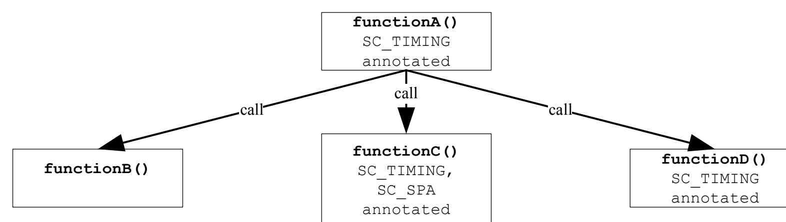


Fig. 2. Simple example of functions which are annotated with security constraints

3.2 Definition of Security Constraints

The proposed concept is not restricted to a certain number or nature of security constraints. A threat model or attack tree can be used to deduce necessary SCs at the beginning of the system's design phase. We derived a list of SCs for the software implementation of a smart card from attacks described in [7,8,9,10]. The list below is not exhaustive but should indicate how the concept of SCs works.

Timing Attack (SC_TIMING): A function which is annotated with this annotation has to provide timing which is independent from any input data to avoid timing attacks.

Simple/Differential Power Analysis (SC_SPA, SC_DPA): Changes of the system's power state can be observed externally and must be avoided.

Differential Fault Analysis (SC_DFA): Fault injection can not be prevented by software but the function can provide e.g. recalculation and comparison of results to detect potential injected faults.

Perturbation Attacks (SC_PERT): An annotated function must check the integrity of data and the integrity of the program flow to detect disturbances of the normal software behavior (e.g. caused by laser beam attacks).

The detailed meaning and implementation of an SC can vary between different systems and different functions. In a real development process the SCs have to be clearly defined and communicated to the developers. Assigning SCs should be followed by a code review done by a different developer to ensure that the system is not corrupted by miss-assigned annotations.

©2010 Springer Verlag. Reprinted, with permission, Lecture Notes in Computer Science, 2010, Volume 6033/2010

3.3 Annotation Based Identification of Security Relevant Functions

The basis of our proposed mechanism is a function call tree. A call tree shown in Figure 3 (a) is extended to a tree representing the chains of trust of the calling functions, shown in Figure 3 (b). When the call tree is transformed to the tree of trust chains each node is replaced by a node including all SCs assigned to the corresponding function.

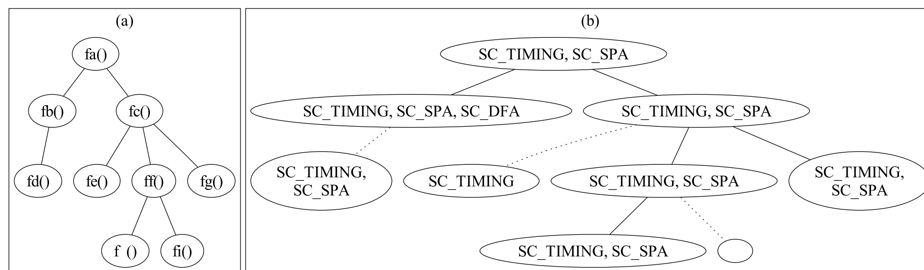


Fig. 3. A call tree in (a) and after its transformation to a tree of trust chains in (b)

A tool runs through all the nodes in the extended call tree. Every node must include all SCs of its parent. If a broken chain of trust is found the tool reports the security constraint violation.

The generation of the call tree can be done by static code analysis or during execution of a use case. It may be difficult in some situations to setup a capable code analysis if different layers of software are used. If, for example, the system is based on a virtual machine (VM), it may be difficult to maintain function calls from the software running on the VM to the underlying software layers. An example is a Java Card [11], a smart card including a Java VM. Such a system can easily be split in four different software layers: the Java application, the Java OS, the native hardware-independent software written in C, and the assembler functions. In such a case, an execution based generation of a call tree may be more promising than a static code analysis through software layers implemented in different programming languages.

3.4 Assertion Based Verification of Security Relevant Functions

For system verification we propose an assertion based mechanism that checks the security constraints during the verification of the system. When a subfunction is called, all SCs of the calling function are passed to the subfunction to be checked. The subfunction provides an assertion and verifies if all necessary SCs are implemented.

The SCs are only used during the system development and verification process. They are not needed anymore when the system operates in the field. Therefore, neither annotations nor assertions have to be included in the final product.

©2010 Springer Verlag. Reprinted, with permission, Lecture Notes in Computer Science, 2010, Volume 6033/2010

4 Implementation

We implemented our concept on the basis of a real but simulated Java Card operating system. As application we chose the JavaPurse application included in the Java Card Development Kit [12]. So far, the OS does not provide any annotations of SCs. Thus, we annotated the security relevant high-level functions of the JavaPurse `processVerifyPIN()`, `processInitializeTransaction()`, and `processCompleteTransaction()`. All three methods are called in `process()` which is called for every command that is received by the JavaPurse application. We identified these three functions as security relevant because they check if the card holder is able to provide the right PIN and initialize respectively complete the payment transaction.

The used simulation environment is a SystemC [13] model of the smart card hardware. We executed one payment transaction and recorded the broken chains of trust which emerge from our annotations in the JavaPurse application. As only functions of the Java Card application were annotated, the mechanism reports all functions in the OS that are called during the payment transaction.

The SystemC model uses an annotation stack which is filled with annotations that have to be implemented by the current called function. All entries of the stack are checked for every simulated function call. Annotations of functions are passed via a Java API to a special function register in the SystemC model of the smart card. The evaluation of the annotations is done in the simulated call-instructions and return-instructions.

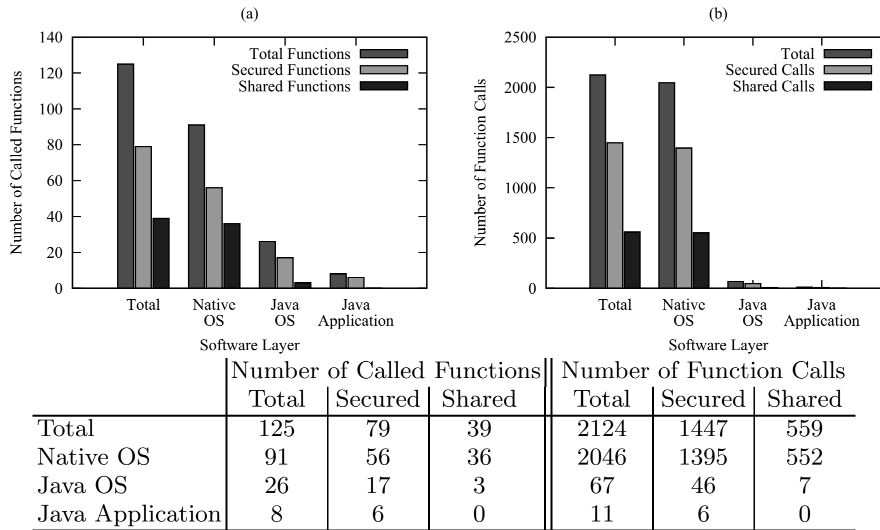
5 Experimental Results

Figure 4 (a) shows the number of functions which were called during one payment transaction of the JavaPurse application. 125 functions were called in total, 91 of them are in the native OS layer, 26 in the Java OS layer, and 8 in the application layer. 63% of them were noted as included in a broken chain of trust, which means that these 79 functions should be checked if they fulfill the security requirement. Notice that for our proof of concept evaluation there was no need to define this requirement in detail. 49% of the functions that have to fulfill special security requirements are also used in a context where the security requirement is not needed (named as shared functions in the diagram). This opens a significant potential for performance optimizations.

Additionally, Figure 4 (a) shows the partitioning of the used functions in the software layers. As expected, most security relevant functions are implemented in the native OS layer. We think that identification of these functions is especially important as they are naturally not secured by the Java VM.

Figure 4 (b) shows the function calls which were executed during the payment transaction. 2124 function calls were executed in total, 2046 in the native OS, and respectively only 67 and 11 function calls in the Java OS and application layer. 68% of all function calls were marked as security relevant by our mechanism. 39% of them were also done in a security neutral state of the system. As can be seen in the diagram the number of Java function calls is minimal in comparison to the native function calls. Therefore, we can argue that the overhead from our

©2010 Springer Verlag. Reprinted, with permission, Lecture Notes in Computer Science, 2010, Volume 6033/2010



	Number of Called Functions			Number of Function Calls		
	Total	Secured	Shared	Total	Secured	Shared
Total	125	79	39	2124	1447	559
Native OS	91	56	36	2046	1395	552
Java OS	26	17	3	67	46	7
Java Application	8	6	0	11	6	0

Fig. 4. Results Diagrams and Table. (a) The number of function calls, (b) the number of called functions.

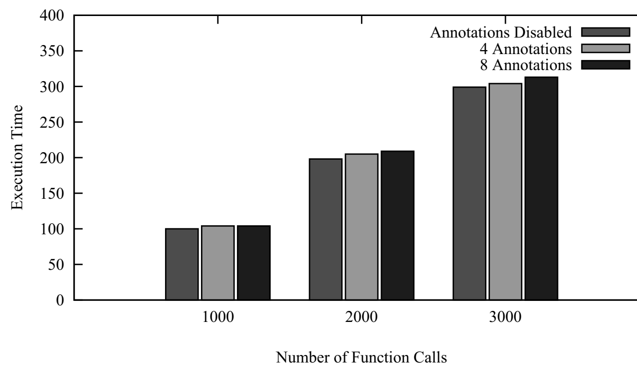


Fig. 5. Simulation Performance of the Annotation/Assertion Mechanism

additional Java API, passing the annotations through the VM to the SystemC model, is irrelevant.

Our modification of the SystemC model, checking the functions' annotations, did not cause any significant performance reduction. This is shown in Figure 5. A simple demo program executed 1000, 2000, and 3000 function calls with 4 annotations, 8 annotations, and disabled annotation mechanism. We normalized the results to 100% for 1000 functions without annotations. According to the results in Figure 5 we can argue that the simulation based verification performance does not suffer drastically from our proposed assertion mechanism.

©2010 Springer Verlag. Reprinted, with permission, Lecture Notes in Computer Science, 2010, Volume 6033/2010

6 Conclusion

In this work we proposed an annotation based method to identify security relevant functions in complex software of embedded systems. This ensures that security constraints of functions are not violated by their called subfunctions and hence increases the system's security. In addition we presented how the same mechanism identifies functions which should be considered for performance optimizations as they do not have to fulfill special security constraints all the time. 49% of all called functions in our demo application were potential targets for such optimizations.

We verified our concept by a proof-of-concept implementation checking assertions for security constraints during simulation time of a SystemC model. Our tests showed that this can be done without significant performance overhead during the simulation of complex systems.

Summarizing this, our approach can be used to optimize the trade-off between security and performance of a complex embedded system.

References

1. Schaumont, P., Verbauwhede, I.: Domain-specific codesign for embedded security. Computer 36(4), 68–74 (2003)
2. Kocher, P., Lee, R., McGraw, G., Raghunathan, A.: Security as a new dimension in embedded system design, pp. 753–760 (2004)
3. Chess, B., McGraw, G.: Static analysis for security. IEEE Security & Privacy 2(6), 76–79 (2004)
4. Bryl, V., Massacci, F., Mylopoulos, J., Zannone, N.: Designing Security Requirements Models Through Planning. In: Dubois, E., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 33–47. Springer, Heidelberg (2006)
5. Ernst, M.D.: Type Annotations Specification (JSR 308) (November 2008)
6. Newkirk, J., Vorontsov, A.: How .NET's custom attributes affect design. IEEE Software 19(5), 18–20 (2002)
7. Eagles, K., Markantonakis, K., Mayes, K.: A comparative analysis of common threats, vulnerabilities, attacks and countermeasures within smart card and wireless sensor network node technologies. In: Sauveron, D., Markantonakis, K., Bilas, A., Quisquater, J.-J. (eds.) WISTP 2007. LNCS, vol. 4462, pp. 161–174. Springer, Heidelberg (2007)
8. Sere, A.A., Iguchi-Cartigny, J., Lanet, J.L.: Automatic detection of fault attack and countermeasures. In: WESS 2009: Proceedings of the 4th Workshop on Embedded Systems Security, pp. 1–7. ACM, New York (2009)
9. Rankl, W., Effing, W.: Smart Card Handbook. John Wiley & Sons, Inc., New York (2003)
10. Common Criteria: Application of Attack Potential to Smartcards Version 2.7 Revision 1 (March 2009)
11. Sun Microsystems, Inc.: Java Card Platform Specification 2.2.2
12. Sun Microsystems: Java Card Development Kit (March 2006), `http://java.sun.com/javacard/devkit/`
13. IEEE: Open SystemC Language Reference Manual (December 2005)

©2010 Springer Verlag. Reprinted, with permission, Lecture Notes in Computer Science, 2010, Volume 6033/2010

Idea: Simulation Based Security Requirement Verification for Transaction Level Models

Johannes Loinig[1], Christian Steger[1],
Reinhold Weiss[1], and Ernst Haselsteiner[2]

[1] Institute for Technical Informatics, Graz University of Technology, Graz, Austria
{johannes.loinig,steger,rweiss}@tugraz.at
[2] NXP Semiconductors Austria GmbH, Gratkorn, Austria
ernst.haselsteiner@nxp.com

Abstract. Verification of security requirements in embedded systems is a crucial task - especially in very dynamic design processes like a hardware/software codesign flow. In such a case the system's modules and components are continuously modified and refined until all constraints are met and the system design is in a stable state. A transaction level model can be used for such a design space exploration in this phase. It is essential that security requirements are considered from the very first beginning. In this work we[1] demonstrate a novel approach how to use meta-information in transaction level models to verify the consistent application of security requirements in embedded systems.

1 Introduction

A lot of modern embedded systems need to provide security functionality. Faults in design and implementation of a system can cause serious security issues. Thus, careful security verification is needed. This is a considerable cost factor. External Common Criteria security evaluation (described later) can cost $100k and more and usually takes months. Finally discovered vulnerabilities in such a late development phase cause serious project delay and cost.

Security has to be considered from the beginning of a development process and in all abstraction levels [6,10]. To support this we propose a methodology to use meta-information in transaction level models (TLMs) for early and continuous security verification in a hardware/software codesign flow. TLMs are abstract functional models of the system. Iterative refinement is used for design space exploration until all system constraints are met. In each iteration our methodology supports security verification appropriate to the model's abstraction level.

The contribution of this work is a novel design and development approach that allows continuous security verification. System designers and developers will gain a better security understanding of the system which reduces the risk for a costly failed Common Criteria security evaluation.

[1] This paper is a result of a project which is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology (contract FFG 816464).

Ú. Erlingsson, R. Wieringa, and N. Zannone (Eds.): ESSoS 2011, LNCS 6542, pp. 264–271, 2011.
© Springer-Verlag Berlin Heidelberg 2011

©2010 Springer Verlag. Reprinted, with permission, from Lecture Notes in Computer Science, 2011, Volume 6542/2011

2 Related Work

Our proposed methodology is based on the Common Criteria (CC) security verification process. It applies the basic CC practices to verify TLMs. Thus, to explain our approach, we first summarize the needed CC and TLM basics. After that we shortly list advantages and disadvantages of formal verification methodologies to motivate our contrary simulation based approach.

2.1 The Common Criteria Process

The Common Criteria [3] is *the* de-facto standard for security evaluations of IT products. The entire process is too extensive to be described here. However, some basics should be clarified to understand our proposed approach.

CC is a rather documentation centric approach. Threats against the IT product are analyzed. Security Objectives are defined to counter these threats. Security Functions provide the functionality for them. Notice that Security Functions are rather a concept and not a concrete implementation. Each Security Function is composed of Security Mechanisms implemented in hardware or in software. Security Functional Requirements (SFRs) describe their functional requirements. These SFRs must be provided by the system to achieve the security objectives. The CC Security Target (ST) describes the relations of security threats, objectives, mechanisms, and functions. It is a design document for the security architecture of one certain IT product. In combination with design documentation an external CC evaluator uses the ST to judge if the security architecture is able to counter threats. This is a manual and extensive process.

The authors of [8] mention a lack of a clear relationship between the CC process and a system development approach. However, security engineering should be integrated in the system development process. [8] describes a CC conform UML based approach for security requirement engineering in a software engineering process. In comparison to that, our approach covers system development (hardware *and* software) and is not restricted to UML. Instead, we support implicit security modeling in a TLM of the system.

2.2 Transaction Level Modeling

Transaction level modeling allows development and analysis of system models [2]. The main concept is the abstraction and separation of the computational part of the system and the communication part of the system. In our approach we extend this by the security part of the system. Meta-information in TLMs allows an early estimation of e.g., the system's performance and power/energy consumption [11]. At the time of writing we were not able to find related work considering security requirements in TLMs.

2.3 Formal System Verification

There is no doubt that a formal verification (FV) would be preferable in comparison to our proposed simulation based verification. However, we think that

©2010 Springer Verlag. Reprinted, with permission, from Lecture Notes in Computer Science, 2011, Volume 6542/2011

today one main aspect still acts against FV: FV is very costly [6] if applied on systems with realistic complexity.

As a consequence, FV can be applied on selected parts of a system only or it can be applied on very abstract meta-models of the system. This has been shown for software [4], for hardware [7], and on system level [1]. FV of selected parts only is unacceptable as system security can not be achieved by single separate modules in a system [10]. A meta-model based approach can be difficult to apply in a dynamic design process. The developer would need to work on both, the functional model and the meta-model.

Meta-models are typically written in special modeling languages and can be created manually as a first design step [4,1] or translated from other models by a verification tool [5]. Manual adaptation of the meta-model would be costly and error-prone. FV of a translated model means, that in a strict sense, not the system's model is verified but a model that was generated by the tool. The quality of the verification strongly depends on the translation tool and the capabilities of the modeling language of the meta-model.

The authors of [9] created formal templates for CC SFRs. These templates can be used for FV of the system's security specification. Again, not the system model itself is verified but its specification. In contrast to that our proposed simulation based methodology allows verification of the relationships of Security Mechanisms and SFRs against the functional system model under development.

