Dipl.-Ing. Michael Lackner, BSc

Hardware/Software Codesign for Secure Java Cards

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

eingereicht an der

Technischen Universität Graz

Betreuer

Em.Univ.-Prof. Dipl.-Ing. Dr.techn. Reinhold Weiß

Institut für Technische Informatik

Graz, Juli 2014

EIDESSTATTLICHE ERKLÄRUNG

AFFIDAVIT

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. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Dissertation identisch.

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 indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present doctoral dissertation.

Datum / Date

Unterschrift / Signature

Kurzfassung

Java Karten sind kleine elektronische Geräte mit stark eingeschränkten Ressourcen. Ihre Hauptaufgabe ist es sicherheitskritische Daten zu verarbeiten und zu speichern. Heutzutage werden diese Karten in verschiedenen Kontexten eingesetzt wie elektronisches Geld, Zugangskontrollen oder elektronische Reisepässe. Die Anzahl dieser Karten wird sich in naher Zukunft, wegen der wachsenden Zahl von Smartphones und mobilen drahtlos verbundenen Geräten, immer weiter erhöhen. Java Karten werden in verschiedenen sicherheitskritischen Systemen eingesetzt wie bei Nahfeldkommunikation (Near Field Communication, Abkürzung NFC) und bei Fingerabdrucksensoren. Java Karten unterstützen verschiedene kryptographische Funktionen wie Checksummenberechnung, Entschlüsselung und Verschlüsselung von Daten mithilfe von geheimen Schlüsseln. Diese Schlüssel werden sicher in der Karte gespeichert und verwaltet. Die Sicherheit der Java Karten Anwendungen werden durch das Java Karten Sandbox Modell gewährleistet. Dieses Modell gewährleistet, dass keine Java Karten Anwendung, die auf der Karte ausgeführt wird, in der Lage ist illegale Operationen auszuführen. Das Modell basiert auf der Tatsache, dass vor der Anwendungsinstallation ein einmaliger Verifikationsprozess ausgeführt wird. Diese Verifikation stellt sicher, dass die Anwendung die Java Karten Spezifikation erfüllt und keine illegalen Operationen oder Speicherzugriffe enthält. Unglücklicherweise ist es mithilfe einer Laufzeitattacke möglich das Sandbox Modell zu umgehen. Eine Laufzeitattacke ist zum Beispiel ein Beschuss mit einem Laser oder ein kurzzeitiger Spannungsabfall in der Stromversorgung.

In dieser Arbeit werden neuartige Gegenmaßnahmen gegen verschiedene Attacken vorgestellt um eine zukünftig sichere Java Karte zu ermöglichen. Diese Gegenmaßnahmen werden entweder rein in Software oder mit Hardwareunterstützung ausgeführt. Neuartige digitale Schutzeinheiten werden in die Karte integriert um Speichergrenzen, Java Datentypen, Kontrollfluss und Datenintegrität während der Laufzeit zu überprüfen. Die Schutzeinheiten werden durch das Einfügen von neuen sicherheitsrelevanten Instruktionen in den Befehlssatz von zwei Java Karten Modellen angesprochen. Das erste Modell ist ein schnell ausführbares und leicht modifizierbares Systemmodell in der Hardwarebeschreibungssprache SystemC. Das zweite Modell ist ein taktgenaues Modell von Oregano Systems welches in VHDL programmiert ist. Der Prozessor der beiden Modelle ist ein 8-Bit 8051 kompatibler Prozessor. Beide Modelle werden während dieser Arbeit zur Erforschung von verschiedenen Software und Hardware Gegenmaßnahmen von Attacken eingesetzt. Die Verschiebung der Sicherheitsfunktionen von Software zur Hardware erhöht die Ausführungsgeschwindigkeit von Anwendungen auf der Karte. Die zusätzlich erforderliche digitale Logik für die neuen digitalen Schutzeinheiten bleibt relativ gering. Solch eine erhöhte Sicherheit für Java Karten wird benötigt, um zum Beispiel die sichere Installation und Ausführung von Anwendungen zu ermöglichen die durch den Kartenbenutzer installiert werden. Eine solche, vom Kartenbesitzer kontrollierte Karte, könnte ein nächstes profitables Geschäfts- und Anwendungsfeld für Java Karten sein.

Abstract

Java Cards are small resource constrained electronic devices which are able to process and store security-critical data. Nowadays, these cards are used in various contexts such as electronic banking, access control or electronic passports. The quantity of these cards will even increase in the near future because of the growing number of smart phones and mobile wireless connected devices. These devices have various built-in security-critical chips like subscriber identification module (SIM) cards, near field communication (NFC) chips, and fingerprint sensors. The Java Cards provide different functionality like hashing, decryption and encryption of data by secret keys. These keys are securely stored on the card. The security of the Java Card applets is ensured by the Java Card sandbox model. This model guarantees that no Java Card applet, which is executed on the card, is able to perform illegal code or data accesses. The model relies on the fact that a static applet verification process is performed. This process ensures that the applet fulfills the Java Card specification and contains no illegal operations or memory accesses. Unfortunately, a fault attack is able to change the Java Virtual Machine code and data which results in a circumvention of the sandbox model.

In this work novel attack countermeasures are explored to enable a secure future Java Card. To increase the overall Java Card security an on-card verified applet is protected by run-time security checks. These checks are either performed purely in software or with hardware support. Novel run-time protection units are integrated into the card to perform memory bound checks, Java data type checks, control flow checks, and data integrity checks. The protection units are addressed by inserting new security-sensitive instructions into the instruction set of two prototype smart card models. The first model is a fast executing and quickly modifiable system level model programmed in the hardware description language SystemC. The second one is a highly accurate register transfer level model from Oregano Systems programmed in the hardware description language VHDL. The processor of both models is an 8-bit 8051 processor. Both models are used for the design space exploration of various software and hardware security hardening techniques for a prototype Java Card. Moving the run-time security checks from software to hardware significantly increases the execution performance of the prototype. The additional digital logic required for the hardware protection units remains relatively small. Such increased security for Java Cards is needed, for example, to enable the secure installation and execution of any applet by the card user. Such a so called user centric ownership model could enable new profitable business fields and applications for Java Cards.

Acknowledgements

This thesis has been pursued as part of the CoCoon research project, which is a successful cooperation between NXP Semiconductors Austria GmbH and the Institute for Technical Informatics of Graz University of Technology. Considering all these organizations and their employees that supported me during the accomplishment of my thesis, I would like to thank them all, but of course I am only able to mention a few.

First of all, I would like to thank Prof. Reinhold Weiss for providing this possibility to work in the hardware/software codesign research group at the Institute for Technical Informatics. Especially, I would thank for the helpful comments and advice given during the review phases of this work. Furthermore, I would like to express my gratitude to Christian Steger for his thoughtful supervision of the project and support concerning the academic publishing process. Our industry partners provided us with continuing insight into state-of-the-art products and industrial design environments. In this regard I would like to thank Johannes Loinig for sharing his knowledge concerning Java Cards and his valuable comments.

Also, I would like to give my thanks to Ernst Haselsteiner for his real interest in the topics of this thesis and his supportive comments especially at the beginning of the project. Special thanks go to my PhD colleagues Massimiliano Zilli and Wolfgang Raschke for their in-depth review of this work and their helpful comments on various publications. Also, I would like to thank Christian Kreiner for his various advices that were a tremendous help during my PhD. Furthermore, my thanks are due to Stefan Orehovec and Erik Gera-Fornvald for their help to improve various papers. In this regard I would also like to give my thanks to my project partner Reinhard Berlach for the great cooperation during the accomplishment of the CoCoon project. I would also like to thank all students contributing to this project: Michael Hraschan, Michael Irauschek, Stephan Oberauer, Mario Wagner, and Christian Zajc.

Graz, July 2014 Michael Lackner

Extended Abstract

A Java Card is a standardized virtual machine running on a resource constrained smart card. The cards are used in a wide range of applications such as banking, transport, access control, and passports. All these applications rely on the security and reliability provided by these cards. To ensure Java Card security, today's card users are not allowed to install their own applets on the card. In future, it is expected that users will be allowed to perform post-issuance installation of their own applets. This post-issuance installation of user applets will enable new business fields and use cases like, for example, an applet store for Java Cards. On the other hand, new security issues and threats will emerge. Therefore, the next major challenge for Java Card platform design is to cover security concerns related to post-issuance installation of user applets on the card.

Current Java Card security is provided by a sandbox security model. This sandbox separates applets from each other and protects their security-critical code and data. Therefore, the sandbox thwarts all illegal access to fields and objects which are not part of the current applet context. The whole sandbox concept relies on the fact that the applet and its corresponding bytecodes are well formed and do not violate the Java Card specification. An applet which does not fulfill the specification is called a malicious applet and is able to perform logical attacks against the virtual machine. Logical attacks are, for example, an overflow of different memory buffers to illegally overwrite the current Java method return address. Such threats of logical attacks are currently thwarted by performing a static off-card applet verification in a secure environment. After a successful verification a signature is calculated over the applet by a secret key. This secret key is only known by authorized companies and authorities.

Unfortunately, attackers are able to mutate a successfully verified and installed applet into a malicious one. This mutation is done by performing a run-time fault attack (e.g., laser beam, clock/power supply glitch) to change the standard control and data flow of an applet to the advantage of an attacker. Such fault attacks can be counteracted by dynamic checks during the run-time of an applet.

Therefore, to enable the next generation of Java Cards with their high security requirements, new static and dynamic security features are needed. The CoCoon project[1], which included and financed this work, aims at enabling the post-issuance installation of applets by the card user. This so called user-centric ownership model has to counteract two main security threats.

1. Logical attacks are performed by an attacker who manipulated a Java Card applet

[1] *CoCoon: Codesign for Countermeasures against Malicious Applications on Java Cards*, collaborative research project of the Graz University of Technology, NXP Semiconductors Austria GmbH. Funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FIT-IT contract FFG 830601.

before its installation. Such a manipulated applet can be thwarted by an *on-card verification* during the install process.

2. Fault attacks can change the data and control flow of an applet during run-time. Such fault attacks are thwarted by new *run-time security checks* which enable a defensive virtual machine.

Both security concepts, *on-card verification* and *run-time security checks*, are illustrated in Figure 1. The on-card verification and the run-time security checks must be designed in respect to the constrained available resources on a Java Card. In particular for the run-time security checks, a tradeoff must be found between the performance/memory consumption and the required level of security. By moving security checks from software to hardware it is possible to improve the execution speed of the run-time checks. This security-related hardware/software codesign approach enables a significant performance improve against an approach where all checks are only performed in software.

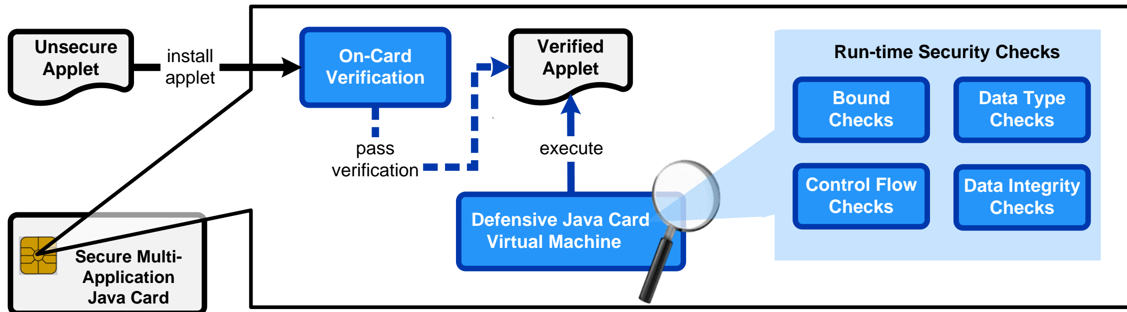


Figure 1: To enable the user-centric ownership model each applet must be verified during the install process. A successfully verified applet is protected by a defensive Java Card virtual machine. This defensive virtual machine performs run-time checks to counteract fault attacks.

This thesis presents security mechanisms to enable the user-centric ownership model for future Java Cards. Based on a hardware/software codesign process, selected security mechanisms are accelerated by dedicated microarchitectural hardware support. The starting point of this work is the research on current state of the art attacks and security threats in the context of Java Cards and the user centric ownership model. The next step is the creation of countermeasure concepts and the assignment of the counteracted threats. Logical attacks are counteracted by an on-card applet verifier[2]. Fault attacks are counteracted by additional run-time checks. The most promising countermeasures are selected and evaluated in a countermeasure exploration phase. The countermeasures are evaluated in terms of their performance impact, memory consumption, detection rate and detection latency. For example, to counteract type confusion attacks three different countermeasure concepts were found: storing[3] the type information for each value during run-time, separating[3] the values to different memory areas depending on the data type,

[2] *Memory-Efficient On-Card Byte Code Verification for Java Cards*, 1st Workshop on Cryptography and Security in Computing Systems (CS2'14), Vienna, Austria, 2014

[3] *Towards the Hardware Accelerated Defensive Virtual Machine - Type and Bound Protection, 11th Smart Card Research and Advanced Applications (CARDIS'12), Graz, Austria, 2012*

and obfuscating[4] the values with different keys depending on the data type.

The next research topic is the effective integration of the selected countermeasures into the virtual machine. For the integration, an additional security layer[5] is added into the virtual machine. This security layer handles all accesses to security-critical memory regions of the virtual machine. The security layer provides a defined security application programming interface.

On the resource constrained Java Cards the impact of run-time countermeasures on the performance is very critical. Executing a lot of software countermeasures slows down the execution performance too much. Such a slow down could make the card useless for industrial use cases which have very strict execution time requirements. Therefore, this work shifts the above described run-time countermeasures from software to hardware during a hardware/software codesign phase. This is done by adding new defensive hardware protection units[6] into the smart card. These protection units are responsible for protecting memory bounds, data types, control flow, and data integrity of the virtual machine. This hardware acceleration shows an impressive increase in speed to the execution time for different Java bytecodes. Also the additional logic area needed on an FPGA development board is quite low.

For a fast evaluation of future software and hardware countermeasures against fault attacks a fault emulation environment[7] is needed. This environment is especially designed to test the security of Java Virtual Machines against run-time attacks on the security-critical memory regions of the virtual machine. During a case study a simplified Java wallet applet has been attacked to gain access to money transfer functionality without knowing the valid PIN.

To summarize, this dissertation presents selected countermeasures against current state of the art security threats and attacks against the Java Card virtual machine. These threats are counteracted by countermeasures during a hardware/software codesign process. Selected countermeasures are accelerated by microarchitectural support of the Java Card. Furthermore, a fault emulation environment is proposed to evaluate the Java Card security against fault attacks. The additionally integrated countermeasures are needed to enable the user-centric ownership model for future Java Cards.

---

[4] *Countering Type Confusion and Buffer Overflow Attacks on Java Smart Cards by Data Type Sensitive Obfuscation*, 1st Workshop on Cryptography and Security in Computing Systems (CS2'14), Vienna, Austria, 2014

[5] *A Defensive Virtual Machine Layer to Counteract Fault Attacks on Java Cards*, 7th Workshop in Information Security Theory and Practice (WISTP'13), Heraklion, Greece, 2013

[6] *A Defensive Java Card Virtual Machine to Thwart Fault Attacks by Microarchitectural Support*, 8th International Conference on Risks and Security of Internet and Systems (CRiSIS'13), La Rochelle, France, 2013

[7] *A Fault Attack Emulation Environment to Evaluate Java Card Virtual-Machine Security*, 17th Euromicro Conference on Digital Systems Design (DSD'14), Verona, Italy, 2014)

Contents

1 Introduction **1**

1.1 Motivation . . . 1
1.2 Security Threats on Java Cards . . . 2
1.2.1 Java Card Virtual Machine Security Concept . . . 3
1.3 Hardware/Software Codesign for a Secure Future Java Card . . . 4
1.3.1 The CoCoon Project . . . 4
1.3.2 Problem Statements . . . 5
1.3.3 Contributions of this Thesis . . . 6
1.3.4 Thesis Structure . . . 7

2 Related Work **8**

2.1 Java Card Security Overview . . . 8
2.2 Java Card Attack Overview . . . 8
2.2.1 Attacks on the Java Card Sandbox . . . 9
2.3 Overview of Java Card Countermeasures . . . 11
2.3.1 Countermeasures to Harden the Java Card Sandbox . . . 11
2.3.2 Drawbacks of Current Countermeasures and Discussion . . . 12
2.4 Summary and Difference to State-of-the-Art Countermeasures . . . 13

3 Design Evaluation for a Secure Future Java Card **14**

3.1 Overview . . . 14
3.2 Run-time Checks . . . 16
3.2.1 Run-time Security Policy . . . 16
3.2.2 Type Confusion Countermeasure Designs . . . 17
3.2.3 Memory Overflow Countermeasure Designs . . . 19
3.3 Countermeasure Integration into Java Card Architecture . . . 19
3.4 Static On-Card Applet Verification . . . 20
3.5 Hardware Support for Run-time Security Checks . . . 20
3.6 Evaluate Java Card Security by Fault Emulation . . . 22

4 Results and Case Studies **23**

4.1 Evaluation Platforms . . . 23
4.1.1 Tool Chain . . . 23
4.1.2 System Level Model . . . 24
4.1.3 Register Transfer Level Model . . . 24
4.2 Case Study on Type Confusion Countermeasures . . . 25
4.2.1 Type Storing . . . 25
4.2.2 Type Separating . . . 28
4.2.3 Type Obfuscating . . . 28
4.2.4 Hardware Support for Run-time Security Checks on 8051 CPU . . . 29

4.2.5 Hardware and Performance Overhead 31
4.3 Memory Efficient On-Card Verification 32
4.4 Fault Emulation Case Study 34

5 Conclusions and Future Work **36**
5.1 Conclusions 36
5.2 Directions for Future Work 37
5.2.1 Fault Attack Countermeasures on Lower Java Card Layers 37
5.2.2 Side Channel Leakage Countermeasures 38

6 Publications **39**
6.1 Towards the Hardware Accelerated Defensive Virtual Machine - Type and Bound Protection 41
6.2 A Defensive Virtual Machine Layer to Counteract Fault Attacks on Java Cards . . 56
6.3 A Defensive Java Card Virtual Machine to Thwart Fault Attacks by Microarchitectural Support 72
6.4 Countering Type Confusion and Buffer Overflow Attacks on Java Smart Cards by Data Type Sensitive Obfuscation 80
6.5 Memory-Efficient On-Card Byte Code Verification for Java Cards 86
6.6 A Fault Attack Emulation Environment to Evaluate Java Card Virtual-Machine Security 90

References **98**

List of Figures

1 Overview of the secure future Java Card concept v

1.1 Number of worldwide mobile subscriptions 1
1.2 Number of shipped secure smart card devices 2
1.3 Security threats and attack scenarios on Java Cards 3
1.4 Future multi-application Java Card 4
1.5 Security topics inside the CoCoon research project 5

2.1 Overview of various Java Card attack points 9
2.2 Memory overflow and underflow attacks on a Java Card 10
2.3 Run-time control flow change by a fault attack 11

3.1 Publication overview for a secure future Java Card 15
3.2 Run-time security policy of a secure future Java Card 17
3.3 Type confusion countermeasure designs 18
3.4 Data integrity and type policy 20
3.5 Statically performed applet verification process. 21
3.6 Additional processor instructions are inserted into the Java Card 21
3.7 Shift run-time security checks to hardware 22
3.8 Overview of the fault emulation environment to evaluate Java Card security 22

4.1 Java Card tool-chain for the evaluation of this work. 24
4.2 High level SystemC model of the 8051 processor 25
4.3 Low level VHDL model of the 8051 processor 26
4.4 Overview of type confusion countermeasure publications 26
4.5 Implementation variants to store the type information during run-time. 27
4.6 Type separating pre-processing step and typed bytecodes overview 29
4.7 Counteract type confusion by type obfuscating countermeasure 30
4.8 Run-time security checks integration with hardware support 31
4.9 Performance overhead overview of the HW accelerated run-time checks 33
4.10 Concept of the statically performed on-card verification process 33
4.11 Main parts of the fault emulation environment 34

5.1 Directions for future work on Java Card security 37

6.1 Publications overview for a secure future Java Card 40

List of Tables

3.1 Comparison between type storing, type obfuscating, and type separating 19

4.1 Performance overhead of the type storing countermeasures 28
4.2 Additional digital hardware resources to enable the hardware protection units . . . 32
4.3 Performance overhead overview of the hardware accelerated run-time checks 32
4.4 Fault emulation speedup 35
4.5 Fault emulation hardware overhead 35

List of Abbreviations

AHB	AMBA High-Performance Bus
AMBA	Advanced Microcontroller Bus Architecture
API	Application Programming Interface
BA	Bytecode Area
BB	Basic Block
BPU	Bound Protection Unit
FPU	Control Flow Protection Unit
CPU	Central Processing Unit
DUT	Device Under Test
D-VM	Defensive Virtual Machine
EEPROM	Electrically Erasable Programmable Read-Only Memory
FIFO	First-In-First-Out
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
HW	Hardware
IPU	Data Integrity Protection Unit
JH	Java Heap
JPC	Java Program Counter
JVM	Java Virtual Machine
LA	Logical Attack
LVs	Local Variables
NFC	Near Field Communication
OS	Operand Stack
PC	Personal Computer
RAM	Random-Access Memory
ROM	Read-Only Memory
SW	Software
SoC	System on Chip
TPU	Type Protection Unit
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VM	Virtual Machine

Glossary

Hardware/Software Codesign
"Hardware/software codesign investigates the concurrent design of hardware and software components of complex electronic systems. It tries to exploit the synergy of hardware and software with the goal to optimize and/or satisfy design constraints such as cost, performance, and power of the final product. At the same time, it targets to reduce the time-to- market frame considerably."[1]

Security
"The term security especially addresses system properties that need to be attained in the presence of malicious threats. Usually, security is either defined in terms of these threats and the objectives to achieve or as a composite of other characteristics. The latter is often called the CIA approach, because secruity is considered as being composed of the attributes confidentiality, integrity, and availability."[2]

Fault Attack
A *fault attack* is performed during run-time when the system performs security-critical operations. As described in [3] there are several ways to induce a fault like a glitch attack, temperature attack, light attack, and magnetic attack. The aim of a fault attack is to change the normal execution of the system to gain access to security-critical information.

Integrity
"Integrity means that assets can be modified only by authorized parties. In this context, modification includes writing, changing, changing status, deleting, and creating."[4]

Defensive Java Card Virtual Machine
Currently each applet is verified by an off-card verification process. Therefore, no additional run-time security checks are needed to ensure that the applet contains no illegal operations. Unfortunately, the data and program flow of such a verified applet can be manipulated by a fault attack. Therefore, to increase the overall security of a Java Card against fault attacks a so called *defensive Java Card Virtual Machine* is needed. This *defensive Java Card Virtual Machine* performs additional security checks during the run-time of a Java Card applet.

Chapter 1

Introduction

1.1 Motivation

By around the end of 2014, the number of mobile phones on the planet will exceed the number of people (currently 7.1 billion). Every mobile phone has a subscriber identity module (SIM) or universal SIM (USIM) that is authenticated by the mobile service provider. These cards perform different cryptographic operations to enable this authentication. The high demand of specialized secure devices will most probably continue in the near future. Their high levels of security will be needed for different use cases such as automotive (e.g., connected/self-driving vehicles), government (e.g., electronic health record) or military (e.g., drones, autonomous robotic vehicles).

The number of mobile subscriptions, shown in Figure 1.1, is due to increase continuously from the year 2014 and will be over 9,000 million in the year 2019. In this time period the number of smart phones will have doubled. The increase in smart phone ownership is an important fact as specialized secure devices are used much more often in smart phones (e.g., fingerprint sensor, near field communication (NFC) chip) than in cheap basic phones. This means that one smart phone can contain several secure chips.

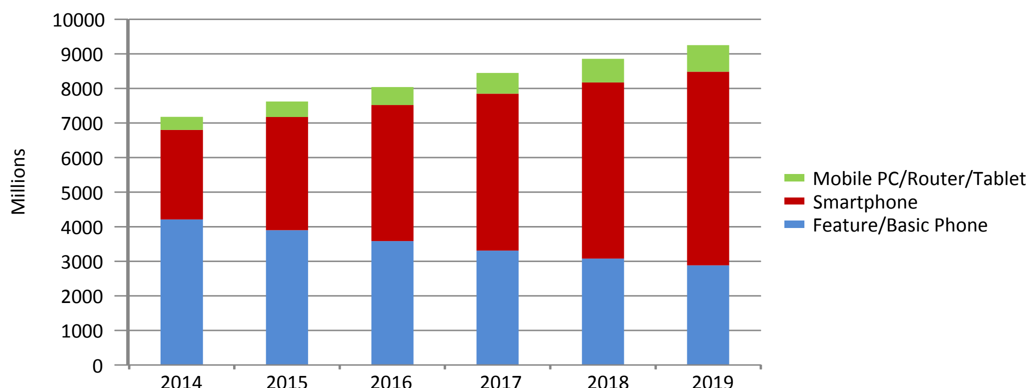


Figure 1.1: Number of worldwide mobile subscriptions for different devices like mobile personal computer, smart phone, and basic phone (obtained with modifications from [5]).

A very successful variant of these secure devices are the so called Java Cards. On these cards runs a Java Card Virtual Machine (VM) which executes different applications.

These applications are, for example, deployed in the context of electronic passports, credit cards, customer loyalty cards, transport, and access control. A famous use case of the Java Cards is in the field of contactless communication. The exponential growth of this business market is shown in Figure 1.2 with quantities of around 650 million units.

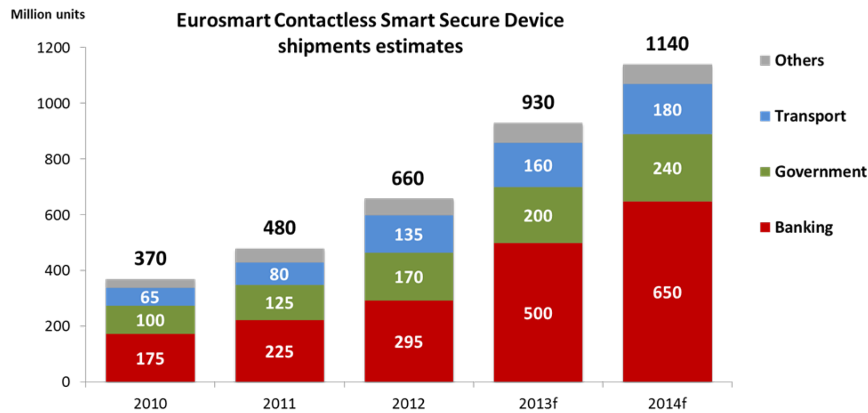


Figure 1.2: Forecasts for the shipment number of contactless smart secure devices (obtained with modifications from [6]).

1.2 Security Threats on Java Cards

The Java Card consists of the software VM which is executed on a smart card. This smart card consists of different digital parts (e.g., processor, volatile/non-volatile memory, bus). During the execution of the Java Card VM various data is transmitted between these digital parts. Data is represented on the electronic circuits by electrical charges. By firing a high-energy optical pulse (laser attack) onto the digital parts it is possible to change the electrical charges and therefore the data which is processed by the VM. This means that it is possible to change the behavior of the VM during run-time using such a laser attack. Besides a laser attack, other security threats for Java Cards also exist which are shown in Figure 1.3.

- **Observation Attack:** By an observation of the power consumption, electromagnetic radiation, or execution time it is possible to draw conclusions on the internal behaviour of the chip.
- **Invasive Attack:** The chip is reverse engineered by optical analysis and a removal of the chip layer.
- **Fault Attack:** The normal chip execution is disturbed by abnormal high/low clock frequency, supply voltage, temperature, or a laser attack. By using a fault attack an adversary is, for example, able to overjump instructions or change data in memories.
- **Logical Attack:** An attacker uploads and installs a malicious applet. This malicious applet performs illegal operations like buffer overflows or type confusion attacks to illegally receive or manipulate security-critical data.

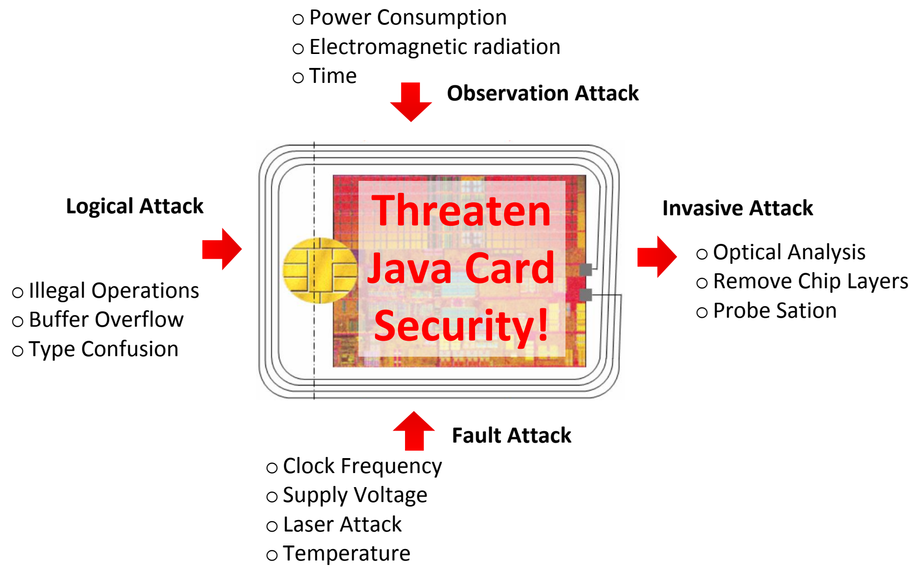


Figure 1.3: The security of the data and code stored on Java Cards is threatened by different security threats and attack scenarios (obtained with modifications from [7] [8]).

All these different security threats must be considered during the design and implementation phases of a secure future Java Card. This work shows countermeasures that can be used against the threat of run-time fault attacks and logical attacks.