3 Simulation Based Security Requirement Verification

In comparison to the work described in Section 2 our contribution is (1) the consideration and verification of security requirements in early design phases, (2) a verification on the basis on a TLM under development instead of a meta-model, and (3) a fast simulation based verification applicable for iterative design processes instead of a complete but extensive formal verification approach.

3.1 Iterative TLM Verification

Figure 1 shows the basic concept of simulation based verification in an iterative refinement process. A pure functional level is the basis for a TLM. In multiple iterations the TLM's modules become refined. During simulation of the TLM, when the test cases are applied, an automated verification generates a functional report and a security report. The reports are the basis for further design decisions. Another refinement cycle is necessary if not all constraints are met.

Different actions can be performed during TLM refinement: modules can be split in several smaller modules, combined to bigger modules, and mapped to software or hardware components of the system. All these actions require consideration of SFRs if the modules are related to Security Functions. To do so, the developer annotates the Security Mechanisms with meta-information about related SFRs. These annotations are evaluated during simulation and verification. If module modifications cause security violations this is reported during the simulation/verification and the developer can react immediately.

©2010 Springer Verlag. Reprinted, with permission, from Lecture Notes in Computer Science, 2011, Volume 6542/2011

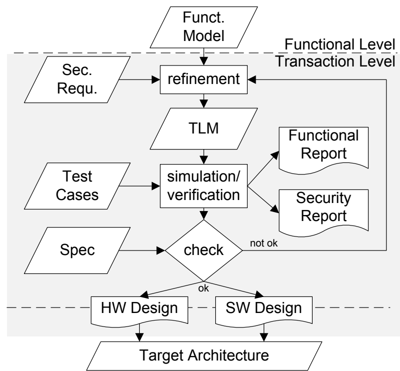


Fig. 1. Simulation based verification

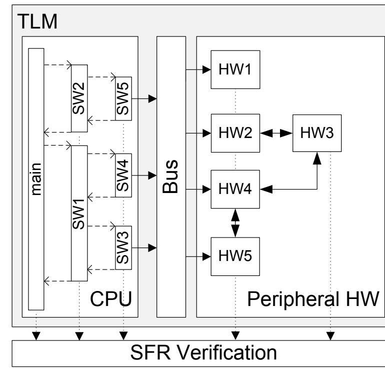


Fig. 2. TLM verification

3.2 HW/SW Verification Approaches

Figure 2 shows a very simple example of a TLM consisting of software modules (embedded in a CPU module) and peripheral hardware modules. Two things should be noted in this figure: the software is shown as a sequence of function calls whereas the hardware is modeled as a set of concurrent hardware blocks. Having sequential software and concurrent hardware is typical for TLMs. Thus, a verification approach has to be able to handle both concepts.

Requirement verification on sequential software is rather straight forward. A call graph clearly shows the dependencies of the security requirements of the software functions. This is not the case if hardware is modeled. Concurrent hardware processes do not provide a call graph as hardware functions usually never terminate. Additionally, hardware exceptions can interrupt the software at any time. This behavior is very difficult to describe in formal models. This is an essential reason why we chose a simulation based approach for our methodology.

Instead of a call graph, a data flow driven approach can be used for hardware verification. Data is passed from one process to another one. This can be done asynchronously - the data producer often has no influence if the receiver consumes the sent data or decides to ignore it. This might depend on the internal state of the receiving hardware module.

In a data flow driven approach data is annotated with SFRs instead of functions or modules. If this data is consumed, the annotated SFRs have to be evaluated by the simulation environment to verify the security capabilities of the module. As shown in the figure, software and hardware interact naturally (sketched as a bus connecting the CPU with the peripheral hardware blocks). As a consequence, SFRs have to be mapped from the call graph to the data flow graph and vice versa.

In our simulation based approach function calls and data generation respectively data consumption is reported to a security verification module. The reporting includes the annotated SFRs of the acting software or hardware modules.

©2010 Springer Verlag. Reprinted, with permission, from Lecture Notes in Computer Science, 2011, Volume 6542/2011

The verification module evaluates the relationship of the reported SFRs and is able to report occurring security violations.

3.3 Verification Rules

Yet we have not discussed the rules our proposed verification module should apply to the reported SFRs. Different sets of rules according to the abstraction level of the TLM are imaginable.

A straight forward approach is a *requires/implements* scenario. Modules that require SFRs are annotated with the SFR's identifier (e.g., a unique number) and a *requires*-flag. Modules that provide the functionality described by the SFR are annotated by the SFR's identifier and an *implements*-flag. The verification module checks if every call graph's node with a *requires*-flag leads to nodes with the according *implements*-flag; respectively if data that has a *requires*-flag is handled in modules that provide the *implements*-flag.

A more sophisticated approach is to annotate software and hardware modules that represent a Security Function and also modules implementing SFRs. The ST can be used to extract the relationship of (1) SFRs required by the Security Functions and (2) SFRs that have dependencies on other SFRs to create verification rules. The transaction level model consists of modules assigned to the software and the hardware domain. It represents the functional behavior and the security behavior of the embedded system. Certain modules might be implemented on different abstraction levels. Communication interfaces, for example, can be modeled on very high abstraction levels as they are usually not security relevant. The software model reports information to the verification module to generate call graphs for Security Functions. Similarly, the verification module generates data flow graphs for data in the hardware model. Call graphs and data flow graphs are verified against the rules extracted from the ST. This is shown in Figure 3.

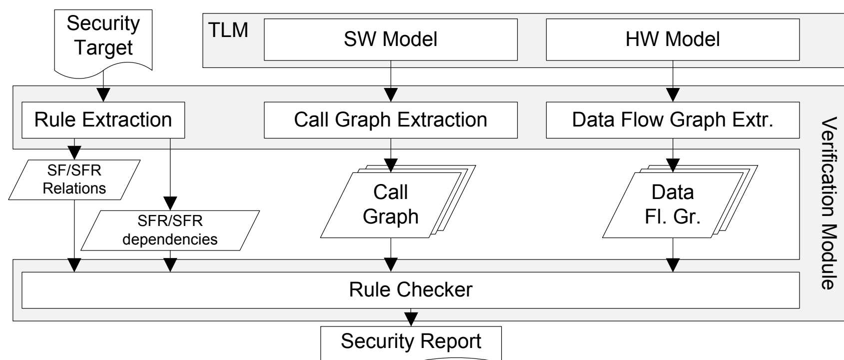


Fig. 3. Verification of rules extracted from a Security Target

©2010 Springer Verlag. Reprinted, with permission, from Lecture Notes in Computer Science, 2011, Volume 6542/2011

4 Proof of Concept Implementation

We implemented a proof of concept verification library for SystemC[2] TLMs. The library provides (1) macros for annotating functions, data, and SystemC processes with SFRs, and (2) the verification module. The macros cause function calls of the static verification module instance. These function-calls cause the verification module to compute the call graphs and data flow graphs and to store the annotated SFRs. The rule set for verification is static and provides checks for *requires*, *implements*, and *supports* flags for SFRs. Modules that *require* SFRs have to utilize modules that *implement* according SFRs. *Supporting* SFRs can be used if interconnecting modules are needed.

So far, we have not evaluated our approach on a use case with realistic complexity. Instead, we first evaluated the general applicability by verification of different smart card STs from different vendors: STMicrosystems ST23YR80A, NXP P5Cx081V1A, Samsung S3CC91A, Infineon SLE66CX680PE, and Fujitsu MB94RS403. These STs are public[3].

In addition we implemented a very small use case example based on some few hardware modules: a CPU, a memory, a DES crypto co-processor, and a CRC co-processor. The software model reads DES key data from the memory and sends it to the DES co-processor. We chose the FDP_SDI.1.1 SFR from the Common Criteria standard [3] - it basically defines, that the DES key has to be integrity protected. The `setDESKey()` function was annotated to *require* FDP_SDI.1.1. Two different implementations to fulfill the security requirement were evaluated: (1) a software implementation calling `checkIntegrity()` within `setDESKey()` and (2) a hardware implementation connecting the DES co-processor with the CRC co-processor.

5 Results and Discussion

Table 1 shows the summarized evaluation results of the STs: the total number of defined Security Functions, the number of selected SFRs, the number of applied SFRs, and the SFRs we think that can be checked automatically by our proposed approach. Notice, that each SFR can be applied on several Security Functions. Thus, the number of applied SFRs can be higher than the number of SFRs. In a strict sense, this number should be even higher than the numbers depicted in Table 1 (third row) because each Security Function consists of several mechanisms. However, the number of Security Mechanisms is depending on the concrete implementation and not given in public STs. Therefore, we took the number of Security Functions as an estimator for our evaluation (we assume that each of them consists of one mechanism in minimum).

26 to 60 SFRs were applied in the selected smart card microprocessors. This should give a feeling about the verification complexity: 26 to 60 applied SFRs means that 26 to 60 times a module has to be verified if it provides the

[2] `www.systemc.org`
[3] `www.commoncriteriaportal.org/products/`

©2010 Springer Verlag. Reprinted, with permission, from Lecture Notes in Computer Science, 2011, Volume 6542/2011

Table 1. Security Target evaluation results

	STMicrosystems	NXP	Samsung	infineon	Fujitsu
Number of Security Functions	10	9	5	9	10
Number of SFRs	15	16	18	21	19
Number of applied SFRs	40	60	26	35	26
Potentially automated check	90%	85%	73%	77%	73%

appropriate SFR. Notice, that the selected STs describe smart card hardware only. If the software has to be verified as well even more Security Functions and SFRs will emerge and the verification effort will be even higher.

We do not expect that all applied SFRs can be verified with our proposed approach. Thus, we checked which used SFRs can be verified by our method. To do so, we evaluated the meaning of used SFRs in the selected STs and checked if we could define rules for our approach that are able to verify the SFRs automatically. Because of the limited space in this publication we can not explain this process in details here. The results are shown in Table 1 in the last row.

We think that 73% to 90% of the used SFRs could be checked with our proposed approach. Accordingly, the verification effort could be reduced by approximately 80%. Of course this is a very optimistic estimation as verification does not only mean to check if a module provides the right SFRs. However, we think that this clearly shows the potential of an automated verification approach.

From our very small case study we can summarize the following results. Without going into the implementation details we can note that the usage of our annotations is very intuitive and fits very well into the iterative design flow of an hw/sw codesign flow. However, we also have to mention that a puristic *requires*/ *implements* rule-set seems not to be sufficient. At least an additional *ignores*-flag was needed to avoid miss-leading incorrect security violations. However, we think that such an *ignores*-flag can come with a high risk of erroneous miss-usage.

In addition, we have to note that the implemented solutions are not identical from a security point of view. If the integrity check is performed in software, sending the sensitive key material to the DES block is unsecured. On the one hand we think that this could be automatically reflected in our verification result (e.g., as the 'distance' of a module that *requires* the SFR to the module that *implements* the SFR). This could help the developer during the design space exploration phase. On the other hand, if needed, there exist certain SFRs that reflect such requirements. FDP_ITT.1.1, for example, requires system module intercommunication to be protected as well.

As a summary we have to conclude that much more use case studies have to be done to extract a rule set that allows a sufficient verification of the SFRs.

6 Conclusion and Future Work

In this work we described a simulation based verification methodology for security requirements in transaction level models of embedded systems. The basic

©2010 Springer Verlag. Reprinted, with permission, from Lecture Notes in Computer Science, 2011, Volume 6542/2011

idea has its roots in the Common Criteria process where Security Functions and according Security Functional Requirements are defined. We showed that our proposed approach is able to verify such requirements by evaluating meta-information in the model during the simulation of its functional behavior. Furthermore, we showed that gained cost savings can be significant.

However, we have to say that our work is in an early stage. More case studies are needed to evaluate the applicability of our methodology. Especially the verification rules have to be refined and their influences on the design have to be evaluated.

References

1. Balarin, F., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A.L.: A formal approach to system level design: metamodels and unified design environments. In: Third ACM and IEEE International Conference on Formal Methods and Models for Co-Design. IEEE, Los Alamitos (2005)
2. Cai, L., Gajski, D.: Transaction level modeling: an overview. In: Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis. ACM, New York (2003)
3. Common Criteria. Common Criteria for Information Technology Security Evaluation - Part 1-3. Version 3.1 Revision 3 Final (July 2009)
4. Deng, Y., Wang, J., Tsai, J.J.P., Beznosov, K.: An approach for modeling and analysis of security system architectures. IEEE Transactions on Knowledge and Data Engineering 15(5), 1099–1119 (2003)
5. Garavel, H., Helmstetter, C., Ponsini, O., Serwe, W.: Verification of an industrial SystemC/TLM model using LOTOS and CADP. In: 7th IEEE/ACM International Conference on Formal Methods and Models for Co-Design. IEEE, Los Alamitos (2009)
6. Kocher, P., Lee, R., McGraw, G., Raghunathan, A.: Security as a new dimension in embedded system design. In: Proceedings of the 41st Annual Design Automation Conference. ACM, New York (2004)
7. Lotz, V., Kessler, V., Walter, G.H.: A formal security model for microprocessor hardware. IEEE Transactions on Software Engineering 26(8), 702–712 (2000)
8. Mellado, D., Fernández-Medina, E., Piattini, M.: A common criteria based security requirements engineering process for the development of secure information systems. Comput. Stand. Interfaces 29(2), 244–253 (2007)
9. Morimoto, S., Shigematsu, S., Goto, Y., Cheng, J.: Formal verification of security specifications with common criteria. In: Proceedings of the 2007 ACM Symposium on Applied Computing. ACM, New York (2007)
10. Schaumont, P., Verbauwhede, I.: Domain-specific codesign for embedded security. Computer 36(4), 68–74 (2003)
11. Trummer, C., Kirchsteiger, C.M., Steger, C., Weiss, R., Pistauer, M., Dalton, D.: Automated simulation-based verification of power requirements for systems-on-chips. In: 13th International Symposium on Design and Diagnostics of Electronic Circuits and Systems. IEEE, Los Alamitos (2010)

©2010 Springer Verlag. Reprinted, with permission, from Lecture Notes in Computer Science, 2011, Volume 6542/2011

TOWARDS FORMAL SYSTEM-LEVEL VERIFICATION OF SECURITY REQUIREMENTS DURING HARDWARE/SOFTWARE CODESIGN

Johannes Loinig[1], Christian Steger[1], Reinhold Weiss[1], and Ernst Haselsteiner[2]

[1] Institute for Technical Informatics
Graz University of Technology, Graz, Austria

[2] NXP Semiconductors Austria GmbH
Gratkorn, Austria

ABSTRACT

Today's embedded systems have to fulfill sophisticated security requirements. They have a deep impact into the system's design. Security can only be achieved if security requirements are considered during the system-wide design and development process. In this concept work[1] we propose a hardware/software codesign approach that addresses functional security requirements during all design and development phases of a system.

I. INTRODUCTION

The authors of [1] state that securing a system requires more than adding single secure modules like cryptographic processors. Security requirements have to be implemented in an organized way. For secure embedded systems this is crucial. Different security verification approaches exist but suffer from too high effort for complex systems. Nevertheless, security critical products have to be security evaluated. Usually, this is done in a documentation-centric approach which suffers form a missing link between the security specification and its implementation. Today, this linking is done with manually written design documents.

In this work we propose a more implementation-centric approach considering security requirements in a hardware/software codesign process. It allows adding security relevant meta-information, representing security requirements, to a system's model. These annotations can be verified during all development phases.

II. RELATED WORK

The authors of [1] describe a security pyramid containing different levels of security from protocol-level down to circuit-level implementations. This clearly shows that security cannot be implemented on one certain level of a system. Implementing security requires a system-wide approach like a hardware/software codesign process where the system's functionality is modeled independently from the final implementation.

Hardware/Software Codesign (HSC) is a well known process for development of hardware and software in parallel [9]. Typical benefits of a HSC are shorter development cycles, less expenses and better system performance. In our proposal we show how security verification fits into a HSC process for high secure embedded systems.

Common Criteria (CC) defines a process for verification of security relevant IT systems [8]. CC is the de-facto standard for security certification of IT products. Security Functional Requirements (SFRs) are defined in the CC standard and applied in a Protection Profile (PP) in a very general way. These requirements are specified in detail in a Security Target (ST) for one certain product. A CC certificate claims that all SFRs in an ST are implemented. The CC verification process is documentation centric as shown in Fig. 1. Design documents describe the implemented SFRs. A CC evaluator first checks the consistency of the documents. Then, these documents are the basis of a manual verification of the system's implementation. In comparison to that, we propose an implementation centric verification methodology where SFRs are included as meta-information into the system's source code.

Formal verification in a codesign approach is explained in [2, 3, 4], and many other publications. Two different concepts are used: stateless model checking and translation based approaches. Stateless model checking suffers from limited scheduling capabilities of modeling languages [4]. An abstract meta-model is used in [2] and [3] to describe the functional behavior and architectural model of the system separately. A special meta-model language has to be used to do so. However, the gen-

[1]This paper is a result of a project which is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology (contract FFG 816464).

978-1-4244-6683-2/10/$26.00 ©2010 IEEE

©2010 IEEE. Reprinted, with permission, from Proceedings of 2010 IEEE International SOC Conference (SOCC)

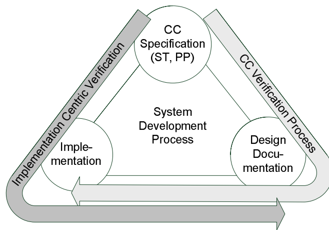


Figure 1: The CC verification process is documentation centric. The implementation is verified against a set of CC documents.

erated meta-models are far away from a concrete implementation which can be used for the product. Translation of the model into a verification language is explained in [4]. The state space explosion problem restricts this approach to a few hundred lines of code. In addition, the verified model is not the model that was developed and executed.

Annotated source code was already proposed for security verification in [11]. However, this was used for security verification of embedded software.

Smart cards are very small embedded systems that have to provide a highly secure environment for use cases like e-government and banking. Typical hardware platforms of smart cards are still based on 8-bit microprocessors. However, modern smart cards have to provide very complex functionalities. A typical example is the Java Card [5], a smart card which provides a Java runtime environment. A huge set of security requirements have to be provided by such a system [6, 7]. The verification of the security requirements is a non-trivial task. Today, for most Java Card products a CC certificate is needed to guarantee the necessary security for the customer's use cases.

III. VERIFICATION OF SECURITY REQUIREMENTS DURING CODESIGN

The basic concept of our proposed secure HSC (SHSC) is that SFRs are included in the model as meta-information called annotations (as described later). These annotations state where in the system-level model SFRs are required respectively implemented. This allows consideration of SFRs from the beginning of the development process and a verification of the SFRs' propagation in the system.

Furthermore, SHSC includes an additional abstraction layer – the secure functional level (SFL) which includes the first model in the development process that implements SFRs.

A. A Novel Security Codesign Approach

Figure 2 shows the details of the topmost SHSC levels: the functional level (FL) and the secure functional level. More detailed abstraction levels and their role in the SHSC are very similar to the regular HSC and are out of scope of this paper.

The starting points of the SHSC are separate functional requirement (FR) specifications and SFR specifications. This separation is done because the SFRs are usually defined independently form the FRs in a CC ST, as explained before.

The FRs are modeled on the functional level model (FLM). This results in a very abstract behavioral model that primary serves simulation-based evaluation of the system's use cases. As a next step, modules of the FLM are annotated according to SFRs that are related to the modules. These security annotations (SAs) do not change the functional behavior of the FLM. We will explain below how SAs can be used to verify the security of the system. The result of this SHSC step is a security annotated functional model (SAFM) which is the starting point for the SFL.

In the SFL all SFRs get implemented on a very abstract level. The result is a complete model of the system including early implementations of the SFRs, the secure functional model (SFM). The SFM is used in further stages of the SHSC process like partitioning and mapping to hardware platform components. SAs can be used similarly at models in these abstraction levels. However, they are not topic of this paper.

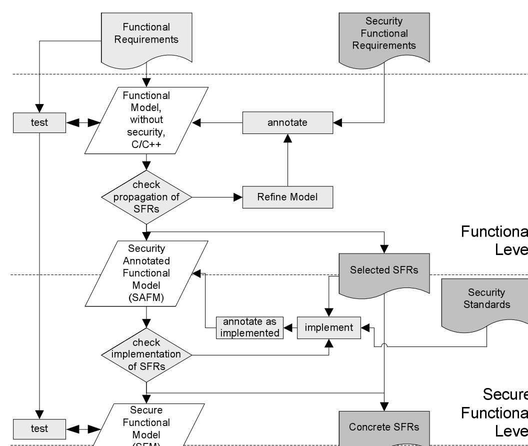


Figure 2: The Functional Level and Secure Functional Level of the proposed SHSC

©2010 IEEE. Reprinted, with permission, from Proceedings of 2010 IEEE International SOC Conference (SOCC)

B. Security Annotations

Security annotations are syntactic meta-information included into the source code of the model. SAs do not change the functional behavior of the model but they define the expected/required security of modules on system-level.

Each SA is related to exactly one SFR and thus defines that the annotated module has to consider the SFR in its implementation. How this implementation is done is not defined by the SA. The SA is not used to verify the exact implementation of the SFR. In early design phases a detailed verification of the implementation is not necessary as the model is still very abstract.

The reason for using SAs in the FL is to evaluate the propagation of SFRs in the model. If a module has to provide an SFR, sub-modules and interacting modules may be affected by the SFR as well. During the refinement process in the FL the model is split into smaller modules. An automated SA verification process allows identification of modules that also have to be aware of the related SFR. As the model is still on a high abstraction level the SA verification is rather simple and can be done on source code basis or simulation basis.

C. The Secure Functional Level

The SFL is the first level where SFRs are implemented. This allows consideration of security relevant implementation details in early development phases.

As the SFM includes now the implementation of SFRs, more detailed verification is possible. To do so an SA includes more details about the SFR. Different types of annotations are used: *requires*, *implements*, *supports*, *fulfills*, and *expects*. Thus, when the FL is refined to the SFL, the SAs have to be refined as well to fit to the abstraction level of the SFM. We define following types for SAs in the SFL that are also depicted in an example in Fig. 3.

Requires/Implements SFR: the annotated module requires the SFR but does not implement it. All modules accessed by the annotated module must implement the SFR.

Fulfills/Expects SFR: if passive modules (like variables), have to provide an SFR but are accessed by modules instead of accessing other modules on their own, requires/implements is not applicable. In such a case the passive module expects that an active module fulfills the SFR.

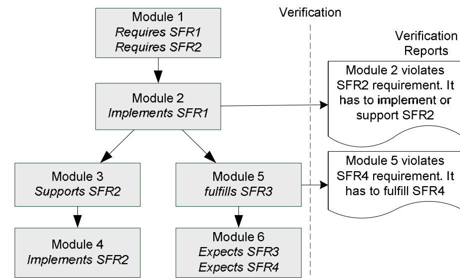


Figure 3: A simple example of SA types in the SFL.

Supports SFR: if a module is accessed via one or more other modules, these modules have to support the annotated SFR (requires or expects). This means that the modules do not violate security conditions related to the according SFR.

The refinement process on the SFL will, again, cause to split existing modules into smaller parts. Similar to the FL, the verification of the SAs ensures a consistent propagation of the SFRs in the system. Furthermore, the verification of the SA-types ensures a valid system-wide integration of all SFR implementations.

IV. IMPLEMENTATION

The codesign development process is a very iterative. This requires that an implementation of our SHSC approach must be very flexible. Utilization of SAs must not cause big overhead during implementation or evaluation of the system. In addition, the implementation has to ensure that accessed annotated modules must not be used, by mistake, without respect to the SA. In practice this means, that a developer must not have to care about, if a module was annotated before or not.

We decided to use SystemC [10] to implement our SHSC. SystemC allows abstract modeling in C++ which is a good starting point for a functional model. Unfortunately, SystemC does not provide any mechanism for annotations. Thus, it was necessary to define a concept for annotations in C++.

As the SA verification should not slow down the refinement process of a codesign approach significantly, we decided to go for simulation based verification. A formal approach would be to translate the model into a language which can be used for pure formal verification. However, we think that, especially for complex systems, this translation is not applicable for an iterative design process. Indeed, a formal verification would provide

©2010 IEEE. Reprinted, with permission, from Proceedings of 2010 IEEE International SOC Conference (SOCC)

better verification results but especially in very early development phases a rapid check would be of better benefit than a complete but slow verification.

Fig. 4 shows the logical concept of a simulation based verification approach. If an annotated module is used, the simulation environment reports that usage via a model-to-checker interface (MCIF) to an SFR checker. Hence, the checker is notified which SAs are applied to the module.

We implemented a set of C++ preprocessor macros that allow annotation of modules. When applying an annotation, the developer replaces the module definition by the macro. Additional macros to access the module are defined because SAs have to be reported to the MCIF at every module usage. To be sure that the developer is not using the module without these functions by mistake, the original module is hidden by renaming.

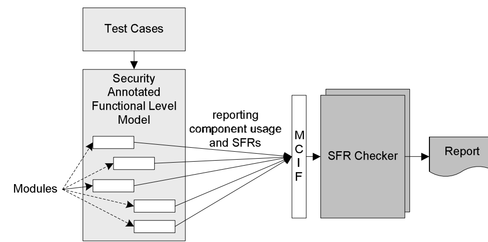


Figure 4: A logical view on our simulation based verification.

V. DISCUSSION

So far we only tried the HSCD on very small examples but did not apply it on a system with realistic complexity. However, the overhead of the SA-reporting mechanism does not cause serious computational overhead. The quality of the verification results depends on the SFR checker implementation which is out of scope of this work. A first SA-stack-based checker prototype was implemented. It is able to verify described SAs on function-level and variable-level. For the FL and SFL this seems to be sufficient, however for a model including hardware modules (and thus concurrent processes) a stack-based approach will not be applicable anymore.

The proposed preprocessor based annotations are very intuitive to use, easy to apply, and efficient to evaluate. No special knowledge about verification methodologies is required for the developer who works on the model. SystemC allows a very fast simulation based verification of the FRs and the SFRs. Thus, we can argue that our proposed methodology is applicable for high-level security verification of embedded systems.

VI. CONCLUSION

In this concept proposal we have shown a methodology for semi-formal verification of functional security requirements in an embedded system. The methodology is included in a secure hardware/software codesign approach. Security requirements are considered and verified from the beginning of the development phases. The concept allows refinement of the security requirements with the remaining system to achieve best system security and performance.

Furthermore, we have shown how an implementation of our concept can be done for SytemC. This implementation is based on preprocessor macros which allow an easy application and verification of security annotations and fast and flexible refinement of the system.

REFERENCES

1. Schaumont P. and Verbauwhede I, "Domain-specific codesign for embedded security", *Computer*, IEEE, Volume 36, pp.68-74, 2003.
2. Sangiovanni-Vincentelli A. and Martin G. "Platform-based design and software design methodology for embedded systems", *Design & Test of Computers*, IEEE, Volume 18, pp.23-33, 2001.
3. Balarin F., Passerone R., Pinto A., and Sangiovanni-Vincentelli A. "A formal approach to system level design: metamodels and unified design environments", *MEMOCODE 2005*, pp.155-163.
4. Garavel H., Helmstetter C., Ponsini O., and Serwe W. "Verification of an industrial SystemC/TLM model using LOTOS and CADP", *MEMOCODE 2009*, pp.46-55.
5. Sun Microsystems, Inc., "Java Card Platform Specification 2.2.2", 2006.
6. Sun Microsystems, Inc., "Java Card™ Platform Security", 2001.
7. Sun Microsystems, Inc., "Java Card™ Protection Profile Collection", Version 1.1, 2006.
8. Common Criteria, "Common Criteria for Information Technology Security Evaluation - Part 1-3", Version 3.1, 2009.
9. Gupta P. "Hardware-software codesign", *Potentials*, IEEE, Volume 20, pp.31-32, 2001.
10. IEEE Computer Society, "Open SystemC Language Reference Manual IEEE Std 1666™-2005", 2006.
11. Loinig J., Steger C, Weiss R., and Haselsteiner E., "Identification and Verification of Security Relevant Functions in Embedded Systems Based on Source Code Annotations and Assertions", WISTP 2010, pp.316-323

©2010 IEEE. Reprinted, with permission, from Proceedings of 2010 IEEE International SOC Conference (SOCC)

2010 Fifth International Conference on Systems

Performance Improvement and Energy Saving based on Increasing Locality of Persistent Data in Embedded Systems

Johannes Loinig, Philipp Maria Glatz, Christian Steger, and Reinhold Weiss
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
Email: {johannes.loinig, philipp.glatz, steger, rweiss}@tugraz.at

***Abstract*—The performance of embedded systems often suffers from strict cost/size and power/energy limitations. This is especially important for very small systems that are sold in a high number of items like smart cards and RFID controllers.**

On one hand, small hardware platforms are developed to minimize cost and power consumption. On the other hand, applications have to fulfill strict performance demands.

We[1] introduce a high-level performance optimization for EEPROM-based embedded systems. Writing to persistent memory is known to be very costly in the meaning of time and energy consumption. Our proposal shows how to reduce the number of write operations. We introduce an approach to increase spatial data locality without losing reliability of data that has to be stored persistently.

***Keywords*-Persistent Memory; Memory Management; Performance Improvement; Energy Consumption Improvement; Data Locality**

I. Introduction

Even very small embedded systems (ES) are widely used today. ESs have either a battery based power supply, which restricts the energy consumption for a given endurance, or they even do not have an own power supply. Latter ones are completely powered by an external device like a contactless smart card reader. This restricts the maximum power drain of the system. A second limitation is the price of the ES - especially if a high number of sold systems is aimed.

As a consequence, the hardware complexity is reduced as far as possible. This has significant negative effects on the performance achieved. However, even very small embedded systems have to meet strict performance limits.

Due to the lack of an own power supply or due to energy saving reasons, ESs are frequently switched off between operation cycles. This requires that data which must not get lost has to be stored in persistent memory. Writing into persistent memory is costly in the meaning of time and energy consumption. Thus, minimizing persistent write operations will reduce energy consumption and increase the performance of the system.

Our approach is based on combining consecutive write operations to an electrically erasable programmable read-only memory (EEPROM) to fewer write operations. This is done by increasing the spatial data locality based on static analysis of the system. EEPROMs are organized in pages. All data on one page can be written with one write cycle only. Thus, the basic idea is to organize selected persistent data segments in a way so that they are located on a smallest possible number of EEPROM pages.

First, we discuss related work and motivate the approach proposed. Next, we explain the three basic characteristics of embedded systems which are important for our approach: spatial data locality, temporal data locality, and atomicity. Based on these characteristics we then introduce our method to increase spatial data locality. Next, we explain an example implementation on the basis of a Java Card and evaluate our proposed approach in terms of needed EEPROM write cycles. Finally, we present our results and conclude the work.

II. Related Work and Motivation

A basic approach for saving costly write cycles is to buffer values in a random access memory (RAM) and only write them to the EEPROM if needed. This may be implemented in software or in the EEPROM hardware. The method described by Choi et al. in [1] proposes a software implementation of a buffer when taking advantage of the high locality of Java class fields. Two buffers in RAM increase the performance when writing data into EEPROM. However, if the system cannot fully rely on the power supply, this is not an appropriate solution. The system must provide a store operation which ensures data consistency even in case of sudden power loss. Typical logging approaches for smart cards are described in [2], but they are general enough to be used in other ESs as well. A common mechanism is *Old Value Logging* (OVL), where the old data is backed up in a persistent buffer before being overwritten by new data. The mechanism requires multiple write cycles to the EEPROM and is thus not very performance and power efficient.

The idea of improving the memory layout to gain better performance is not new and is described in [3] for embedded Java. Cache misses are analyzed dynamically to gather information for a better performance of accessing multidimensional arrays. Furthermore, Kim et al. propose two approaches for better memory performance of embedded Java applications in [4]. First, frequently used objects are allocated in local memory instead of external memory. Second, for achieving better caching results, the garbage

[1]This paper is a result of the *HiPerSec* project which is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the contract FFG 816464.

978-0-7695-3980-5/10 $26.00 © 2010 IEEE
DOI 10.1109/ICONS.2010.37

IEEE computer society

©2010 IEEE. Reprinted, with permission, from Proceedings of 2010 Fifth International Conference on Systems (ICONS)

collector moves objects during the compaction phase. Both approaches use a static study based on tracing executed bytecodes of the application. Utilizing locality-awareness for persistent memory was also described before by Lee et al. in [5] for NAND flash-memory disks in general purpose computers. A *Flash-Translation-Layer* was introduced that exploits sequential locality and temporal locality of the file accesses.

Motivation of our Proposed Approach: Dynamic methods are hardly feasible for very small ESs: first, the dynamic overhead is too high and second, because there is too few space in RAM to store the intermediate results. Additionally, very often the application does not run long enough to extract sufficient information. For example, if the memory access patterns for persistent memory should be analyzed dynamically, the application has to run at least once to gather information. For taking advantage of the results either the application has to be executed again before shutting down the system, or the results have to be stored persistently. This is in contradiction to our goal to reduce costly write operations.

On very small ESs based on Java, running the garbage collector (GC) automatically is avoided because of the nondeterministic performance side-effects. Therefore, Java Cards, for example, provide a GC, but it is only executed, if the application explicitly requests for it [6]. Moving persistent objects during the GC's compaction phase is very time and energy consuming, thus it is very unlikely that such an approach would cope with our goal.

Our proposal is based on increasing the spatial locality of data fields in EEPROM when they are allocated. We will show that the needed information can be easily extracted (e.g., from the source code) before compilation of the software. The proposed approach can be applied on ESs to improve the performance and energy consumption significantly.

III. Concept and Approach

Our concept is based on three characteristics of embedded software: the *spatial data locality*, the *temporal data locality*, and *atomicity*. Next we will explain these characteristics with simple source code examples of embedded Java.

A. Data Characteristics

Spatial Data Locality: The basic principle of our concept is that EEPROMs are organized in pages. All data located on one page can be written with only one write cycle. Thus, to save costly write cycles it is desirable to have the data to be written on only one page - in other words it is desirable to have a high spatial data locality (SDL). The SDL of a system strongly depends on its implementation. The locality of certain allocated variables can be influenced by the developer and compiler. In object oriented systems fields within objects have shown to have a high SDL because they are allocated 'at once' within one data block representing the class instance. This may not hold for static class fields, because they are not part of one certain class instance and are thus allocated in a separate data area for static fields. The SDL of multiple objects depends on the implementation of the object allocation mechanism of the underlying operating system (OS).

Listing 1 A simple Java code example to explain data locality.

```
1: Object a = new PersistentObject();
2: Object b = new PersistentObject();
3: // ...
4: Object z = new PersistentObject();
5:
6: a.field1 = 0x01;
7: a.field2 = calculateSomething();
8: z.field1 = 0x03;
```

Temporal Data Locality: It is unrealistic to have all potentially written persistent data on one and the same page. Small platforms provide EEPROMs with a page size of 64 or 128 bytes only. Embedded Java object instances have a typical size of 10-20 bytes. For taking advantage of the spatial data locality, it is necessary that the write operations are temporally close to each other, which means a high temporal data locality (TDL). In case of copying arrays, for example, this may be obvious but it is not that clear for object oriented assignments to class fields as shown in Listing 1. The data written in Lines 6, 7, and 8 have a high TDL as the assignments are executed consecutively. However, only the data written in Line 6 and 7 may have a high SDL as the written fields are located in the same object. For Line 8 we do not know this, because object `z` might be allocated on a different EEPROM page. The TDL depends on the program flow of the application. It may change during different execution phases.