The Java VM runs on a resource constrained smart card. These cards have to withstand logical attacks and physical attacks. To counteract these attacks it is necessary to integrate countermeasures into the hardware (HW) and software (SW). Unfortunately these countermeasures, particularly in SW, decrease the execution performance of the card or need a lot of memory for intermediate calculations. These cards are used in mass production which means that millions or billions of such cards are produced. Even a small amount of money saved per device results in a large overall saving. Therefore, this work shows various run-time countermeasures which have a low impact on the execution performance and HW overhead and are able counteract different proposed attack scenarios.

1.2.1 Java Card Virtual Machine Security Concept

The Java Card security is ensured by a sandbox concept. This sandbox ensures that each applet is somehow trapped in its own memory regions. Therefore, applets are not able to illegally access code or data of other applets installed on the card. This sandbox concept relies on the fact that each applet fulfills the Java Card VM specification [9, 10]. This specification defines, for example, that bytecodes are not allowed to access data outside of reserved memory regions by a buffer overflow attack. The compliance of an applet with the specification is currently checked by an off-card verification step in a secure environment. This means that, for example, such buffer overflows are currently checked by the static byte code verification process but not during run-time.

These run-time attacks are performed by firing a laser onto the card, dropping the supply voltage, applying an illegally high or low clock frequency or a strong electro magnetic field. With such attacks it is possible to overjump or change the execution behaviour

of Java bytecodes. Therefore, an adversary is able to change the control and data flow of a Java applet to his will. By such a manipulation it is possible to bypass security checks during run-time. Such checks protect, for example, money transfer functions from users who do not know the secret personal identification number (PIN) of a credit card. An adversary which removes the PIN check functionality is able to withdraw or transfer money to the bank account of an adversary.

1.3 Hardware/Software Codesign for a Secure Future Java Card

This chapter provides an overview of the research project in which this thesis is established. Furthermore, this chapter explains the problem statements and the contributions and significance of this work.

1.3.1 The CoCoon Project

This thesis is part of the "Codesign for Countermeasures against Malicious Applications on Java Cards" (CoCoon) collaborative research project, shared between the Institute for Technical Informatics at the Graz University of Technology and NXP Semiconductors Austria GmbH. This combined effort targets the HW/SW codesign of a secure future Java Card. These future cards should support the user centric ownership model (UCOM). With the UCOM every user is able to upload any applet onto the card without any security problems. Therefore, with the help of the UCOM only one card is needed for different applications as shown in Figure 1.4.

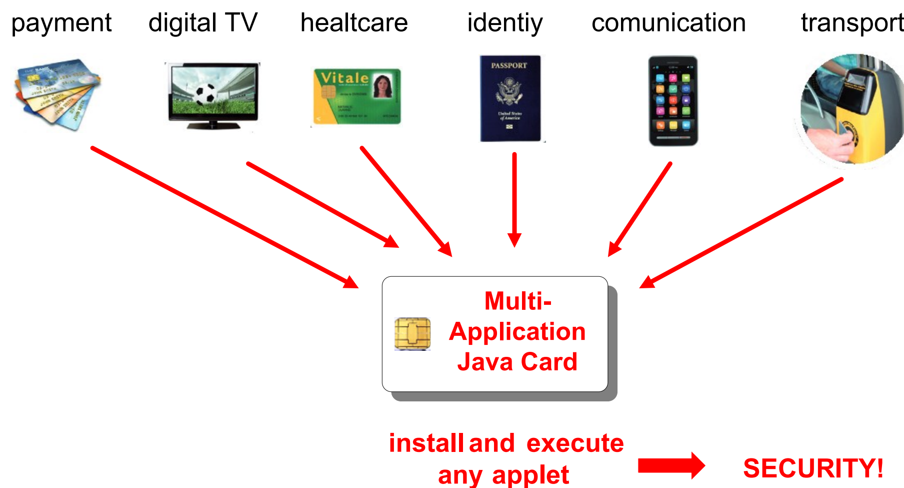


Figure 1.4: In the UCOM every user is able to upload applications for different contexts. Therefore, the overall security of such future multi-application cards must be increased to counteract different security threats (obtained with modifications from [11]).

The main goal of the project is to increase the security of Java Cards as much as possible with the least amount of expense possible. A great benefit of the CoCoon project is the fact that NXP Semiconductors Austria GmbH is the developer of the Java Card

SW and the smart card HW. Other Java Card manufacturers only produce the SW but without the HW and vice versa. This holistic view of the Java Card in the CoCoon project enables a HW/SW codesign process which particularly focuses on security.

An overview of the complex field of HW/SW codesign is presented in [12]. Another overview is given in the lecture notes from the ETH Zürich [13]. In general, as described in [13], the HW/SW codesign process can be seen as the integrated design of HW and SW components. During this process different alternative designs are planned, compared and evaluated during a design space exploration phase.

An overview of the different security topics inside the CoCoon project is shown in Figure 1.5. The CoCoon approach covers *security checks at the Java layer*, the *Java Card firewall* mechanism, *run-time (defensive) VM mechanisms*, static *bytecode verification*, and *HW support* of these security features. This work will concentrate on the last three topics. This thesis performs HW/SW codesign to enable a secure future Java Card for the UCOM.

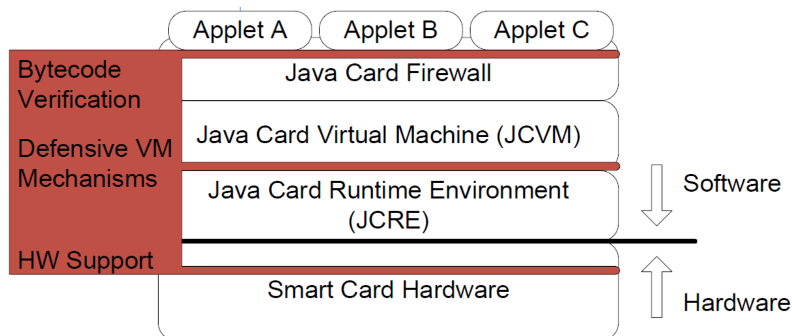


Figure 1.5: Overview of the different security topics inside the CoCoon project (obtained with modifications from [14]).

1.3.2 Problem Statements

A Java VM runs on a Java Card which can be seen as a SW processor running on the real smart card processor. This abstraction adds an additional execution time overhead. This overhead is very critical especially for Java Cards which are used as a mass product. Therefore, appropriate run-time countermeasures must be designed to consume as few additional resources (computational power and memory) as possible. Furthermore, these countermeasures should counteract as many attacks as possible. State-of-the-art countermeasures on Java Cards are only implemented in SW and not accelerated by special HW mechanisms as proposed in this work. This means that we shift functionality from SW into HW during a HW/SW codesign phase shown in this work. Compared to currently proposed SW countermeasures against fault attacks, the following deficiencies can be identified:

- Currently proposed countermeasures are executed in SW which slows down the execution performance.
- Synergy effects from designing HW and SW in parallel for Java Cards is less researched especially from a security point of view.

- Countermeasure designs are quite well known by industrial Java Card manufacturers but not published in the scientific community.
- Lack of knowledge of how and which countermeasures are needed during run-time leads to a long countermeasure design space exploration phase or inefficient countermeasures.
- Missing knowledge about appropriate countermeasure designs creates insecure systems which can lead to economic, material, and human losses.

These limitations have been addressed by the creation and evaluation of different countermeasure proposals. Early evaluations are based on memory consumption, run-time overhead, detection latency, and the general security increase. To reduce the computational overhead the countermeasures are accelerated by HW support. The HW acceleration mechanisms are developed during a HW/SW codesign process. For this acceleration a smart card prototype was used which is implemented at different abstraction levels. For a fast design space exploration a high-abstracted transaction level model is used. For a finer computational and HW overhead analysis a low-abstracted register transfer level model is used. The countermeasures are seamlessly integrated into the VM by adding a new security related SW layer into the Java Card architecture.

1.3.3 Contributions of this Thesis

In summary, this thesis provides contributions to the following fields:

1. **Identify Security Threats on Future Java Cards and Create New Concepts of Countermeasures:** Logical attacks are counteracted in this work by a statically performed bytecode verification process during the install time of an applet. During run-time additional checks were added to counteract fault attacks. Different security holes were identified and thwarted by additional checks. Only selected checks are performed during run-time to counteract as many attacks as possible with as little memory and performance overhead as needed. The new run-time checks are based on memory bound checks, data type checks, control flow checks, and data integrity checks. Different countermeasure designs are explored based on a design space exploration phase. For example to counteract type confusion attacks we proposed three general ideas: type storing, type separating, and type obfuscating.
2. **Hardware Support for Countermeasures:** SW checks add an additional memory and performance overhead. Therefore, in this work we shifted different run-time checks from SW to specially designed HW units. The units are namely the memory bound protection, data type protection, control flow protection, and data integrity protection units. An execution speedup is reached by performing the checks of the HW units in parallel to the normal processor operations. To communicate with the HW units new specially designed processor instructions were added. These new instructions are used to access the security-critical memory regions of the VM. This novel approach enables a significant improvement in the execution speed of the run-time checks compared to a full SW implementation. The performance and HW overhead of the additional protection units were evaluated on a Java Card prototype implementation running on an FPGA board.

3. **Seamless Integration of Countermeasures into Java Card:** A novel security abstraction layer was added into the Java Card. This layer is used to access the security-critical memory regions of the VM. All run-time security checks of this work are added into this layer and can be easily enabled, disabled, or modified. The main benefits of the new layer are: high flexibility by partitioning security features from SW to HW or from one SW implementation to another; increased maintainability of security checks; accelerating further development of new security features for the VM.

1.3.4 Thesis Structure

Section 2 describes the related work in the field of Java Card security. Various attack scenarios against the Java Card sandbox model are presented such as memory overflow or Java data type confusion. Furthermore, an overview of state-of-the-art countermeasures and the drawbacks of these countermeasures are discussed. This section finishes with a summary of the main improvements to the countermeasures that have been developed in this thesis compared to state-of-the-art countermeasures.

Section 3 presents an evaluation of the design of a secure future Java Card. The run-time security policy of such a future card is discussed. Furthermore, various general countermeasure designs are introduced which fulfill the proposed run-time security policy. Also the integration of the countermeasures into the Java Card architecture and static on-card applet verification is discussed. Furthermore, the fault emulation environment of this work is presented.

Section 4 shows the results of HW, performance, and memory measurements on two evaluation platforms. The evaluation platforms are first described with their changes for the new run-time security checks. Followed by a case study on different type confusion countermeasure implementation variants. The acceleration of the run-time security checks by newly added HW protection units which are integrated into an 8051 processor is described. Followed by a discussion and presentation of the static on-card applet verification algorithm. Furthermore, the fault emulation environment is discussed in more detail followed by a case study of an attacked Java wallet applet.

Section 5 draws conclusions and future works from this thesis. Directions for future work are presented which could focus in the direction of HW/SW codesigned countermeasures against side channel attacks. Furthermore, the integration of countermeasures into deeper Java Card layers like the operating system or directly into the HW are suggested.

Section 6 presents the collection of publications which were published during this thesis.

Chapter 2

Related Work

This section first presents an overview of the Java Card technology with a particular focus on security. Furthermore, an overview of attacks on Java Cards is given followed by countermeasures against these attacks. Finally, a summary of this section is shown with the main differences of this thesis compared to related work.

2.1 Java Card Security Overview

Security requirements of embedded systems like Java Card are continuously increasing. These security requirements also have to be considered during the design phase of a Java Card [15]. Java Cards have to withstand various attacks (e.g., microprobing, SW attacks, eavesdropping, fault generation) to protect the security-critical data stored on them [16].

The current Java Card security concept relies on the fact that every Java Card applet is verified. This verification counteracts logical attacks against the Java Card sand-box model. Nowadays, this verification is performed off-card in a secure environment. On commercially available Java Cards like banking cards, the card user is not able to install new applets. To install new applets the card user would need to know a secret key which is only known by authorized companies and authorities. With this secret key it is possible to create a valid signature which is checked during the install process of an applet on the card.

The next big shift in the Java Card market will be the user centric ownership model. In this model the card user is able to install any applet he wants onto the card. Such a shift also increases the security requirements for Java Cards to counteract logical attacks by an on-card verification process. Furthermore, the Java Card must persist in an environment where attackers perform different attacks on the card. These attacks are, for example, side channel analysis, glitch attacks, and laser attacks.

2.2 Java Card Attack Overview

The Java Card has various contact or contactless physical interfaces to communicate with other electronic devices. Furthermore, over the chip surface physical quantities can be measured or the internal processing of the chip can be disturbed by fault attacks shown in Figure 2.1. By the *electrical measurement and analysis* of the power consumption of the

chip an attacker can gain knowledge of internal processes [17]. Over the *communication* interface SW viruses can be uploaded onto the Java Card [18]. By *electrical stimulation* (e.g., digital clock or power supply glitch) an adversary is able to change the data which is read out or written into volatile and non-volatile memories [19]. Over *electrical stimulation* or *energy and particle exposure* an attacker can manipulate internal calculations. Such manipulations during cryptographic operations can lead to the loss of secret cryptographic keys [17]. Furthermore, there is a threat of *reverse-engineering* and *physical manipulation* of the silicon chip [8]. Also so called side channel attacks, performed by *electro-magnetic radiation analysis* [20, 21, 22], are a serious security threat and a very active field of research.

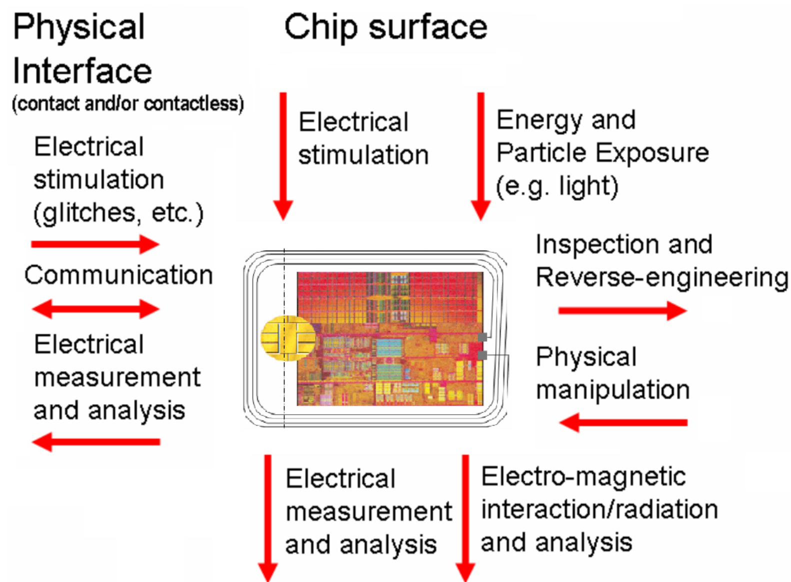


Figure 2.1: A Java Card has several attack points such as physical interfaces or the measurement or manipulation by physical quantities (obtained with modifications from [7]).

2.2.1 Attacks on the Java Card Sandbox

An attacker is able to break the sandbox model of the Java VM using a malicious applet which does not fulfill the Java Card specification [9, 10]. Such a malicious applet can be created by an applet manipulation before the upload onto the card [23] or by a fault attack during run-time. In literature four general sorts of attack are described which are memory overflow attacks, control flow attacks, type confusion attacks, and data integrity attacks.

Memory Overflow Attacks

For every invoked Java method a new so called Java Frame is created. When a Java method returns then the execution flow returns to the previous frame. This frame data generally consists of three fixed sized memory regions:

- Operand Stack (OS): The OS is used by the VM for most logical and arithmetic operations. Values are pushed and popped onto the top element of the OS.
- Local Variables (LVs): The LVs are freely accessible like standard registers in a processor.
- Frame Data: The frame data consists of internal data of the VM like the return address to the previous frame. Note that the specific internal frame data is implementation dependent and not specified inside the VM specification.

An attacker is able to provoke an overflow of the OS or LVs in order to illegally access data outside the reserved memory regions. An overview of such overflow attacks is shown in Figure 2.2. The authors in [24] show an attack, which is called EMAN2, where they provoke an overflow of the LVs and illegally gain access to the frame data to overwrite the return address. Therefore, they are able to change the control flow of an applet. In 2013 an OS underflow attack was proposed in [25]. By the help of illegal *swap* bytecodes they were able to illegally access the return address of a Java Frame.

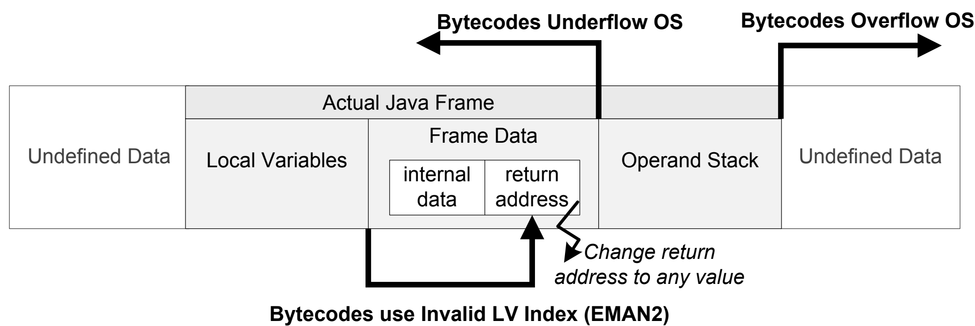


Figure 2.2: On the Java Frame different attacks are proposed like an OS underflow attack [25] or an overflow attack on the LVs [24] memory (obtained with modifications from [26]).

Control Flow Attack

During the execution of an applet the VM fetches the bytecodes from the fixed sized bytecode area (BA). In 2011 a run-time attack, called EMAN4, was found by the authors in [24] which is shown in Figure 2.3. By a laser attack during run-time they were able to change the control flow of an applet to illegally jump outside the BA by a manipulated *goto* bytecode. The jump destination can then be, for example, a data array filled with malicious code. This leads to the security threat of executing data instead of code. Such a manipulation of the control flow can lead to serious security attacks like the memory dump of security-critical code and data on the card as described by the authors in [27].

Data Type Confusion Attack

By the Java Card specification [9, 10] it is exactly specified which data types are expected on the OS or LVs and which data types are written back. The data types can be divided into two main data types *integralData* (boolean, byte, short) and *reference* (object references). Type confusion between these two data types is often performed and mentioned

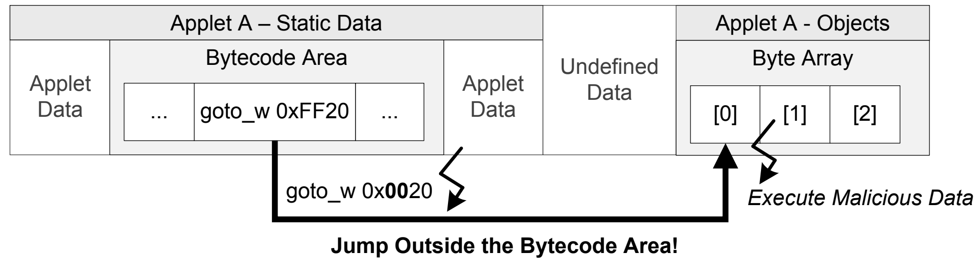


Figure 2.3: An attacker is able to change the bytecode operands to 0x00 or 0xff by a fault attack. By applying a fault attack on the *goto_w* bytecode Barbu [24] was able to illegally change the control flow of an applet to execute malicious data (obtained with modifications from [28]).

in related works [29, 30, 31, 32, 33, 27, 24, 34] and is therefore a serious security threat. Such type confusion attacks are often the starting point for more advanced attacks like executing a data array instead of code as shown in [33, 24], the access to forbidden objects [34], and to characterize the Java Card API [30]. This thesis shows in Section 3.2.2 three different countermeasure designs which counteract type confusion attacks on the OS and LVs. Furthermore, these type checks have been shifted from SW into HW which is explained in more detail in Section 6.3.

Data Integrity Attack

The Java Card VM security and the Java Card sandbox model relies on the integrity of the data in the memory. This is especially true for the integrity of the OS and LVs because almost every bytecode reads or writes data into these memory regions. In 2011 the authors in [35] proposed a run-time data integrity attack on the OS using a laser beam. They abuse the fact that most conditional jump bytecodes (e.g., *ifeq*, *ifne*, *iflt*, *ifge*) rely their jump conditions on data on the OS. Therefore, by manipulating the OS data they were able to change the jump destination to their will. Therefore, this integrity attack enables the bypass of security checks (e.g., pin checks, control flow checks).

2.3 Overview of Java Card Countermeasures

Today various countermeasures are used to counteract attacks on Java Cards. These countermeasures start from additional physical sensors (e.g., light, high/low clock frequency or supply voltage), bus scrambling, additional protection layer in silicon or glue logic [8]. Various researcher proposed countermeasures against optical fault attacks [36], power analysis attacks [37, 38], and electromagnetic analysis attacks [20].

2.3.1 Countermeasures to Harden the Java Card Sandbox

In 2010 Barbu [29] showed the first practical attack where the Java sandbox model was skipped by a fault attack. Theoretical works on such attacks were earlier described [32, 31]. After Barbu's attack, researchers worked on the question of how to harden the sandbox against fault attacks on resource constrained Java Cards. Countermeasures in Java Cards should consume as little additional performance, memory and power as possible. To

reach these requirements various techniques are proposed which can be split into countermeasures where a Java Card applet is pre-processed and enriched with security relevant information and countermeasures which work on every Java Card applet without a pre-processing step.

Sere proposes in [39] that the value of the *nop* (no operation) bytecode be changed from 0x00 to another value. Such a modification prevents from the threat of skipping bytecodes by injecting 0x00 during a fetch operation of the VM. To protect against such skipping attacks Sere also proposes [39, 40] the calculation of a bit field over the bytecode. In this bit field is stored whether the actual fetched data is an opcode or an operand. Therefore, a confusion between opcodes and operands can be detected during run-time. Another countermeasure to detect skipped bytecodes is the off-card calculation of checksums over basic blocks (BBs) of the bytecode [41]. This is done by checking the pre-calculated checksum against the checksum calculated during run-time. Furthermore, by pre-calculating the control flow between BBs anomalies in the run-time control flow are detected. Such anomalies are illegal jumps from a BB to another BB which is not reachable. All of Seres countermeasures increase the size of the applet by additional security related data to store bit fields, checksums, and control flows. A more memory friendly countermeasure is proposed in [42] which is based on an off-card encryption of the bytecode. During run-time the bytecodes are then encrypted with a secret key. Such a bytecode encryption prevents from the threat of jumping outside the BA and executing attacker defined data. An attacker has to know the secret key to fill an array with valid code. Another countermeasure proposed in [43, 44] prevents the threat of type confusion between the two main data types *integralData* and *reference* on the OS. The countermeasure is based on splitting the OS into two memory regions. Depending on the expected main data type (*integralData*, *reference*) data is read or written into one of these memory regions. This countermeasure requires an off-card step to remove incompatible bytecodes (e.g., *swap*, *pop*, *dup*). In [45] a so called fingerprint is calculated off-card over the bytecodes of an applet. During run-time it is checked whether the actual applet execution is compliant with the fingerprint. In [46] an automation array is calculated during the linking time for every method. Based on this automation information the control flow of the applet is checked during run-time. In 2011 Barbu [35] proposed three countermeasure types against data integrity attacks on the OS. The first approach is based on an additional memory read and comparison operation after every read or write operation in the OS. Barbus second approach is performing a data integrity evaluation when the Java Card firewall checks are performed. By this approach the number of checks is reduced and therefore the overall execution performance of an applet increases. The third approach is based on calculating checksums for every written or read element into the OS. These checksums are evaluated at security-critical points in time during the execution of a Java applet.

2.3.2 Drawbacks of Current Countermeasures and Discussion

The currently proposed countermeasures against run-time attacks are implemented in SW and have two main drawbacks which are additional execution time overhead and memory consumption. The additional memory consumption comes from the fact that various countermeasures rely on a pre-processing step where checksums and control flow graphs are additionally added into the Java Card applet [39, 40, 45]. During run-time the

calculation and comparison of the checksums and control flow graphs requires additional execution time. Another drawback is that these SW checks can be skipped by a double fault injection [47]. Such a double fault injection gets easier when the time between the two faults is relatively long. To skip our new proposed HW checks in Section 6.3 an attacker has to inject faults very quickly which increases the difficulty of a successfully performed double fault injection [48]. Furthermore, the overall success rate of the attack decreases if more than one fault injection is needed. For example if one fault injection has a success rate of 10% than the overall success of an attack decreases to $10\% * 10\% = 1\%$ if two attacks are needed [49]. Therefore, the required number of attack runs for one sucessfully performed attack increases from 10 to 100. For an attacker this means that the card will most probably be unusable because the attack trigger will in all likelihood reach its maximum tolerable number which results in a card lock.

2.4 Summary and Difference to State-of-the-Art Countermeasures

This thesis aims at introducing the following main improvements to the state-of-the-art, with respect to the goals defined in Section 1.3.3:

- Run-time attacks are used by attackers to break out of the Java sandbox model to access security-critical data. In related work all proposed countermeasures which harden the VM against run-time attacks are performed in SW. In this work countermeasures to counteract, for example, type confusion attacks are accelerated by HW mechanisms during a HW/SW codesign process. This shift of functionality from SW to HW decreases the execution time overhead of the countermeasures.
- This thesis shows countermeasures which do not rely on an off-card pre-processing step to harden the Java Card sandbox. In related works pre-processing is needed to calculate, for example, checksums over the bytecode to counteract integrity attacks. Such integrity attacks are counteracted in this thesis by double checking the values which are read from or written into the memory.
- Related work does not answer the question of how countermeasures can be smoothly integrated into the VM to attain easy maintainability. Therefore, this work proposes to add an additional security layer into the VM architecture. In this security layer all run-time checks from this work are integrated.

Furthermore, this work shows the following additional advancements:

- Exploration of various countermeasure designs and implementation variants (e.g., type confusion and memory overflow countermeasures).
- Countermeasures are explored in terms of execution time, memory, and hardware overhead on a Java Card prototype.
- To counteract fault attacks this work moves checks which are statically performed at install time to the run-time.

Chapter 3

Design Evaluation for a Secure Future Java Card

This section provides an overview of publications concerning a HW/SW codesigned secure future Java Card. Various run-time security policies are shown which are fulfilled by this card. Run-time countermeasure designs are discussed which counteract, for example, type confusion or memory overflow attacks. Furthermore, this section shows the acceleration of the run-time security checks by specific HW support and the integration of the security mechanisms into the Java Card architecture.

3.1 Overview

Different attacks exist which threaten Java Card security. To counteract these threats new security mechanisms must be integrated into the Java Card VM. The proposed mechanisms of this work aim at increasing security against fault attacks and logical attacks. The increased security of the card is reached by a static on-card verification step performed on every newly installed applet. Furthermore, run-time checks are executed which counteract various fault attacks during run-time. These run-time checks are accelerated in this work by microarchitectural HW mechanisms on the card. An overview of the individual contributions of this work and their mapping to scientific publications are illustrated in Figure 3.1.

The basis of such a secure future multi-application Java Card is that an attacker is not able to install a malicious applet onto the card. In case a malicious applet can be installed onto the Java Card, then an attacker is able to illegally access security-critical code or data by a logical attack. To prevent this security threat, a static on-card verification step is needed. Such a memory efficient on-card bytecode verification process is described in detail in *Publication 1*.

A verified applet cannot break the Java Card sandbox model and gain illegal memory accesses. Unfortunately, the sandbox model can be circumvented by fault attacks during the applet execution. Therefore, new countermeasures must be integrated into the Java Card VM. To counteract, for example, memory overflow attacks and type confusion attacks on Java Cards two different approaches are presented in *Publication 2*. These approaches counteract type confusion attacks between *integralData* (e.g., short, byte, boolean) and

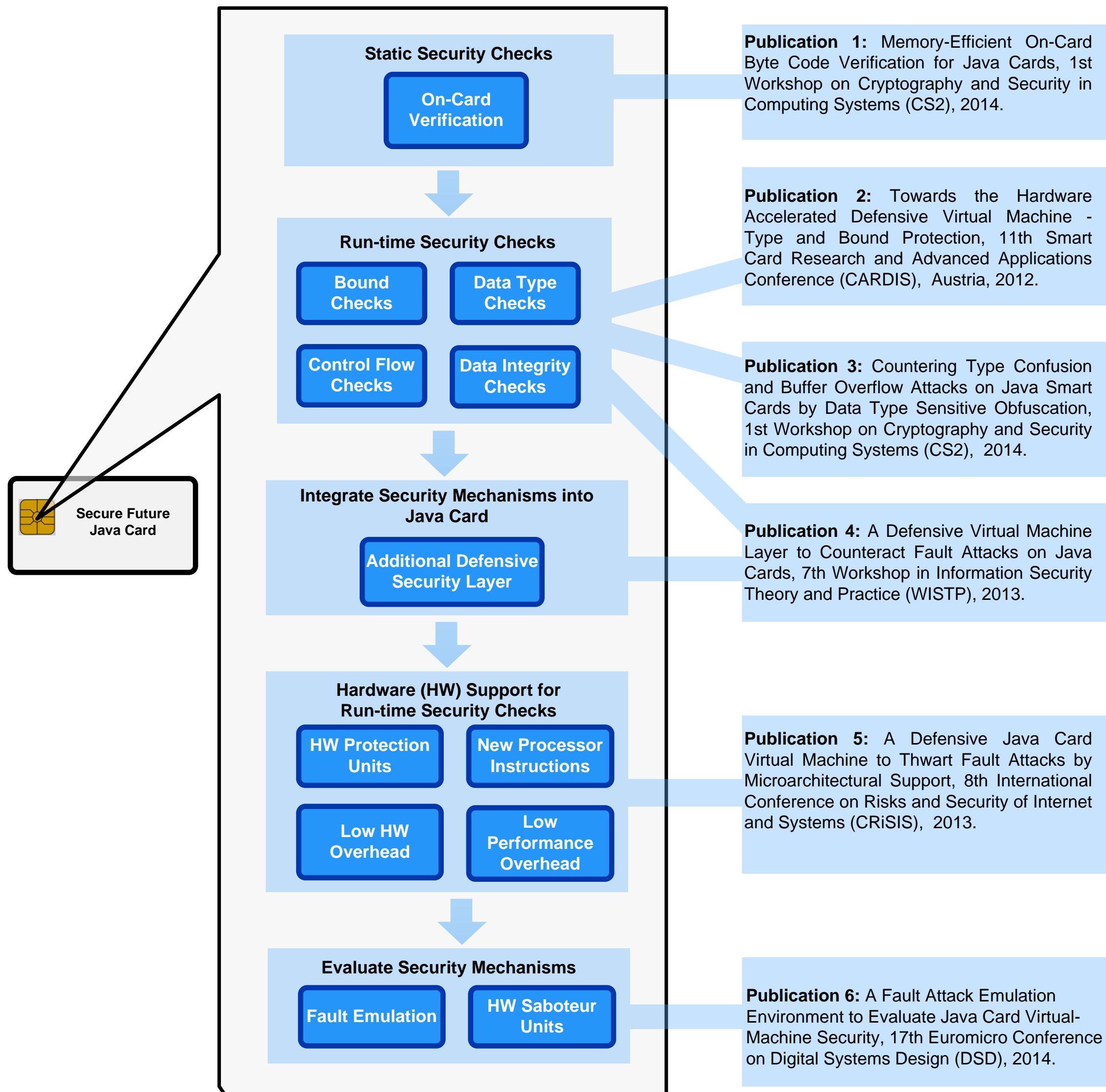


Figure 3.1: Publication overview concerning a secure future Java Card with new HW and SW security features.

object *references* (e.g., short[], byte[], class A). The first countermeasure approach is called type storing and stores the data type information during run-time. Based on this stored information additional checks are performed during run-time to detect type confusion. The second approach is called type separating and separates the data into different memory regions depending on the data type. By this type separation it is no more possible to perform type confusion between *integralData* and *reference* during run-time. Furthermore, both approaches perform memory bound checks to counteract run-time buffer overflow attacks.