Atomicity: So far, we have not yet considered that an unexpected power loss might happen at any point in time during execution. An application developer may expect that the assignment in Line 6 of Listing 1 was completely and correctly executed even if the power supply drops during execution of Line 7. Even if `a.field1` and `a.field2` have a high TDL and SDL, it must not be assumed that the separate write operations can be combined to save time and energy. Notice that the method `calculateSomething()` in Line 7 may take a while until it returns. The longer the method takes, the higher the risk for data inconsistencies will be, if write cycles are combined. Therefore, for not violating atomicity requirements, arbitrary write operations with high TDL and/or SDL can not be used for combining write operations.

B. Increasing Spatial Locality

Our proposed approach is to find and group persistent write operations that do not violate atomicity requirements. The basis is a static analysis of the system. It is shown in Figure 1. A loop searches for potential groups of write

©2010 IEEE. Reprinted, with permission, from Proceedings of 2010 Fifth International Conference on Systems (ICONS)

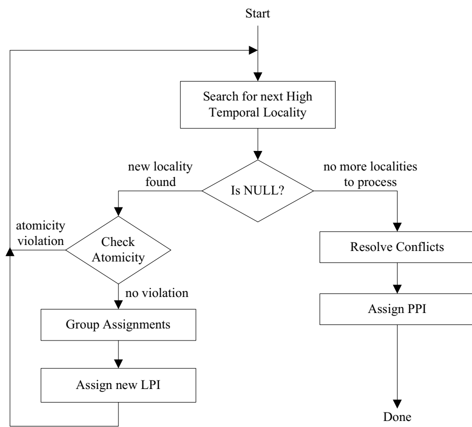


Figure 1. The algorithm of our proposed approach.

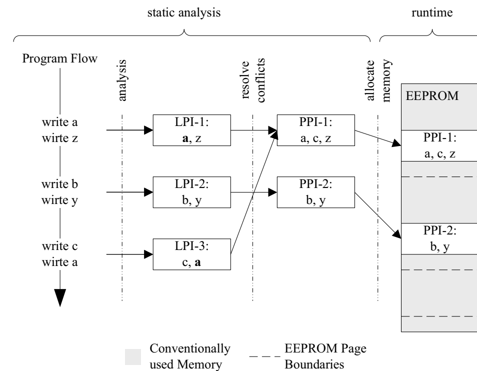


Figure 2. Mapping assignments to LPIs, PPIs, and physical EEPROM pages.

operations. Each finding is checked if a merge would cause any atomicity violations. If not, the group is set up and assigned to a *Logical Page Identifier* (LPI). LPIs are mapped to *Physical Page Identifiers* (PPI) - it would be a trivial mapping to use one PPI for each LPI. However, it might happen that a written data variable is included in more than one LPI. Such conflicts have to be resolved before the final PPIs can be assigned. As a consequence more than one LPI might be mapped to one single PPI as shown in Figure 2.

Notice that a PPI doesn't necessarily have to be a concrete physical page in the EEPROM. The concrete physical page can be selected by the OS during runtime. In such a case the PPI can be seen as an annotation for data fields. It provides information about which data fields should be allocated on the same EEPROM page - independent from the page which will be chosen by the OS.

The static analysis can be done on different representative data sets, for example the source code of the application and/or memory traces. Usually dynamic persistent data is not allocated with levity on small ESs as the cost for allocation and deallocation is very high and the OS cannot ensure that there is enough persistent memory left when the system operates in the field. The latter is especially critical if the system's end user cannot easily react to failures that may happen during runtime (e.g., for wireless sensor nodes or smart cards). Therefore an embedded application allocates the memory needed during installation when the system is in an environment which can react to potential failures.

Our approach benefits from these quasi-static data, because they are easier to handle. However, real dynamic allocations can also be handled if the system can reserve physical EEPROM pages for future allocated data that was identified to be in the same PPI. If the OS provides a garbage collector it must be ensured that the spatial locality achieved remains after the compaction phase.

For primitive data values (byte, short, ...) an assignment to an LPI is trivial. The situation is more complex for bigger blocks of data like whole objects or arrays. Not all data within a big data block may be changed and, even worse, different parts of one data block may be assigned to different LPIs. In theory an arbitrary high granularity is obtainable by our method but the implementation specific overhead of splitting data blocks in parts and dereferencing them in different EEPROM pages may overcome the achieved time and energy savings.

IV. Implementation

We have chosen an implementation of a Java Card [6] to evaluate our approach. Java Cards are small embedded systems (smart cards) including a Java virtual machine.

Java Cards include a 'natural' group of assignments that do not violate atomicity requirements if combined: transactions. All other assignments to persistent data fields are atomic and therefore cannot be combined. Transactions like shown in Listing 2 are a group of assignments where either all of them are executed completely and correctly or none of them - so they can be seen as one atomic assignment.

Listing 2 A simple code example using Java Card transactions.

```
1: JCSystem.beginTransaction();
2: a.field1 = 0x01;
3: a.field2 = calculateSomething();
4: z.field1 = 0x03;
5: JCSystem.commitTransaction();
```

A. Java Card Transactions with Old Value Logging

We have chosen the Old Value Logging because it is a common way to implement the logging mechanism for Java

©2010 IEEE. Reprinted, with permission, from Proceedings of 2010 Fifth International Conference on Systems (ICONS)

Card transactions. We will now discuss the OVL's cost in terms of numbers of EEPROM write cycles needed.

As described in [2], OVL requires the steps shown in Listing 3 to be executed. Each described step requires costly EEPROM write cycles. For each assignment, at least three of them are needed: First, the backup of the old data is written to the transaction buffer (TB). Second, the TB becomes validated to signal that after a potential card reset the data in the TB must be restored. Finally, the old data is replaced by the new values. When the transaction ends, the TB becomes invalidated.

Listing 3 Basic steps of the OVL mechanism.

```
1: // After JCSystem.beginTransaction()
2: for All assignments in the transaction do
3:     writeToTransactionBuffer(oldData)
4:     validateTransactionBuffer()
5:     writeToDataArea(newData)
6: end for
7:
8: // After JCSystem.commitTransaction()
9: invalidateTransactionBuffer()
```

How many write operations are actually needed strongly depends on:

- n: the number of assignments within the transaction.
- s^{TB}: the size of each backup written to the TB.
- s^{NEW}: the size of the new data.
- p: the EEPROM page size.

More implementation specific parameters are the location of the transaction buffer in the EEPROM, the address length of data stored in the persistent memory and the location of the data to be overwritten, but they are out of scope of this work. Therefore, later presented equations can be interpreted as a lower boundary for the number of write operations needed.

B. Java Card Code Analysis and Annotation

Java Card's transactions always begin with the command `JCSystem.beginTransaction()` and end with the command `JCSystem.commitTransaction()`. Therefore, the source code of the Java application can be used for our static analysis. Java is a fully object oriented programming language. Each data field is stored somehow in a class (static fields) or object instance (non-static fields). Local function variables are allocated on the Java stack in RAM and thus are out of interest for this work. As a result of the analysis process, complete object instances or certain object fields which are modified within one transaction are assigned to one LPI. Because transactions are atomic as a whole, there is no need to check for violations of atomicity.

Class fields can be provided with *Runtime Retention Java Annotations* [7] including the PPI. Thus the PPI is stored within the field definition in the Java executable and is evaluated by the virtual machine (VM) during runtime. The VM allocates the affected object or field on a physical page related to the PPI. This concept for Java Cards is shown in Figure 3. Java source files are searched for transactions. LPIs and PPIs are extracted and used to create new annotated Java source files. Finally, these files are compiled. This is done off-card during development of the Java Card application. After installation the annotations are evaluated during runtime on the smart card (on-card).

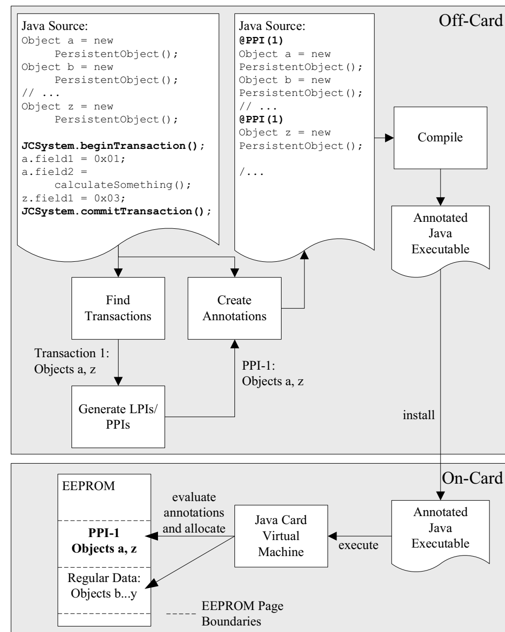


Figure 3. An example of our proposed Java Card work flow.

C. Proof of Concept Example Implementation

We implemented our proposed mechanism on a memory accurate Java Card simulation including a fully implemented Java Card OS. Class fields can be annotated in the source file of the Java Card application. If so, they are allocated on a reserved page in the EEPROM. This increases the SDL of the selected fields. So far, we have not implemented the annotation of entire objects or arrays.

The OVL in the Java Card OS is modified so that data is not written to persistent memory immediately but not until the `commitTransaction()` call. A similar buffered logging mechanism was already mentioned in [8]. For that purpose a very small cache (some few bytes) in RAM is used to hold the data in the meantime. The implementation fulfills all atomicity requirements of the Java Card specification.

We chose the *JavaPurse* example implementation of an electronic purse, which can be downloaded from the Java Card website [9], as a test application. For our observation

©2010 IEEE. Reprinted, with permission, from Proceedings of 2010 Fifth International Conference on Systems (ICONS)

only the code part listed in Listing 4 is of importance. Multiple assignments are executed within one single transaction. In Line 3, a short value is written, Lines 5 and 6 modify byte arrays. Finally, Line 7 calls a method which again includes field assignments in a different object instance. We annotated the field `TN` and two fields written in `updateNewLogRecord()` to be allocated at one and the same physical EEPROM page.

Listing 4 A code part of the JavaPurse example implementation.

```
1: // The following few steps have to be performed atomically!
2: JCSystem.beginTransaction();
3: TN = transientShorts[TN_IX];
4: // Update balance
5: Util.setShort(balancesRecord, START, newBalance);
6: Util.arrayCopy(buffer, START, theRecord, START,
   TRANSACTION_RECORD_LENGTH);
7: transactionLogFile.updateNewLogRecord();
8: JCSystem.commitTransaction();
```

V. Evaluation

In general, the number of write cycles C_{OVL} for the OVL can be represented as shown in Equation 1. The first summand represents the writing of the backups to the transaction buffer, the second one the TB's validation, the third one the writing of the new data, and the constant $+1$ is the final invalidation of the TB.

$$C_{OVL} = \sum_{i}^{n} \left\lceil \frac{s_i^{TB}}{p} \right\rceil + n + \sum_{i}^{n} \left\lceil \frac{s_i^{NEW}}{p} \right\rceil + 1 \quad (1)$$

As the summands are related to different phases of the OVL, we will not summarize them in the following equations. To partially simplify the equation, let's assume that the size of the new data S^{NEW}, and thus also the size of the backup S^{TB}, is constant for all assignments. This is shown in Equation 2.

$$C'_{OVL} = n \left\lceil \frac{s^{TB}}{p} \right\rceil + n + n \left\lceil \frac{s^{NEW}}{p} \right\rceil + 1 \quad (2)$$

As described before, the transaction buffer and the original data area are both written alternating for each assignment in the transaction. As both data areas usually have a very poor SDL we have to separate their write operations before we can apply our proposed approach. For example, the backups can be buffered in the RAM before writing them to the TB. Equation 2 changes to Equation 3 where f_b is an implementation dependent function for writing the values to the TB more efficiently.

$$C''_{OVL} = f_b(n, s^{TB}) + n \left\lceil \frac{s^{NEW}}{p} \right\rceil + 1 \quad (3)$$

What remains costly is the term representing write operations of the new data values. As the TB naturally has a high spatial data locality, the newly written data usually has not.

After applying our approach each transaction is located on pages with a maximum of SDL. Equation 3 changes to Equation 4. Because in common applications s^{NEW} and n are rather small (some few bytes per assignment and some few assignments per transaction), we can further assume that $n\frac{s^{NEW}}{p} < 1$. This means that most of the transactions will only need one constant write cycle for all new values.

$$C'''_{OVL} = f_b(n, s^{TB}) + \left\lceil n \frac{s^{NEW}}{p} \right\rceil + 1 \quad (4)$$

VI. Experimental Results

Figure 4 shows a qualitative comparison of the write cycles needed for the OVL mechanism, an ideally buffered OVL mechanism, and our approach with increased SDL for transactions. These results are deduced from the equations described above. The data size is assumed to be equal for the backup into the TB and the new values. As can be seen, the number of write cycles needed for the OVL is strictly monotonic increasing with the number of assignments in a transaction and the size of the written data.

This does not hold for the two optimized variants of the anti-tearing mechanism. The buffered OVL does not require writing the TB with every new assignment within a transaction. Only, if the buffer in RAM is full, the TB has to be modified. However, in such a case, several write cycles for the new data values might be needed.

In contrast to that, with a high SDL, not only the number of write operations to the TB but also the write operations for new values can be reduced. Therefore, the characteristic in Figure 4 becomes more flat. In comparison to the buffered OVL our approach reduces the number of write cycles by 20% to 50% for $2 \leq n \leq 5$, $10 \leq s \leq 20$, and $p = 64$ bytes.

We expect that our approach results in unused data areas, because physical EEPROM pages are reserved for PPIs which do not require a full page. LPIs may be mapped to PPIs in a more sophisticated and memory-saving way. A discussion and evaluation of such mapping algorithms and their memory consumption overhead is out of scope of this work.

A. JavaPurse Simulation Results

One payment with JavaPurse is spread over 4 commands (APDUs) sent to the Java Card. Only the last APDU causes EEPROM write operations (saving the new balance and some logging information). In total, our simulation reported 16 EEPROM write cycles with the unchanged OVL mechanism. Our implemented mechanism reduces this to only 10 write cycles. This is a reduction of 37.5% of costly write operations.

The execution time of the APDUs is shown in Figure 5. The precise execution time is strongly dependent on the

©2010 IEEE. Reprinted, with permission, from Proceedings of 2010 Fifth International Conference on Systems (ICONS)

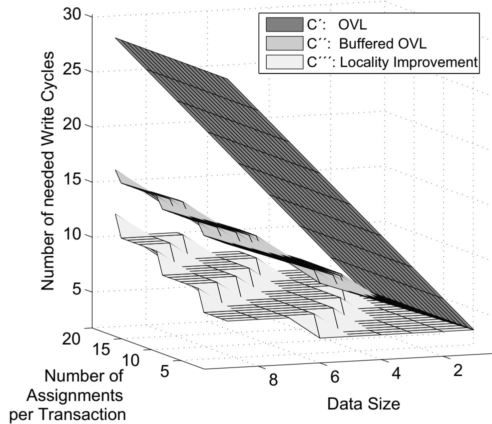


Figure 4. A qualitative comparison of needed write operations.

choice of the hardware platform and its configuration. Therefore, we normalized the measurement results to the total execution time of the not optimized system. As only the 4th APDU causes EEPROM write cycles, the execution time of APDU 1 to APDU 3 stay unchanged. However, the 4th APDU causes the longest execution time and is thus a worthwhile target for optimizations.

Due to our proposed mechanism the execution time of APDU 4 is reduced considerably by 33.8%. Even including the unchanged APDUs, the total execution time of the payment process is reduced by 17.2%. This was basically achieved just by adding some bytes of cache in RAM and increasing the SPL.

Reducing EEPROM write cycles has direct impact to the energy consumption. However, for Java Cards the total energy consumption is not essential and thus not considered in our measurements.

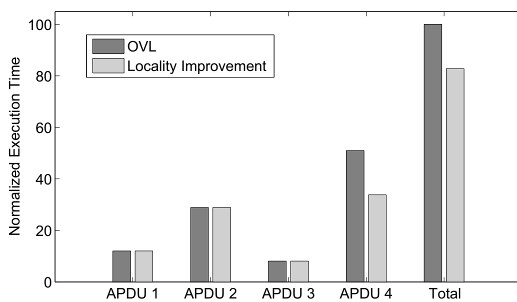


Figure 5. Execution time for the JavaPurse application.

VII. Conclusion and Future Work

Write cycles to persistent memory in small embedded systems are necessary but costly in terms of power and time consumption. Properties that have to be taken into account when the number of write cycles becomes reduced are being discussed. Furthermore, we have proposed an approach to increase spatial data locality depending on given temporal data locality. Finally, we have evaluated this method for Java Cards. We have shown that costly write cycles can be reduced considerably by just increasing spatial data locality for fields that are modified within transactions. However our approach is general enough to be used for other embedded applications and operating systems as well.

Ongoing work is to investigate in the granularity in which data is assigned to LPIs and in the overhead of memory holes which are a consequence of PPIs that are smaller than physical EEPROM pages.

References

[1] W.-H. Choi, H.-Y. Jeon, R. Rosholt, G. Jung, and M.-S. Jung, "A Novel Buffer Cache Scheme Using Java Card Object with High Locality for Efficient Java Card Applications," *Advances in Hybrid Information Technology*, vol. Volume 4413, pp. 500–510, 2007.

[2] M. Oestreicher, "Transactions in Java Card," in *ACSAC '99: Proceedings of the 15th Annual Computer Security Applications Conference*. Washington, DC, USA: IEEE Computer Society, 1999, p. 291.

[3] F. Li, P. Agrawal, G. Eberhardt, E. Manavoglu, S. Ugurel, and M. Kandemir, "Improving memory performance of embedded Java applications by dynamic layout modifications," April 2004.

[4] S. Kim, S. Tomar, N. Vijaykrishnan, M. Kandemir, and M. Irwin, "Energy-efficient Java execution using local memory and object co-location," *Computers and Digital Techniques, IEE Proceedings*, vol. 151, no. 1, pp. 33–42, Jan. 2004.

[5] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, "LAST: locality-aware sector translation for NAND flash memory-based storage systems," New York, NY, USA, 2008.

[6] *Java Card Platform Specification 2.2.2*, Sun Microsystems, Inc., March 2006. [Online]. Available: http://java.sun.com/javacard/specs.html

[7] M. D. Ernst, "Type Annotations Specification (JSR 308)," November 2008. [Online]. Available: http://groups.csail.mit.edu/pag/jsr308/

[8] J. Loinig, C. Steger, R. Weiss, and E. Haselsteiner, "Java Card Performance Optimization of Secure Transaction Atomicity Based on Increasing the Class Field Locality," in *Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on*, July 2009, pp. 342–347.

[9] "Java Card Website." [Online]. Available: http://java.sun.com/javacard/

©2010 IEEE. Reprinted, with permission, from Proceedings of 2010 Fifth International Conference on Systems (ICONS)

2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement

SSIRI 2009 Short Paper
Java Card Performance Optimization of Secure Transaction Atomicity based on Increasing the Class Field Locality

Johannes Loinig, Christian Steger, and Reinhold Weiss
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
Email: {johannes.loinig, steger, rweiss}@tugraz.at

Ernst Haselsteiner
NXP Semiconductors Austria GmbH
BU A&I, BL ID
Gratkorn, Austria
Email: ernst.haselsteiner@nxp.com

Abstract

Java Cards are embedded systems, very often implemented as smart cards. They are used in banking, e-government, telecommunication, and ticketing. Due to these use cases they have to provide a wide range of security mechanisms and a high performance in relation to the available hardware cost. One of these security features is the transaction mechanism. It ensures that data in persistent memory stays consistent in case of the execution of the application is interrupted unexpectedly by e.g. loss of power. Such transaction mechanisms are very time consuming. Therefore, we[1] *propose a caching mechanism for transactions. The mechanism uses a buffer located in RAM and reduces costly write cycles into persistent memory without any loss of security. In order to further increase the performance of this caching mechanism, we additionally introduce a concept to maximize the locality of selected Java fields which are written very often.*

1. Introduction

Java Cards are small platforms (usually smart cards) that provide a runtime environment (JCRE) and a virtual machine (JCVM) [1]. Sun Microsystems defined them and a subset of the Java programming language in [2]. The embedded implementation of a smart card and the security mechanisms of a Java Card allow the execution of highly secure applications directly on the card. Typical use cases are transport, ticketing, banking, telecommunication, and e-government applications. In comparison to other smart card systems without any computational capability (e.g. memory cards), these cards achieve a much higher security level based on integrity, authentication and confidentiality. All of these use cases come with a high number of issued cards which leads to a price pressure for the manufacturers. This results in a small footprint for the microchip. Smart cards are very often based on the old 8-bit 8051 CPU architecture. In order to achieve specified execution time, maximum performance of these embedded devices is necessary. For Java Cards, this performance condition is even tightened because of the included virtual machine which provides additional security features but represents a computational overhead in comparison to native smart cards which execute machine instructions instead of Java bytecodes.

In the next sections we will introduce a method how to accelerate transaction atomicity significantly. In Section 2 we summarize related work to optimize the performance of Java Cards. Furthermore Section 2 gives the basics of Java Card's transaction atomicity and how this can be achieved. Section 3 introduces our methodology to gain better performance for transactions. First, our hash table based cache is described, and then a optimization using Java annotations is proposed. Section 4 describes the simulation environment and the achieved results. Finally Section 5 summarizes this paper and lists related further work.

2. Related Work

The virtual machine (executing the compiled Java programming language) is not the only security feature of Java Cards. An applet firewall provides a secure border between multiple installed applications. Cryptographic classes provide algorithms for user- and data authentication and confidentiality. Transaction atomicity (more detailed explained below) ensures that all necessary assignments to persistent memory are executed completely and correctly - or not at all if the execution was aborted [3]. All of these features cause a computational overhead which was tried to overcome in different approaches. To accelerate the virtual machine the authors of [4] (and others as well) propose a Java processor executing Java bytecode directly instead of interpreting and 'converting' them to native machine instructions. Another approach to accelerate bytecode execution is to use instruction folding [5] [6] to combine Java bytecodes and eliminate redundant sequences of machine instructions. Optimizations executed during runtime like dynamic compilation [7] are

1. This paper is a result of the *HiPerSec* project which has been supported by the Austrian government under grant number 816464.

978-0-7695-3758-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSIRI.2009.39

IEEE computer society

©2009 IEEE. Reprinted, with permission, from Proceedings of Third IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI)

```
Object o1 = new Object();
// ...

JCSystem.beginTransaction();
o1.setState(...)

// b1 is a class field located
// in EEPROM
b1 = (byte) 0xFF;
JCSystem.commitTransaction();

// Either both (o1 and b1) are
// changed or none.
```

Figure 1. Java Card Code Example with Transactions

not yet applicable for smart cards because they have too little RAM to store intermediate results.

Another well known performance bottleneck of smart cards is writing data to persistent memory. In general, smart cards do not have their own power supply. They are powered by the smart card reader via the contact- or the contactless interface. Therefore smart cards include usually an EEPROM which is able to store data that must not get lost between sessions. Writing into EEPROM is about 1000 times slower than writing into RAM.

2.1. Transaction Atomicity

Furthermore, the application execution can be aborted at any time if, for example, the reader shuts down the power supply or the card is removed from the reader. *Transaction atomicity* guarantees in such a case that one or multiple variable assignments are executed completely and correctly or not at all. This guarantees data integrity in case of an unexpected interruption of the executed program. Transaction atomicity is performed by multiple consecutive EEPROM write cycles what decreases the performance of a Java Card additionally.

The Java Card specification [2] defines two mechanisms for transaction atomicity: *atomic writes* and *transactions*. Atomic writes are used implicitly if a single field assignment is executed and the field is located in persistent memory. Transactions can include several assignments. Either all of these assignments are executed or none. Transactions start with a `JCSystem.beginTransaction()` and end with a `JCSystem.commitTransaction()` or a `JCSystem.abortTransaction()`. A simple example is shown in Figure 1.

2.2. Transaction Mechanisms

Two different approaches for implementing a transaction mechanism in software are described in [8] and referred to as *Old Value Logging* (OVL) and *New Value Logging* (NVL).

OVL backups the *old* data values altered within a transaction before overwriting them: For each assignment within a transaction, the old value is stored into a reserved area in the persistent memory - the transaction buffer (TB). After checking and validating the TB the variable is overwritten in the persistent heap with the new data value. Finally the new value is checked. After commitment of the transaction the data in the TB is not needed anymore and the TB gets marked as invalid again. If the execution is interrupted before commitment of the transaction (or the transaction was aborted), the old values are restored from the TB.

NVL stores the *new* values at a different location and updates the reference to the fields: For each assignment in a transaction a new variable is allocated and filled with the new value. After checking the data the reference of the Java field is updated to the new location. When the transaction is committed all new data values are copied to their primary location. After checking the values the additional allocated memory is freed again and the variable references are changed back to the primary location. If the transaction is interrupted it has to be ensured that the references point to the old unchanged values.

OVL needs a minimum of $1+3n$ EEPROM write cycles, NVL needs a minimum of $2+n$ write cycles where n is the number of assignments within one transaction.

3. Annotation Based Methodology to Accelerate Transactions

Our proposal is based on two steps. First, in Sections 3.1 and 3.2 we show how class field locality can be used to optimize the performance of transactions. In the second step we propose to annotate class fields that are written often within transactions. An annotation as described later in Section 3.4 signals the virtual machine to allocate the needed memory on a reserved EEPROM page instead of the regular persistent heap. Figure 2 shows the resulting memory layout. This ensures a higher locality for often used class fields. The additional performance gain happens implicitly in the caching mechanism described in the first step.

3.1. Utilizing Class Field Locality

OVL and NVL have their own advantages and disadvantages which are described in detail in [8]. The big advantage of the NVL is that a buffer in RAM can accelerate transactions. However, it must be ensured that no data becomes corrupted when it is copied from RAM to the primary location in EEPROM. Therefore a mixed approach consisting of OVL and NVL is used as shown in Figure 3. Lines 1 to 8 belong to the NVL. Lines 8 to 15 belong to the OVL. Line 8 is both, NVL and OVL.

©2009 IEEE. Reprinted, with permission, from Proceedings of Third IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI)

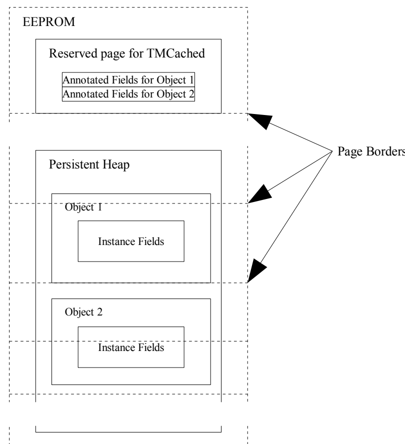


Figure 2. Proposed Memory Layout

1: **if** transaction is active **then**
2: **for** all variable assignments into EEPROM **do**
3: Allocate a new variable in RAM.
4: Store the new value into the new variable.
5: Update the reference to the new variable.
6: **end for**
7: **else if** transaction commit **then**
8: Copy the content of the new variables in the RAM into transaction buffer.
9: Check the content of the transaction buffer.
10: Validate the transaction buffer.
11: Overwrite all old variables in persistent heap with their new content in RAM.
12: Check the new content in the persistent heap.
13: Invalidate the transaction buffer.
14: Free the allocated memory in RAM.
15: **end if**

Figure 3. A Buffered Transaction Mechanism.

The better transaction performance is reached when the RAM buffer is copied into the transaction buffer in EEPROM (Line 8). EEPROMs are usually organized in pages. One write cycle affects one page, independent of the number of changed bytes. If the content of the RAM buffer fits into one EEPROM page, only one write cycle is needed to fill the transaction buffer. In an unbuffered OVL or NVL sequential write cycles would be needed instead.

3.2. A Hash-Table based Cache for Transactions

EEPROM pages can be utilized to accelerate transactions. Java Card class fields have a high locality: they are located close together in the memory. A reason for this lays in the Java Card specification. Instance fields, for example, are stored in one memory area which size is defined in the CAP file (Class component `declared_instance_size`, see JCVM specification [2] for more details). As described later in Section 3.5 CAP files are the binary representation of Java classes. The individual instance fields are represented by consecutive 16-bit slots which are numbered serially by their tokens. These tokens are used as references to the variables. Therefore the probability that class fields are located in the same EEPROM page is very high.

Hash (n bits)	EEPROM Address	New Value	Status (dirty)
0			
1			
...			
2^n			

Figure 4. Our Proposed Cache Structure

To accelerate transactions we propose a software cache located in RAM. It is a modification of the buffered mechanism described above. New values of a transaction are stored in a hash table. A hash value is used to lookup the right entry in this table. The lower bits of the accessed class field's address are used as a hash value. How many bits are used depends on the size of the cache. The entries of the cache are shown in Figure 4. They store information of the class field's address (to detect hash collisions), the new value, and some status information. Only field accesses within transactions are handled over this cache.

One transaction is executed as shown in Figure 5. This algorithm takes advantage of EEPROM page writes at two positions. In the for-loop starting at line 9 multiple values are written into the transaction buffer instead of only one. As usually only a few bytes are written, the probability that this affects only one EEPROM page is very high. In the second for-loop starting at line 14 multiple values are stored into their primal position. As only these entries which refer to the same page were selected before, this can be done in one EEPROM page write as well.

As usual for NVL mechanisms, reading variables within transactions is more costly than for OVL mechanisms because the buffer has to be checked. However, this lookup into RAM is much faster than a EEPROM write cycle into. Thus, saving write cycles overcomes the lookup overhead by far. To minimize this overhead, we have chosen a hash table as a buffer which allows a lookup in constant time.

The proposed algorithm does not violate the Java Card specification. New values are located in the RAM first. In case of an interruption of the execution, the content of RAM is invalid but as the transaction is not committed and the old values in EEPROM are not changed, this is not an

©2009 IEEE. Reprinted, with permission, from Proceedings of Third IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI)

```
1:  if transaction active then
2:    for all variable assignments into EEPROM do
3:      Calculate hash using the address of the variable.
4:      Select entry in hash table with the hash value.
5:      if selected entry is not marked as dirty then
6:        Store the new value and the EEPROM address
          into selected entry.
7:        Set entry's dirty-bit.
8:      else
9:        for all dirty entries whose variable is located in
          the same page as the selected one do
10:         Write the old value into the transaction buffer.
11:         Check the value.
12:       end for
13:       Validate the transaction buffer.
14:       for the same entries as before do
15:         Overwrite the old value with the new content
            in the hash table.
16:         Mark entry as not dirty.
17:       end for
18:       Write the new value in selected entry.
19:       Mark entry as dirty.
20:     end if
21:   end for
22: else if transaction commit then
23:   while entries are marked as dirty do
24:     Select next dirty entry.
25:     Repeat line 9 to 17.
26:   end while
27:   Invalidate transaction buffer.
28: end if
```

Figure 5. A Cached Transaction Mechanism Based on a Hash Table

issue. In case of a hash value collision, or a transaction commitment the values in RAM are stored in EEPROM. The old values were backed up before into the transaction buffer like in an OVL. As the OVL is compliant to the Java Card specification, our proposed algorithm is as well.

3.3. Increasing Java Field Locality

The performance of our proposed caching algorithm depends strongly on two criteria. First, the performance is higher if the accessed fields are in the same EEPROM page. And second, collision of the hash value may occur and decrease the performance.

Increasing the locality of Java class fields will influence both criteria beneficially: If class fields are located consecutively in the memory, maximum locality is achieved. For our cache, this results in adjoining hash values where collisions only can happen if the number of assignments does not fit into the cache. Furthermore, write cycles can be reduced