In *Publication 3* it is shown how type confusion and buffer overflow attacks can be counteracted by data type sensitive obfuscation. The values inside the VM are obfuscated by different secret keys depending on the data type. For *integralData* the key $K_{integralData}$

is used and for *references* the key $K_{reference}$. By this countermeasure an attacker will create unpredictable values with a type confusion attack. Furthermore, buffer overflow attacks are counteracted because the data which is read/written outside the buffer is also unpredictable for an attacker. The type obfuscation approach is more memory efficient and consumes less additional computational performance than the type storing and type separating approach.

All these run-time countermeasures have to be seamlessly integrated into the Java Card architecture. Therefore, *Publication 4* shows an additional SW security layer inside the Java Card architecture. All memory accesses to security-critical memory regions are processed over this novel layer. This security layer contains all the run-time security operations. Therefore, security operations can be easily added, disabled or modified inside the layer.

The execution of the run-time security operations in SW slows down the overall execution performance of the card. Therefore, *Publication 5* adds new HW protection units into the card to shift the run-time security operations from SW to HW. These units are the type protection unit, memory bound protection unit, control flow protection unit, and the data integrity protection unit. With the help of these additional HW units the performance overhead due to the run-time checks is dramatically reduced compared to a SW implementation.

A fault emulation environment is presented in *Publication 6* to test different HW and SW countermeasures. By this environment different fault attacks can be recreated by a HW bus saboteur unit. This environment enables the design space exploration of new countermeasures. Furthermore, the effects of various fault attacks on the behavior of the VM can be analyzed.

3.2 Run-time Checks

The possible security threats for a future multi-application Java Card have been analyzed based on related work described in Section 2.2.1, the Java Card protection profile [50], and the Java Card specification [9, 10]. Based on these documents a run-time security policy has been created which counteracts the most dangerous threats. A detailed description of the policy and the countermeasures to fulfill this policy are shown in Section 6.3.

3.2.1 Run-time Security Policy

The goal of the run-time security policy is to counteract different run-time security threats on the security-critical data of the VM. Figure 3.2 depicts a schematic overview of the policy and the security-critical memory regions involved. The run-time security policy is based on four different sub-policies: (i) memory bound policy, (ii) data type policy, (iii) control flow policy, and (iv) data integrity policy.

- **Memory Bound Policy:** The memory bound policy counteracts buffer overflow attacks on the OS and LVs which is described in Section 6.1 [26]. The security of the bound policy is then improved in Section 6.3 by taking into account the fact that the OS can be seen as a FIFO (first in, first out) buffer [51]. Therefore, it is only

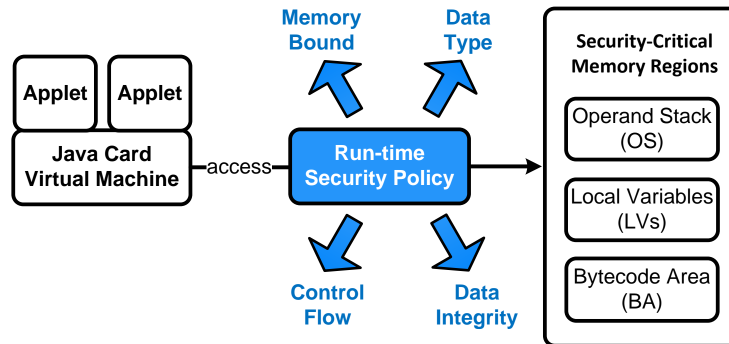


Figure 3.2: Overview of the run-time security policies for a future multi-application Java Card (obtained with modifications from [51]).

allowed to write data onto the top element of the OS. A read or write operation in the middle of the OS is not allowed.

- **Control Flow Policy:** The control flow policy checks that attacks are not able to illegally change the control flow of an applet by jumping outside the bytecode area of the applet. This control flow policy counteracts, for example, the threat of executing data instead of code and is described in more detail in Section 6.2 [28] and Section 6.3 [51].

- **Data Integrity Policy:** The data integrity policy counteracts the illegal manipulation of data which is read or written into the run-time data of the VM (OS and LVs) by a fault attack. The integrity policy is described in more details in Section 6.3 [51].

- **Data Type Policy:** The data type policy counteracts run-time type confusion attacks between the main data types *integralData* and *reference*. The counteracted threats and a more detailed description of the policy are shown in Section 6.1 [26].

3.2.2 Type Confusion Countermeasure Designs

Based on the previously defined run-time security policies different implementations of countermeasures are designed and evaluated to fulfill the policies. For example to fulfill the type policy three different general countermeasure designs have been found. Type confusion can be counteracted by (i) storing the type information for every element, (ii) separating the data based on the data type or (iii) obfuscating the data dependent on the data type. A schematic overview of the three general type confusion countermeasure designs is illustrated in Figure 3.3.

- **Type Storing:** For every element on the OS and LVs the type information is stored. Due to the fact that it is statically defined which bytecodes work on which types, it is possible to perform type checks during run-time. An evaluation of two different type storing designs is described in more detail in Section 6.2 [28] which are namely the word storing and bit storing approach. The difference between them is that the type information is either stored into a memory area with the same data size as

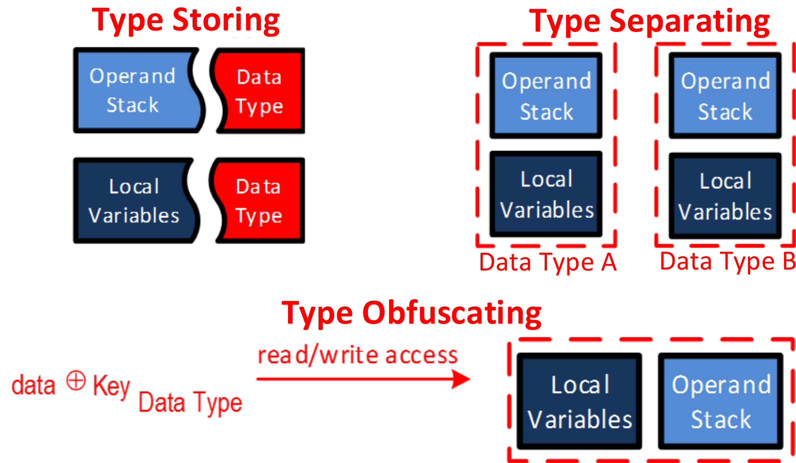


Figure 3.3: To fulfill the type policy of this work three different general countermeasure designs have been evaluated which are type storing, type separating, and type obfuscating.

the processor or into a type bit-map where every bit represents one entry of the OS or LVs. The two different designs vary in the memory consumption and execution speed.

- **Type Separating:** This countermeasure stores the OS and LV values into different separated memory areas depending on the data type. For example there is one OS for all elements of type *integralData* and one OS for all elements of the data type *reference*. Type confusion is now avoided from an architectural point of view because every bytecode takes and stores its values to the right OS or LVs. Therefore, run-time type checks are no longer needed in comparison to the type storing approach, where they are. A disadvantage of the type separating approach is that so called untyped bytecodes (*pop*, *swap*, *dup*) must be replaced by typed ones during a pre-processing step. The type separating countermeasure is described with its advantages and disadvantages in Section 6.1 [26].

- **Type Obfuscating:** Depending on the data type the values on the OS and LVs are obfuscated with a secret key. An adversary is no longer able to perform useful type confusion attacks without knowing the secret key. Type confusion attacks are not detected by this countermeasure as they are with the type storing approach but the data which is created by a type confusion is useless for an attacker without knowing the secret key. A description of the key creation and more details of the design are presented in Section 6.4 [52].

The different type confusion countermeasure designs have been evaluated during a countermeasure design space exploration phase. The five attributes which are used for the evaluation are: Is a pre-processing of the Java Card applet necessary (pre-processing required)?; What is the expected execution time increase of the VM (time overhead)?; What is the expected memory overhead of the countermeasure (memory overhead)?; Is the attack immediately detected by the countermeasure or later during its execution (detection latency)?; Is an attack really detected or does an attacker still receive data (detection

rate)?. For example the detection rate for the type obfuscation countermeasure is low because an attacker is still able to perform the type confusion attack. Nevertheless, the received obfuscated data is useless for the attacker. An overview of the different type confusion countermeasures and their evaluation is shown in Table 3.1. The table input is evaluated based on the type confusion countermeasure designs in the Sections 6.1 6.2 6.4 [26, 28, 52].

Table 3.1: The SW type confusion countermeasures of this work have different properties. This table compares the type storing approach (word storing, bit storing) with type obfuscating and type separating.

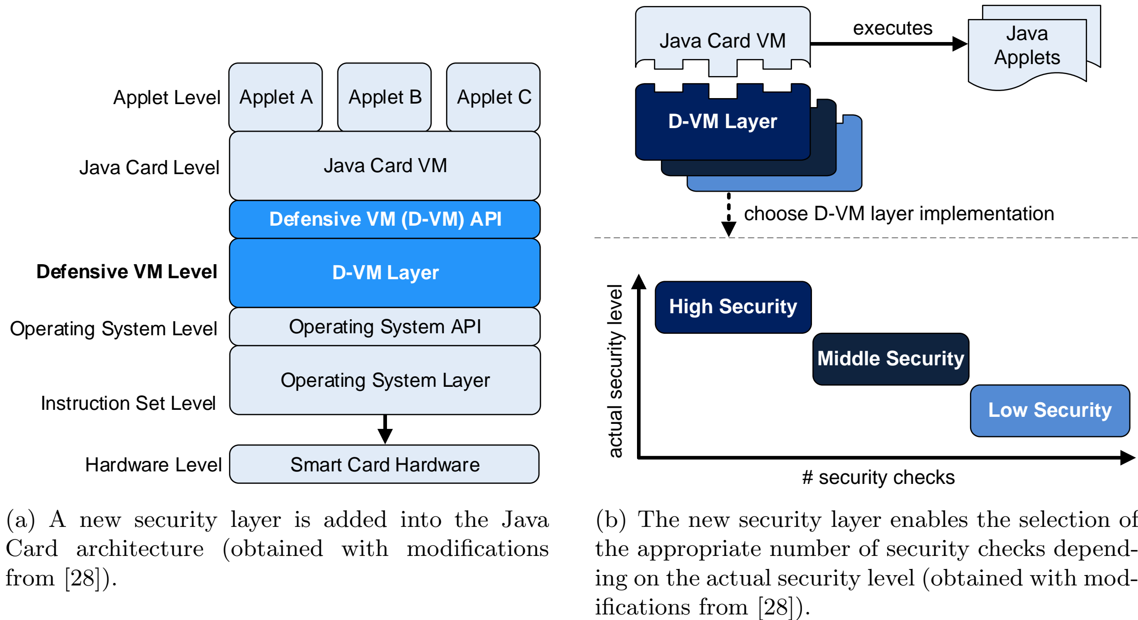
Type Confusion Countermeasure	Pre-Processing Required	Time Overhead	Memory Overhead	Detection Latency	Detection Rate
Type (Word) Storing	**No**	**Middle**	**High**	**Low**	**High**
Type (Bit) Storing	**No**	**High**	**Middle**	**Low**	**High**
Type Obfuscating	**No**	**Low**	**Low**	**Middle**	**Low**
Type Separating	**Yes**	**Middle**	**Middle**	**Middle**	**Middle**

3.2.3 Memory Overflow Countermeasure Designs

The memory bound policy says that the fixed sized OS and LVs memory should never overflow. To fulfill this policy it is possible to perform run-time checks if the accessed memory is inside the higher and lower memory bound address. This bound checking approach is described in Section 6.1 [26]. Another approach is the obfuscation of the values inside the OS and LVs. When the values are obfuscated with a secret key then every read or write outside the memory bounds will create useless data for an attacker. The obfuscation approach is presented in Section 6.4 with a countered attack example [52].

3.3 Countermeasure Integration into Java Card Architecture

The proposed run-time countermeasures must be seamlessly integrated into the Java Card architecture. Therefore, a new layer is proposed in this thesis as shown in Figure 3.4(a). During the execution of a bytecode the access to the security-critical data of the VM is performed over this new layer. A more detailed description of the layer and its usage to enable dynamic countermeasures is shown in Section 6.2 [28]. A schematic overview of the dynamic countermeasure approach is shown in Figure 3.4(b). Based on the required security level an appropriate security layer implementation is chosen. If the security needs are very high, an implementation with a lot of checks is used. If the security needs are low, a layer implementation which consumes less memory and run-time performance can be chosen as described in Section 6.2 [28].



(a) A new security layer is added into the Java Card architecture (obtained with modifications from [28]).

(b) The new security layer enables the selection of the appropriate number of security checks depending on the actual security level (obtained with modifications from [28]).

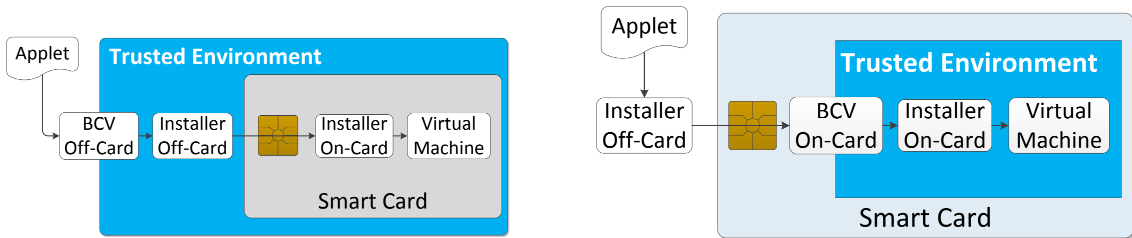
Figure 3.4: The overall security of the Java Card VM against fault attacks is increased by adding an additional security related SW layer (defensive VM (D-VM) layer) into the Java Card architecture. This new layer contains the security checks and provides the VM access to security-critical memory regions.

3.4 Static On-Card Applet Verification

One of the most important use cases for a secure future multi-application card is that everybody can install any applet on this Java Card. Therefore, a static check during the install time must be performed to confirm that the applet contains no illegal operations. Currently, the applet verification process is performed off-card in a secure environment as shown in Figure 3.5(a). This work shifts the on-card verification onto the card as shown in Figure 3.5(b). The difficulty in enabling the off-card verification relies on the constrained available memory resources of a Java Card. Therefore, the new on-card applet verification of this thesis relies on a generation of BBs and a control flow analysis with the goal of reducing the memory requirements. Using this verification process the bytecodes of every Java method is split into BBs. Based on the BBs a control flow graph is calculated which connects the BBs together. Subsequently, every BB is verified by analyzing all its containing bytecodes and marking them verified if the verification was successful. A more detailed description and memory analysis of the new on-card verification process is discussed in Section 6.5 [53].

3.5 Hardware Support for Run-time Security Checks

Performing the different run-time security checks in SW adds an additional overhead to the execution performance of the VM. Java Cards are very resource limited devices which have strict timing requirements for industrial applets. A Java Card, even if it is more secure



(a) Currently the bytecode verification is performed off-card (obtained with modifications from [53]).

(b) The future secure Java Card of this work performs on-card bytecode verification (obtained with modifications from [53]).

Figure 3.5: Every newly uploaded applet must be analyzed before installation. The new on-card verification process performs this analysis to detect applets which could break the Java Card sandbox model by invalid operations.

than another, cannot be sold if it does not meet these strict execution time requirements. Therefore, this thesis shifts the computationally expensive run-time checks from SW to HW as shown in Figure 3.6. Based on the security policy of this work from Section 3.2.1 the instruction set of the Java Card processor has been split into two different classes. The first class are security-critical memory access instructions (defensive instructions) which are enhanced with different HW mechanisms to fulfill the security policy. The second class are standard processor instructions which are not related to the run-time security policy of Section 3.2.1.

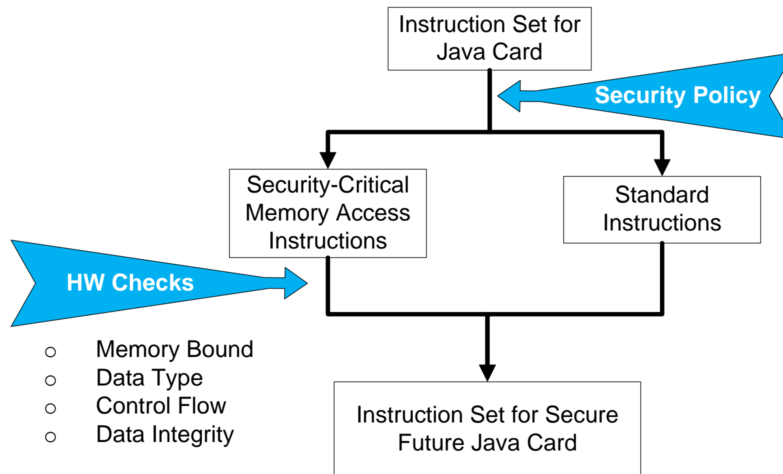


Figure 3.6: Additional processor instructions are inserted into the Java Card to support HW accelerated security checks during run-time to fulfill the security policy of this work (obtained with modifications from [54]).

This work adds a protection unit, control flow protection unit, data type protection unit, and data integrity protection unit into the Java Card architecture as shown in Figure 3.7. These different protection units are responsible for fulfilling the security policy from Section 3.2.1. The VM communicates with the protection units over newly added so called defensive processor instructions. Different Java related information is decoded into these

new instructions. The new HW protection units are able to validate the memory accesses based on the decoded information.

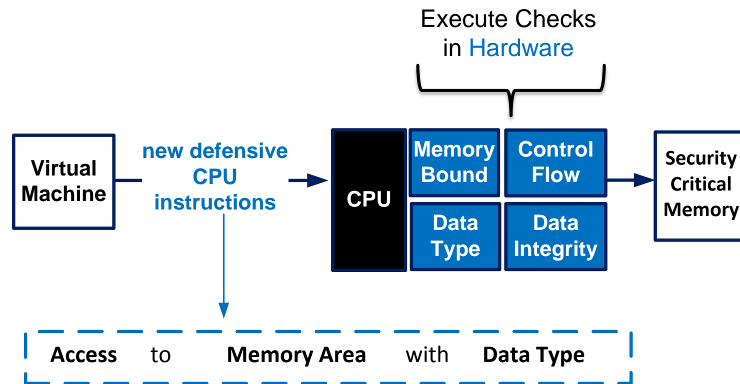


Figure 3.7: To overcome the execution slow down of run-time security checks, this work shifts the run-time checks from SW to HW (obtained with modifications from [51]).

3.6 Evaluate Java Card Security by Fault Emulation

To evaluate new countermeasures either implemented in HW or SW a fault emulation environment is proposed in this thesis. Furthermore, this environment can be used to find new attack paths using automatically performed fault attacks. The fault emulation environment is able to inject faults into the security-critical memory regions during the execution of the Java Card VM. An overview of the environment is outlined in Figure 3.8. The environment user is able to inject faults at specific bytecodes into specific security-critical memory regions. The appropriate fault is automatically injected by the environment based on the actual state of the VM and the system bus. The emulation approach on an FPGA is much faster compared to a simulation of the faults on a personal computer. A more detailed description of the fault emulation environment is given in Section 6.6 [55].

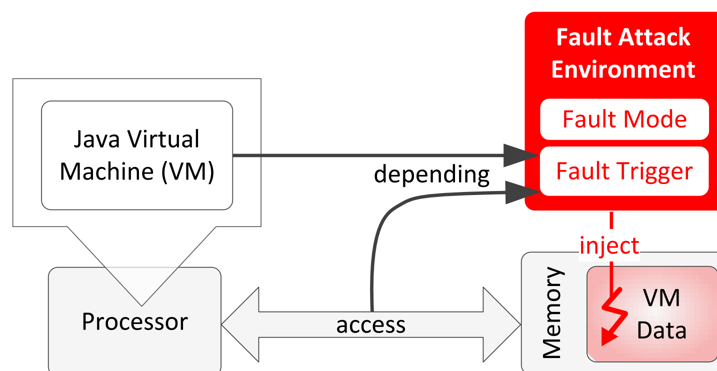


Figure 3.8: With the help of the fault emulation environment SW/HW countermeasures can be evaluated. Furthermore, the behaviour of the VM can be studied by automatically injecting and evaluating the faults (obtained from [55]).

Chapter 4

Results and Case Studies

This section first gives an overview of the HW models which are used in different case studies. During these case studies various countermeasures were evaluated. The countermeasures are either implemented in SW or HW and evaluated in terms of run-time performance, memory overhead, and HW overhead. Furthermore, this section shows the modifications needed to the processor architecture to enable the newly added HW protection units. These protection units secure the VM in terms of memory bound checks, Java data type checks, control flow checks and data integrity checks.

4.1 Evaluation Platforms

The SW and HW countermeasures to counteract run-time fault attacks on the Java Card sandbox model are evaluated in this section. The evaluation is done on two different smart card models using an 8-bit 8051 [56] architecture. Derivates of this architecture are still used in current industrial Java Cards. The two models of this work are implemented at two different HW abstraction levels. The first model is implemented at the system level in SystemC [57] and is used for a fast design space exploration [58]. The second model is implemented at the register transfer level in the very high speed integrated circuit HW description language (VHDL) [59]. The VHDL model enables exact measurements of the timing overhead due to the additionally added functionality of the HW protection units. This VHDL model has been synthesized on a Xilinx Virtex-5 field programmable gate array (FPGA) development board.

4.1.1 Tool Chain

The whole development flow used to evaluate our prototype is shown in Figure 4.1. The prototype Java Card VM is implemented in C code and in 8051 assembly language. Assembly is needed to integrate the newly added defensive CPU instructions into the VM. As a compiler and linker the Keil 8051 development tools[1] are used to generate a hex file for both HW models. The VHDL HW model is then executed in ModelSim[2] or on a Virtex-6 FPGA development board. The obtained performance and HW results are used

[1] http://www.keil.com/
[2] http://www.model.com/

to enhance the SystemC model. The SystemC HW model is executed in Microsoft Visual Studio using the programming language C++. The SystemC extension for transaction level modeling (TLM) 2.0 is used for modeling the data transactions in the processor bus.

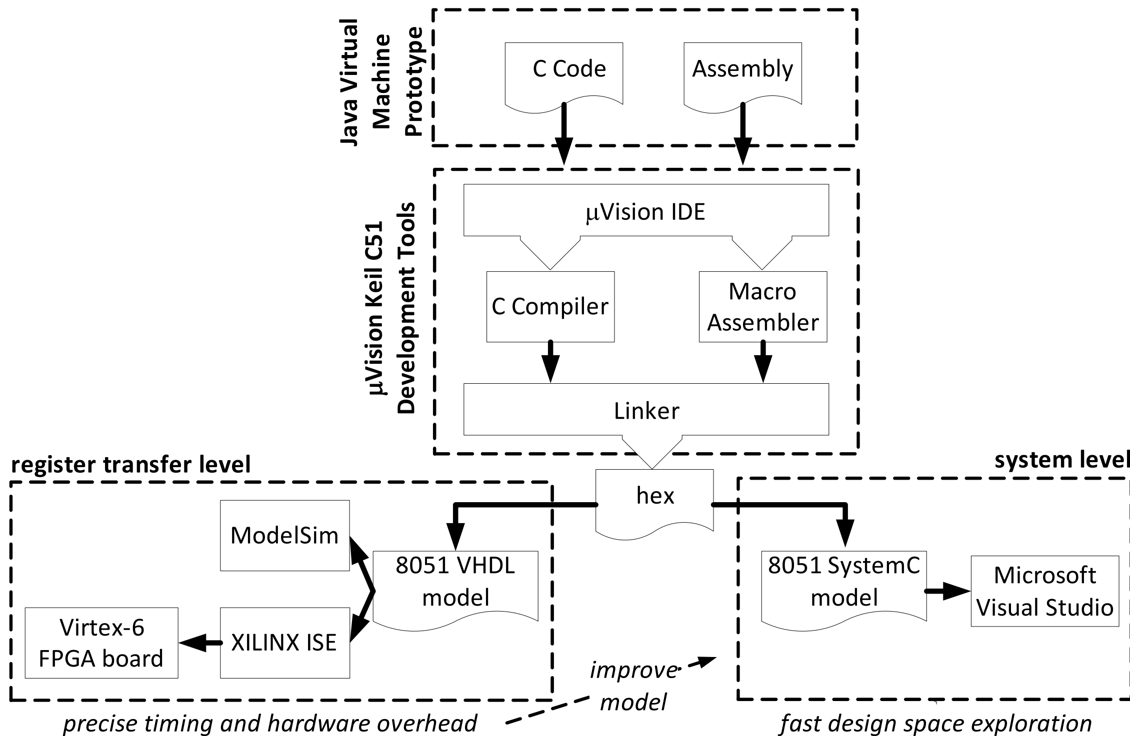


Figure 4.1: The Java Card VM prototype is implemented in the programming languages C and assembly. A hex file of the VM is created by the Keil C51 development tool chain. This hex file is either executed on a register transfer level model programmed in VHDL or a system level model programmed in SystemC.

4.1.2 System Level Model

An overview of the main parts of the SystemC model is shown in Figure 4.2. To enable the HW accelerated run-time checks different protection units (bound, type, control flow, data integrity) are integrated into the controller unit of the model. Furthermore, new defensive instructions are added into the instruction set of the processor model. The external memory (XRAM) has been extended with an additional type bit which holds the data type information for the type protection unit. This fast executing high-level model was used for the proof of concept implementation and the case studies which are presented in the Sections 6.1 6.2 6.4.

4.1.3 Register Transfer Level Model

The main parts of the VHDL model are shown in Figure 4.3. The parameterizable and synthesizeable 8051 processor core model is freely available[3] even for commercial purposes.

[3] www.oreganosystems.at/?page_id=96

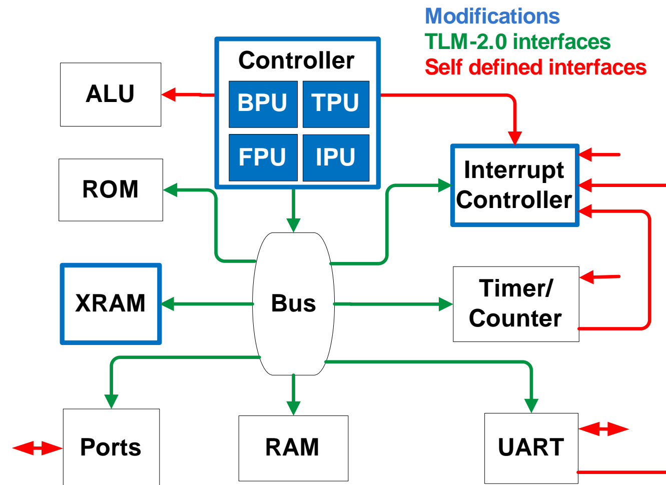


Figure 4.2: The high level SystemC model of an 8051 processor consists of different main parts such as the controller unit, memories or timers which are connected over a bus. During this work different HW protection units are integrated into the model (obtained with modifications from [58]).

The model is developed from Oregano systems in cooperation with the Vienna University of Technology. In this processor model our new HW security features have been incorporated[4]. The bound, type, and control flow protection units were integrated into the 8051 core component. Furthermore, the new defensive instructions are added into the instruction fetch, decode, execute, and write back logic. The data integrity protection unit was integrated as a separate component into the top level architecture of the 8051 and has its own interface to the external memory. The external memory was extended with an additional type bit for the type protection unit. When a violation of the security policy is detected then a newly added security interrupt occurs. This very accurate model was used for the implementation of the secure future Java Card of Section 6.3.

4.2 Case Study on Type Confusion Countermeasures

In this work different innovative techniques are presented and evaluated to counteract runtime type confusion attacks between the main data types *integralData* and *reference*. As presented in Section 3.2.2 this thesis shows three type confusion countermeasure designs which are (i) type storing, (ii) type separating, and (iii) type obfuscating. In this work different implementations for the three designs were evaluated in different publications as shown in Figure 4.4.