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TMCached {}
```

Figure 6. The Proposed Annotation

because a lot of class fields are located in one and the same page.

An analysis of Java Card Applets of different application domains has shown that during one ordinary session just a few Java fields in EEPROM change their value. These few fields can fit easily into one EEPROM page which has usually a minimum of 64 bytes. As for single object instances the probability is very high that all accessed class fields are in one page, this is not the case if more than one object instance is used. Creating multiple object instances is common for object oriented programming languages and can increase the clearness of an application.

Java Card encapsulates the hardware platform completely. The application developer has no influence where the Java fields are going to be stored in the card's memory. One method against this is to store all data into one array in one object. This reduces the clearness of the application and prohibits dynamic data growth. We propose a methodology such that the developer can annotate fields to store them on a reserved page, independent which object instance they belong to. We assume that a developer knows which fields will be written very often during usage. Thereby, the locality for selected fields can be increased.

3.4. Java Annotations

The Java Language Specification [9] defines annotations for different Java targets like classes, methods, and fields. These annotations can be evaluated during compile-time, when processing the class file, or during runtime. Annotations can be pre-defined (e.g. `@deprecated`) or defined by the Java developer. They can include fields that are assigned during compile-time. Runtime and class-file annotations are represented as attributes [10] [11] and stored in the constant-pool of a class.

We propose a new annotation shown in Figure 6. The target is set to `ElementType.FIELD`. This is evaluated during compile time and restricts the appliance of the annotation to class fields. The retention policy `RetentionPolicy.RUNTIME` causes the Java compiler and class loader not to eliminate the annotation. Thus, it will be available during runtime for the virtual machine. Our proposed annotation does not need fields.

Runtime annotations can be accessed by the Java application via the reflection-API. However, they can also be evaluated by the virtual machine itself. Virtual machines, which do not support annotations or the used type of

©2009 IEEE. Reprinted, with permission, from Proceedings of Third IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI)

annotation, have to ignore them and execute the application if it would not be annotated. That is why runtime annotations give application developers the possibility to use features of virtual machines with special capabilities without losing compatibility with common virtual machines.

3.5. Annotations for Java Cards

Annotations are not defined for Java Cards. However, as Java for Java Card is a subset of the Java programming language [2] [9], the common Java compiler can be used for Java Cards as well. This allows usage of annotations with following considerations:

- For Java Cards an own API is defined in the specification [2]. This API does not include the necessary `java.lang.annotation` package. The package has to be ported to Java Card.
- Java Cards do not execute class files. The JCVM is split into an off-card and an on-card part. The off-card VM is a converter which loads and pre-processes the class file. This results into the CAP file which is uploaded to the smart card. A common CAP file converter may delete annotations as they are not defined for Java Cards. An extended converter is needed.
- The CAP file does not support attributes and thus also not annotations. The CAP file does not leave a lot of space for extensions. However, the annotated CAP file must be fully compatible with Java Cards that do not support annotations.

4. Experimental Results

Our implementation is based on two simulators: a low-level Java Card simulator and a high level cache simulator. The low-level simulator is accurate enough to provide occurrence of EEPROM accesses and the related addresses. The simulation results were used as input for the cache simulator to investigate the performance for different cache-parameters.

Selected applets were run on the Java Card simulator to determine representative EEPROM access patterns for Java Cards. Only a few fields are usually written within transactions (2-5 fields). Arrays are also copied in transactions, but as `Utils.arrrayCopy()` is mapped to a native function, these accesses are not considered here. The class fields can be defined in one class or can be spread over several objects.

To benchmark the caching mechanism a Java Card applet was implemented that accesses fields randomly in single objects or in an array of objects. Latter one was implemented in two variants. Variant one is straight forward with all fields in the concerning objects. The second one uses a factory class that allocates members at once. The idea was to try if a factory pattern in Java suffices to achieve a higher field locality. The factory class provides arrays which are

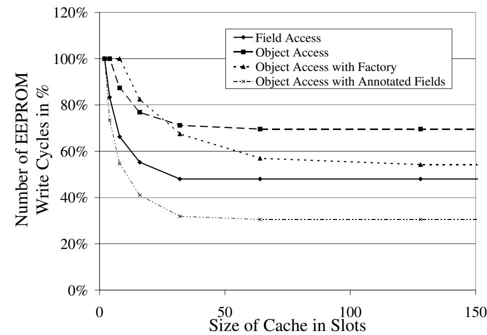


Figure 7. Achieved Performance Gain

all created at the same time. This increases their locality. Objects that need an array, request it at the factory instead of using an array annotated with `@TMCached`.

Our existing Java Card simulator was extended by the proposed annotation. During instantiation of a new object, the system checks if a field is annotated. Annotated fields are allocated in a reserved area which exactly fits into one EEPROM page. Not annotated fields are allocated as before.

4.1. Simulation Results

Figure 7 shows the results in performance gain in comparison to an OVL. Different access patterns were executed. The shown values are an average of 255 test runs writing to 5 fields (per transaction) in a set of 16 fields respectively 16 objects. Even with a small cache size of only 2 entries 16% of write cycles can be saved if the fields are located in one and the same object. A maximum performance gain of more than 50% can be achieved with 32 entries. A bigger cache size has no positive influence anymore because the entire transaction fits into the cache. For fields in multiple objects a performance gain of 20-30% can be achieved. The gain is lower because the variables are spread over multiple pages. The factory pattern performs better than a pure object based approach but does not catch up with pure field access. As can be seen in Figure 8 the factory pattern causes more collisions of the hash values than the other access patterns.

The usage of annotated fields shows the most significant performance gain of more then 26% for only 4 cache slots (see Figure 7). A maximum of almost 70% can be achieved with 32 cache slots.

Figure 8 furthermore shows that the change from a one-bit hash to two bits never reduces the number of collisions. The reason for this is the Java Card memory structure defined by the Java Card specification [2]. All primitive data types (except of integers) are represented by 16 bits. This means that the addresses of two consecutive primitive type fields

©2009 IEEE. Reprinted, with permission, from Proceedings of Third IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI)

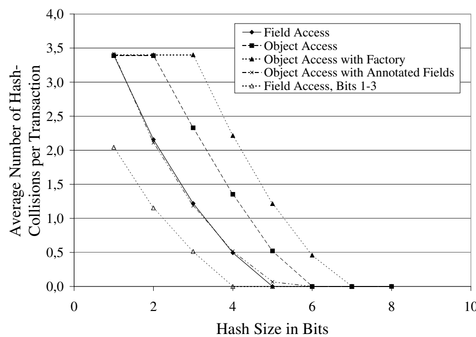


Figure 8. Hash-Collisions Depending on Hash Size

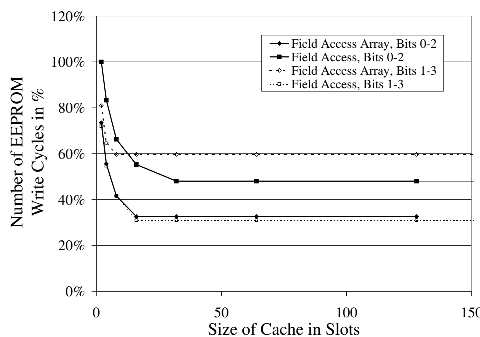


Figure 9. Performance Gain with Bits 1-3 for Hash

in an 8-bit architecture vary in the second bit but not in the lowest one. Figures 9 and 8 show the performance gain and number of collisions if bits 1 to 3 of the EEPROM address are used for the hash value instead of bits 0 to 2.

The performance gain of field access increases from 16% to 45% for a buffer with two entries. This comes with a reduction of performance gain for accesses into byte arrays, also shown in Figure 9. Byte arrays are not 16-bit oriented. Ignoring the lowest bit of the address reduces the performance gain from 44% to 35%. Figure 8 shows that the number of collisions is reduced for field accesses that are not targeted to arrays.

5. Conclusion and Future Work

Java Cards are commonly used for secure applications. Therefore they provide a lot of security features that cause a computational overhead. We have shown that one of the performance bottlenecks, the transaction mechanism, can be accelerated significantly. By using of a buffer in RAM transactions with Java fields in objects can be accelerated by 20%-30%. Furthermore we have shown that Java annotations can help to utilize special features of certain virtual machines without losing compatibility. Using our proposed `@TMCached` annotation for selected fields in objects increases their locality in persistent memory and accelerates transactions by 60% with moderate costs of some few additional bytes in RAM.

Investigation can be done to find other caching algorithms reducing the number of costly collisions. Atomic assignments which use transactions implicitly do not benefit from our proposed cache. As these atomic writes appear very often, it should be considered if they can be accelerated by using annotations for selected fields as well.

References

[1] Z. Chen, *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[2] *Java Card Platform Specification 2.2.2*, Sun Microsystems, Inc. [Online]. Available: http://java.sun.com/javacard/specs.html

[3] *Java Card Platform Security*, Sun Microsystems, Inc. [Online]. Available: http://java.sun.com/products/javacard

[4] Z. Jianjie, L. Feihui, G. Yuanqing, Y. Zhenwu, and Y. Zhilian, "A Java processor suitable for applications of smart card," *ASIC, 2001. Proceedings. 4th International Conference on*, pp. 736–739, 2001.

[5] A. Azevedo, A. Kejariwal, A. Veidenbaum, and A. Nicolau, "High performance annotation-aware JVM for Java cards," in *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*. New York, NY, USA: ACM, 2005, pp. 52–61.

[6] H. McGhan and M. O'Connor, "PicoJava: a direct execution engine for Java bytecode," *Computer*, vol. 31, no. 10, pp. 22–30, Oct 1998.

[7] M. Debbabi, A. Mourad, C. Talhi, and H. Yahyaoui, "Accelerating embedded Java for mobile devices," *Communications Magazine, IEEE*, vol. 43, no. 9, pp. 80–85, Sept. 2005.

[8] M. Oestreicher, "Transactions in Java Card," in *ACSAC '99: Proceedings of the 15th Annual Computer Security Applications Conference*. Washington, DC, USA: IEEE Computer Society, 1999, p. 291.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*. ADDISON-WESLEY, 2005. [Online]. Available: http://java.sun.com/docs/books/jls/

[10] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, 2nd ed. Prentice Hall PTR, 1999. [Online]. Available: http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html

[11] M. D. Ernst, "Type Annotations Specification (JSR 308)," November 2008. [Online]. Available: http://groups.csail.mit.edu/pag/jsr308/

©2009 IEEE. Reprinted, with permission, from Proceedings of Third IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI)

Fast Simulation Based Testing of Anti-Tearing Mechanisms for Small Embedded Systems

Johannes Loinig, Christian Steger, and Reinhold Weiss
Institute for Technical Informatics
Graz University of Technology, Graz, Austria
Email: {johannes.loinig, steger, rweiss}@tugraz.at

Ernst Haselsteiner
NXP Semiconductors Austria GmbH, Gratkorn, Austria
Email: ernst.haselsteiner@nxp.com

***Abstract*—Small embedded systems are often powered by unreliable power supplies like energy harvesting systems or external power supplies. For secure embedded systems a sudden loss of power can violate data integrity. The power has just to drop when data is written to non-volatile memory. In order to guarantee data integrity, a secure embedded system has to provide an anti-tearing mechanism. In this work we[1] summarize a fast simulation based test method for such mechanisms.**

I. Introduction, Related Work, and Motivation

Small embedded systems (ES) have often no reliable power supply. However, it is important that a system is able to guarantee integrity for data written to persistent memory. To ensure this in case of sudden power loss, ESs provide an anti-tearing mechanism, or so called transaction mechanism (TM). Testing a TM can be very difficult and may require a lot of knowledge about implementation details. The basic concept of most TMs is a backup strategy. A backup of important data is made before overwriting it with new data. In [1] different backup strategies are explained in detail. Other approaches are based on memory redundancy [2].

In [3] a black box test for TMs is described. An atomic operation is interrupted at multiple specified points in time t_i. After each interruption it is checked if the system is still in a consistent state. In detail, it may be difficult to determine the 'right' values for t_i.

A simulated fault injection is a promising approach to overcome this drawback. Simulated fault injection [4] allows observing a complex system in situations which are difficult to reproduce on real hardware. A remaining drawback is that a lot of code has to be re-executed for every test case.

II. A Test Vector Based Fault Injection Approach

If the repeated execution of embedded code leads to the same write operations to persistent memory, there is no need to run it several times for testing purposes. The TM would perform anyway the same operations as before. Therefore, we run code only once without any fault injection and record the write operations to generate test vectors (TV). Because the TVs are independent it is possible to skip and merge TVs which are identified to cause execution of identical code. A compaction process identifies such TVs.

[1]This paper is a result of a project which is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology (contract FFG 816464).

To apply a TV, its associated write operations are performed by the simulation environment. Afterwards the simulated system starts up, the TM is executed, and the consistency check can be done. This is repeated for each TV and is equivalent to the model's simulation after an injected tearing event.

This method requires less code to be executed. The TVs are generated out of recorded write operations and are therefore created without any knowledge of the TM's implementation.

III. Implementation, Evaluation, and Results

We implemented our test environment for the TM of a Java Card [5]. The software runs on a memory accurate simulation implemented in SystemC [6]. The test cases are implemented in a Java Card application.

A theoretical evaluation of our method has shown that the TV approach is faster than a regular fault injection approach if $\sum_i^{x_B-1} t_{B_i} + \sum_i^{x_R-1} t_{R_i} > 0$ holds, where x_B and x_R are the numbers of needed write operations for backup and restore of data, and t_B and t_R are the execution time until the i-th tearing event. This equation holds for all of our known TMs.

A measurement of the execution time of our TV based mechanism shows a linear behavior which scales much better than the regular fault injection method. For 16 test cases with more than 70 write operations, we achieved a performance gain of more than 20% without TV compaction. For a basic compaction method we achieved up to 13% faster execution times in comparison to the approach without compaction.

IV. Conclusion

We explained a test vector based fault injection method for testing anti-tearing mechanisms. The approach reduces code re-execution during testing without loss of test coverage. A proof of concept implementation achieved a performance gain of more than 20%. Furthermore, our approach allows reducing the number of test cases by test vector compaction.

References

[1] M. Oestreicher, "Transactions in Java Card," in *ACSAC '99* . Washington, DC, USA: IEEE Computer Society, 1999, p. 291.
[2] G. Lisimaque and P. Paradinas, "Method and device for updating information elements in a memory," US Patent 5,479,63, Dec. 26, 1995.
[3] W. Rankl and W. Effing, *Smart Card Handbook*. New York, NY, USA: John Wiley & Sons, Inc., 2003.
[4] K. Rothbart et al., "High level fault injection for attack simulation in smart cards," *Test Symposium, 2004. 13th Asian*, pp. 118–121, Nov. 2004.
[5] *Java Card Platform Specification 2.2.2*, Sun Microsystems, Inc.
[6] *Open SystemC Language Reference Manual IEEE Std 1666-2005*, IEEE.

978-1-4244-5833-2/10/$26.00 ©2010 IEEE **242**

©2010 IEEE. Reprinted, with permission, from Proceedings of 2010 15th IEEE European Test Symposium (ETS)

Fast Simulation Based Testing of Anti-Tearing Mechanisms for Small Embedded Systems

Johannes Loinig, Christian Steger, and Reinhold Weiss
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
Email: {johannes.loinig, steger, rweiss}@tugraz.at

Ernst Haselsteiner
NXP Semiconductors Austria GmbH
Gratkorn, Austria
Email: ernst.haselsteiner@nxp.com

***Abstract*—Small embedded systems are often powered by unreliable power supplies like energy harvesting systems (e.g., for sensor nodes) or external power supplies (for smart cards). For secure embedded systems a sudden loss of power can violate data integrity. The power has just to drop when data is written to non-volatile memory. Thinking about a byte array in a smart card representing some digital money of an e-purse, this becomes obvious. In order to guarantee data integrity a secure embedded system has to provide an anti-tearing mechanism. Testing this mechanism is very difficult, extensive, and requires deep inside knowledge into its implementation details.**

In this work we show how a simulation of an embedded system can be used to test the anti-tearing mechanism. High-level test cases are used to generate test vectors automatically. The proposed approach allows a fast and comprehensive test of the anti-tearing mechanism. We[1] explain our proposed mechanism on the basis of a case study of a smart card system. However, the mechanism is general enough to be used for secure embedded systems of any kind.

I. Introduction

Small embedded systems have often no reliable power supply. Typical examples are sensor nodes with energy harvesting systems and smart cards which are externally powered by the smart card reader. In the first case the gained power is dependent on the environment wherein the system operates, e.g., the lighting conditions in case of photovoltaic cells. In the case of smart cards, the user may just remove the card form the reader. In both cases the execution of the embedded application will be aborted immediately. Consequently, the application can not rely on its execution to its regular end.

However, it is important that a system is able to guarantee integrity for information stored in persistent memory. This is especially important if the system is used for secure use cases as banking cards or e-government applications. For example, the amount of digital money stored on a banking card must stay consistent in any case of operation. To guarantee the necessary data integrity in case of sudden power loss most smart cards have to support atomic write operations to persistent memory. Anti-tearing mechanisms, or also called transaction mechanisms (TM), provide this functionality.

[1]This paper is a result of a project which is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology (contract FFG 816464).

A TM is usually part of the operating system (OS). Therefore, an application does not need to take care about tearing-events on its own. For example, in case of Java Cards [1] (a smart card system including a Java virtual machine) each assignment to a variable located in the persistent memory is defined to be atomic.