4.2.1 Type Storing

During the design space exploration phase three different implementation techniques for type storing were found and evaluated, as shown in Figure 4.5. These three implementa-

[4] Michael Hraschan, "Hardware Based Security Features for a Defending Java Card Virtual Machine", Telematic It-Project, Graz University of Technology, 2013

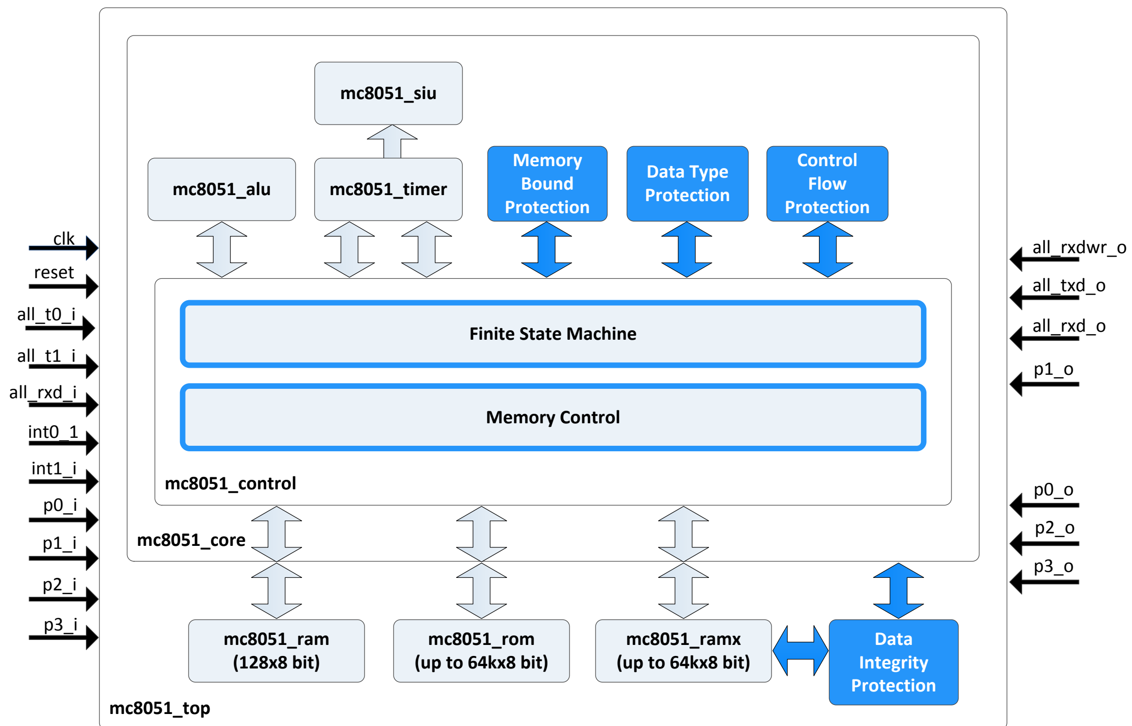


Figure 4.3: The low level VHDL model of an 8051 processor consists of different main parts like a finite state machine or a memory control unit. The new HW protection units were integrated into this processor model (obtained with modifications from [60]).

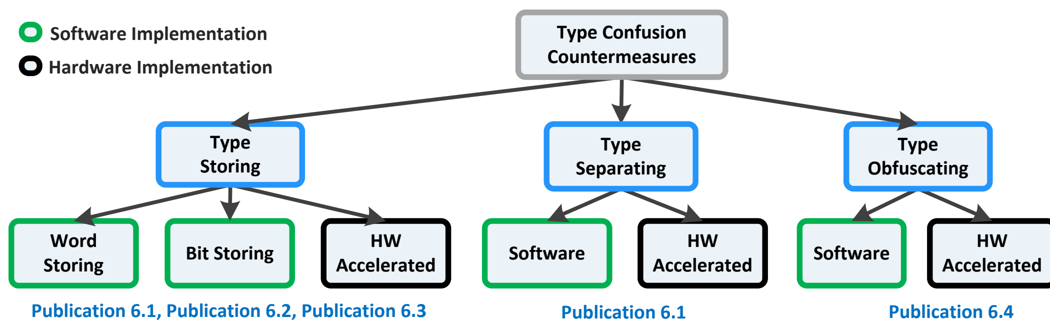


Figure 4.4: Various type confusion countermeasure designs and implementation variants were evaluated based on run-time performance, memory overhead, and HW overhead. The three main countermeasure designs are type storing, type separating and type obfuscating which were implemented either in SW or with HW support.

tions are compared and evaluated between each other based on their additional HW, main memory, and run-time performance overhead shown in Section 6.2.

- **Bit Storing:** The *bit storing* countermeasure holds the type information in a specially designed bit-map structure. In this structure each bit represents one entry of the OS or LVs. The bit value 0 represents the Java data type *integralData* and the bit value 1 represents a *reference*. Read or write operations into the bit-map

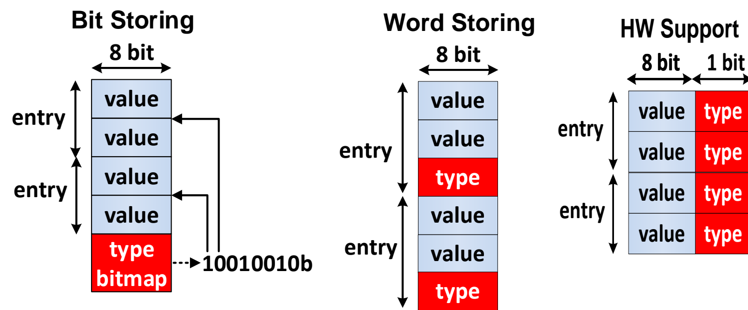


Figure 4.5: Type confusion attacks can be counteracted by storing the type information during run-time for every value on the OS and LVs. Note that this figure shows an abstract implementation overview for the 8-bit 8051 architecture (obtained with modifications from [28]).

structure are relatively costly and time-consuming. The overhead comes from the fact that each access needs some shift and modulo operation to find the right bit for the searched entry [28]. The *bit storing* approach increases the memory needs of the OS and LVs by 6,25% because 1 type bit is needed to represent 16 data bits.

- **Word Storing:** The *word storing* countermeasure significantly increases the run-time performance compared to the *bit storing* approach. Unfortunately, the memory needs also increase because each type entry of the OS or LVs is added to this entry in the word size of the processor. Values stored in the standard word size can be quickly accessed by the processor and needs no additional shift or modulo operations. The word size of the 8051 processor is 8-bit [28]. Therefore, the memory needs of the OS and LVs increase the *word storing* approach by 50% because 8 type bits are needed to represent 16 data bits.

- **HW Supported:** The *HW supported* type storing approach moves the time costly type comparing and type storing operations from SW to HW. This means that the run-time overhead dramatically decreases compared to the previously explained SW countermeasures *bit storing* and *word storing*. The type information is directly stored as an additional bit into the memory. Therefore, on an 8051 architecture the main memory increases by 12,5% because 1 type bit is added to 8 data bits. The newly added HW type protection unit is able to validate whether the accessed memory has the expected data type. If the type is not valid then a HW exception is thrown by the type protection unit [26, 28].

Performance Overhead - Type Storing Countermeasures

The performance overhead for the three type storing countermeasures is presented in Table 4.1 for different bytecode groups. The *bit storing* approach significantly decreases the overall performance by adding 210% of overhead. The *word storing* approach exchanges speed with more memory consumption and has an overhead of 140%. The *HW supported* countermeasure has an overhead of only 6% [28].

Table 4.1: Performance overhead overview of the three type storing countermeasure approaches. Note that the performance overhead measurements are normalized to a VM implementation which does not perform these type checks during run-time (obtained with modifications from [28]).

Bytecode Groups	Bit Storing	Word Storing	HW Supported
Arithmetic/Logic	+240%	+150%	+7%
LV Access	+240%	+190%	+5%
OS Manipulation	+230%	+150%	+5%
Control Transfer	+170%	+110%	+7%
Array Access	+170%	+130%	+5%
Overall	+210%	+140%	+6%

4.2.2 Type Separating

By separating the values on the OS and LVs by their types it is no longer necessary to perform type checks. Unfortunately some bytecodes are not compatible with the type separating approach. These incompatible bytecodes need information about which data type they operate on. Therefore, an off-card preprocessing step is necessary which exchanges these so called untyped bytecodes (*pop_x*, *dup_x*, *swap_x*) to newly added typed bytecodes as explained in Section 6.1 [26] and shown in Figure 4.6(b). To perform this preprocessing step the type information of the values on the OS are needed. This type information can be extracted during the static on-card bytecode verification process described in Section 6.5 [53]. The off-card tool[5] which performs the replacement of the untyped bytecodes to typed ones is outlined in Figure 4.6(a). Also besides this replacement additional security information can be added to the applet like CRC checksums. Run-time performance overhead measurements of a type separating approach implemented in SW and in HW are explained in more detail in Section 6.1 [26].

4.2.3 Type Obfuscating

An execution time and memory efficient countermeasure against type confusion attacks between *integralData* and *reference* is type obfuscating. Type obfuscating obfuscates the values on the OS and LVs with a secret key depending on their main data type. Therefore, the secret key $K_{integralData}$ is used for *integraldData* values.

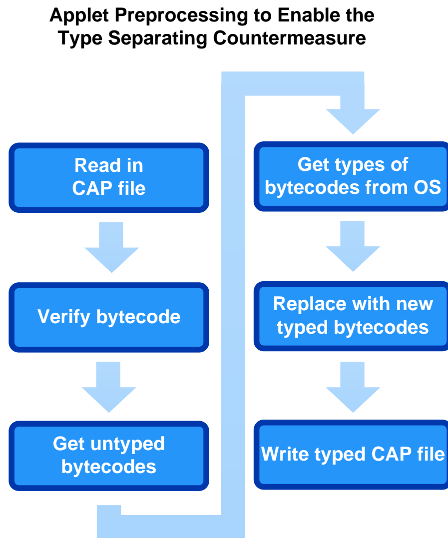
$$integralData_{hidden} = integralData \oplus K_{integralData}$$

All elements of type *reference* are obfuscated with the key $K_{reference}$.

$$reference_{hidden} = reference \oplus K_{reference}$$

This obfuscating technique has quite low performance and memory requirements which makes this countermeasure especially usable in an industrial context. This comes from the fact that no types must be stored and no type checks are needed during run-time. Furthermore, the applet must not be pre-processed as it is for the type separating countermeasure. Note that also bound checks of the OS and LVs are no longer necessarily

[5]Stephan Oberauer, "CAP File Enhancement to Enable a Defensive Virtual Machine", Telematic It-Project, Graz University of Technology, 2013



(a) Outline of the tool chain to enable the type separating countermeasure (obtained with modifications from [61]).

untyped bytecodes	*typed bytecodes*
pop	pop_reference pop_integral
pop2	pop2_reference pop2_integral pop2_reference_integral pop2_integral_reference
dup	dup_reference dup_integral
dup2	dup2_reference dup2_integral dup2_reference_integral dup2_integral_reference
dup_x	dup_x_reference dup_x_integral
swap_x	swap_x_diverse

(b) The untyped bytecodes are replaced by typed bytecodes to enable the type separating countermeasure (obtained from [61]).

Figure 4.6: The type separating countermeasure against type confusion attacks is enabled by a pre-processing step on the CAP file to replace untyped bytecodes with newly added typed ones.

needed. An attacker which performs a memory overflow attack will write unpredictable data outside the OS and LVs. A SW implementation of type obfuscating adds an overhead of around 16% measured over different bytecode groups. A more detailed analysis, description and performance measurement is shown in Section 6.4 [52].

Case Study - Counteract Type Confusion Attack by Type Obfuscating

A type confusion example is shown in Figure 4.7. Java code is first compiled to bytecodes which are uploaded onto the Java Card. An attacker now changes the bytecodes by a fault attack changing the values 0x10 0x19 to 0x00 0x19. This attack results in a type confusion attack on the OS between *integralData* and *reference*. Fortunately, in this case study the type obfuscating countermeasure is used which obfuscates the OS with the secret key $K_{integralData}$ and $K_{reference}$. Based on this obfuscating the attacker receives an obfuscated reference of an array object.

$$reference \neq (reference \oplus K_{reference}) \oplus K_{integralData}$$

A more detailed description of this case study and how type obfuscating counteracts type confusion attacks is shown in Section 6.4 [52].

4.2.4 Hardware Support for Run-time Security Checks on 8051 CPU

An overview of the HW security checks integration into the Java Card architecture, of an 8-bit 8051 processor, is shown in Figure 4.8. The VM and its operating system use new defensive processor instructions to access security-critical memory regions. Specific

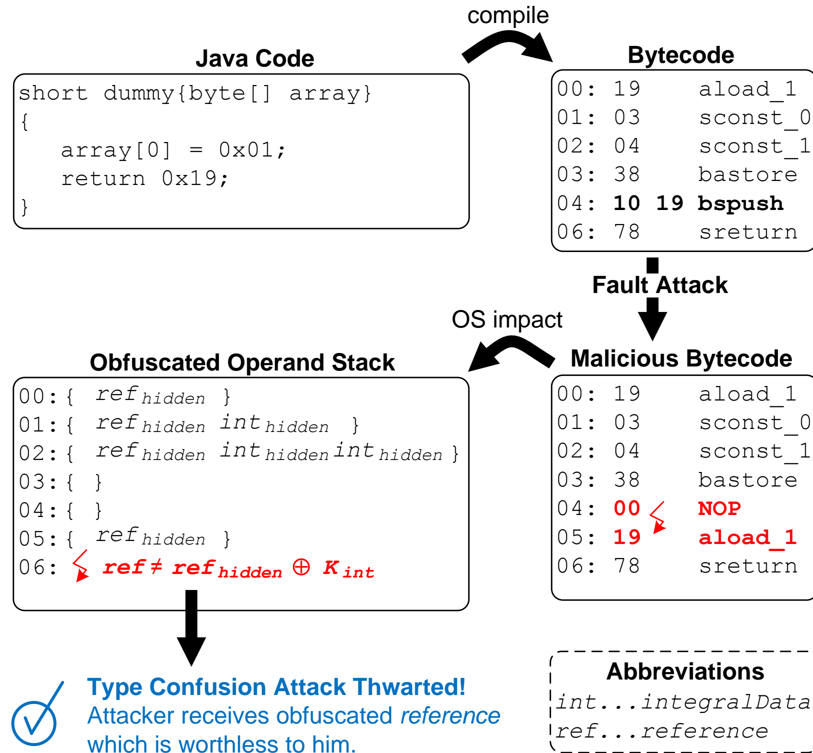


Figure 4.7: Type confusion is counteracted by type obfuscating. Every element in the OS and LVs is obfuscated with a data type depending secret key (obtained from [52]).

security relevant information is decoded into the new defensive instructions which are the access direction (read or write), the accessed memory area (OS, LVs, BA), and the Java data type (*integralData*, *reference*, and *untyped*). This information is used to allow the newly added HW protection units to perform different run-time checks.

- **Memory Bound Protection Unit (BPU):** The BPU protects the bounds of the OS and LVs against buffer overflow attacks. When a new defensive CPU instruction is used with the information that the memory destination is the OS or LVs then the BPU checks if the accessed address is inside the upper and lower OS or LV bounds.
- **Data Type Protection Unit (TPU):** The TPU uses the decoded type information of the defensive instructions and the additional type bit in the memory word to perform type checks during run-time.
- **Control Flow Protection Unit (FPU):** The FPU is responsible for checking if the currently fetched bytecode is inside the bytecode area to counteract the threat of executing data instead of code.
- **Data Integrity Protection Unit (IPU):** The IPU protects the integrity of the OS and LV entries by performing an automatically performed double read. This double read is triggered at every read or write access into the OS or LVs. After reading or writing the value v the IPU automatically reads the inverted value $v_{inverted}$ from the

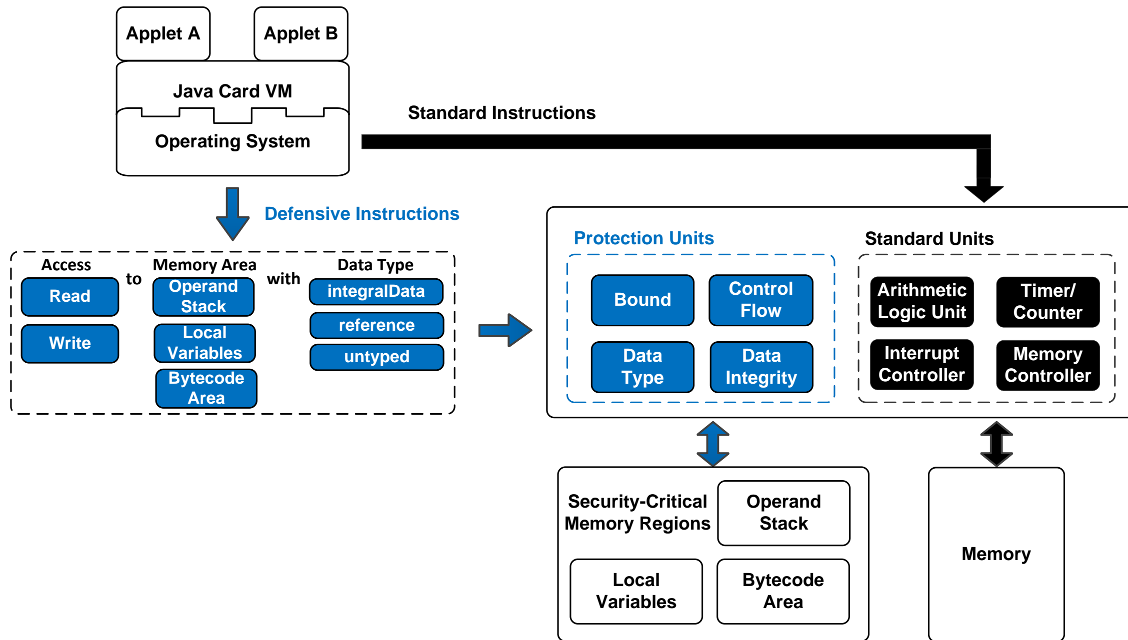


Figure 4.8: Overview of the integration of the protection units into the Java Card architecture. The instruction set of the 8051 processor is extended with new so called defensive instructions. These instructions decode different Java security related information which are used by the HW protection units to perform various high-performance run-time checks (obtained with modifications from [51]).

same accessed memory address to calculate $\psi = v \oplus v_{inverted}$. The IPU triggers a security interrupt when $\psi \neq$ 0xff .

The standard processor instructions are used to configure the standard HW units or standard memory regions on the card. A more detailed description of the case study of integrating the BPU, TPU, FPU, and IPU into an 8051 processor is presented in Section 6.3 [51].

4.2.5 Hardware and Performance Overhead

This section presents the HW and performance overhead measurements of the HW accelerated run-time security checks.

Hardware Overhead

The HW overhead of the newly added HW protection units (BPU, TPU, FPU, IPU) were evaluated on an VHDL 8051 processor model. To enable the new HW protection units additional processor instructions, new special function registers (SFRs), and additional interrupt lines were integrated into the model. Furthermore, the data width of the main memory interface has been increased. The digital HW overhead was received by emulating the system on a Virtex-6 FPGA development board. The HW overhead is listed in Table 4.2. The total HW overhead of the additional security features is 11%. Note that

the processor model does not contain any crypto-coprocessor or any other specific security techniques.

Table 4.2: This table shows the digital HW resources needed to add the HW protection units into the processor (obtained from [51]).

Hardware Components	Area Overhead
Bound Protection Unit	+5%
Type Protection Unit	+1%
Control Flow Protection Unit	+1%
Data Integrity Protection Unit	+1%
Defensive Instructions	+3%
Total	+11%

The type protection unit is responsible for checking if the VM executes the bytecodes with valid types on the OS and LVs. To store the type information, an additional type bit was added into the main memory word. This means that the main memory increases from 8 bit to 9 bit (8 data bit + 1 type bit). This main memory increase of 12,5% and the additional HW overhead of 11% are discussed in more detail in Section 6.3 [51].

Performance Overhead

The newly added HW protection units benefit from the fact that they can work in parallel to the normal processor instructions. The shifting of security operations from SW to HW significantly decreases the execution performance as shown in Table 4.3. Overall the HW checks increase the run-time by 4% which is very low compared to 160% if all checks are performed in SW. A more detailed analysis of the overhead for selected bytecode examples is presented in Figure 4.9. A more detailed discussion of the performance overhead is given in Section 6.3 [51].

Table 4.3: Performance overhead overview of the HW accelerated run-time checks. Note that the performance overhead measurements are normalized to a VM implementation which does not perform run-time checks (obtained with modifications from [51]).

Bytecode Groups	HW Accelerated Checks	SW Checks
Arithmetic/Logic	+6%	+180%
LV Access	+3%	+200%
OS Manipulation	+4%	+150%
Control Transfer	+4%	+120%
Array Access	+4%	+160%
Overall	+4%	+160%

4.3 Memory Efficient On-Card Verification

For a secure future Java Card every applet which is to be installed must be verified during the install process. Basic checks[6] can be quite easily performed during the installation of

[6] Michael Irauschek, "Java Card Attacks and Countermeasures against Malicious Applications in a User Centric Ownership Model", Master Thesis, Graz University of Technology, 2013

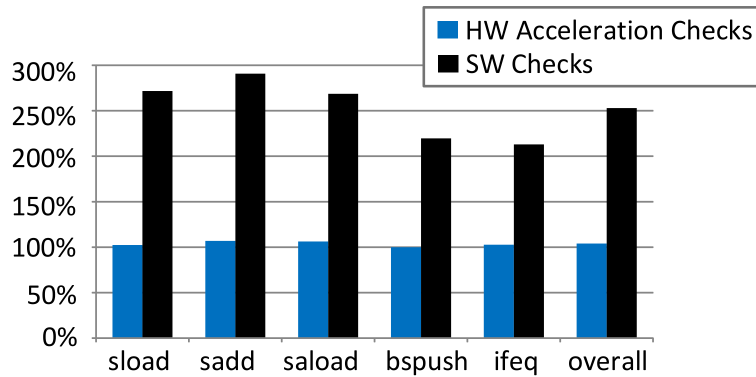
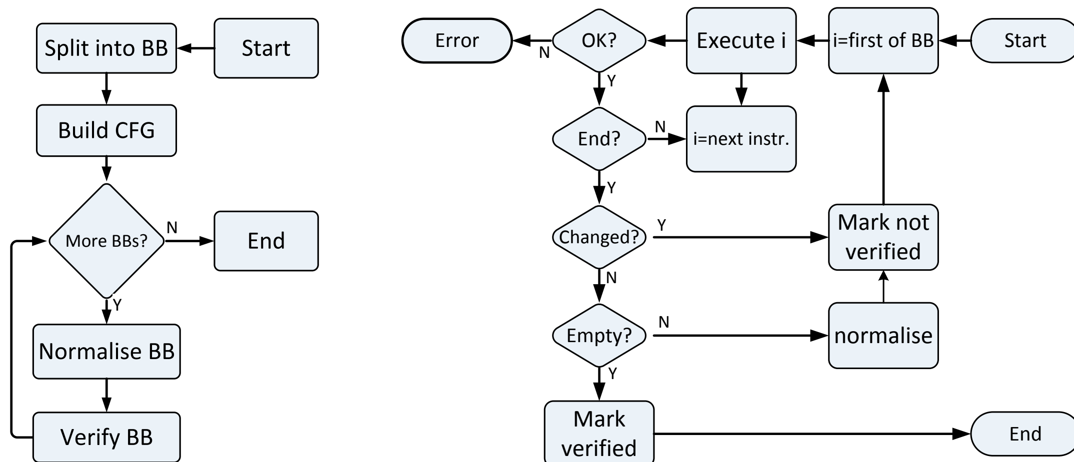


Figure 4.9: Performance overhead overview of the HW accelerated run-time checks. Note that the performance overhead measurements are normalized to a VM implementation which do not perform these type checks during run-time (obtained from [51]).

an applet on the card. These simple checks are for example: check that the applet contains no illegal opcodes; jump targets of bytecodes are inside the bytecode area; LV indexes of various bytecodes (e.g., *sstore*, *astore*) are inside the maximum number of allowed LVs.

A more sophisticated problem is the Java data type checking during install time. Therefore, this thesis presents a memory efficient on-card verification algorithm programmed in SW. A conceptual overview of the steps needed to verify a Java method is illustrated in Figure 4.10(a) and in more detail in Figure 4.10(b). The whole algorithm and its use case for future Java Cards is explained in more detail in Section 6.5 [53].



(a) Basic blocks are calculated over the bytecode of every method (obtained with modifications from [53]).

(b) Every bytecode instruction inside a basic block is checked during the statically performed verification (obtained with modifications from [53]).

Figure 4.10: Concept of the statically performed on-card verification process.

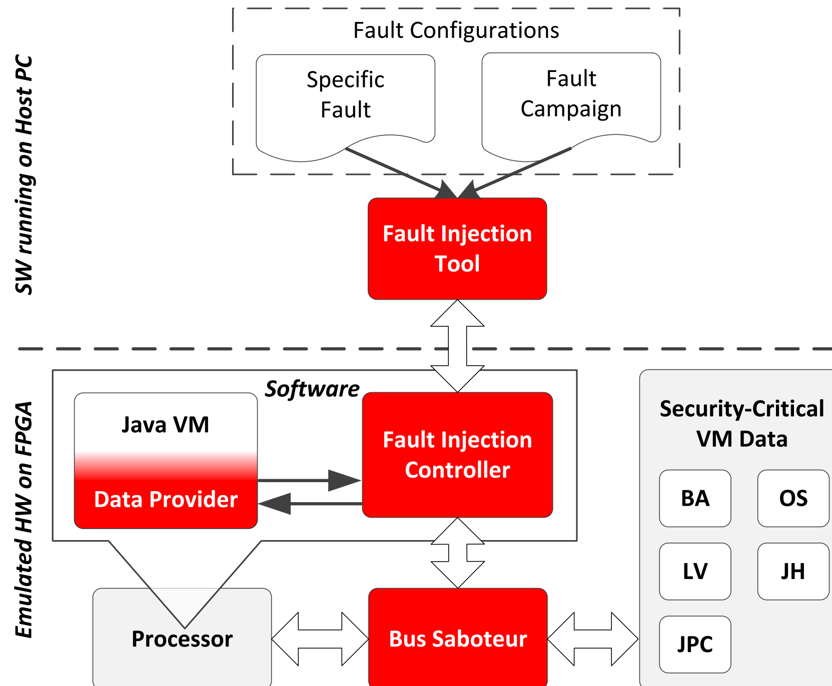


Figure 4.11: Detailed overview of the main parts of the fault emulation environment. Faults can be injected into the security-critical VM data of the memory. These faults are configured over the fault injection tool which configures the fault injection controller (obtained from [55]).

4.4 Fault Emulation Case Study

The prototype fault emulation environment[7] of this work is implemented on a Leon3 processor from Aeroflex Gaisler[8]. On this processor runs the freely available Java VM SimpleRTJ[9]. An overview of the main parts of the environment are shown in Figure 4.11. Specific faults or fault campaigns are configured over a fault injection tool running on a host computer. This tool automatically configures the fault injection controller which runs on the emulated Leon3 processor. The controller gets the actual state of the Java VM from the data provider and automatically configures the bus saboteur units for the appropriate faults. The bus saboteur unit is able to inject faults at different precisions like bit-precise, set-all-one, set-all-zero, indetermination, and inverting.

During a case study a Java wallet applet was attacked. The attack was based on manipulating the PIN verification by a fault injection into the operand stack memory region of the VM. Using this attack the money transfer functions of the wallet applet can be used without knowing the valid PIN. By the emulation approach a speedup of 6,600 is reached compared to a simulation shown in Table 4.4.

Table 4.5 shows the additional HW needed to enable the fault injection into the Leon3 processor. For example the wallet applet case study runs on the smallest possible config-

[7] Michael Hraschan, "Design and Implementation of a Fault Emulation Environment for a Java Virtual Machine", Master Thesis, Graz University of Technology, 2014

[8] www.gaisler.com/index.php/products/processors/leon3

[9] web.archive.org/web/20130803012237/www.rtjcom.com

Table 4.4: By the emulation approach a high acceleration of the fault injection is reached compared to a simulation (obtained from [55]).

Fault Injection Scenario	Simulation Time	Emulation Time	Achieved Speedup
Manipulate PIN Check	1 hrs 50 min	<1 s	6,600
Fault Campaign (500 Runs)	2 hrs	6.5 s	1,100

uration with an HW overhead of +14.5% FPGA look-up-tables and +13% FPGA slices. The HW overhead nearly linearly increases based on the number of configurable faults needed per executed bytecode. A more detailed description of the whole environment, the attacked wallet applet and the resources needed are described in Section 6.6 [55].

Table 4.5: The HW overhead of the fault emulation environment is listed based on the number of possible fault injections per bytecode (obtained from [55]).

Monitored Memory Regions	Fault Configurations per Memory Region	FPGA Look-Up-Tables	FPGA Slices
1	2	+14.5%	+13%
1	4	+28.3%	+24.7%
2	2	+28.4%	+27.1%
3	2	+46.8%	+41.7%
2	4	+64.2%	+57.7%
3	4	+85.5%	+68.9%

Chapter 5

Conclusions and Future Work

This section provides conclusions of this work for a HW/SW codesigned secure future Java Card. Furthermore, the directions for future work are presented with examples of recent publications.

5.1 Conclusions

Java Cards are increasingly used in different applications like mobile payment over NFC, specialized security-critical electronic chips like fingerprint sensors or as USIM cards in mobile phones. Java Cards are responsible for the confidentiality, integrity, and authenticity of the code and data stored and processed on these cards. The security needs of these cards will rise again in the future because of new use cases where everybody is allowed to install any applet on the card or the appearance of more advanced and precise attacks. Therefore, new countermeasure concepts are needed for these future Java Cards. These countermeasures must be carefully designed because of the resource constrained nature of these cards. Executing all countermeasures in SW would slow down the execution performance too much. Therefore, a security related HW/SW codesign must be performed to move countermeasures from SW to HW and evaluate different countermeasure design possibilities in a design space exploration phase.

This work shows this HW/SW codesign process for different security threats which threaten the security concept of the Java Card. These threats are memory overflow attacks, Java data type confusion attacks, control flow attacks, and data integrity attacks. First, different countermeasure concepts were conceptually proposed and evaluated based on their resource consumption. The evaluation takes into account the effectiveness of the countermeasures, and their run-time performance and memory consumption. Unfortunately, SW countermeasures add a significant execution time overhead on the applet execution. Therefore, this work proposes the shift of SW countermeasures to microarchitectural supported HW countermeasures. By moving functionality into silicon it is possible to significantly speed up the run-time memory bound checks, data type checks, control flow checks, and data integrity checks. The new HW checks are addressed over newly added processor instructions. These new instructions are used to perform read and write accesses to security-critical VM memory regions. Therefore, this work presents an application specific processor to increase the security against attacks on the Java VM

security model.

The new HW and SW countermeasures were integrated into a Java Card prototype running on two different smart card models. The first model is fast executing at a highly abstracted system level and used for fast design space exploration of countermeasures designs. The second model is running at a highly accurate register transfer level and was synthesized on a FPGA development board. The prototype VM was enhanced with an innovative so called defensive abstraction layer. This layer is used by the VM to integrate our newly proposed countermeasures and to access the security-critical memory regions of the card. The layer enables increased maintainability and flexibility of the countermeasures by their concentration into this layer. The run-time countermeasures of this work, either in SW or HW, are integrated into the Java Card prototype and evaluated based on their HW overhead, memory overhead, and performance overhead.

5.2 Directions for Future Work

The research directions for future work to increase the security of a Java Card against other security attacks is shown in Figure 5.1. Next research could concentrate on the operating system and HW layer of the Java Card to find new countermeasures using a HW/SW codesign approach. Furthermore, side channel attacks [62, 63, 64, 22] must be taken into account so that the card does not leak security-critical information to a possible attacker during execution. For example stabilizing or adding noise to the power supply of the card will hamper an attacker from creating a successful side-channel attack on the power supply.

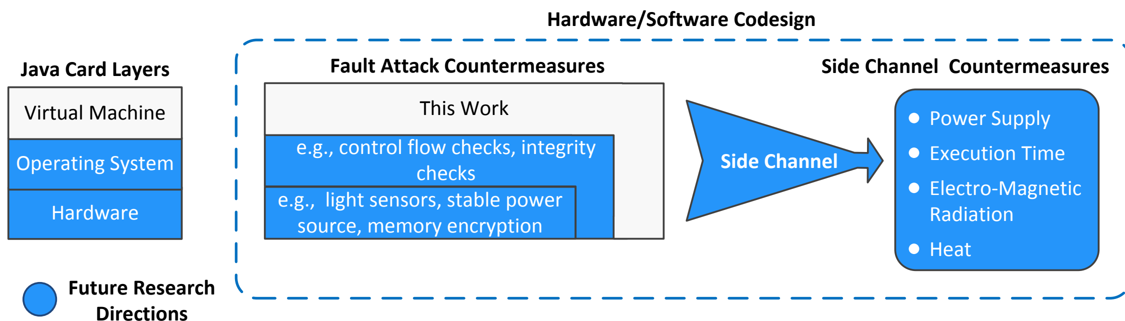


Figure 5.1: Future research could concentrate on adding countermeasures into deeper Java Card layers than this work. Furthermore, side-channel attacks must be taken into account during the design and implementation of countermeasures by novel design techniques, tools, or work flows.

5.2.1 Fault Attack Countermeasures on Lower Java Card Layers

HW glitch detection sensors (e.g., power supply glitch, digital clock glitch) could counteract the threat of manipulating security-critical data or operations on a smart card. Such a glitch detection sensor at a low Java Card layer as proposed in [65] would counteract attacks which manifest themselves on higher abstraction levels. A comprehensive analysis of various current SW countermeasures is shown in [66]. This countermeasure analysis could be used as a starting point for countermeasure design space explorations at the

operating system layer. The effectiveness of such SW countermeasures could be proved by formal verification as proposed by the authors in [67].

5.2.2 Side Channel Leakage Countermeasures

During the HW/SW codesign of different countermeasures the threat side channel attacks must be taken into account. The countermeasures must be designed keeping the threats of side-channel attacks in mind. A newly added countermeasure which creates a new security hole into the system using a side-channel attack could decrease the overall security more than the new countermeasure adds. Therefore, a direction for future research is novel side-channel countermeasures, exploring side channel threats using novel tools, and the automatic integration of side-channel countermeasures either in SW or HW into the card. In 2013 Hutter showed in his work [63] that heat can be used as a side channel and also a source of fault attacks. In 2014 Rodriguez shows a toolbox for differential power analysis attacks to smart cards [62]. These attacks must be counteracted by a novel side-channel related HW/SW codesign process. A possible side-channel countermeasure is published in 2014 by Gunman to counteract such attacks on cryptographic operations [64].

Chapter 6

Publications

This chapter presents the collection of publications which were created during this thesis. The publications explain the related work, methodology, results, and contributions of this thesis in a more detail.

Publication 1: *Towards the Hardware Accelerated Defensive Virtual Machine - Type and Bound Protection*, 11th Smart Card Research and Advanced Applications Conference (CARDIS), Springer Berlin Heidelberg, Lecture Notes in Computer Science, pages 1-15, Graz, Austria, 2012.

Publication 2: *A Defensive Virtual Machine Layer to Counteract Fault Attacks on Java Cards*, 7th Workshop in Information Security Theory and Practice (WISTP), Springer Berlin Heidelberg, Lecture Notes in Computer Science, pages 82-97, Heraklion, Greece, 2013.

Publication 3: *A Defensive Java Card Virtual Machine to Thwart Fault Attacks by Microarchitectural Support*, 8th International Conference on Risks and Security of Internet and Systems (CRiSIS), IEEE, pages 1-8, La Rochelle, France, 2013.

Publication 4: *Countering Type Confusion and Buffer Overflow Attacks on Java Smart Cards by Data Type Sensitive Obfuscation*, 1st Workshop on Cryptography and Security in Computing Systems (CS2), ACM, pages 19 - 24, Vienna, Austria, 2014.

Publication 5: *Memory-Efficient On-Card Byte Code Verification for Java Cards*, 1st Workshop on Cryptography and Security in Computing Systems (CS2), ACM, pages 37 - 40, Vienna, Austria, 2014.

Publication 6: *A Fault Attack Emulation Environment to Evaluate Java Card Virtual-Machine Security*, 17th Euromicro Conference on Digital Systems Design (DSD), Verona, Italy, 2014, in press.

The publications in this chapter discuss various HW/SW codesign aspects for a future secure Java Card. This secure Java Card enables the secure installation and execution of different applications. Various checks and countermeasures against attacks are proposed in this thesis. The checks and countermeasures are either implemented in SW or HW.

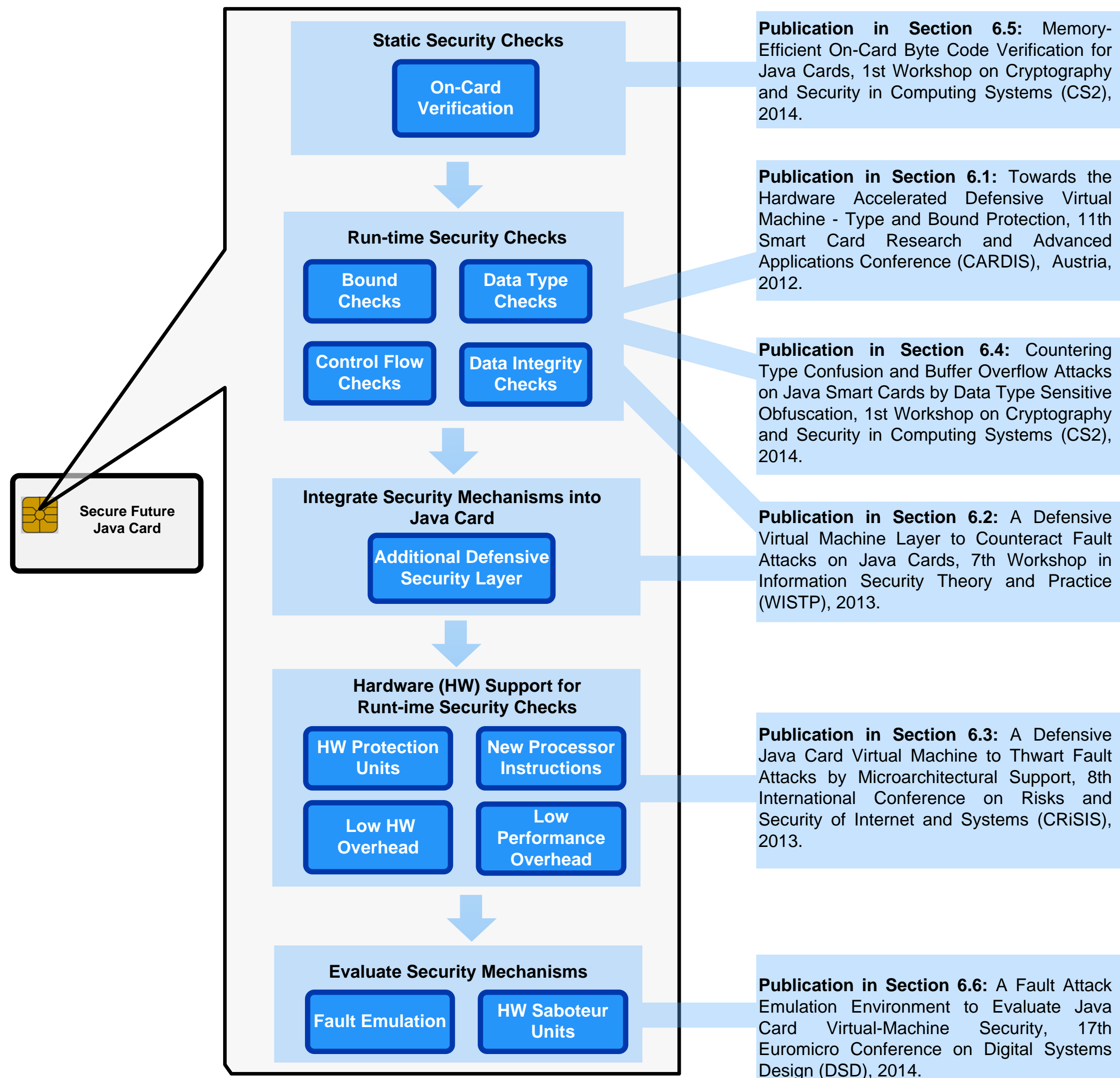


Figure 6.1: Enable a secure future Java Card by various security mechanisms.

The publication in Section 6.5 shows a memory efficient approach for a static applet verification process. When a verified applet is executed various run-time checks are performed to counteract run-time attacks. Different concepts of such run-time checks are proposed in Sections 6.1, 6.4, and 6.2. To integrate these security checks into a Java Card a new security layer is proposed in Section 6.2. To speed up the run-time checks we moved the checks from SW to HW which is shown in Section 6.3. To evaluate and test the security mechanisms in SW and HW we created a fault attack emulation environment which is presented in Section 6.6.

Towards the Hardware Accelerated Defensive Virtual Machine – Type and Bound Protection

Michael Lackner[1], Reinhard Berlach[1], Johannes Loinig[2], Reinhold Weiss[1], and Christian Steger[1]

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

Abstract. Currently, security checks on Java Card applets are performed by a static verification process before executing an applet. A verified and later unmodified applet is not able to break the Java Card sand-box model. Unfortunately, this static verification process is not a countermeasure against physical run-time attacks corrupting the control or data flow of an applet. In this piece of work, designs for Java Card Virtual Machines are investigated in relation to their ability to perform run-time security checks. These security checks are accelerated by hardware units and performed in parallel to CPU instructions that are executing concurrently. Attacks on the Java operand stack and local variables, which are elementary components for the Virtual Machine, are thwarted by type and bound protection. To enable these hardware checks, different designs of a defensive Java Card Virtual Machine are compared to their overheads on a prototype platform.

Keywords: Java Card, Defensive Virtual Machine, Hardware Countermeasure, Fault Attack, Logical Attack.

1 Introduction

Current applied static verification of Java Card applets provides insufficient security protection against run-time Fault Attacks. This is especially a problem in the field of multi-application Java Cards. In this field, cards are used in a wide range of applications (e.g., passport, e-money) and have the ability to perform post-issuance loading of new applets. An adversary provoking a Logical Attack by changing the bytecode or internal representation of an uploaded Java applet can get access to security related data from other applets or the Java Card Virtual Machine (JCVM) [12]. To thwart Logical Attacks, verification of applets is performed either off-card or on-card. This verification procedure is currently a static process performed once before an applet is executed. One of the most time and memory consuming checks performed is the bytecode verification [9,15].

Java Card applets are stored into non-volatile memories such as EEPROM. With the help of physical Fault Attacks it is possible for a JCVM to read out

S. Mangard (Ed.): CARDIS 2012, LNCS 7771, pp. 1–15, 2013.
© Springer-Verlag Berlin Heidelberg 2013

© 2013 Springer-Verlag Berlin Heidelberg. Reprinted, with permission, from Proceedings of 11th Smart Card Research and Advanced Applications Conference (CARDIS).

incorrect values from these memories or skip CPU instructions. Therefore, it is possible to change the bytecode of stored applets to execute ill-formed Java instructions. Knowledge about these attack possibilities is used in [3] to create a new class of attacks called Combined Attacks. To perform Combined Attacks, applets which pass the verification process are used and become malicious in combination with a Fault Attack. Combined Attacks are used to bypass the Java Card sand-box, mounted by the static verification process, JCVM and Java Card Runtime Environment [11].

To guard the Java Card against run-time attacks, a so called defensive JCVM is needed [4]. This defensive JCVM can be reached by performing all checks done by the static verification process during run-time. However, this is currently not achievable because of the constrained hardware resources of today's Java Cards. In this work specific security checks, extracted from the Java Card specification [12], are performed on the executing bytecode during run-time. In this specification the data flow is exactly defined for every bytecode with some additional constraints which must be fulfilled.

To speed up bytecode checking during run-time, new hardware protection units are introduced in this work to speed up the checks performed on every bytecode. These hardware checks can be performed in parallel while the CPU performs its operations. Therefore hardware checks are a good solution for run-time checks that are performed very often, in contrast to software checks. Software checks slow down the whole system if they are performed on the same CPU that the standard operations are performed on. Beside this benefit, hardware checks also have the advantage of being more immune against additional fault injects onto the Java Card. This is due to the fact that software checks [13,5,2] are vulnerable to skipping them by additional Fault Attacks. This threat of skipping software security operations leads Vertanen in [16] to the conclusion that hardware assisted run-time checks are mandatory for enhancing run-time security for Java Cards.

This work introduces a hardware accelerated defensive JCVM which performs selected security checks on the executing bytecodes by hardware with a low computational overhead. These checks are also part of the verification process which is done statically before executing a Java applet. As far as we know, such a hardware accelerated defensive JCVM has not been introduced in literature before.

The contribution of this work is the definition of a run-time security policy extracted from the Java Card specification [12] to ensure that the executing bytecode performs valid operations. This run-time policy prevents type confusion and overflow/underflow attacks on the operand stack (OS) and local variable (LV) memory inside the JCVM. With this policy it is not possible for an adversary to perform type confusion between values of type *integralData* and object *references* on the OS and LV. Furthermore, two hardware accelerated defensive JCVM designs are presented with their main parts, such as additional hardware protection units and new CPU instructions. The new CPU instructions are used inside the JCVM to process bytecodes and communicate directly with the

hardware protection units leading to a very low computational overhead. This communication is depicted in Figure 1.

Section 2 gives an overview of attacks on Java Cards, bytecodes violating the Java frame bounds and how to enable a defensive JCVM. Section 3 describes the security policy and the design of all defensive JCVMs introduced in this work. Section 4 presents the prototype implementation of these designs on a SystemC 8051 derivate. Section 5 analyses the run-time costs on execution speed and hardware changes needed to activate our JCVM designs. Finally, conclusions and future work are drawn in Section 6.

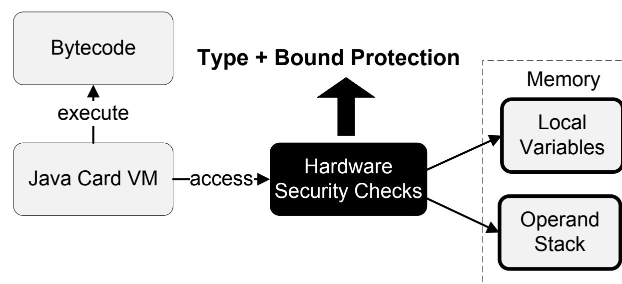


Fig. 1. In this work the operand stack and the local variables are protected during run-time by hardware accelerated security checks

2 Related Work

In this section an overview of possible attacks on the Java Card is given with focus on run-time fault attacks. Following this, previous work on run-time countermeasures and an overview of defensive JCVMs are presented.

2.1 Attack Overview

Attack scenarios on Java Cards are manifold [18,19]. Side Channel Attacks are used to draw conclusions of internal operations by studying physical phenomena of the chip. Invasive Attacks are used for optical or measurement analysis of internal components. Fault Attacks (FA) change the physical environment of the chip under attack [1]. These are for example, temperature changes, additional light of a laser or spikes in the power supply or clock source. These FA lead to an undefined behavior of the chip by skipping instructions or read/write errors to memory like the EEPROM. This is especially a problem for post-issuance loaded applets due to the fact that they are mostly installed in non-volatile memory. Therefore, a FA during the fetch process of a JCVM can lead to ill-formed applets even if a static verification was performed. This ill-formed code enables an adversary to circumvent the Java Card security model and enables an applet to have access to unauthorized resources. This security problem of FA to verified applets is well known in literature and is used to enable different attack paths [10,3,17,14].

2.2 Frame Bound Violation Attacks

Generally, inside every Java Frame, specific memory areas are reserved for OS, LV and internal frame data. Every time a new Java method is invoked, the JCVM creates a new frame and pushes it onto the Java stack. Specific implementation details for the Java Frame are not provided by Java Card specification. Therefore, the specific frame data depends on the particular implementation. In general the frame data contains a return address so that it is possible to return to the code of the old frame. The size needed for a frame and all its containing elements (e.g., OS, LV) is ascertained when the method is invoked and is not changing during method execution.

Ill formed bytecode can now access illegal memory regions by performing an OS or LV out of bound access as illustrated in Figure 2. In [5] an attack called EMAN2 was performed. There an invalid LV index was used by an ill formed bytecode *sstore* to access the memory region of the frame data where the return address of the current frame is stored. With the help of this ill formed bytecode, an adversary can set the return address to any value. In their attack the return address was set to the address of an array which leads to the security threat of executing adversary definable data. This illegal execution of data opens new security issues not treated here in detail. The threats of OS overflow/underflow and bytecodes using invalid LV index are thwarted by the run-time policy of this work.

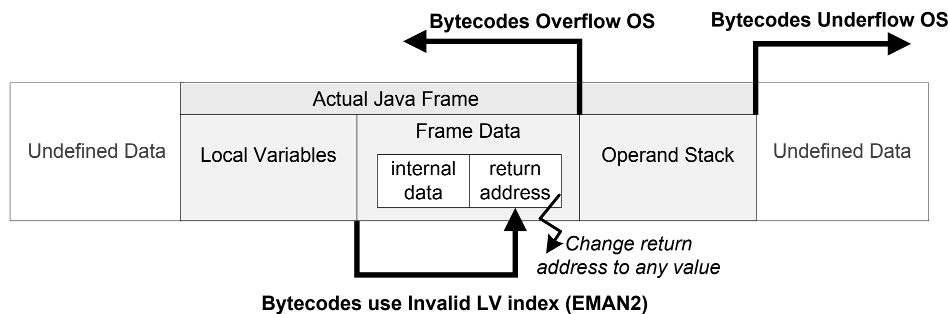


Fig. 2. OS overflow and underflow leads to illegal memory access outside the reserved OS memory space. An adversary who uses bytecode with invalid LV index can overwrite the return address of the current active frame [5].

2.3 Enabling a Defensive Virtual Machine

Currently the Java Card research community concentrates on finding attack paths to bypass the Java Card security model by FAs and Combined Attacks. [3,10,17,14]. Also a lot of effort is invested in exploiting and thwarting Side Channel Attacks [8]. In contrast to these big research topics, the question of how to enable a defensive JCVM is a research topic with little public attention. However, in the Java Card industry the know-how to enable defensive JCVM designs is of course available. This fact is proven by different works bringing

the defensive nature of current available industrial Java Card products to light [10,7]. Techniques and knowledge that provide such a defensive design are of course not freely available. Currently research related to FA countermeasures is focussed on static verification of an applet and checking that the exact verified code is executed. This can be done by code integrity and control flow checks in software (SW) during run-time [13,5]. The annotations that enable these checks are stored in an additional component of a verified CAP-file. Research was also done to check the OS integrity against FA by performing double reads by SW [2].

This work focuses on a hardware accelerated defensive JCVM performing security checks during run-time. Based on a policy it checks if the executing bytecode is behaving correctly. Compared to current countermeasures in literature the approach in this work does not just check the integrity of the bytecode or OS, it performs checks based on a policy. This approach stops either manipulated applets loaded onto the card or run-time FA from violating this policy. Furthermore, performing these checks in hardware makes it more resistant to additional FA which are also able to skip additional software checks.

3 Design of the Defensive Virtual Machine

In this section the run-time security policies for all defensive JCVM designs in this work are shown. This is followed by our method of reducing all Java data types to two main types.

3.1 Defensive Run-Time Policy

The OS and LV, located in the Java Frame, are main parts of the JCVM. The JCVM is a stack machine and performs most operations on the OS. Therefore, securing these parts of the JCVM are the first steps to hampering or stopping Fault Attacks during run-time. The following two main policies for the OS and LV are retained by our defensive designs during run-time:

- **Frame Type Policy:** All bytecodes which access the OS or LV must use the right main data type (*integralData* or *reference*) which is expected by the bytecode during its execution. In this work all numerical types are combined (*boolean*, *byte*, *short*) to the main data type *integralData*. All object references (e.g., *short array*, *byte array*, *Class A*) are combined to the main data type *reference*.
- **Frame Bound Policy:** Bytecodes operating on the OS or LV are not allowed to access data outside the frame bounds. This means that bytecodes are not allowed to overflow or underflow the OS. Furthermore, all bytecodes accessing the LV must be inside the borders of the reserved LV memory area.

Policy Creation: The two policies above were extracted from the JCVM specification [12] where for every bytecode a textual description of the operation is

given. In this specification it is for every bytecode defined from which JCVM component needed operands are taken and results are written back. Also the type information is specified for every operand and result value. Such a bytecode specification for the *sstore* instruction is listed below. This bytecode consists of two bytes, the opcode (0x29) and an index referencing to an item of the LV.

> *"The index is an unsigned byte that must be a valid index into the local variables of the current frame (Section 3.5, "Frames"). The value on top of the operand stack must be of type short. It is popped from the operand stack, and the value of the local variable at index is set to value."* [12]

The requirement that bytecodes perform no OS stack overflow/underflow, access the right LV index and operate with the right types on the OS and LV is crucial for the security concept of the Java Card and therefore checked by all defensive JCVM designs of this work.

3.2 Design of the Defensive JCVMs

In this section we introduce two designs for a defensive JCVM which fulfill the security policies defined in the previous section. A general overview of the defensive JCVM designs is shown in Figure 3 and described how they are used in more detail below. Note that our defensive JCVM designs are not able to thwart all sort of attacks on a Java Card. Examples of such undetected attacks are control flow changes (skipping a branch instruction) or data corruption (read corrupted values from the RAM).

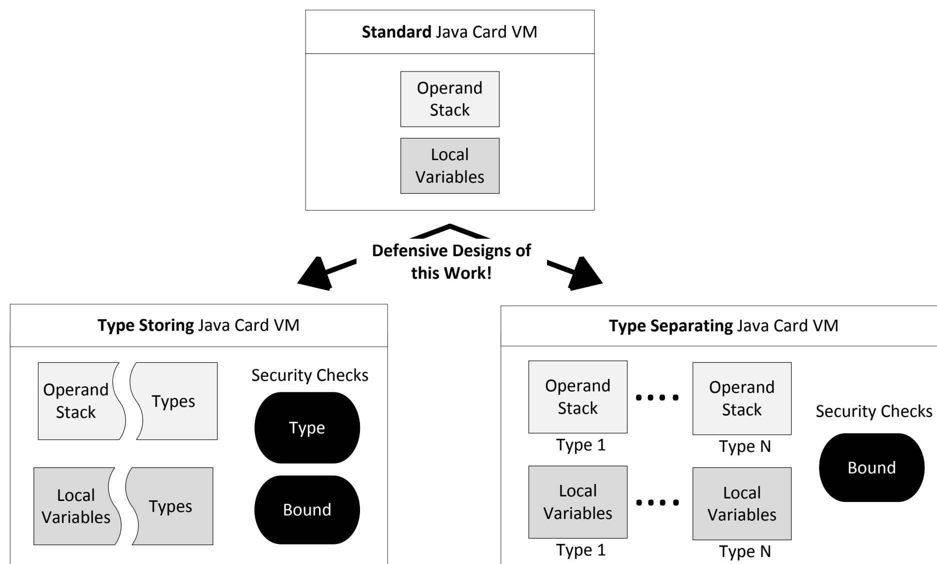


Fig. 3. In this work two designs are used to fulfill the run-time security policy

- **Type Storing:** Every entry on the OS or LV is extended with type information in order to distinguish between *integralData* and *reference* during run-time. During run-time it is now possible to check if the expected type for the bytecode is on the OS or LV which is the obvious defensive approach to enable a Defensive JCVM. A disadvantage of this approach is the additional memory needed for type storing and the computational overhead to perform type checking.
- **Type Separating:** Every Java main data type (*integralData* and *reference*) operates on its own OS and LV memory area. No general OS and LV area where all data types occur exists. Type confusion between the two main data types is therefore no longer possible during run-time because every bytecode always receives the right type. A disadvantage of the Type Separating design is that an attack is only detected by its security related side effects on the current frame. Such a side effect is for example an OS underflow for a specific type.

3.3 Two Types for Type Storing and Type Separating

In this section we introduce our approach for separating the Java Card types into two main data types to enable the Type Storing and Type Separating JCVM that was presented in the previous chapter. Java bytecodes are highly typed. This means that based on the data type different opcodes exist for the same operation [12, Table 3-1]. For example, only the *sstore* bytecode is allowed to push integral data types (boolean, byte, short) into the LV. Another Java bytecode is used to store an element of type *reference*, pointing to an object, into the LV. It is therefore possible to differentiate between two main data types and distinguish them just by looking at the bytecode. In this work they are called *integralData* and *reference*:

- ***integralData.*** These are the primitive constant data types that represent the numerical values of the JCVM: *boolean*, *byte*, *short*. Elements of this data type can be deliberately created by executing the bytecode *sconst_1*. This bytecode pushes an integral value 1 with type *short* on the OS.
- ***reference.*** These are all kinds of *references* to objects and the *returnAddress* type to enable sub-routines. An applet programmer can only indirectly create elements of type *reference*. For example the bytecode *new_array* pushes the reference of a newly created array object onto the OS. This address can have any logical structure and does not have to correlate with physical addresses of objects stored on the card. For example the JCVM can create a random number and a look up table maps this number to a real memory address.

By separating these two main data types it is no longer possible for an adversary to create object references with a defined value by confusing *integralData* and *reference*. It is also not possible for an adversary to get deeper insight into how the JCVM represents references. For this insight an adversary would have to perform type confusion between *reference* and *integralData* by sending the reference out of the card from the APDU buffer. The APDU class is responsible for receiving and sending data to off-card applications.

Thwarted Threats in Literature. This sort of type confusion between integral data and object references is well known and often used as the first step of an attack path [16,10,7,5,17]. This attack path can enable an adversary creating self mutable code by executing data from a Java array [7,5] or even gain access to forbidden methods of objects [17].

Note that type confusion between different objects such as *short array* and *Class A* object is not detected by the defensive JCVM designs in this work. This is because all object references are assigned to the *reference* main data type and cannot be distinguished. This determination also applies to the main data type *integralData* where it is not possible to detect type confusion between *byte* and *short*.

4 Prototype Implementation

In this work five different prototype JCVMs were implemented in C and assembly language. The JCVMs are based on the Classic Edition of the Java Card specification [12]. The hardware (HW) platform on which they run is an 8-bit Smart Card model written in SystemC [6]. This model is memory and instruction cycle accurate. Into this HW platform new typed CPU instructions were implemented. Furthermore, additional HW protection units were added to enable HW accelerated security checks for bytecodes accessing the OS and LV memory area.

4.1 Additional CPU Instructions

The information decoded into the new CPU instructions is illustrated in Figure 4. New typed CPU instructions are used by our JCVMs to process the bytecodes and perform access to the OS and LV memory regions. These decoded pieces of information are the access type (Read, Write), the destination of the accessing memory (OS, LV) and the type which should be written/read (*integralData*, *reference*, *untyped*). With the help of these pieces of information the protection units are able to check if the new CPU instruction doesn't perform a security policy violation during run-time. Such a violation is for example a LV element address which is outside the actual LV memory bounds.

Two examples of how to use these new instructions inside the JCVM program code in order to process the Java bytecodes is outlined in Figure 5. The *sadd* bytecode first reads two values from the OS by the new CPU instruction *Read_OS_integralData*. The result of the addition of these values is then written back by the instruction *Write_OS_integralData*. Another example is the bytecode *astore_0*. This bytecode uses the CPU instruction *Read_OS_reference* to read a reference value from the OS and stores it into the LV by the instruction *Write_LV_reference*. The big advantage in the sense of computational overhead of the new typed CPU instructions is that the JCVM can communicate very effectively with the HW protection units by using the new CPU instructions.

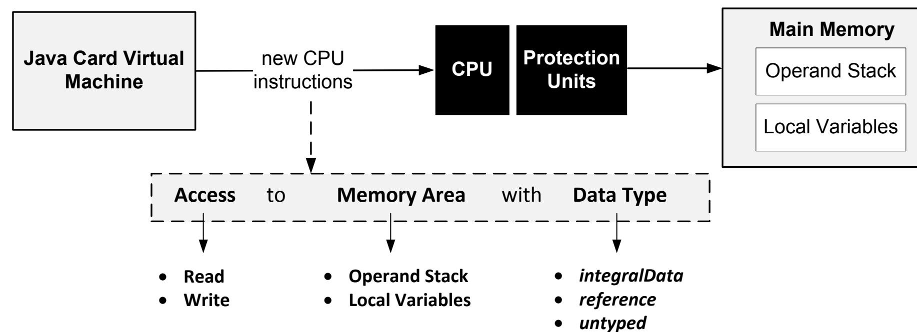


Fig. 4. The prototype JCVMs proposed in this work uses new assembly instructions to access the run-time protected OS and LV memory regions

```
sadd:                               //Add two short values on the OS
   short VAR1, VAR2, SUM;           //Create variables for sadd
   VAR1 = Read_OS_integralData;     //Read first operand from OS
   VAR2 = Read_OS_integralData;     //Read second operand from OS
   SUM  = VAR1 + VAR2;              //Sum the two operands
   Write_OS_integralData = SUM;     //Write the sum back onto the OS

astore_0:                           //Store reference from OS to LV
   short REF1;                      //Create variables for astore_0
   REF1 = Read_OS_reference;        //Read reference from OS
   Write_LV_reference(0) = REF1;    //Write reference into LV element 0
```

Fig. 5. Pseudocode example of the two bytecodes *sadd* and *astore*, processed by the JCVM. The JCVM uses our new CPU instructions to access the OS and LV memory.