Testing a TM can be very difficult and may require a lot of deep knowledge about the implementation details. Additionally, the TM runs on an embedded system which is supposed to be a secure environment. Hence, observability and controllability of the system for testing purposes is strongly limited. Having a reliable, implementation-independent, and rapidly executed test in place would be a great benefit.

In this work we propose an approach which fulfills these requirements. Test vectors are generated automatically from high-level test cases which are independent from the TM's implementation. A fast simulation based execution allows a rapid and simple test setup and execution. Furthermore, we evaluate the performance of the test mechanism in comparison to a common fault injection based testing approach.

The remainder of this paper is structured as follows. First, we summarize related work, explain anti-tearing mechanisms, and give our motivation. Then, we describe a simulation based fault injection method which is the basis for further comparison. After that, we propose our test vector based approach including test vector generation and compaction. Finally, we introduce our proof of concept implementation and evaluate our proposed approach on theoretical basis and simulation results.

II. Related Work and Motivation

The requirement to write data atomically to persistent memory is well known for smart cards. However, as already mentioned, it is not restricted to them. Smart cards are very small embedded devices with limited computational resources: some few hundred bytes of RAM and some few hundred kilobytes of ROM and persistent memory (typically EEPROM). Smart cards are usually powered via inductive coupling by the smart card reader what raised the need for reliable and efficient TMs.

In the following section we explain some published TMs for smart cards. This list is not complete, there are more

©2010 IEEE. Reprinted, with permission, from Informal Proceedings of 2010 15th IEEE European Test Symposium (ETS)

mechanisms published especially in patents. Afterwards, we summarize related work about verification methods for atomicity.

A. Anti-Tearing Mechanisms

The basic concept of the introduced anti-tearing mechanisms is a backup strategy. Simply overwriting values in persistent memory is very insecure [2]. If the execution is interrupted (e.g., caused by a breakdown of the power supply), the old value may be deleted, but the new value may not yet be in a consistent state. Thus, the basic principle is to make a backup of important data before writing the new data. When the system boots up it searches for existing backups. If a backup exists (in case of a tearing event) the system restores the data to bring the memory back to a consistent state. For the remainder of this paper we call these phases the *Backup Phase* (BP) and the *Restore Phase* (RP).

Oestreicher explains two basic mechanisms in [3]: *Old Value Logging* (OVL) and *New Value Logging* (NVL). Both mechanisms use a transaction buffer (TB) in the persistent memory to store the backup information. In the BP OVL saves the old value going to be overwritten, its type, and its address to the TB. This is a roll-backward strategy as the old value can be restored in case of a tearing event. NVL saves the new value, type, and address to the TB. Thus, the RP is able to finalize the write operations if necessary. This is a roll-forward strategy.

The patent [4] describes a different concept based on memory redundancy. The persistent memory is split into multiple parts. Each data field is allocated in all these parts. A write operation does not overwrite the old data directly but stores the new values in an according data field on a different memory part. If the write operation was successful the field's reference is changed to the new location. The old data field may be overwritten with the next atomic write operation. The critical write operation in this mechanism is the update of the reference. As the reference to persistent data fields is also located in the persistent memory it can not be overwritten without tearing countermeasures. An OVL or a NVL may be used to update the reference in a secure way.

B. Verification of Atomicity

Verification and testing of software in small embedded systems is very crucial. The software (e.g., the operating system) is often stored in the ROM during fabrication. Thus, it is practically impossible to update the software in case of a bug that should be fixed [5, page 574]. Additionally, highly secure use cases require a fully trustworthy implementation which makes testing even more important.

In [5, page 586] it is mentioned that testing the TM is a typical black box test. An atomic operation is executed and interrupted at multiple specified points in time. After each interruption it is checked if the system is still in a consistent state. This is explained in more detail in Section III. However, it is mentioned that the number of needed tests is fairly large.

There exists a lot of related work around the TM for Java Cards [1], [6]. The authors of [7] and [8] used the *Java Modeling Language* to model the Java Card TM. The concepts allow formal verification of atomicity of Java Card applications. However, a fully functional and reliable TM is assumed to be in place when the verified application is executed.

In [9] the author introduces a method to formally prove embedded C code of anti tearing mechanisms. First, a transition system is extracted automatically from a formal specification of the atomic function to be verified. Then, a program verification tool (Caduceus) is used to evaluate if the implementation fulfills the transition system. The verification tool requires that the C code is annotated with functional properties like the precondition and the post-condition.

Simulation based testing is common for hardware development. Simulated fault injection allows observing the system in situations which are difficult to reproduce on real hardware. Rothbart et al. has proposed a high-level fault injection approach for smart cards in [10]. The system was modeled in SystemC [11]. Faults can be injected into functional blocks and interconnections without recompilation of the model.

A simulation based test environment must provide a suitable performance if it should be used during the development process. Misera et al. describes how to achieve higher testing performance for SystemC based fault injection [12]. The basic principle of the performance increase is to parallelize tests for rather low level modules.

C. Motivation of our Work

We were not able to find related work about fast and reliable verification of transaction mechanisms. The found formal verification methods are done on a high abstraction level. As great parts of operating systems are implemented at least partially in assembler these approaches may not be applicable for verification during development. In addition these approaches require additional high-level information like functional properties. This information may not always be available during a development process.

Fault injection is a promising approach. A tearing event does not need a complicated fault model and can be easily applied on a simulated embedded system. However, we will show in this work that a fault injection based approach only may not be the best choice. First, we explain the basics of the fault injection based approach. After that, we explain how the test performance can be increased by reduction of code that is executed by the simulated system. Furthermore, our approach allows a reduction of test cases which is not easily possible with a basic fault injection based approach.

III. A Simulation Based Fault Injection Approach

In this section we explain how to move a common test for anti-tearing mechanisms for smart cards explained in [5] to a simulation based environment. Furthermore, we identify the advantages and disadvantages of this proposed test method.

A. Fault Injection Without Simulation

As mentioned before, [5] describes a black box test for anti-tearing mechanisms of smart cards. Figure 1 shows this

©2010 IEEE. Reprinted, with permission, from Informal Proceedings of 2010 15th IEEE European Test Symposium (ETS)

basic test approach. The host system sends a command that causes the smart card to start execution of the application. The host turns off the power supply of the smart card at multiple points in time t_i. This aborts the execution on the smart card's application. The implementation of the test seems to be trivial.

In detail, it may be difficult to determine the 'right' values for t_i. Notice that a secure embedded system is designed not to leak information which may be used for attacks. Thus, the choice of values for t_i can be, in the worst case, more or less just guessing. Additionally, as this is a pure black box test, not only the right values have to be guessed, the right number of tests is also difficult to determine.

It should be mentioned that just switching off the reader may turn up as not so easy as expected. A commercial smart card reader may not provide a command to switch of the power supply. Furthermore, smart card readers do usually not provide a mechanism to define exact arbitrary timing for commands as the timing is defined by the underlying protocol like ISO 14443.

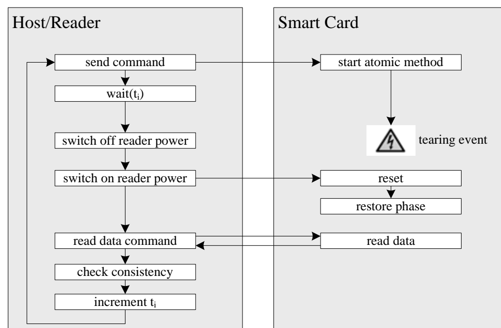


Fig. 1. Fault Injection Approach Without Simulation

B. Simulation Based Fault Injection

Applying the fault injection to a simulation avoids the difficulties to choose appropriate time values because the simulation provides full observability and controllability of simulated system. A memory accurate model is able to identify write operations to persistent memory. The simulation can notify a fault injection block which decides if a tearing event has to be executed. The basic principle is shown in Figure 2. The hardware model can be implemented with a language like SystemC. SystemC allows modeling in different abstraction levels. A suitable abstraction level that allows a simulation speed that is fast enough for testing in an appropriate time should be chosen. The requirements to a model that provides the proposed fault injection mechanism are minimal. It only must be able to report the persistent memory's write operations, and it must have a mechanism to simulate the loss of the power supply on a rather high abstraction level (e.g., by enforcing a system reset).

In detail, the explained approach is more complex. The fault injection module as shown in Figure 2 must be able to decide if a fault injection should be executed. Therefore, it has to identify the write operations and distinguish between:

- write operations that have to cause a tearing event,
- write operations that already have caused a tearing event and are thus tested,
- and write operations that should not cause a tearing event because they are out of scope of the active test case.

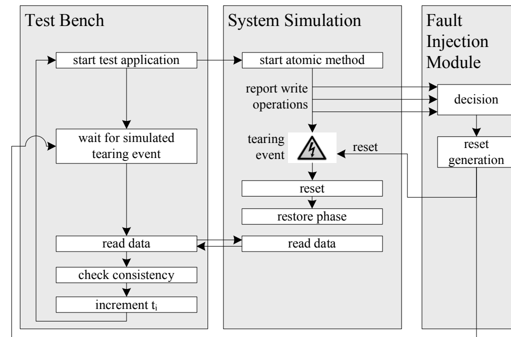


Fig. 2. Simulation Based Fault Injection Approach

Latter ones are, for example, write operations in the RP. Notice that an interrupted BP is followed by a RP and the RP also includes write operations which may be necessary to be tested in a separated test case.

C. Estimation of the Test Effort

The number of test cases strongly depends on the implementation of the embedded system. Different data types which can be written in an atomic way may have different implementations in the TM. Thus, they have to be tested separately. Additionally, the system can provide different kinds of atomic write operations. Java Cards, for example, provide atomicity for single variable assignments and for groups of assignments. Latter ones are called transactions [1] and are encapsulated within the `JCSystem.beginTransaction()` and `JCSystem.commitTransaction()` function calls. Finally, there may be functions like `arrayCopy()` that are atomic by definition. Each of these write operations for Java Cards are conceptually different. It is obvious that they may be implemented at least partially independently and that they require different test cases.

Furthermore, these test cases consist of multiple write operations and each of these write operations results in a simulated tearing event. Last but not least, each test case starts again from the beginning after a tearing event (after the RP). Thus, huge parts of the system are simulated again and again.

Therefore, the test effort can be very high in total. This is a clear drawback of the proposed approach. We describe a more efficient test mechanism in the next section.

©2010 IEEE. Reprinted, with permission, from Informal Proceedings of 2010 15th IEEE European Test Symposium (ETS)

IV. A Test Vector Based Approach

For our proposed test vector based approach we define the internal state of the embedded system by the content of the TB. The state is evaluated during the RP. Both, BP and RP are modifying the state during their execution. Figure 3 shows the state diagrams for the BP and the RP. The BP always starts in the *invalid* state and ends in the *invalid* state if the BP is executed to its end without occurred tearing event. The RP may start at any state because it must be assumed that a tearing event has interrupted an atomic write operation. If the RP is executed without interruption *invalid* is the final state.

The state diagram in Figure 3 includes the state *written and partially valid*. Latter state is needed if more than one data field must be written like for mentioned Java Card transactions. In such a case the backups are written and validated sequentially one after another. However, the state diagram is very general and does not include states for different data types.

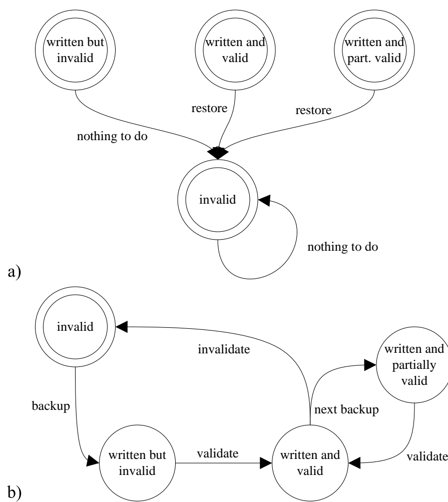


Fig. 3. TB-States during a) the RP and b) the BP

As shown, the BP does not depend on the state of the TB. The executed instructions in the BP only depend on the data which has to be written. This data is defined by the test case. In the strict sense the BP also depends on the initial state of the persistent memory. This initial state can easily be pre-defined and is therefore out of interest for our test approach.

Contrary to the BP, the RP depends on the state of the TM only. The test case can only influence the RP by the content of the TB. This is obvious as the RP runs at startup of the system before a test case is executed.

A. Test Vector Generation

In our proposed vector based test approach we take advantage of these two properties. If the repeated execution of code leads to the same write operations, and therefore to the same TB state, there is no need to run it several times. The RP would perform anyway the same operations as before. Therefore, we run code that is modifying the TB's state only once and record the write operations. These records represent the TB's state changes. Afterwards we use the obtained write operations to create test vectors. Each test vector contains addresses of persistent values and related new values to be changed before the test is executed. An overview is shown in Figure 4 a).

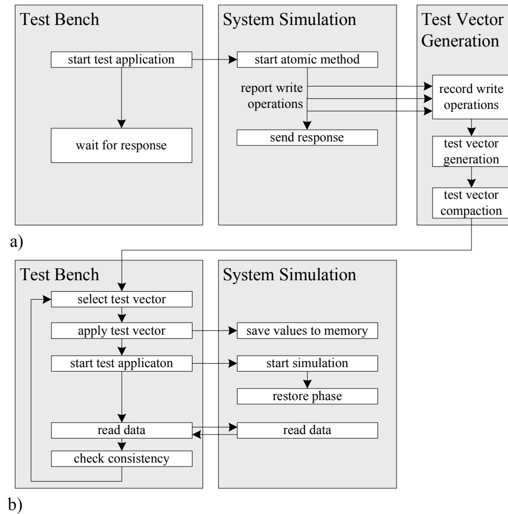


Fig. 4. A Test Vector Based Approach. a) Test Vector Generation b) Test Execution

The test case is executed only once. All write operations to the persistent memory are reported by the simulation. One tearing event may be simulated to run the RP and to store its write operations as well (not shown in the figure). Afterwards the stored write operations are combined cumulatively to test vectors as shown in Figure 5. This is done to generate test vectors that are independent from each other.

Because the test vectors are independent it is possible to run them in any arbitrary order. This also allows to skip or merge test vectors which are identified to cause execution of identical code parts. The compaction process shown in Figure 4 runs a static analysis on the test vectors and identifies test vectors that can be skipped. This may require that the compaction process considers implementation details of the TM.

B. Test Execution

To apply a test vector, its associated write operations are performed by the simulation environment like shown in Figure 4 b). Notice that these write operations are executed by the simulation environment and not by the simulated model. Therefore, this can be performed very fast. Afterwards the simulated system starts up, the RP is executed, and the consistency check can be done. This is repeated for each test

©2010 IEEE. Reprinted, with permission, from Informal Proceedings of 2010 15th IEEE European Test Symposium (ETS)

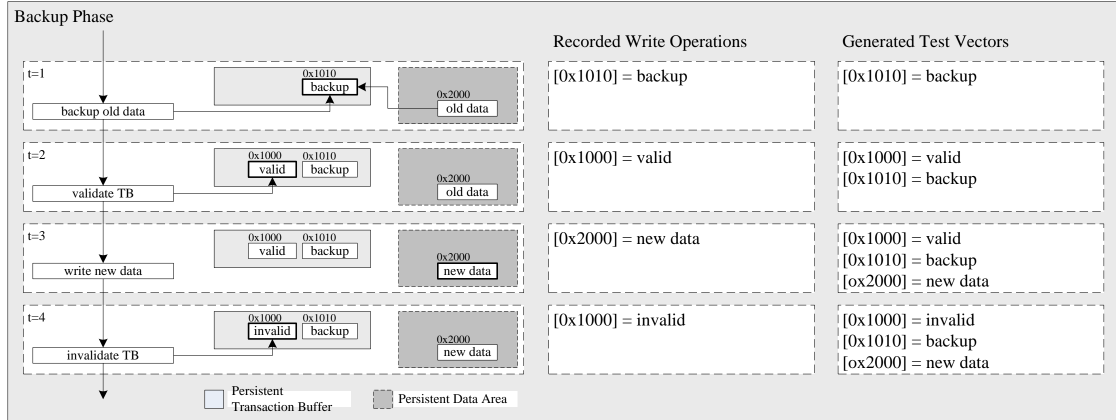


Fig. 5. Example for Generation of Test Vectors for the OVL's BP

vector and is equivalent to the model's simulation after an injected tearing event described in Section III.

This proposed test method requires less code to be executed in the simulated system. The simulation environment must be able to record write operations to the persistent memory and it must provide a mechanism to write data to selected addresses without running the model. Both requirements are independent from the TM. Therefore, the test vectors can be generated and applied without knowledge of implementation details of the TM.

V. Implementation

We implemented our test environment for the anti-tearing mechanisms of a complete Java Card OS. The OS runs on a cycle accurate and memory accurate instruction set simulation and provides NVL and OVL functionality. However, cycle accuracy is not needed for our proposed approach. The simulated smart card was modeled with SystemC.

The test cases are implemented in a Java Card application. Furthermore, this application also includes the consistency checks after the RP. Several test cases for different types (byte, short ...) and atomic write operations (transactions, `arrayCopy()` ...) are implemented.

The fault injection module described in Section III and the record functionality and test vector generation explained in Section IV are implemented in the EEPROM module of the smart card's SystemC model. Test vectors are stored bytewise in a file. So far, the appliance of the test vectors is done manually.

The test vector compaction is just done rudimentarily as the detailed compaction process is out of scope of this work. Only test vectors that are identical (e.g., if the TB is erased) are combined in our proof-of-concept implementation.

VI. Evaluation and Experimental Results

First, we want to evaluate if our proposed test vector based approach is in general faster than a simple fault injection based approach. Equation 1 describes the time t_{fi} which is needed to execute test cases with the fault injection approach where:

- n is the number of test cases emerging from the different data types and write operations,
- x_{BP} and x_{RP} are the number of write cycles during BP and RP,
- t_{BP_i} and t_{RP_i} are the execution time of BP and RP until the i-th tearing event is injected,
- and t_{BP} and t_{RP} are the execution time of BP and RP in total.

The two sums in the equation represent the partially executed (interrupted by a tearing event) test runs for the BP and the RP. The last term represents the execution of the RP for every injected tearing event.

$$t_{fi} = n \cdot \left(\sum_{i}^{x_{BP}} t_{BP_i} + \sum_{i}^{x_{RP}} t_{RP_i} + (x_{BP} + x_{RP}) \cdot t_{RP} \right) \quad (1)$$

Equation 2 describes the time t_{tv} which our proposed test vector based approach needs for execution. In this equation v is the number of generated test vectors. The worst case is that $v = x_{BP} + x_{RP}$ if no test vector compaction is performed and all test vectors are used. For the further evaluation we assume this case.

$$t_{tv} = n \cdot (t_{BP} + t_{RP} + v \cdot t_{RP}) \quad (2)$$

If we use these equations to evaluate $t_{fi} > t_{tv}$ we come to the result shown in Equation 3. This equation holds for all known TMs based on backup strategy. Therefore, we can argue for all TMs based on a backup strategy, that our test vector based mechanism is in general faster than a fault injection mechanism.

$$\sum_{i}^{x_{BP}-1} t_{BP_i} + \sum_{i}^{x_{RP}-1} t_{RP_i} > 0 \quad (3)$$