4.2 Additional Hardware Protection Units

An overview of the new CPU instructions and the protection units needed to activate the Type Storing and Type Separating JCVM is presented in Figure 6. Based on the new CPU instructions introduced in the previous section, our HW protection units restrict the access to the security critical memory regions of the OS and LV. The Type Storing JCVM needs a type protection unit to check if the type expected by the bytecode is also available on the OS or LV.

- **Bound Protection Unit (BPU):** This unit is responsible for thwarting attacks performing an OS overflow or underflow. Furthermore, all bytecodes accessing the LV using a wrong index are detected. The BPU is used by the Type Storing and Type Separating JCVM prototype.
- **Type Protection Unit (TPU):** The TPU is responsible for checking that the bytecodes that are accessing the OS and LV are operating with the right data type. The TPU is only needed to enable the Type Storing JCVM.

4.3 Type Storing JCVM Implementation Details

To enable a Type Storing JCVM, the TPU must store additional type information for every element held by the OS and LV. Due to the fact that these two parts are located in RAM, one additional type bit was added to every 8-bit word. This bit enables the distinction between the two main data types *integralData* and *reference*.

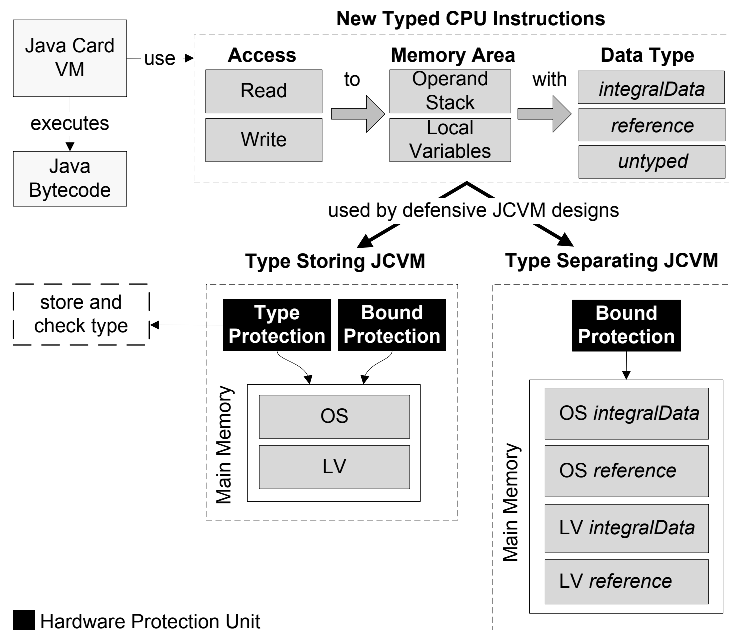


Fig. 6. Implementation overview of the two JCVM prototypes. Hardware protection units perform run-time checks on the OS and LV.

4.4 Type Separating JCVM Implementation Details

In this section we give insight into the detail of how the Type Separating JCVM was implemented and describe a tool chain to enable it. The Type Separating JCVM performs all bytecode operations on the right typed OS and LV. This Type Separating approach avoids type confusion. The type checking problem is reduced to a bound checking problem.

Most bytecodes work well with our run-time type separating approach to two main data types (*integalData*, *reference*). An exception are the bytecodes operating with undefined types on the OS: *pop*, *pop2*, *dup*, *dup2*, *dup_x* and *swap_x*. In this paper we call them untyped bytecodes. For these untyped bytecodes the JCVM does not know on which of the two separated OS, specific operations are performed during run-time.

As a solution for this problem the missing type information was added directly into the bytecode by using unused bytecodes which are not defined in the Java Card specification. For example, the *pop* instruction is either convert

to *pop_reference* or *pop_integralData*. The JCVM now knows right after fetching a new bytecode, which typed OS it must operate on. Therefore, no execution speed is wasted searching for the type information in additional components uploaded on the card. In the JCVM specification [12] only 185 (0x00 to 0xb8) of all available 8-bit bytecodes are specified. The unused bytecodes from 0xb9 to 0xfd can be used to decode the operand stack type information that is needed directly into the instruction.

To perform the exchange of untyped bytecodes with new typed bytecodes we propose a static replacement process performed once for every method. To speed up this process, type information obtained during the bytecode verification process can be used to exchange the untyped bytecodes with typed ones. However, in this work the replacement process for the untyped bytecodes is not looked at in detail.

5 Prototype Results and Discussion

In this section we show the computational overhead coming from full software (SW) implementations compared to running our HW accelerated prototypes. The SW implementations perform the same security checks that are performed by the HW protection units. Furthermore, the additional hardware overhead is compared between all prototypes.

5.1 Computational Overhead

Performing all security checks in SW increases the computational overhead significantly for frequently executed bytecodes, as illustrated in Figure 7. For example the *sload* bytecode executed by a Type Storing prototype in SW has a computational overhead of around 107% caused by the following run-time SW operations:

- Check if the index parameter to the LV is valid.
- Check if the element at the LV index is of type *integralData*.
- Check if pushing a value from the LV index to the OS provokes an overflow.
- Store the fact that the new value on the OS is of type *integralData*.

If the *sload* bytecode is executed on a HW accelerated prototype the overhead decreases to 5%. In Table 1 different groups of bytecodes are compared to their computational overhead. As expected, the HW accelerated prototypes consume much less computational overhead compared to prototypes which implement the checks in SW.

5.2 Hardware Overhead

In this section we give an overview of the HW modifications used to activate our HW accelerated prototypes, as depicted in Table 2. The instruction set of a standard 8051 microcontroller consists of 255 opcodes. Adding our new CPU

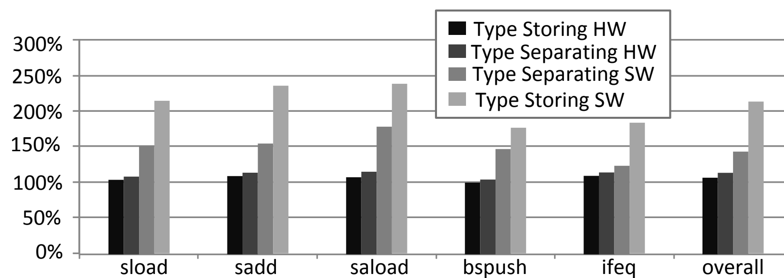


Fig. 7. Run-time measurement for specific bytecodes and the overall time of all implemented bytecodes for different JCVM implementations. Measurements are normalized to a JCVM without any run-time security checks.

Table 1. Computational overhead for all prototypes, normalized to a JCVM without performing run-time security checks

	Type Storing		Type Separating	
Bytecode Groups	HW	SW	HW	SW
1: Arithmetic/Logic	+7%	+123%	+7%	+47%
2: Local Variable Access	+5%	+152%	+9%	+52%
3: Operand Stack Manipulation	+5%	+119%	+3%	+54%
4: Control Transfer	+8%	+77%	+9%	+23%
5: Array Creation/Manipulation	+6%	+111%	+9%	+59%
Overall	+6%	+107%	+8%	+38%

Table 2. HW modifications needed to activate the HW accelerated run-time security checks of the prototypes

Additional Hardware	Type Storing	Type Separating
New 8051 CPU Instructions	9 (+3,5%)	8 (+3,1%)
New 8-bit Control Registers (SFRs)	11 (+52,4%)	15 (+71,4%)
New Bound Protection Unit (BPU)	Yes	Yes
New Type Protection Unit (TPU)	Yes	No
Extend RAM word with type bit	Yes	No

instructions means an overall CPU instruction increase of around only 3,5%. Another important hardware modification is that the RAM module of the HW accelerated Type Storing JCVM was extended with an additional type bit for every memory word in order to differ between the main data types *integralData* und *reference*. Therefore, overall RAM memory size increases to 12,5%.

5.3 Type Confusion Attack Example

An example of a run-time attack on the Java Card prototypes in order to perform type confusion between *integralData* and *reference* is illustrated in Figure 8.

There, a run-time attack changes the *bspush* code 0x10 0x19 to 0x00 0x19. The JCVM interprets 0x00 as NOP instruction and performs no action. The following byte 0x19 is interpreted as the bytecode *aload_1* which pushes an array reference onto the OS. The *sreturn* instruction would now take the array reference and push it back to the calling function. An adversary is now able to use the array *reference* as an *integralData* which enables different attack paths such as executing the data inside the array [16,10,7,5,17]. Both defensive JCVMs designs from this work are able to thwart this type confusion attack on the OS between *integralData* and *reference*.

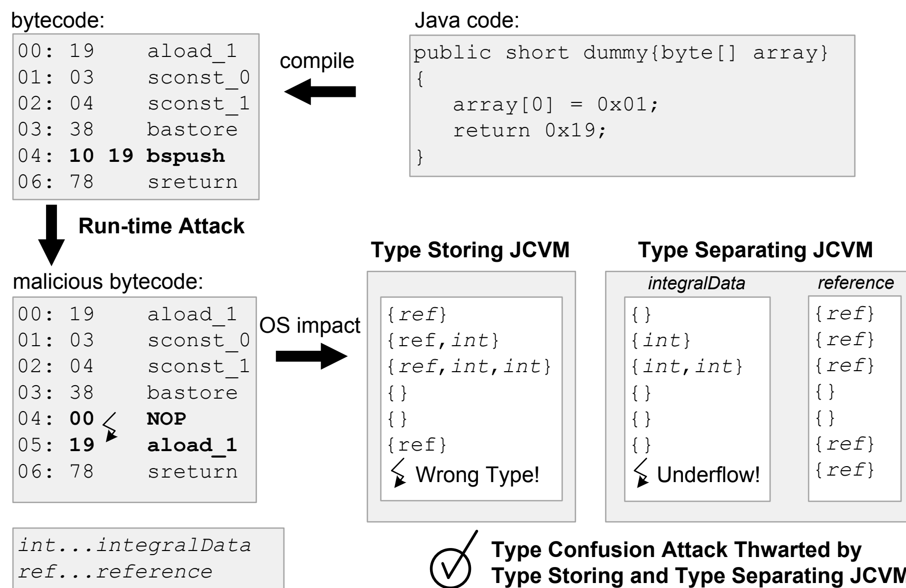


Fig. 8. Run-time type confusion attack on the OS to receive the address of an array. All defensive JCVM implementations of this work are able to thwart this attack.

Type Storing: By using the new typed CPU instructions, it is decoded inside the JCVM code that the *sreturn* instruction expects a value of type *integralData* on the OS. The previously executed instruction *aload_1* pushed the reference of an array with type *reference* on the OS. Therefore, the TPU hardware module finds the wrong type for the *sreturn* bytecode on the OS and throws a security exception.

Type Separating: Both main data types have their own OS and LV memory areas during run-time. Therefore, the malicious instruction *aload_1* pushes an array reference onto the *reference* OS containing all values from type *reference*. The *sreturn* bytecode tries to pop data from the *integralData* OS which is empty. The return operation is now aborted by a security exception because an underflow on the *integralData* OS is detected by the BPU hardware module.

6 Conclusion and Future Work

This work presents Java Card Virtual Machine (JCVM) designs to counter different Fault Attacks. This is done by performing run-time security checks based on a security policy for each bytecode. These policies ensure that each bytecode which operates on the operand stack or the local variables memory area uses the right data type (*integralData* or *reference*). Furthermore bytecodes which overflow or underflow the OS or LV are detected. These run-time checks are accelerated by hardware protection units to make it harder to skip these checks with additional Fault Attacks. Furthermore, the defensive JCVM designs are profiting from the parallel execution of the hardware checks by having very low computational overhead.

In this work the design of a Type Storing and Type Separating JCVM were shown. Both designs were implemented on a Java Card prototype platform with several additional hardware changes. The requirements of the hardware to enable run-time checking by hardware protection units were listed for both defensive JCVM designs. We measured that these hardware accelerated prototypes consume 6% and 8% more execution time overall compared to a JCVM without any additional run-time security checks. This overhead is very low compared to prototypes which perform all run-time security checks in software and consume around 107% and 38% more execution time. Therefore, we have shown that our approach of performing additional security checks during run-time by using hardware units is feasible especially in the case of resource constrained Java Cards.

For future work we will focus on the bytecode replacement process of untyped bytecodes required by the defensive Type Separating approach. This transformation is needed to give the JCVM type information needed during run-time to process untyped bytecodes like *pop*. Furthermore, we are working on increasing the number of separated main types so that it is also possible to detect type confusion between *integralData* like *short* and *byte*.

Acknowledgement. The authors would like to thank the Austrian Federal Ministry for Transport, Innovation, and Technology, which funded the CoCoon project under the FIT-IT contract FFG 830601. We would also like to thank our project partner NXP Semiconductors Austria GmbH.

References

1. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The Sorcerer's Apprentice Guide to Fault Attacks. Proceedings of the IEEE 94(2), 370–382 (2006)
2. Barbu, G., Duc, G., Hoogvorst, P.: Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 297–313. Springer, Heidelberg (2011)
3. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) CARDIS 2010. LNCS, vol. 6035, pp. 148–163. Springer, Heidelberg (2010)

4. Barthe, G., Dufay, G., Jakubiec, L., de Sousa, S.M.: A Formal Correspondence between Offensive and Defensive JavaCard Virtual Machines. In: Cortesi, A. (ed.) VMCAI 2002. LNCS, vol. 2294, pp. 32–45. Springer, Heidelberg (2002)
5. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.-L.: Combined Software and Hardware Attacks on the Java Card Control Flow. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 283–296. Springer, Heidelberg (2011)
6. IEEE: Open SystemC Language Reference Manual IEEE Std 1666-2005, IEEE
7. Iguchi-Cartigny, J., Lanet, J.L.: Developing a Trojan applets in a smart card. Journal in Computer Virology 6, 343–351 (2010)
8. Krieg, A., Grinschgl, J., Steger, C., Weiss, R., Haid, J.: A Side Channel Attack Countermeasure using System-On-Chip Power Profile Scrambling. In: 2011 IEEE 17th International On-Line Testing Symposium (IOLTS), pp. 222–227 (July 2011)
9. Leroy, X.: Java Bytecode Verification: An Overview. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 265–285. Springer, Heidelberg (2001)
10. Mostowski, W., Poll, E.: Malicious Code on Java Card Smartcards: Attacks and Countermeasures. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 1–16. Springer, Heidelberg (2008)
11. Oracle: Runtime Environment Specification. Java Card Platform, Version 3.0.4, Classic Edition (2011)
12. Oracle: Virtual Machine Specification. Java Card Platform, Version 3.0.4, Classic Edition (2011)
13. Séré, A.A.K., Iguchi-Cartigny, J., Lanet, J.-L.: Checking the Paths to Identify Mutant Application on Embedded Systems. In: Kim, T.-H., Lee, Y.-H., Kang, B.-H., Ślęzak, D. (eds.) FGIT 2010. LNCS, vol. 6485, pp. 459–468. Springer, Heidelberg (2010)
14. Sere, A., Iguchi-Cartigny, J., Lanet, J.L.: Evaluation of Countermeasures Against Fault Attacks on Smart Cards. International Journal of Security and Its Applications 5(2), 49–61 (2011)
15. Sun Microsystems Inc.: Java Card 2.2 Off-card Verifier. White Paper (June 2002)
16. Vertanen, O.: Java Type Confusion and Fault Attacks. In: Breveglieri, L., Koren, I., Naccache, D., Seifert, J.-P. (eds.) FDTC 2006. LNCS, vol. 4236, pp. 237–251. Springer, Heidelberg (2006)
17. Vetillard, E., Ferrari, A.: Combined Attacks and Countermeasures. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) CARDIS 2010. LNCS, vol. 6035, pp. 133–147. Springer, Heidelberg (2010)
18. Witteman, M.: Advances in Smartcard Security. Information Security Bulletin, 11–22 (July 2002)
19. Witteman, M.: Java Card Security. Information Security Bulletin, 291–298 (July 2003)

A Defensive Virtual Machine Layer to Counteract Fault Attacks on Java Cards

Michael Lackner, Reinhard Berlach, Wolfgang Raschke, Reinhold Weiss, and Christian Steger

Institute for Technical Informatics,
Graz University of Technology, Graz, Austria
{michael.lackner,reinhard.berlach,wolfgang.raschke,rweiss,steger}@tugraz.at

Abstract. The objective of Java Cards is to protect security-critical code and data against a hostile environment. Adversaries perform fault attacks on these cards to change the control and data flow of the Java Card Virtual Machine. These attacks confuse the Java type system, jump to forbidden code or remove run-time security checks. This work introduces a novel security layer for a defensive Java Card Virtual Machine to counteract fault attacks. The advantages of this layer from the security and design perspectives of the virtual machine are demonstrated. In a case study, we demonstrate three implementations of the abstraction layer running on a Java Card prototype. Two implementations use software checks that are optimized for either memory consumption or execution speed. The third implementation accelerates the run-time verification process by using the dedicated hardware protection units of the Java Card.

Keywords: Java Card, Defensive Virtual Machine, Countermeasure, Fault Attack.

1 Introduction

A Java Card enables Java applets to run on a smart card. The primary purpose of using a Java Card is the write-once, run-everywhere approach and the ability of post-issuance installation of applets [21]. These cards are used in a wide range of applications (e.g., digital wallets and transport tickets) to store security-critical code, data and cryptographic keys. Currently, these cards are still very resource-constrained devices that include an 8- or 16-bit processor, 4kB of volatile memory and 128kB of non-volatile memory. To make a Java Card Virtual Machine run on such a constrained device, a subset of Java is used [19]. Furthermore, special Java Card security concepts, such as the Java Card firewall [18] and a verification process for every applet [15], were added. The Java Card firewall is a run-time security feature that protects an applet against illegal access from other applets. For every access to a field or method of an object, this check is performed. Unfortunately, the firewall security mechanism can be circumvented by applets

L. Cavallaro and D. Gollmann (Eds.): WISTP 2013, LNCS 7886, pp. 82–97, 2013.
© IFIP International Federation for Information Processing 2013

© 2013 Springer-Verlag Berlin Heidelberg. Reprinted, with permission, from 7th Workshop in Information Security Theory and Practice (WISTP).

that do not comply with the Java Card specification. Such applets are called malicious applets.

To counteract malicious applets, a bytecode verification process is performed. This verification is performed either on-card or off-card for every applet [15]. Note that this bytecode verification is a static process and not performed during applet execution. The reasons for this static approach are the high resource needs of the verification process and the hardware constraints of the Java Card. This behavior is now abused by adversaries. They upload a valid applet onto the card and perform a fault attack (FA) during applet execution. Adversaries are now able to create a malicious applet out of a valid one [5].

A favorite time for performing a FA is during the fetching process. At this time, the virtual machine (VM) reads the next Java bytecode values from the memory. An adversary that performs an FA at this time can change the readout values. The VM then decodes the malicious bytecodes and executes them, which leads to a change in the control and data flow of the applet. A valid applet is mutated by such an FA to a malicious applet [5,17,11] and gains unauthorized access to secret code and data [16,2].

To counteract an FA, a VM must perform run-time security checks to determine if the bytecode behaves correctly. In the literature, different countermeasures, such as control-flow checks [23], double checks [4], integrity checks [8] and method encryption [20], have been proposed. Barbu [3] proposed a dynamic attack countermeasure in which the VM executes either standard bytecodes or bytecodes with additional security checks.

All these works do not concentrate on the question of how these security mechanisms can be smoothly integrated into a Java Card VM. For this integration, we propose adding an additional security layer into the VM. This layer abstracts the access to internal VM resources and performs run-time security checks to counteract FAs. The primary contributions of this paper are the following:

– Introduction of a novel defensive VM (D-VM) layer to counteract FAs during run-time. Access to security-critical resources of the VM, such as the operand stack (OS), local variables (LV) and bytecode area (BA), is handled using this layer.
– Usage of the D-VM layer as a dynamic countermeasure. Based on the actual security level of the card, different implementations of the D-VM layer are used. For a low-security level, the D-VM implementation uses fewer checks than for a high-security level. The security level depends on the credibility of the currently executed applet and run-time information received by hardware or software modules.
– A case study of a defensive VM using three different D-VM layer implementations. The API of the D-VM layer is used by the Java Card VM to perform run-time checks on the currently executing bytecode.
– The defensive VMs are executed on a smart card prototype with specific HW security features to speed up the run-time verification process. The resulting run-time and main memory consumption of all implemented D-VM layers are presented.

Section 2 provides an overview of attacks on Java Cards and the current countermeasures against them. Section 3 describes the novel D-VM layer presented in this work and its integration into the Java Card design. Furthermore, the method by which the D-VM layer enables the concept of dynamic countermeasures is presented. Section 4 presents implementation details regarding how the three D-VM implementations are inserted into the smart card prototype. Section 5 analyzes the additional costs for the D-VM implementations based on the execution and main memory overhead. Finally, the conclusions and future work are discussed in Section 6.

2 Related Work

In this section, the basics of the Java Card VM and work related to FA on Java Cards are presented. Then, an analysis of work regarding methods of counteracting FAs and securing the VM are presented. Finally, an FA example is presented to demonstrate the danger posed by such run-time attacks for the security of Java Cards.

2.1 Java Card Virtual Machine

A Java Card VM is software that is executed on a microprocessor. The VM itself can be considered a virtual computer that executes Java applets stored in the data area of the physical microprocessor. To be able to execute Java applets, the VM uses internal data structures, such as the OS or the LV, to store interim results of logical and combinatorial operations. All of these internal data structures are general objects for adversaries that attack the Java Card [4,20,24].

For every method invocation performed by the VM, a new Java frame [19] is created. This frame is pushed to the Java stack and removed from it when the method returns. In most VM implementations, this frame internally consists of three primary parts. These parts have static sizes during the execution of a method. The first frame part is the OS on which most Java operations are performed. The OS is the source and destination for most of the Java bytecodes. The second part is the LV memory region. The LV are used in the same manner as the registers on a standard CPU. The third part is the frame data, which holds all additional information needed by the VM and Java Card Runtime Environment (JCRE) [18]. This additional information includes, for example, return addresses and pointers to internal VM-related data structures.

2.2 Attacks on Java Cards

Loading an applet that does not conform to the specification defined in [19] onto a Java Card is a well-known problem called a logical attack (LA). After an LA, different applets on the card are no longer protected by the so-called Java

sandbox model. Through this sandbox, an applet is protected from illegal write and read operations of other applets. To perform an LA, an adversary must know the secret key to install applets. This key is known for development cards, but it is highly protected for industrial cards and only known by authorized companies and authorities. In conclusion, LAs are no longer security threats for current Java Cards.

Side-channel analyses are used to gather information about the currently executing method or instructions by measuring how the card changes environment parameters (e.g., power consumption and electromagnetic emission) during run-time. Integrated circuits influence the environment around them but can also be influenced by the environment. This influence is abused by an FA to change the normal control and data flow of the integrated circuit. Such FAs include glitch attacks on the power supply and laser attacks on the cards [2,24]. By performing side-channel analyses and FAs in combination, it is possible to break cryptographic algorithms to receive secret data or keys [16].

In 2010, a new group of attacks called combined attacks (CA) was introduced. These CAs combine LAs and FAs to enable the execution of ill-formed code during run-time [5]. An example of a CA is the removal of the *checkcast* bytecode to cause type confusion during run-time. Then, an adversary is able to break the Java sandbox model and obtain access to secret data and code stored on the card [5,17]. In this work work, we concentrate on countering FAs during the execution of an applet using our D-VM layer.

2.3 Countermeasures against Java Card Attacks

Since approximately 2010, an increasing number of researchers have started concentrating on the question of what tasks must be performed to make a VM more robust against FAs and CAs. Several authors [22,8] suggest adding an additional security component to the Java Card applet. In this component, they store checksums calculated over basic blocks of bytecodes. These checksums are calculated off-card in a static process and added to a new component of the applet. During run-time, the checksum of executed bytecodes is calculated using software and compared with the stored checksums. If these checksums are not the same, a security exception is thrown.

Another FA countermeasure is the use of control-flow graph information [23]. To enable this approach, a control-flow graph over basic blocks is calculated off-card and stored in an additional applet component. During run-time, the current control-flow graph is calculated and compared with the stored control graph.

In [20], the authors propose storing a countermeasure flag in a new applet component to indicate whether the method is encrypted. They perform this encryption using a secret key and the Java program counter for the bytecode of every method. Through this encryption, they are able to counteract attacks that change the control-flow of an applet to execute illegal code or data.

Another countermeasure against FAs that target the data stored on the OS is presented in [4]. In this work, integrity checks are performed when data are pushed or popped onto the OS. Through this approach, the OS is protected against FAs that corrupt the OS data.

Another run-time check against FAs is proposed in [10,14], in which they create separate OSes for each of the two data types, *integralValue* and *reference*. With this approach of splitting the OS, it is possible to counteract type-confusion attacks. A drawback is that in both works, the applet must be preprocessed.

In [3], the authors propose a dynamic countermeasure to counteract FAs. Bytecodes are implemented in different versions inside the VM, a standard version and an advanced version that performs additional security checks. The VM is now able to switch during run-time from the standard to the advanced version. By using unused Java bytecodes, an applet programmer can explicitly call the advanced bytecode versions.

The drawbacks of current FA countermeasures are that most of them add an additional security component to the applet or rely on preprocessing of the applet. This has different drawbacks, such as increased applet size or compatibility problems for VMs that do not support these new applet components. In this work, we propose a D-VM layer that performs checks on the currently executing bytecode. These checks are performed based on a run-time policy and do not require an off-card preprocessing step or an additional applet component.

2.4 EMAN4 Attack: Jump Outside the Bytecode Area

In 2011, the run-time attack EMAN4 was found [6]. In this work a laser was used to manipulate the read out values from the EEPROM to 0x00. By this laser attack an adversary is able to change the Java bytecode of post-issuance installed applets during their execution.

The target time of the attack is when the VM fetches the operands of the *goto_w* bytecode from the EEPROM. Generally the *goto_w* bytecode is used to perform a jump operation inside a method. The *goto_w* bytecode consists of the operand byte 0xa8 and two offset bytes for the branch destination [19]. This branch offset is added to the actual Java program counter to determine the next executing bytecode. An adversary which changes this offset is able to manipulate the control flow of the applet.

With the help of the EMAN4 attack it is possible to jump with the Java program counter outside the applet bytecode area (BA), as illustrated in Figure 1. This is done by changing the offset parameters of the *goto_w* bytecode from 0xFF20 to 0x0020 during the fetch process of the VM. The jump destination address of the EMAN4 attack is a data array outside the bytecode area. This data array was previously filled with adversary defined data. After the laser attack the VM executes the values of the data array. This execution of adversary definable data leads to considerably more critical security problems, such as memory dumps [7]. In this work we counteract the EMAN4 attack by our control flow policy. This policy only allows to fetch bytecodes which are inside the bytecode area.

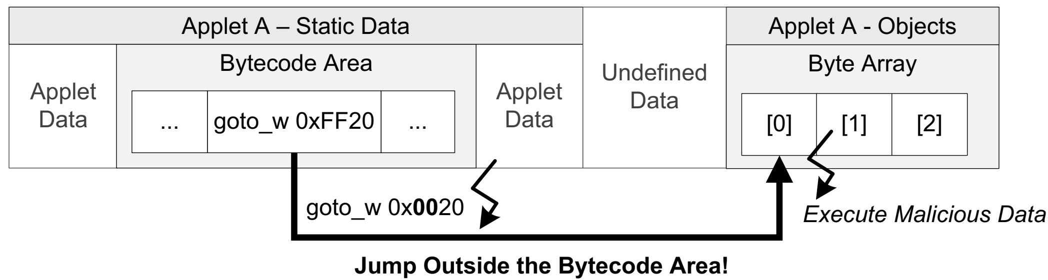


Fig. 1. The EMAN4 run-time attack changes the jump address 0xFF20 to 0x0020, which leads to the security threat of executing bytecode outside the defined BA of the current applet [6]

3 Defensive VM Layer

In this work, we propose adding a novel security layer to the Java Card. Through this layer, access to internal structures (e.g., OS, LV and BA) of the VM is handled. In reference to its defensive nature and its primary use for enabling a defensive VM, we name this layer the defensive VM (D-VM) layer. An overview of the D-VM layer and the D-VM API, which is used by the VM, is depicted in Figure 2 and is explained in detail below.

Functionalities offered by the D-VM API include, for example, pushing and popping data onto the OS, writing and reading from the LV and fetching Java bytecodes. It is possible for the VM to implement all Java bytecodes by using these API functions. The pseudo-code example in Listing 1.1 shows the process of fetching a bytecode and the implementation of the *sadd* bytecode using our D-VM API approach. The *sadd* bytecode pops two values of *integral data* type from the OS and pops the sum as an *integral data* type back onto the OS.

Listing 1.1. Pseudo-code of the VM using the API functions of the newly introduced D-VM layer.

```
//use the D-VM API to fetch the next bytecode from the BA
switch(dvm_fetch_bytecode())
{
  ...
  case sadd: //implementation of the sadd bytecode.
  {
    //use the D-VM API to obtain the two values from the OS
    result = dvm_pop_integralData() + dvm_pop_integralData();
    //use the D-VM API to write the sum back onto the OS
    dvm_push_integralData(result);
  }
  ...
}
```

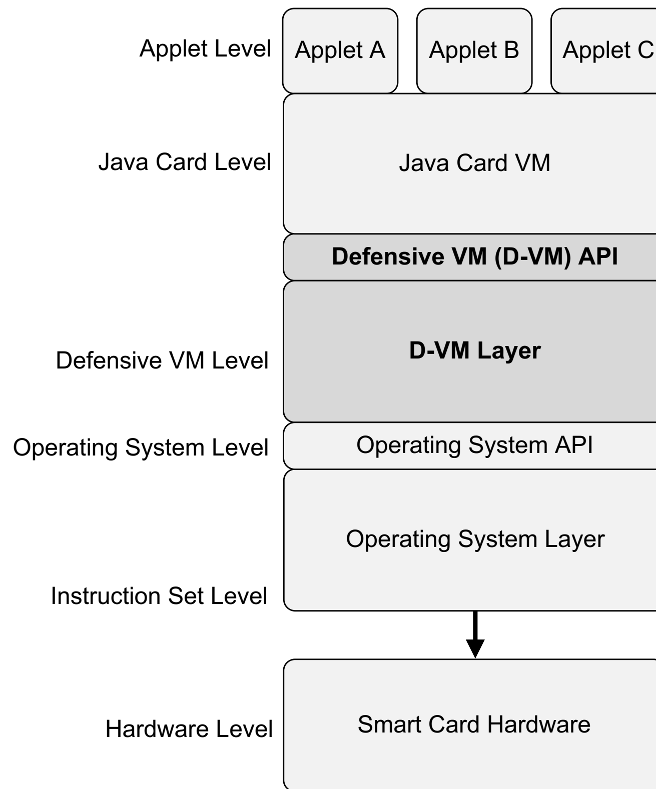


Fig. 2. The VM executes Java Card applets and uses the newly introduced D-VM layer to secure the Java Card against FAs

The security mechanisms within the security layer intended to protect the VM from FAs are hidden from the VM programmer. A security architect, specialized for VM security, is able to implement and choose the appropriate countermeasures within the D-VM layer. These countermeasures are based on state-of-the-art knowledge and the hardware constraints of the smart card architecture. Programmers implementing the VM do not need to know these security techniques in detail but rather just use the D-VM API functions.

If HW features are used, the D-VM layer communicates with these units and configures them through specific instructions. Through this approach, it is also very easy to alter the SW implementations by changing the D-VM layer implementation without changing specific Java bytecode implementations. It is possible to fulfill the same security policy on different smart card platforms where specific HW features are available.

On a code size-constrained smart card platform, an implementation that has a small code size but requires more main memory or execution time is used. The appropriate implementations of security features within the D-VM API are used without the need to change the entire VM.

Dynamic Countermeasures: The D-VM layer is also a further step to enable dynamic fault attack countermeasures such as that proposed by Barbu in [3]. In this work, he proposes a VM that uses different bytecode implementations depending on the actual security level of the smart card. If an attack or malicious behavior is detected, the security level is decreased. This decreased security leads to an exchange of the implemented bytecodes with more secure versions. In these more secure bytecodes, different additional checks, such as double reads, are implemented, which leads to decreased run-time performance.

Our D-VM layer further advances this dynamic countermeasure concept. Depending on the actual security level, an appropriate D-VM layer implementation is used. Therefore, the entire bytecode implementation remains unchanged, but it is possible to dynamically add and change security checks during run-time. An overview of this dynamic approach is outlined in Figure 3.

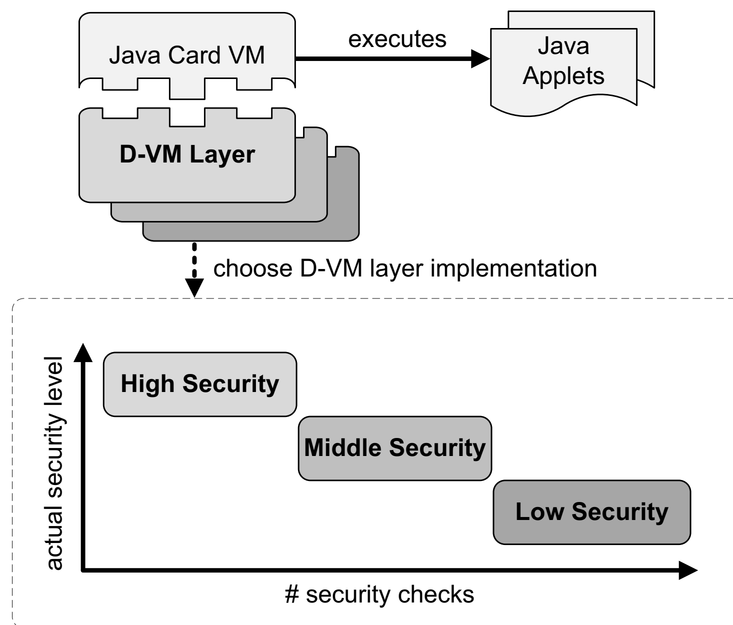


Fig. 3. Based on the current security level of the VM, an appropriate D-VM layer implementation is chosen

The actual security level of the card is determined by HW sensors (e.g., brightness and supply voltage) and the behavior of the executing applet. For example, at a high security level, the D-VM layer can perform a read operation after pushing a value into the OS memory to detect an FA. At a lower security level, the D-VM layer performs additional bound, type and control-flow checks.

Security Context of an Applet: Another use case for the D-VM layer is the post-issuance installation of applets on the card. We focus on the user-centric ownership model (UCOM) [1] in which Java Card users are able to load their own

applets onto the card. For the UCOM approach, each newly installed applet is assigned a defined security level at installation time. The security level depends on how trustworthy the applet is. For example, the security level for an applet signed with a valid key from the service provider is quite high, which results in a high execution speed. Such an applet should be contrasted with an applet that has no valid signature and is loaded onto the card by the Java Card owner. This applet will run at a low security level with many run-time checks but a slower execution speed. Furthermore, access to internal resources and applets installed on the card could be restricted by the low security level.

3.1 Security Policy

This chapter introduces the three security policies used in this work. With the help of these policies, it is possible to counteract the most dangerous threats that jeopardize security-critical data on the card. The type and bound policies are taken from [14] and are augmented with a control-flow policy. The fulfillment of the three policies on every bytecode is checked by three different D-VM layer implementations using our D-VM API.

Control-Flow Policy: The VM is only allowed to fetch bytecodes that are within the borders of the currently active method's BA. Fetching of bytecodes that are outside of this area is not allowed. The actual valid method BA changes when a new method is invoked or a return statement is executed. Because of this policy, it is no longer possible for control-flow changing bytecodes (e.g., *goto_w* and *if_scmp_w*) to jump outside of the reserved bytecode memory area. This policy counters the EMAN4 attack [6] on the Java Card and all other attacks that rely on the execution of a data array or code of an-other applet that is not inside the current BA.

Type Policy: Java bytecodes are strongly typed in the VM specification [19]. This typing means that for every Java bytecode, the type of operand that the bytecode expects and the type of the result stored in the OS or LV are clearly defined. An example is the *sastore* bytecode, which stores a *short* value in an element of a *short* array object. The *sastore* bytecode uses the top three elements from the OS as operands. The first element is the address of the array object, which is of type *reference*. The second element is the index operand of the array, which must be of type *short*. The third element is the value, which is stored within the array element and is of type *short*.

Type confusion between values of integral data (*boolean*, *byte* or *short*) and object references (*byte[]*, *short[]* or *class A*, for example) is a serious problem for Java Cards [24,17,13,25,6,11]. To counter these attacks, we divide all data types into the two main types, *integralData* and *reference*. Note that this policy does not prevent type confusion inside the main type *reference* between array and class types.

Bound Policy: Most Java Card bytecodes push and pop data onto the OS or read and write data into the LV, which can be considered similar to registers. The OS is the main component for most Java bytecode operations. Similar to buffer overflow attacks in C programs [9], it is possible to overflow the reserved memory space for the OS and LV. An adversary is then able to set the return address of a method to any value. Such an attack was first found in 2011 by Bouffard [6,7]. An overflow of the OS happens by pushing or popping too many values onto the OS. An LV overflow happens when an incorrect LV index is accessed. This index parameter is decoded as an operand for several LV-related bytecodes (e.g., *sstore*, *sload* and *sinc*). This operand is therefore stored permanently in the non-volatile memory. Thus, changing this operand through an FA gives an attacker access to memory regions outside the reserved LV memory region. These memory regions are created for every method invoked and are not changed during the method execution. Therefore in this work, we permit Java bytecodes to operate only within the reserved OS and LV memory regions.

4 Java Card Prototype Implementation

In this work three implementations of the D-VM layer are proposed to perform run-time security checks on the currently executing bytecode. Two implementations perform all checks in SW to ensure our security policies. One implementation uses dedicated HW protection units to accelerate the run-time verification process. The implementations of the D-VM layer were added into a Java Card VM and executed on a smart card prototype. This prototype is a cycle-accurate SystemC [12] model of an 8051 instruction set-compatible processor. All software components, such as the D-VM layer and the VM, are written in C and 8051 assembly language.

4.1 D-VM Layer Implementations

This section presents the implementation details for the three implemented D-VM layers used to create a defensive VM. All three implemented D-VM layers fulfill our security policy presented in Chapter 3 but differ from each other in the detailed manner in which the policies are satisfied. The key characteristic of the two SW D-VM implementations is that they use a different implementation of the type-storing approach to counteract type confusion. The run-time type information (*integralData* or *reference*) used to perform run-time checks can be stored either in a type bit-map (memory optimization) or in the actual word size of the microprocessor (speed optimization). The HW Accelerated D-VM uses a third approach and stores the type information in an additional bit of the main memory. Through this approach, the HW can easily store and check the type information for every OS and LV entry. An overview of how the type-storing policy is ensured by our D-VM implementations and a memory layout overview are shown in Figure 4 and explained in detail in the next sections.

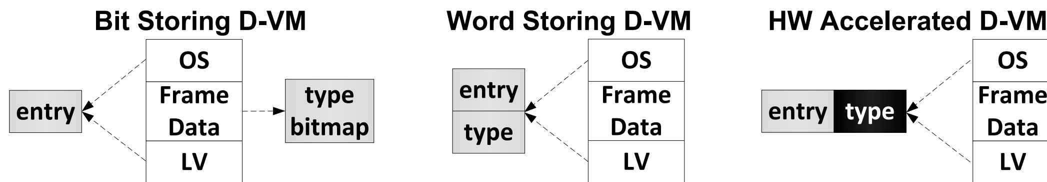


Fig. 4. The Bit Storing D-VM stores the type information for every OS and LV entry in a type bitmap. The Word Storing D-VM stores the type information below the value in the reserved OS and LV spaces. The HW Accelerated D-VM holds the type information as an additional type bit, which increases the memory size of a word from 8 bits to 9 bits.

Bit Storing D-VM: This D-VM layer implementation stores the type information for every element on the OS and LV in a type bitmap. The type information for every entry of the OS and LV is now represented by a one-bit entry. A problem with this approach is that the run-time overhead is quite high because different shift and modulo operations must be performed to store and read the type information from the type bitmap. These operations (shift and modulo) are, for the 8051 architecture, computationally expensive operations and thus lead to longer execution times. An advantage of the bit-storing approach is the low memory overhead required to hold the type information in the type bitmap.

Word Storing D-VM: The run-time performance of the type storing and reading process is increased by storing the type information using the natural word size of the processor and data bus on which the memory for the OS and LV is located. Every element in the OS and LV is extended with a type element of a word size such that it can be processed very quickly by the architecture. By choosing this implementation, the memory consumption of the type-storing process increases compared with the previously introduced SW Bit Storing D-VM. Pseudo-codes for writing to the top of the stack of the OS for the bit- and word-storing approach are shown in Listings 1.2 and 1.3.

Listing 1.2. Operations needed to push an element on the OS by the Bit Storing D-VM

```
dvm_push_integralData(value)
{
  //push value onto OS and
  //increase OS size
 OS[size++] = value;
  //store type information
  //into type bitmap,
  //INT->integralData type
 bitmap[size/8] = INT<<(size%8);
}
```

Listing 1.3. Operations needed to push an element on the OS by the Word Storing D-VM

```
dvm_push_integralData(value)
{
  //push value onto OS
  //increase OS size
 OS[size++] = value;
  //store type information
  //into next memory word,
  //INT->integralData type
 OS[size++] = INT;
}
```

HW Accelerated D-VM: Performing type and bound checks in SW to fulfill our security policy consumes a lot of computational power. Types must be loaded, checked and stored for almost every bytecode. The bounds of the OS and LV must be checked such that no bytecode performs an overflow. The HW Accelerated D-VM layer uses specific HW protection units of the smart card to accelerate these security checks. New protection units (bound protection and type protection) are able to check if the current memory move (MOV) operation is operating in the correct memory bounds. The type information for the OS and LV entries is stored as an additional type bit for every main memory word. The information is decoded into new assembly instructions to specify which memory region (OS, LV or BA) and with which data type (*integralData* or *reference*) the MOV operation should write or read data. An overview of the HW Accelerated D-VM is shown in Figure 5. Depending on the assembly instruction, the HW protection units perform four security operations:

- Check if the Java opcode is fetched from the current active BA.
- Check if the destination address of the operation is within the memory area of the OS or LV. If the operation is not within these two bounded areas, a HW security exception is thrown.
- For every write operation write the type decoded in the CPU instruction into the accessed memory word.
- For every read operation, check if the stored type is equal to the type decoded in the CPU instruction. If they are not equal, throw a hardware security exception.

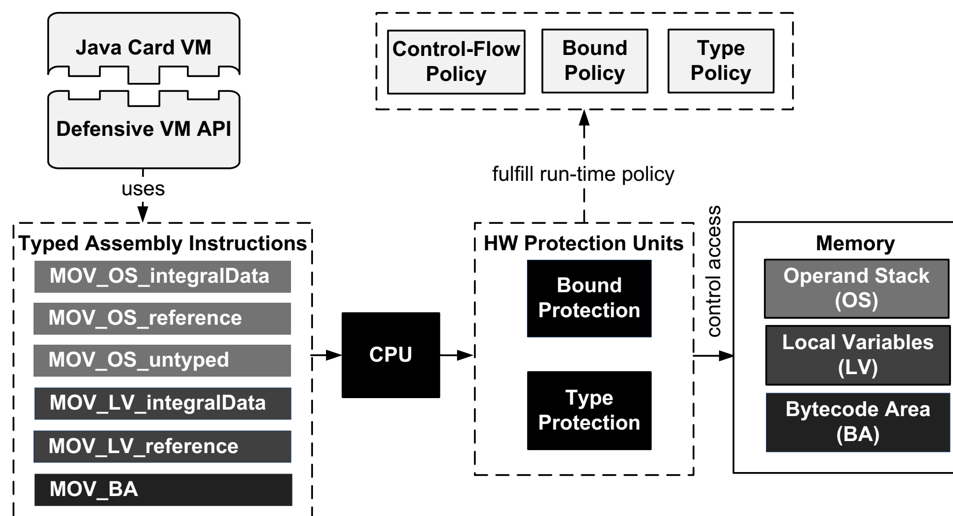


Fig. 5. Overview of the HW Accelerated D-VM implementation using new typed assembly instructions to access VM resources (OS, LV and BA). Malicious Java bytecodes violating our run-time policy will be detected by new introduced HW protection units.

5 Prototype Results

In this section, we present the overall computational overhead of the three implemented D-VM layers and their main memory consumption. All of them are compared with a VM implementation without the D-VM layer. The speed comparison is performed for different groups of bytecodes by self written micro-benchmarks where all bytecodes under test are measured. These test programs first perform an initialization phase where the needed operands for the bytecode under test are written into the OS or LV. After the execution of the bytecode under test the effects on the OS or LV are removed. Note that our smart card platform has no data or instruction cache. Therefore, no caching effects must be taken into account for all test programs.

5.1 Computational Overhead

Speed comparisons for specific bytecodes are shown in Figure 6. For example, the Java bytecode *sload* requires 148% more execution time for the Word Storing D-VM. For the Bit Storing D-VM, the execution overhead is 212%. The increased overhead is because of the expensive calculations used to store the type information in a bitmap. For the HW Accelerated D-VM, the execution speed decreases by only 4% because all type and bound checks are performed using HW. Additional run-time statistics for groups of bytecodes are listed in Table 1. As expected, the Bit Storing D-VM consumes the most overall run-time, with an increase of 208%. The Word Storing D-VM needs 142% more run-time. The HW Accelerated D-VM has only 6% more overhead.

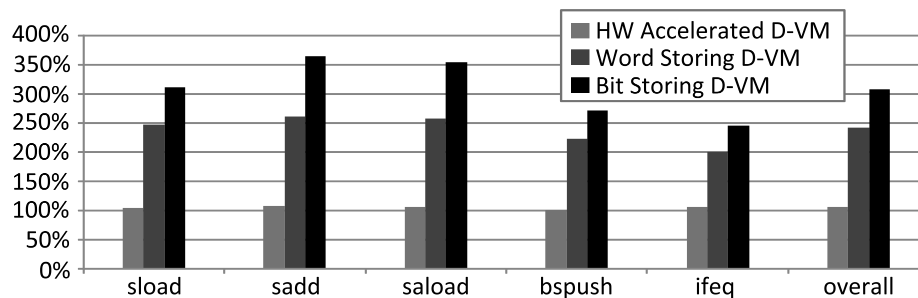


Fig. 6. Speed comparison of individual bytecodes for the different D-VM layer implementations proposed in this work. The results are compared with a VM without the D-VM layer.

5.2 Main Memory Consumption

The HW Accelerated D-VM requires one type bit per 8 bits of data to store the type information during run-time. This results in an overall main memory increase of 12.5%. The Word Storing D-VM requires in the worst case 33% more memoy because one type byte holds the type information for two data bytes.

Table 1. Speed comparison for different groups of bytecodes compared with a VM without the D-VM layer

Bytecode Groups	HW Accelerated D-VM	Word Storing D-VM	Bit Storing D-VM
Arithmetic/Logic	+7%	+146%	+240%
LV Access	+5%	+185%	+243%
OS Manipulation	+5%	+151%	+231%
Control Transfer	+7%	+113%	+173%
Array Access	+5%	+130%	+166%
Overall	+6%	+142%	+208%

The Bit Storing D-VM requires approximately 6.25% more memory in the case in which the entire memory is filled with OS and LV data. This is because the Bit Storing D-VM requires one type bit per 16 bits of data.

6 Conclusions and Future Work

This work presents a novel security layer for the virtual machine (VM) on Java Cards. Because it is intended to defend against fault attacks (FAs), it is called the defensive VM (D-VM) layer. This layer provides access to security-critical resources of the VM, such as the operand stack, local variables and the bytecode area. Inside this layer, security checks, such as type checking, bound checking and control-flow checks, are performed to protect the card against FAs. These FAs are executed during run-time to change the control and data flow of the currently executing bytecode.

By storing different implementations of the D-VM layer on the card, it is possible to choose the appropriate security implementation based on the actual security level of the card. Through this approach, the number of security checks can be increased during run-time by switching among different D-VM implementations. Furthermore, it is possible to assign a trustworthy applet a low security level, which results in high execution performance, and vice versa. One D-VM layer implementation can be, for example, low security with high execution speed or high security with low execution speed. Another advantage is the concentration of the security checks inside the layer.

To demonstrate this novel security concept, we implemented three D-VM layers on a smart card prototype. All three layers fulfill the same security policy (control-flow, type and bound) for bytecodes but differ in their implementation details. Two D-VM layer implementations are fully implemented in software but differ in the manner in which the type information is stored. The Bit Storing D-VM has the highest run-time overhead, 208%, but the lowest memory increase, 6.25%. The Word Storing D-VM decreases the run-time overhead to 142% but consumes approximately 33% more memory. The HW Accelerated D-VM uses dedicated Java Card HW to accelerate the run-time verification process and has an execution overhead of only 6% and a memory increase of 12.5%.

In future work, we will focus on the question of which sensor data should be used to increase the internal security of the Java Card. Another question is how many security states are required and how much they differ in their security needs.

Acknowledgments. The authors would like to thank the Austrian Federal Ministry for Transport, Innovation, and Technology, which funded the CoCoon project under the FIT-IT contract FFG 830601. We would also like to thank our project partner NXP Semiconductors Austria GmbH.

References

1. Akram, R., Markantonakis, K., Mayes, K.: A Paradigm Shift in Smart Card Ownership Model. In: 2010 International Conference on Computational Science and Its Applications (ICCSA), pp. 191–200 (March 2010)
2. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The Sorcerer's Apprentice Guide to Fault Attacks. Proceedings of the IEEE 94(2), 370–382 (2006)
3. Barbu, G., Andouard, P., Giraud, C.: Dynamic Fault Injection Countermeasure. In: Mangard, S. (ed.) CARDIS 2012. LNCS, vol. 7771, pp. 16–30. Springer, Heidelberg (2013)
4. Barbu, G., Duc, G., Hoogvorst, P.: Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 297–313. Springer, Heidelberg (2011)
5. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) CARDIS 2010. LNCS, vol. 6035, pp. 148–163. Springer, Heidelberg (2010)
6. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.-L.: Combined Software and Hardware Attacks on the Java Card Control Flow. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 283–296. Springer, Heidelberg (2011)
7. Bouffard, G., Lanet, J.-L.: The Next Smart Card Nightmare. In: Naccache, D. (ed.) Cryphtography and Security: From Theory to Applications. LNCS, vol. 6805, pp. 405–424. Springer, Heidelberg (2012)
8. Bouffard, G., Lanet, J.-L., Machemie, J.-B., Poichotte, J.-Y., Wary, J.-P.: Evaluation of the Ability to Transform SIM Applications into Hostile Applications. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 1–17. Springer, Heidelberg (2011)
9. Cowan, C., Wagle, P., Pu, C., Beattie, S., Walpole, J.: Buffer overflows: attacks and defenses for the vulnerability of the decade. In: Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems], pp. 227–237 (2003)
10. Dubreuil, J., Bouffard, G., Lanet, J.-L., Cartigny, J.: Type Classification against Fault Enabled Mutant in Java Based Smart Card. In: 2012 Seventh International Conference on Availability, Reliability and Security (ARES), pp. 551–556 (August 2012)
11. Hamadouche, S., Bouffard, G., Lanet, J.-L., Dorsemaine, B., Nouhant, B., Magloire, A., Reygnaud, A.: Subverting Byte Code Linker service to characterize Java Card API. In: Proceedings of the 7th Conference on Network and Information Systems Security (SAR-SSI), pp. 122–128 (2012)

12. IEEE: Open SystemC Language Reference Manual IEEE Std 1666-2005, IEEE
13. Iguchi-Cartigny, J., Lanet, J.-L.: Developing a Trojan applets in a smart card. Journal in Computer Virology 6, 343–351 (2010)
14. Lackner, M., Berlach, R., Loinig, J., Weiss, R., Steger, C.: Towards the Hardware Accelerated Defensive Virtual Machine – Type and Bound Protection. In: Mangard, S. (ed.) CARDIS 2012. LNCS, vol. 7771, pp. 1–15. Springer, Heidelberg (2013)
15. Leroy, X.: Bytecode verification on Java smart cards. Software: Practice and Experience 32(4), 319–340 (2002)
16. Markantonakis, K., Mayes, K., Tunstall, M., Sauveron, D., Piper, F.: Smart card security. In: Nedjah, N., Abraham, A., de Macedo Mourelle, L. (eds.) Computational Intelligence in Information Assurance and Security. SCI, vol. 57, pp. 201–233. Springer, Heidelberg (2007), http://dx.doi.org/10.1007/978-3-540-71078-3_8
17. Mostowski, W., Poll, E.: Malicious Code on Java Card Smartcards: Attacks and Countermeasures. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 1–16. Springer, Heidelberg (2008)
18. Oracle: Runtime Environment Specification. Java Card Platform, Version 3.0.4, Classic Edition (2011)
19. Oracle: Virtual Machine Specification. Java Card Platform, Version 3.0.4, Classic Edition (2011)
20. Razafindralambo, T., Bouffard, G., Thampi, B.N., Lanet, J.-L.: A Dynamic Syntax Interpretation for Java Based Smart Card to Mitigate Logical Attacks. In: Thampi, S.M., Zomaya, A.Y., Strufe, T., Alcaraz Calero, J.M., Thomas, T. (eds.) SNDS 2012. CCIS, vol. 335, pp. 185–194. Springer, Heidelberg (2012)
21. Sauveron, D.: Multiapplication smart card: Towards an open smart card? Information Security Technical Report 14(2), 70–78 (2009); Smart Card Applications and Security
22. Séré, A.A.K., Iguchi-Cartigny, J., Lanet, J.-L.: Checking the Paths to Identify Mutant Application on Embedded Systems. In: Kim, T.-H., Lee, Y.-H., Kang, B.-H., Ślęzak, D. (eds.) FGIT 2010. LNCS, vol. 6485, pp. 459–468. Springer, Heidelberg (2010)
23. Séré, A.A.K., Iguchi-Cartigny, J., Lanet, J.-L.: Evaluation of Countermeasures Against Fault Attacks on Smart Cards. International Journal of Security and Its Applications 5(2), 49–61 (2011)
24. Vertanen, O.: Java Type Confusion and Fault Attacks. In: Breveglieri, L., Koren, I., Naccache, D., Seifert, J.-P. (eds.) FDTC 2006. LNCS, vol. 4236, pp. 237–251. Springer, Heidelberg (2006)
25. Vetillard, E., Ferrari, A.: Combined Attacks and Countermeasures. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) CARDIS 2010. LNCS, vol. 6035, pp. 133–147. Springer, Heidelberg (2010)

2013 International Conference on Risks and Security of Internet and Systems (CRiSIS)

A Defensive Java Card Virtual Machine to Thwart Fault Attacks by Microarchitectural Support

Michael Lackner, Reinhard Berlach, Michael Hraschan, Reinhold Weiss and Christian Steger
Institute for Technical Informatics
Graz University of Technology
Inffeldgasse 16/1, A-8010 Graz, Austria
{michael.lackner, reinhard.berlach, rweiss, steger}@tugraz.at
michael.hraschan@student.tugraz.at

***Abstract*—Java Cards, which are primarily used to store security-sensitive data, are employed in a wide range of applications, such as authentication and banking. Because these data must be protected against logical and fault attacks, static and runtime verification must be performed to assure the security of Java applets. Currently, this verification is performed in the software. Runtime verification for counteracting fault attacks is costly due to additional execution time and memory consumption. To circumvent the drawbacks of software verification, we propose incorporating a microarchitectural support of runtime verification directly into smart card hardware. These new hardware features enable a defensive virtual machine to counteract buffer overflow attacks, type confusion attacks, control flow attacks, and data integrity attacks. To measure the additional overhead of hardware and performance, the new microarchitectural security features are integrated into a smart card prototype on a field programmable gate array board.**

I. Introduction

When engineers initially considered installing a Java virtual machine onto a smart card in 1996, they did not predict the success of this concept. Today, more than 1 billion cards are produced each year and used in daily life. Transport systems, passport verification systems and banking systems rely on the security provided by applets, which are executed on these cards. The success story of Java Card can continue in the area of near field communication (NFC). In this NFC area, the vision of a multi-application Java Card that is controlled by the card user may constitute the next stage for these cards [1]. In this multi-application context, applications from different sources are installed and executed on the card. This multi-application function increases the security requirements of Java Cards, which protect against logical and fault attacks.

Today, the security of Java Cards is reliant on a static verification process that is performed either on-card or off-card [2]. This verification, which requires a significant amount of memory and time, examines the Java applet for malicious behavior, such as overflow, type confusion or control flow attacks. During runtime, one of the last lines of defense is the Java Card firewall, which protects access permissions for methods and fields of different applets [3], [4].

Unfortunately, the security concepts of static verification and a runtime applet firewall can be circumvented by fault attacks during runtime (e.g., laser attack). These attacks enable an adversary to create a malicious applet and gain unauthorized access to code and data from other applets or the virtual machine (VM) [5], [6], [7], [8], [9], [10].

In this study, we propose a defensive VM for Java Cards, which is accelerated by microarchitectural support of the platform. A bound protection unit, type protection unit, control flow protection unit, and data integrity protection unit are inserted into the smart card hardware (HW) to ensure the security of the VM. These new units protect the VM against fault attacks, as shown in Figure 1. These protection units are integrated into a smart card prototype on a field programmable gate array (FPGA) board and protect the security-critical memory regions of the VM. These regions are the operand stack (OS), local variables (LV) and the bytecode area (BA) which have increasingly become the targets of fault attacks [11], [12], [13].

In this work a Java Card VM was modified to facilitate the incorporation of these new protection units. To obtain precise results of the HW and runtime overheads of the additional HW security features, a smart card prototype is functionally emulated on a FPGA platform. Based on this implementation, the runtime and HW overhead are measured. Our approach, which involves a HW acceleration defensive VM, enables runtime security checks with low HW and execution overheads. The combination of a high verification speed and low HW overhead is prevalent in the Java Card market, which incorporates the advantage of a large unit volume and high security requirements.

Section II provides an overview of the Java Card VM and current attacks and countermeasures. Section III provides insight into general design concepts of a defensive VM. Section IV describes the five security policies that are satisfied by our defensive VM design. Section V presents insight into the integration of HW protection units within a smart card prototype to fulfill our security policies. Section VI analyzes the total cost of implementation of a smart card prototype, including runtime costs of execution speed, memory consumption and HW overhead. Last, conclusions and future studies are presented in Section VII.

978-1-4799-3488-1/13/$31.00 ©2013 IEEE

© 2013 IEEE. Reprinted, with permission, from Proceedings of 8th International Conference on Risks and Security of Internet and Systems (CRiSIS).

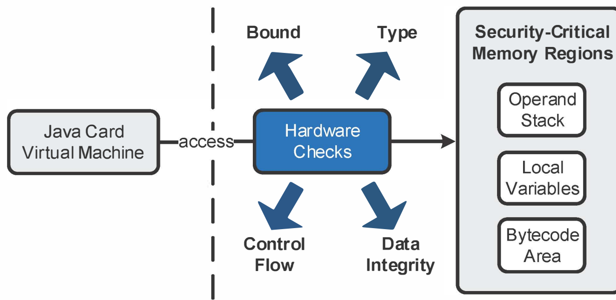


Fig. 1. The Java Card VM uses HW security features to perform runtime security checks on Java bytecodes based on bound checks, type checks, control flow checks, and data integrity checks.