©2010 IEEE. Reprinted, with permission, from Informal Proceedings of 2010 15th IEEE European Test Symposium (ETS)

We executed both of our proposed testing approaches with the test environment described in Section V. The experimental results are shown in Figure 6. Notice that the absolute simulation times are out of interest as they are anyway strongly depended on the abstraction level of the model and the performance of the simulation environment. Therefore, we have normalized the values in the graphs: the shortest time value for the fault injection approach is normalized to 1. All other values are relative to this basis value.

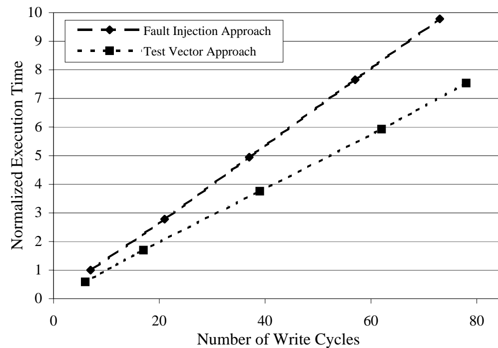


Fig. 6. Normalized Values of Required Testing Time for Fault Injection Approach and Test Vector Approach

The graph in Figure 6 shows the expected linear behavior explained in Equation 1 and Equation 2. As can be seen, our test vector based approach scales much better than the regular fault injection method. For 16 test cases with more than 70 write operations, respectively simulated tearing events or test vectors, we achieved a significant performance gain of more than 20%.

Figure 7 shows the measured and normalized execution time for testing after the test vector compaction. As compaction reduces the number of test vectors, and thus the number of write cycles, we chose the number of test cases for the x-axis. However, these results have to be interpreted with care. The result of the test vector compaction strongly depends on the choice of test cases. Thus, it would have been possible to construct or select test cases with better or worse results as shown in the graph. Therefore, the graph is just given for seek of completeness to show that even a very simple compaction method results in a notable performance gain. For our very basic compaction method we achieved up to 13% faster execution times in comparison to the test vector based approach without compaction.

VII. Conclusion

In this work we explained the importance of anti-tearing mechanisms for embedded system without reliable power supply. Furthermore, we summarized existing mechanisms and common basics. These basics were used to propose two testing approaches: a simulation based fault injection approach educed

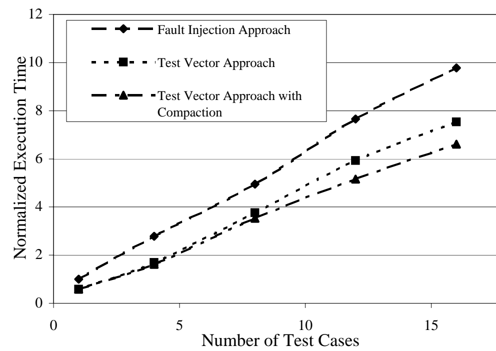


Fig. 7. Normalized Values of Required Testing Time with and without Test Vector Compaction

from a very similar black box test, and an implementation independent test vector based approach.

The test vector based approach reduces unnecessary code re-execution during testing without loss of test coverage. Thus, it is able to achieve better performance for a large number of test cases. The applied test vectors are generated implicitly by the system under test. This allows implementation independent testing of the TM. Furthermore, our approach allows reducing the number of test cases by test vector compaction.

We evaluated our proposed mechanism on a theoretical basis and by usage of a proof of concept implementation of a Java Card test environment based on a SystemC simulation. The results show that a significant performance gain can be achieved without loss of test coverage.

References

[1] *Java Card Platform Specification 2.2.2*, Sun Microsystems, Inc.
[2] Z. Chen, *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.
[3] M. Oestreicher, "Transactions in Java Card," in *ACSAC '99: Proceedings of the 15th Annual Computer Security Applications Conference*. Washington, DC, USA: IEEE Computer Society, 1999, p. 291.
[4] G. Lisimaque and P. Paradinas, "Method and device for updating information elements in a memory," US Patent 5,479,63, Dec. 26, 1995.
[5] W. Rankl and W. Effing, *Smart Card Handbook*. New York, NY, USA: John Wiley & Sons, Inc., 2003.
[6] *Java Card Platform Security*, Sun Microsystems, Inc.
[7] C. Marche and N. Rousset, "Verification of JAVA CARD Applets Behavior with Respect to Transactions and Card Tears," in *Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on*, Sept. 2006, pp. 137–146.
[8] E. Hubbers and E. Poll, "Reasoning about card tears and transactions in Java Card," Tech. Rep., 2004.
[9] J. Andronick, "Formally Proved Anti-tearing Properties of Embedded C Code," in *Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006. Second International Symposium on*, Nov. 2006, pp. 129–136.
[10] K. Rothbart, U. Neffe, C. Steger, R. Weiss, E. Rieger, and A. Muehlberger, "High level fault injection for attack simulation in smart cards," *Test Symposium, 2004. 13th Asian*, pp. 118–121, Nov. 2004.
[11] *Open SystemC Language Reference Manual*, IEEE, December 2005.
[12] S. Misera, H. T. Vierhaus, and A. Sieber, "Simulated fault injections and their acceleration in SystemC," *Microprocess. Microsyst.*, vol. 32, no. 5-6, pp. 270–278, 2008.

©2010 IEEE. Reprinted, with permission, from Informal Proceedings of 2010 15th IEEE European Test Symposium (ETS)

Bibliography

[1] W. Rankl and W. Effing, *Smart Card Handbook*. New York, NY, USA: John Wiley & Sons, Inc., 2003.

[2] P. Schaumont and I. Verbauwhede, "Domain-specific codesign for embedded security," *Computer*, vol. 36, pp. 68–74, April 2003.

[3] P. Gupta, "Hardware-software codesign," *Potentials, IEEE*, vol. 20, pp. 31–32, January 2001.

[4] J. Loinig, C. Steger, R. Weiss, and E. Haselsteiner, "Idea: Simulation Based Security Requirement Verification for Transaction Level Models," in *Engineering Secure Software and Systems* (l. Erlingsson, R. Wieringa, and N. Zannone, eds.), vol. 6542 of *Lecture Notes in Computer Science*, pp. 264–271, Springer Berlin / Heidelberg, 2011.

[5] J. Loinig, C. Steger, R. Weiss, and E. Haselsteiner, "Towards formal system-level verification of security requirements during hardware/software codesign," in *SOC Conference (SOCC), 2010 IEEE International*, pp. 388–391, September 2010.

[6] J. Loinig, P. Glatz, C. Steger, and R. Weiss, "Performance Improvement and Energy Saving Based on Increasing Locality of Persistent Data in Embedded Systems," in *Systems (ICONS), 2010 Fifth International Conference on*, pp. 175–180, April 2010.

[7] J. Loinig, C. Steger, R. Weiss, and E. Haselsteiner, "Java Card Performance Optimization of Secure Transaction Atomicity Based on Increasing the Class Field Locality," in *Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on*, pp. 342–347, July 2009.

[8] J. Loinig, C. Steger, R. Weiss, and E. Haselsteiner, "Identification and Verification of Security Relevant Functions in Embedded Systems Based on Source Code Annotations and Assertions," in *Information Security Theory and Practices. Security and Privacy of Pervasive Systems and Smart Devices*, pp. 316–323, Springer, 2010.

[9] J. Loinig, C. Steger, R. Weiss, and E. Haselsteiner, "Fast simulation based testing of anti-tearing mechanisms for small embedded systems," in *Test Symposium (ETS), 2010 15th IEEE European*, p. 242, May 2010.

[10] J. Loinig, C. Steger, R. Weiss, and E. Haselsteiner, "UNPUBLISHED.".

[11] P. Kocher, R. Lee, G. McGraw, and A. Raghunathan, "Security as a new dimension in embedded system design," in *DAC '04: Proceedings of the 41st annual Design Automation Conference*, (New York, NY, USA), pp. 753–760, ACM, 2004.

[12] D. S. Herrmann, *Using the Common Criteria for It Security Evaluation*. Boca Raton, FL, USA: CRC Press, Inc., 2002.

[13] Common Criteria, "Common Methodology for Information Technology Security Evaluation - Evaluation methodology," July 2009. Version 3.1 Revision 3 Final.

[14] Common Criteria, "Common Criteria for Information Technology Security Evaluation - Part 1-3," July 2009. Version 3.1 Revision 3 Final.

[15] F. Keblawi and D. Sullivan, "Applying the common criteria in systems engineering," *Security Privacy, IEEE*, vol. 4, pp. 50–55, March 2006.

[16] MAOSCO Limited, *MULTOS Developer's Reference Manual*.

[17] Sun Microsystems, Inc., *Java Card Platform Specification 3.0.1*, March 2009.

[18] Sun Microsystems, Inc., *Java Card Platform Security*.

[19] B. Schneier and A. Shostack, "Breaking up is hard to do: Modeling Security Threats for Smart Cards," in *Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology*, p. 19, USENIX Association, 1999.

[20] D. Mellado, E. Fernández-Medina, and M. Piattini, "A common criteria based security requirements engineering process for the development of secure information systems," *Comput. Stand. Interfaces*, vol. 29, no. 2, pp. 244–253, 2007.

[21] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, (New York, NY, USA), pp. 19–24, ACM, 2003.

[22] U. Neffe, K. Rothbart, C. Steger, R. Weiss, E. Rieger, and A. Muhlberger, "Energy estimation based on hierarchical bus models for power-aware smart cards," in *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, vol. 3, pp. 300–305, February 2004.

[23] J. Zambreno, A. Choudhary, R. Simha, B. Narahari, and N. Memon, "SAFE-OPS: An approach to embedded software security," *ACM Trans. Embed. Comput. Syst.*, vol. 4, no. 1, pp. 189–210, 2005.

[24] S. Park and S.-I. Chae, "A C/C++-based functional verification framework using the SystemC verification library," in *Rapid System Prototyping, 2005. (RSP 2005). The 16th IEEE International Workshop on*, pp. 237–239, June 2005.

[25] N. Bombieri, A. Fedeli, and F. Fummi, "Extended abstract: on the property-based verification in SoC design flow founded on transaction level modeling," in *Formal Methods and Models for Co-Design, 2005. MEMOCODE '05. Proceedings. Third ACM and IEEE International Conference on*, pp. 239–240, July 2005.

[26] C. Trummer, C. Kirchsteiger, C. Steger, R. Weiß and, M. Pistauer, and D. Dalton, “Automated simulation-based verification of power requirements for Systems-on-Chips,” in *Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2010 IEEE 13th International Symposium on*, pp. 8–11, April 2010.

[27] A. A. Sere, J. Iguchi-Cartigny, and J.-L. Lanet, “Automatic detection of fault attack and countermeasures,” in *WESS '09: Proceedings of the 4th Workshop on Embedded Systems Security*, (New York, NY, USA), pp. 1–7, ACM, 2009.

[28] E. Kylikowski, R. Scandariato, and W. Joosen, “Using Multi-Level Security Annotations to Improve Software Assurance,” in *High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE*, pp. 471–474, December 2008.

[29] F. Balarin, R. Passerone, A. Pinto, and A. Sangiovanni-Vincentelli, “A formal approach to system level design: metamodels and unified design environments,” in *Formal Methods and Models for Co-Design, 2005. MEMOCODE '05. Proceedings. Third ACM and IEEE International Conference on*, pp. 155 –163, July 2005.

[30] S. Morimoto, S. Shigematsu, Y. Goto, and J. Cheng, “Formal verification of security specifications with common criteria,” in *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, (New York, NY, USA), pp. 1506–1512, ACM, 2007.

[31] V. Lotz, V. Kessler, and G. Walter, “A formal security model for microprocessor hardware,” *Software Engineering, IEEE Transactions on*, vol. 26, pp. 702–712, August 2000.

[32] J. Ma, D. Zhang, G. Xu, and Y. Yang, “Model Checking Based Security Policy Verification and Validation,” in *Intelligent Systems and Applications (ISA), 2010 2nd International Workshop on*, pp. 1–4, May 2010.

[33] D. Kroening and N. Sharygina, “Formal verification of SystemC by automatic hardware/software partitioning,” in *Formal Methods and Models for Co-Design, 2005. MEMOCODE '05. Proceedings. Third ACM and IEEE International Conference on*, pp. 101–110, July 2005.

[34] K. Rothbart, U. Neffe, C. Steger, R. Weiss, E. Rieger, and A. Muehlberger, “High level fault injection for attack simulation in smart cards,” in *Test Symposium, 2004. 13th Asian*, pp. 118–121, November 2004.

[35] E. Fourneret, M. Ochoa, F. Bouquet, J. Botella, J. Jurjens, and P. Yousefi, “Model-Based Security Verification and Testing for Smart-cards,” in *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, pp. 272–279, August 2011.

[36] B. Chess and G. McGraw, “Static analysis for security,” *Security & Privacy, IEEE*, vol. 2, pp. 76–79, December 2004.

[37] M. Shanahan, “The event calculus explained,” in *Artificial intelligence today* (M. J. Wooldridge and M. Veloso, eds.), ch. The event calculus explained, pp. 409–430, Berlin, Heidelberg: Springer-Verlag, 1999.

[38] E. T. Mueller, *Commonsense Reasoning.* Morgan Kaufmann, 2006.

[39] E. T. Mueller, "Discrete Event Calculus Reasoner Documentation," tech. rep., IBM Thomas J. Watson Research Center, March 2008.

[40] J. Lee and R. Palla, "Classical logic event calculus as answer set programming," in *Working Notes of the Workshop on Answer Set Programming and Other Computing Paradigms*, 2008.

[41] N. Amalio, "Suspicion-Driven Formal Analysis of Security Requirements," in *Emerging Security Information, Systems and Technologies, 2009. SECURWARE '09. Third International Conference on*, pp. 217–223, June 2009.

[42] T. T. Tun, Y. Yu, C. Haley, and B. Nuseibeh, "Model-Based Argument Analysis for Evolving Security Requirements," in *Secure Software Integration and Reliability Improvement (SSIRI), 2010 Fourth International Conference on*, pp. 88–97, June 2010.

[43] A. Mana and G. Pujol, "Towards Formal Specification of Abstract Security Properties," in *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*, pp. 80–87, March 2008.

[44] M. Oestreicher, "Transactions in Java Card," in *ACSAC '99: Proceedings of the 15th Annual Computer Security Applications Conference*, (Washington, DC, USA), p. 291, IEEE Computer Society, 1999.

[45] K. T. A. F., "PKI for Machine Readable Travel Documents offering ICC Read-Only Access Version 1.1," tech. rep., International Civil Aviation Organization, 2004.

Appendix A

Additional Information

A.1 Axioms of the Event Calculus Domain Description

The list in Table A.1 is the complete set of Event Calculus axioms in the domain used in Section 6.1. *enables* and *disables* are events indicating activation and deactivation of a Security Functional Requirement (SFR). The associated fluent is *provided*. The events *requestsR* and *releasesR* indicate begin and end time points where a certain SFR is needed by a module while *requires* is the related fluent. The fluent *unsatisfied* is set whenever a module requires an SFR. It requires the event *satisfy* to happen to reset *unsatisfied*. *accesses* and *frees* represent events that setup respectively abort connections between modules. The related fluent is *connected*.

Axiom Nr.	Description
1	*enables* initiates *provides*.
2	*disables* terminates *provides*.
3	*requestsR* initiates *requires*.
4	*releasesR* terminates *requires*.
5	*requestsR* initiates *unsatisfied*.
6	*satisfy* terminates *unsatisfied*.
7	*accesses* initiates *connected*.
8	*frees* terminates *connected*.
9	*requestsR* only happens if a modules does not already require an SFR. A module must only require one SFR at a time.
10	*releasesR* only happens if a module actually requires the same SFR.
11	*releasesR* only happens if the module is not unsatisfied. If a module is still unsatisfied it must not release the requirement. Only solved requirements are released.
12	Every *requestsR* needs an *releasesR* with the same SFR.

Continued on next page

Axiom Nr.	Description
13	*accesses* only happens if the module is not already connected. Only one connection is allowed at a time.
14	*accesses* only happens when *provides* and *requires* are not active and *enables* does not happen. Modules which are currently active are not accessed. If SFR processing is currently in progress the module cannot handle a new incoming connection.
15	*enables* only happens when the module is not connected and not momentarily performing an *accesses*. Avoids circular connections.
16	*frees* only happens if the module is connected to another module.
17	$frees(M_1, M_2)$ only happens if M_2 is not unsatisfied. If a connected module is not satisfied the SFR might still in progress.
18	$frees(M_1, M_2)$ only happens if M_1 is not unsatisfied. A module cannot give up a connection before it is satisfied. This avoids accessing several modules sequentially where the last one provides the SFR.
19	A module does never access itself.
20	There are no multiple connections to a module at one time point. A module cannot handle multiple SFR requests at a same time.
21	*enables* happens only if the requirement is not already enabled.
22	*disables* happens only if the requirement is already enabled and if the module is not unsatisfied.
23 and 24	*satisfy* happens if a module is connected to a module that provides the right requirement and is not satisfied (2 axioms, see Section 3.2 for details).
25	A module does never provide its own requirement.
26	A module does never access a connected module.

Table A.1: Event Calculus Domain Descriptions of Security Verifications

A.2 Call Graph of the Java Card Application JavaPurse

Figure A.1 shows the call graph of the JavaPurse Java Card application. JavaPurse is a demo e-purse application in the Java Card Development Kit[1]. Figure A.1 shows the application method calls and the Java Card API calls only. Obviously the API functions also call operating system functions which are not shown in the figure. The figure should indicate how complex call graphs for Java Card applications can be and how time consuming a manual verification of such a call graph could become.

[1] http://java.sun.com/javacard/devkit/

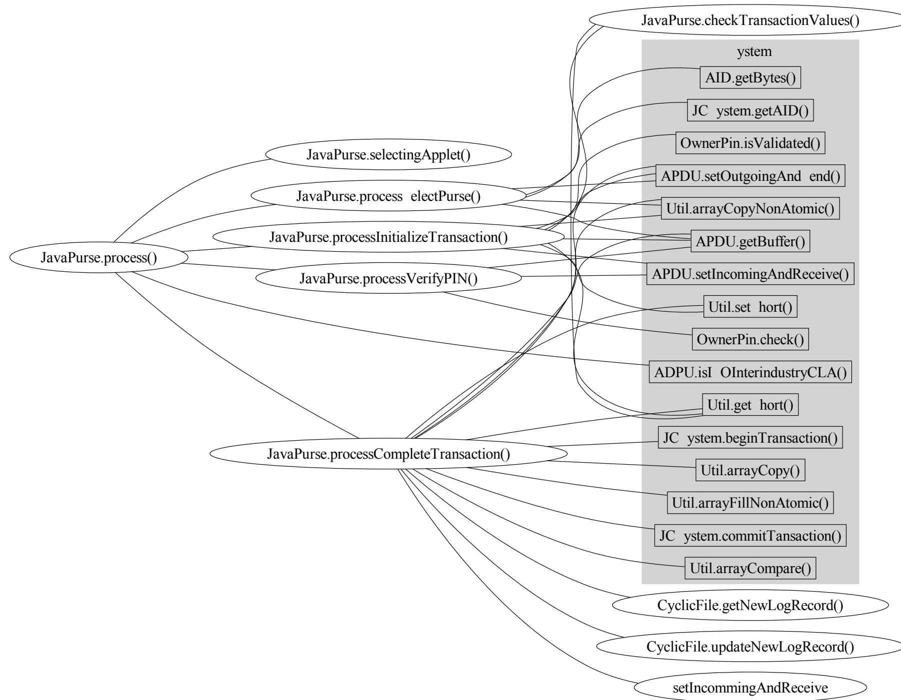


Figure A.1: Call Graph of JavaPurse

A.3 Event Calculus Symbols of Security Requirements

Functional requirements are represented as meta-information in the methodology of this work. This means that additionally to the modeling or implementation of these requirements (in the system's modules) textual information is added to the module's source code. This textual information includes a machine readable symbol uniquely identifying the security requirement. The requirements themselves are not bound to the methodology but can be defined for every system under development depending on use cases and threat scenarios. Table A.2 lists and explains the security requirements used in the case study discussed in Section 4.4.

Symbol	Explanation
r_key_usage	A secure cryptographic key has to be used to ensure confidentiality of any kind of data.
r_delete_key	After usage the cryptographic key it must not stay in the key buffer (see r_crypto_key_buffer). This requires to remove the key from the buffer which can be done by overwriting it with random data.
r_key_usage	The cryptographic key has to be used in an secure way that does not leak information about the key. In addition, the integrity of the key has to be ensured.

Continued on next page

Symbol	Explanation
r_read_key	The cryptographic key has to be read from a key store. Confidentiality and integrity of the key has to be ensured.
r_sec_copy	A copy routine that does not leak data while moving data in memory. This can be achieved by using a randomized and permuted order of memory accesses.
r_rng	Reliable random numbers have to be used. This random numbers must be true random numbers (no pseudo random numbers). They shall be statistically tested to ensure an applicable entropy.
r_key_storage	Cryptographic keys need to be stored in a key storage (e.g., in EEPROM). This key storage provides confidentiality while the keys are not actively used.
r_enc_keys	To provide confidentiality for keys themselves they need to be encrypted as well.
r_master_key	For any kind of encryption a cryptographic key is needed. For encryption of keys (r_enc_keys) a master key is needed.
r_derived_key	If a master key is used (r_master_key), this key is highly security sensitive if it is shared between different systems. To avoid that many systems are compromised if one secret master key gets known by attackers, master keys have to be derived. This ensures that each system has its own master key.
r_crypto_key_buffer	Cryptological operations shall have their dedicated memory area for cryptological keys.

Table A.2: Symbols and Explanations of Functional Security Requirements