II. Related Work

This section provides an overview of the Java Card VM and the manner in which Java bytecodes are executed in the VM. Previous studies of attacks on Java Cards are also presented. The contribution of our research to existing state-of-the-art countermeasures is discussed at the end of this section.

A. Java Card Virtual Machine

Running a Java VM on resource-constrained devices, such as a smart card, is a challenging task. Modern desktop computers contain multicore processors with approximately 4 processing units, 8 GB of volatile memory and 2 TB of nonvolatile memory. Smart card HW contains a single 8/16 bit processing unit, approximately 10 kB of volatile memory and 100 kb of non-volatile memory (e.g., EEPROM, flash memory). These HW constraints reduce the functionality of the Java Card VM. For example, memory and computational intensive data types, such as float, double or long, are not supported. Multidimensional arrays and the execution of concurrent threads are also not supported.

Conversely, new security features, such as special application programming interface (API) functions to access cryptographic operations for authentication and signature, have also been added to the Java Card. New runtime security mechanisms were also incorporated, such as the Java Card software firewall. This firewall determines whether objects are allowed to access the methods and fields of other objects.

The behaviors of the Java Card VM for all bytecodes and the format of the applets that are loaded on the card are described in [3], [4]. The VM is a stack machine that retrieves the Java bytecode from the bytecode area (BA). For computational operations, two main memory regions are employed: the operand stack (OS) and the local variables (LV). Each method invocation creates a new fixed-sized OS and LV in the main memory.

The OS is employed for all arithmetic and logical operations. For example, to perform the calculation 3+3, the value 3 is initially pushed onto the OS two times. The arithmetic bytecode *sadd*, which reads and removes both values from the OS and pushes the result 6, is performed.

The LV, which is used to store the intermediate results of a variable, is the source of the method parameters when a new method is invoked. The LV is logically related to the registers of a physical processor.

The OS and LV memory regions are used substantially during VM execution. As a result, they are a rewarding destination for attacks, such as type confusion attacks [13], [7], [8], memory overflow attacks [14] or data integrity attacks [6]. Therefore, the runtime checks in this study focus on restricting the memory accesses of the VM to these memory regions.

B. Attacks on Java Cards

Attacks on Java Cards are classified into three large groups: physical attacks, observation attacks and fault attacks. Our research focuses on countering fault attacks. Nevertheless, all three attack groups must be considered during the design and implementation phase of our new HW security features.

Physical Attacks: An adversary removes different physical layers of a Java Card chip by etching and accessing different digital or analog HW blocks. By placing probes on buses or input/output units, it is possible to detect or recover internal functions of the chip or recover data messages, such as keys. By using a focused ion beam, an adversary is able to break or connect lines on the chip to inject a fault [9].

Observation Attacks: These non-invasive attacks exploit the notion that the computation of software (SW) is executed on a physical smart card. This smart card interacts with its surrounding environment and requires electrical energy for computation. This power consumption and electromagnetic radiation of the digital circuits vary based on the actual executed instructions. This means that an adversary can develop conclusions about the internal Java Card SW by observing these physical processes. Using advanced techniques, such as simple or differential power analyses, an adversary can break cryptographic algorithms and receive a secret key [15], [16].

Fault Attacks: Fault attacks are provoked by running a digital chip beyond its technical specifications, such as an excessive high/low supply voltage amplitude or high/low clock signal frequency. An attack on the supply voltage or clock signal during a short period of time is named a glitch attack [17]. The bombardment of the chip with high-energy optical pulses (laser attack) produces undefined chip behavior. An application of this type of fault attack is the disturbance of cryptographic operations. In this type of attack, an adversary can obtain the secret key of a cryptographic operation [18].

Currently, fault attacks are utilized to directly attack the execution and behavior of the Java Card VM. For example, by changing the supply voltage during a read access to the non-volatile memory, it is possible to modify the reference voltage to determine if the bits stored in memory are interpreted as zeros or ones. This modification causes the conversion of a valid applet to a malicious applet [13], [12].

Other VM attacks change the control and data flow of a Java applet to obtain illegal access to secret codes or data stored in the memory of the card [7], [8]. Adversaries manipulate Java branch instructions by faults, which causes the Java VM to jump outside the current BA. The jump target can be an array filled with data by an adversary, which becomes capable of executing any code on the Java Card and performing, for example, a memory dump of the card [14].

Integrity attacks on OS memory are also performed. The values on the OS are used to perform comparison operations to determine which code path Java branch bytecodes prefer to execute. For example, the branch bytecode *if_scmpeq* compares the top two operands to determine whether they are equivalent; the branch *ifeq* compares the top OS element with zero. An attacker that manipulates OS data during runtime can manipulate the outcome of these conditional branches [6]. This action leads to the threat of entering branches in the applet, which are typically inaccessible without the knowledge of a secret key or data.

C. Actual Java Card Security Concept

The current security concept for Java Cards is dependent on a static verification of the uploaded application. Due to the high memory requirements of the type verification, this static verification is performed off-card [2]. For an on-card verification, more powerful and expensive cards are required. This static off-card verification verifies whether the application contains any security violations, such as type confusion, memory overflows, and violations to the Java Card specification or Java application format [3], [4]. After the verification, the application is signed with a secret key and uploaded onto the card using a secure channel. The signature on the card is verified and the installation process commences.

D. Fault Attack Countermeasures

In 2009, researcher began to combine logical attacks with fault attacks to create combined attacks, which are capable of breaking the sandbox security model of the VM [8], [5]. Different new attacks that are based on OS/LV overflow [14], Java type confusion [13], [8], control flow changes [19], or data integrity attacks [6] against the VM were discovered

Simultaneously, research on ways to prevent a VM against this new class of combined attacks was initiated. Because combined attacks explicitly attack the computation of the VM, new additional SW security checks are directly added to the VM. To counteract attacks that change the values of elements on the OS, data integrity checks are performed by [6]. In [12] and [14], the authors propose a calculation of off-card checksums over the bytecode of a Java applet. The granularity of these checksums is derived from basic blocks (BB). These BB are used in [19] to create a control flow graph. This pre-computed control flow graph is subsequently verified against the current program flow to detect illegal jumps or branches of the VM during runtime. An additional countermeasure, which is reliant on an off-card step, is the encryption of the Java bytecode with the Java program counter and a secret key [20]. Using this encryption, the threat of executing data instead of Java bytecode was countered.

To counteract type confusion attacks between *integralData* (e.g., *boolean*, *byte*, and *short*) and *references* (e.g., *byte[]*, *short[]*, and *class A*) a separation of the single OS into two OSs is proposed [10], [21]. This change enables all values of a specific type to be read and written on the OS for *integralData* or on the OS for *references.* With this separation approach, type confusion between the two main types is prevented by an architectural change of the VM and detection during runtime by an overflow or underflow of the two OSs. An off-card pre-computation must be performed to exchange bytecodes, in which the VM does not know during runtime on which of the two OSs the operation must be performed.

In a previous work [21] we introduced the concept of a hardware accelerated defensive VM to fulfill a type policy (prevent type confusion) and bound policy (prevent OS and LV overflow). To fulfill the type policy the concept of a type storing and type separating defensive VM are presented and accelerated by new hardware protection units. First run-time overhead measurements are shown based on a smart card simulation. Detailed hardware overhead measurements of the new protection units were outstanding.

E. Drawbacks of Current Countermeasures

SW checks to enable a defensive VM have considerable disadvantages, such as high computational overhead and additional memory requirements. Checksums and pre-computed control flow graphs are permanently stored in the memory. The verification of these data against the current state of the Java Card creates additional performance overhead. Another significant disadvantage is that SW checks, which should prevent a single fault attack, can be removed by a second fault attack. For example, an adversary injects one fault attack to create an overflow attack on the VM and injects a second fault when the overflow check happens.

In this study HW-assisted runtime checks are used to circumvent the issues of SW checks. Our HW checks require no pre-computed data and are performed parallel to standard processor operations. Therefore, the performance and memory overhead remains low. Furthermore, our security approach is fully compatible with all standard Java Card bytecodes and is not dependent on off-card computed data.

III. Defensive Virtual Machine Design

A defensive Java Card VM performs specific security checks on the bytecode during runtime. The following critical points must be considered during the selection and design of these runtime checks:

Security Increase: The selection of correct runtime countermeasures is an important prerequisite of our defensive approach. By selecting the correct runtime checks, our defensive VM is able to counteract the most dangerous attacks on Java Cards. By countering an attack, such as type confusion, an adversary is prevented from obtaining the

address of a data array, which is required for more advanced attacks [7], [13], [23], [8].

Smoothly Integration: The countermeasure should not rely on additional data that is calculated by off-card tools because industrial applet providers do not want their applets modified in any way. Another reason is the decreased interoperability that results when security related data are integrated into certain applets and is only supported by certain Java Cards.

Low Overhead: The HW countermeasures should consume as little power, memory and performance overhead as possible. To prevent side channel attacks, the countermeasures must be evaluated to ensure that no information about the internal state of the chip is disclosed to the outside world.

IV. Defensive Policies of Our Work

This section provides an overview of the security policies of our HW protection units during runtime. Furthermore, an overview and analysis of countered attacks are presented. The following bound policy, top of stack policy, type policy, and control flow policy are extracted from the Java Card specifications [3]. The data integrity policy is based on the security aspect of protecting Java Card VM related data against illegal data changes, as described in the Java Card protection profile [24].

Bound Policy: During the execution of an applet, the VM writes and reads data onto the OS and LV. The sizes of the OS and LV are known during the installation time of the applet. Therefore, the VM reserves the required memory size during runtime for the OS and LV in the main memory. An attacker who performs a fault attack can overflow the OS and/or LV by creating malicious bytecodes. Using an overflow, an adversary accesses undefined data regions. These data regions can contain security-critical data, such as the return address of the actual method. With an overflow attack, it is possible to return to any address mentioned by an adversary, as illustrated in the EMAN2 attack [14]. To counteract an overflow attack, our bound policy verifies all bytecodes during runtime, if the operation overflows the OS or LV.

Top of Stack Policy: The OS is a fundamental memory region for the VM, which is accessed during the majority of Java bytecode executions. The OS is logically accessed similar to a stack. With the top of stack policy, all bytecodes are only allowed to pop their elements from the actual top element of the OS and push them on the top element of the OS. This policy encounters attacks on the VM in which the counter of the actual number of OS elements is not updated or a wrong counter value is read from memory. Using such an attack, it is possible to read values that are logically removed from the OS; however, the values are retained in memory. By the top of stack policy it is furthermore not possible to write to elements below the top element of the OS.

Type Policy: Java data types are distinguished between the two main types *integralData* and *reference*. The *integralData* type contains *boolean*, *byte*, *short*, and *integer*, which are used to perform arithmetic/logic operations.

The second main type represents all references to objects, such as arrays (e.g., *boolean[]*, *byte[]*, *short[]*). This *reference* type includes also general class objects that are for example defined by the applet programmer. All references are created by the VM when the bytecode *newarray* or the *new* bytecode is summoned to create objects from classes. A trustworthy applet is unable to create a *reference* by a cast out of *integralData* as in C/C++.

Type confusion attacks between the two main types *integralData* and *reference* are a critical problem in the Java Card world [13], [7], [8]. Type confusion attacks are used to initiate other attacks, including the EMAN2 attack [14] in which the address of a data array is extracted from the Java Card by a type confusion attack. This array address is subsequently used as the destination of a second control flow attack. After the control flow attack, the VM begins executing the data of the array. By filling the data array with values similar to bytecodes, an adversary is able to execute security-critical code. Further attacks, which are dependent on the type confusion of *integralData* and *reference* involve the execution of a data array [25], [14], the characterization of the Java Card API [7] or access to forbidden object methods [5].

Our type policy prohibits type confusion between *integralData* and *reference* on the OS and LV. Note that this policy does not counteract type confusion inside our two main data types, such as between a *byte[]* and *short[] reference*. Type confusion inside the *reference* main type must be detected by additional SW checks when an object is accessed.

Control Flow Policy: Fault attacks are used to change the valid control flow of an applet. An example of a successful attack is the EMAN4 attack [26]. In this attack, an adversary is able to execute any bytecode on a Java Card. This action generates critical security threats, such as memory dumps of the entire memory. The EMAN4 attack is based on the notion that control-flow-changing bytecodes, such as *goto*, contain operands that indicate the relative jump destination address. By manipulating this address by a fault attack, the *goto* bytecode jumps outside the current BA and executes defined adversary code.

Under the control flow policy, the VM is only allowed to retrieve bytecodes that are inside the BA. All bytecodes required for method execution are located in the BA. Dynamic data structures that are created or changed during execution are not located within this area. This security policy circumvents all attacks that aim to execute data instead of bytecodes.

Data Integrity Policy: The OS and LV are elementary

memory regions of the VM. Attackers perform fault attacks into these memory regions by corrupting the data during a read/write access [17]. With this attack, adversaries are able to manipulate the outcome of conditional branch instructions because the branch destination is dependent on data of the OS [6]. For example, the outcome of the branch bytecode *ifeq* is dependent on the comparison of the top OS element with 0x00. By manipulating the read-out OS element to 0x00 or 0xFF, an adversary can manipulate the outcome of the branch. Under our integrity policy, we protect the values of the OS memory and LV memory against a fault injection during a read/write operation.

V. Java Card Prototype Implementation

The five runtime security policies in this study are satisfied by new HW protection units that are integrated into a prototype smart card. This prototype consists of an 8051 instruction set-compatible 8 bit processor. To evaluate the effects of the changes on the VM and the effects of the smart card HW on the execution time and HW overhead, the smart card is functionally emulated on a Virtex-6 FPGA evaluation kit. The smart card HW is implemented in the HW description language VHDL. The VM is written in standard C and 8051 assembly language using our new defensive assembly instructions.

A. Modifications to the Smart Card Hardware

An overview of the implementation of the Java Card in this study is shown in Figure 2. Java applets are executed on a Java Card VM using functionality provided by the operating system of the card. To accelerate communication with the protection units, security-related information is directly decoded into new defensive assembly instructions of the 8051 processor. The VM uses these new defensive instructions to communicate with the new bound, type, control flow, and data integrity protection units. These units perform the HW acceleration checks on the security-critical memory regions of the VM, which comprise the OS, LV, and BA.

Into the defensive instructions are decoded the access direction, the accessible memory regions, and the Java data type information which are presented in Table I. This approach of decoding information into new instructions has the advantage of significantly lower communication and configuration overhead of the protection units. For example, to read an *integralData* value from the OS, the VM uses the new defensive assembly instruction *MOV_READ_OPERAND_STACK_INTEGRAL_DATA*. With the information decoded into this instruction, the bound protection unit is able to verify that the accessed memory address is inside the OS. Furthermore, the type protection unit verifies that the accessed OS value is of the type *integralData*. Parallel to this step, the data integrity protection unit performs an integrity check of the read OS value. Any security policy violations detected by the protection units results in a HW exception. This exception brings the card to a secure state and increases its attack counter. The card is deactivated when the attack counter reaches the maximum number of allowed attacks.

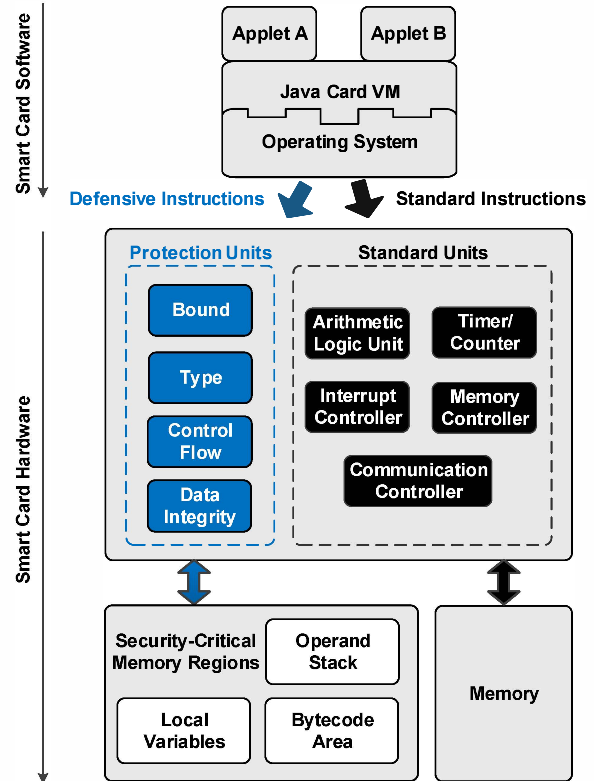


Fig. 2. Implementation overview of the Java Card prototype. A Java Card VM uses new defensive assembly instructions to access data inside security-critical memory regions. Using the information decoded into these instructions, HW checks are performed to counter fault attacks. These checks include bound, type, control flow, and integrity checks on the current executing Java bytecode.

TABLE I

Decoded information (access direction, memory area and Java data type) into the new added defensive assembly instructions. This information is used by the HW protection units to perform security checks during runtime.

Defensive Assembly Instructions		
Access Direction	Memory Area	Java Data Type
read	operand stack	*integralData*
		reference
		untyped
	local variables	*integralData*
		reference
	bytecode area	*untyped*
write	operand stack	*integralData*
		reference
	local variables	*integralData*
		reference

B. Bound Protection Unit (BPU)

The BPU verifies whether the current bytecode is malicious by overflowing the actual OS or LV memory region. To set the memory bounds of the OS and LV, new HW registers are

inserted into the BPU. These registers are updated for every Java method invocation to the actual bounds of the OS and LV.

The BPU also incorporates the top of stack policy in which data are read and written from the OS memory, such as from a HW FIFO (first in, first out) buffer. Therefore, the BPU internally stores the actual address of the top element of the OS (OS_{top}). The BPU only allows write operations to the ($OS_{top}+1$) address and read operations to the OS_{top} address. If a valid read operation is performed on the OS, then the actual top element of the OS is updated to OS_{top} = ($OS_{top}-1$).

C. Type Protection Unit (TPU)

The TPU counteracts type confusion attacks between *integralData* and *references* on the OS and LV memory. To store the Java type information, the main memory word size is increased by one additional type bit. By combining this type bit with the expected type information decoded into the new defensive instructions, it is possible for the TPU to verify whether the type is valid.

D. Control Flow Protection Unit (CFPU)

The CFPU unit contains the upper and lower bound addresses of the BA memory region. The BA contains all bytecodes of the actual executing Java method. During runtime, the CFPU confirms whether the current fetched bytecode is inside the BA. The only assembly instruction that is allowed to read from the BA memory region is the new defensive assembly instruction *MOV_READ_BYTECODE_AREA*.

E. Data Integrity Protection Unit (IPU)

The IPU is implemented with its own data and control signals to the main memory. It is independent to the state machine of the processor and performs read operations to the main memory parallel to the execution of the processor instructions. The IPU has its own state machine and begins its integrity checks when data are read or written into the security-critical OS and LV memory regions.

- **Read Operation:** When data are read from the OS or LV, the IPU initially performs a standard read operation to obtain the memory value. This standard $read_value$ is stored into an internal IPU register and transmitted to the logic of the processor for additional computations. The IPU begins in parallel to the state machine of the processor to read the inverted value, which is the $not(read_value)$ from the same memory address. The two values, which consist of the $read_value$ and the $not(read_value)$, are employed by the IPU to create σ by the following operation:

  $$\sigma = not(read_value) \oplus read_value$$

  The IPU verifies if σ = 0xff. When $\sigma \neq$ 0xff, an attack is detected and a HW security interrupt is emitted.

- **Write Operation:** When a write operation is performed to the OS or LV memory region, the IPU initially receives the $write_value$, which will be written into the memory. The $write_value$ is stored in an internal IPU register and written into the memory. After the write operation is completed, the IPU reads the inverted value, which is the $not(write_value)$, from the same memory address in which the previous write operation was performed. Subsequently, the following computational operation is performed to receive the value ς:

  $$\varsigma = not(write_value) \oplus write_value$$

  After this operation, the IPU verifies whether $\varsigma \neq$ 0xff and raises a security exception if an integrity violation is detected.

VI. Prototype Results and Discussion

This section presents a comparison of our HW acceleration checks with the SW checks. The SW checks perform the same security checks and policies as the HW implementation with the exception of the top of stack policy. The additional overhead to enable the SW and HW acceleration checks are compared with an unmodified smart card prototype without any runtime checks.

The computational overhead of the bytecodes is measured in two steps. First, the pre-conditions for the bytecodes under investigation are established inside the VM. The required data for the bytecode under investigation are written into the OS or LV or specific objects are created. Second, the bytecodes are executed in the bench phase and the results of the operation are removed. Note that the benchmarks are not dependent on previously executed instructions and their data because the processor contains no instructions or data cache.

A. Computational Overhead

The execution overhead of the additional security checks for specific Java bytecodes are shown in Figure 3. The performance of runtime security checks in SW significantly increases the execution time overhead. For example, the overhead for the *sload* bytecode is 172% compared with a VM without these checks. The *sload* bytecode reads an LV element of type *short* and pushes this value onto the OS. The SW overhead for the *sload* bytecode is derived from the following SW checks:

1) Verify that the retrieved bytecode is inside the BA.
2) Verify that the read LV element is inside the LV memory area.
3) Verify the integrity of the read LV element (double read).
4) Verify that the LV element is of the type *integralData.*
5) Verify that a new element will not provoke an overflow of the OS.
6) Verify the integrity of the written OS element (double read).
7) Store that the new OS element is of the type *integralData.*

The relocation of these time-consuming computations to the HW decreases the overhead for the *sload* bytecode to 3%.

These execution speed measurements are performed for different groups of bytecodes, as shown in Table II. The HW acceleration checks generate only 4% additional overhead for

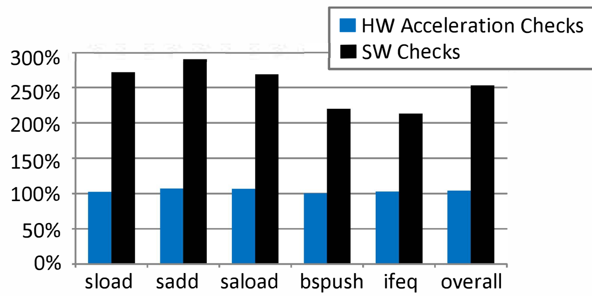


Fig. 3. Computational overhead for specific bytecodes and the total time of all implemented bytecodes for the SW implementation and HW acceleration checks. The measurements are normalized to a Java Card VM, which performs no additional runtime security checks.

all bytecodes. When the checks are performed in the SW, they generate an overall overhead of 159%.

TABLE II
MEASUREMENTS OF THE ADDITIONAL OVERHEAD FOR GROUPS OF BYTECODES BY PERFORMING THE RUNTIME CHECKS IN SW OR WITH HW SUPPORT. THE MEASUREMENTS ARE NORMALIZED TO A JAVA CARD VM WHICH PERFORMS NO ADDITIONAL RUNTIME SECURITY CHECKS.

Bytecode Groups	HW Acceleration Checks	SW Checks
Arithmetic/Logic	+4%	+147%
LV Access	+3%	+196%
OS Manipulation	+6%	+176%
Control Transfer	+4%	+119%
Array Access	+4%	+161%
Overall	+4%	+159%

B. Hardware Overhead

This section demonstrates the HW overhead that is required to integrate the new defensive instructions and the protection units within a smart card prototype. This smart card consists of the basic HW components like arithmetic logic unit, timer/counter, interrupt controller, memory controller and communication controller. This prototype is emulated on a Virtex-6 development board. The HW overhead for our new components and total overhead are listed in Table III. The total area overhead on the FPGA board is approximately 11%. Note that these size measurements are calculated without a crypto coprocessor and the main memory.

The word size of the main memory was increased by one bit to store the type information for every element on the OS and LV during runtime. With this additional type bit per 8-bit memory word, the total main memory increases to 12.5%.

C. Countered Attacks and Limitations

The HW security mechanism of this study performs runtime checks on the current executing bytecodes and the security-critical memory regions in which these operations are performed.

- The BPU counteracts all types of malicious bytecodes, which overflow the OS and LV area during runtime. Furthermore, this unit only allows the pop and push of data onto the actual top element of the OS. Note that overflow attacks on other memory regions like on the static field image [25] are still possible and must be counteracted by additional checks.
- The TPU thwarts type violations between *integralData* and *reference*. Note that type violations inside the class of *integralData* or *reference* are not detected and must be thwarted in the SW.
- The CFPU verifies if the current method execution occurs within the bounds of the current BA. Therefore, no execution of data, with the exception of code, is possible. Note that control flow attacks, which occur within the current BA, are not prevented.
- The IPU checks each read and write access to the security-critical OS and LV areas. This is accomplished by an additional read operation of the inverted value from the same address. The inversion is performed internally by HW logic inside the main memory. By comparing the standard value with the inverted value, single fault injections are detected. These single faults corrupt the value, which is read or written into the OS memory and LV memory. Using this approach of reading the inverted value, 2nd order attacks with the same value are also thwarted. To circumvent this countermeasure, an attacker would have to perform an initial 2nd order attack with the value he wants to inject and a subsequent 2nd order attack with the inverted value.

TABLE III
HW OVERHEAD TO ENABLE THE RUNTIME PROTECTION UNITS IN THE SMART CARD PROTOTYPE ON A FPGA BOARD (VIRTEX-6). THE HW OVERHEAD IS COMPARED WITH AN IMPLEMENTATION WITHOUT ANY SECURITY CHECKS.

Hardware Components	Area Overhead
Bound Protection Unit	+5%
Type Protection Unit	+1%
Control Flow Protection Unit	+1%
Data Integrity Protection Unit	+1%
Defensive Instructions	+3%
Total	+11%

VII. CONCLUSIONS AND FUTURE WORK

This study presents an approach to perform hardware (HW) acceleration runtime checks on Java bytecodes executed by a Virtual Machine (VM) on a smart card. Using this approach, it is possible to detect malicious behavior of bytecodes caused by a fault attack during runtime. These checks enable proper functioning of our bound policy, top of stack policy, control flow policy and data integrity policy, which are fulfilled during runtime. By implementing these checks in the HW instead of the software (SW), it is more challenging for an adversary to counter these checks by a second fault attack. Furthermore, acceleration of the verification speed is achieved by moving the expensive computational check operations into HW.

To enable these HW checks, new HW protection units were inserted into a Java Card prototype. Based on this prototype, the additional resource requirements and modifications are

evaluated. This prototype consists of an 8051 compatible processor that is functionally emulated on a field programmable gate array (FPGA) board. New defensive assembly instructions are employed by the VM to communicate with the HW protection units. The protection units use information that is directly decoded into the defensive assembly instructions to perform runtime checks. The information comprise the access direction, accessed memory area and the Java data type. The protection units verify all accesses to security-critical memory regions of the VM, which comprise the bytecode area, operand stack or local variables.

A complete SW implementation of all checks has an overhead of 159%. By performing the checks in HW the overall execution time overhead is only 4%. The integration of the protection units into the smart card prototype increases the digital area by approximately 11%. To store the Java type information during runtime the main memory increases by 12.5%.

The relocation of the runtime security checks from the SW to the HW is a promising approach, especially for the high-security requirements of the Java Card market. This increased runtime security is required by new Java Card applications, such as near field communication (NFC) or multiple applications on one card. It is possible to counteract the steps of a first attack that is executed by adversaries to perform more advanced attacks, such as memory dumps. This study demonstrates that minimal HW resources and computational overhead are required for our runtime checks.

In future work, we will focus on the practical evaluation of our security mechanisms. Furthermore, the modified smart card must be evaluated to verify its capability of withstanding side channel attacks. These evaluations are required for the implementation of our HW features and their industrial usage.

VIII. Acknowledgments

The authors would like to thank the Austrian Federal Ministry for Transport, Innovation, and Technology, which funded the CoCoon project under the FIT-IT contract FFG 830601. We would also like to thank our project partner NXP Semiconductors Austria GmbH.

References

[1] R. Akram, K. Markantonakis, and K. Mayes, "Building the Bridges – A Proposal for Merging Different Paradigms in Mobile NFC Ecosystem," in *Computational Intelligence and Security (CIS), 2012 Eighth International Conference on*, 2012, pp. 646–652.

[2] X. Leroy, "Bytecode verification on Java smart cards," *Software: Practice and Experience*, vol. 32, no. 4, pp. 319–340, 2002.

[3] Oracle, *Virtual Machine Specification*. Java Card Platform, Version 3.0.4, Classic Edition, 2011.

[4] Oracle, *Runtime Environment Specification*. Java Card Platform, Version 3.0.4, Classic Edition, 2011.

[5] E. Vetillard and A. Ferrari, "Combined Attacks and Countermeasures," in *Smart Card Research and Advanced Application*, ser. Lecture Notes in Computer Science, D. Gollmann, J.-L. Lanet, and J. Iguchi-Cartigny, Eds. Springer Berlin Heidelberg, 2010, vol. 6035, pp. 133–147.

[6] G. Barbu, G. Duc, and P. Hoogvorst, "Java Card Operand Stack:Fault Attacks, Combined Attacks and Countermeasures," in *Smart Card Research and Advanced Applications*, ser. Lecture Notes in Computer Science, E. Prouff, Ed. Springer Berlin Heidelberg, 2011, vol. 7079, pp. 297–313.

[7] S. Hamadouche, G. Bouffard, J.-L. Lanet, B. Dorsemaine, B. Nouhant, A. Magloire, and A. Reygnaud, "Subverting Byte Code Linker service to characterize Java Card API," ser. Proceedings of the 7th Conference on Network and Information Systems Security (SAR-SSI), 2012, pp. 122–128.

[8] W. Mostowski and E. Poll, "Malicious Code on Java Card Smartcards: Attacks and Countermeasures," in *Smart Card Research and Advanced Applications*, ser. Lecture Notes in Computer Science, G. Grimaud and F.-X. Standaert, Eds. Springer Berlin / Heidelberg, 2008, vol. 5189, pp. 1–16.

[9] M. Witteman, "Advances in Smartcard Security," *Information Security Bulletin*, July 2002.

[10] J. Dubreuil, G. Bouffard, J.-L. Lanet, and J. Cartigny, "Type Classification against Fault Enabled Mutant in Java Based Smart Card," in *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*, aug. 2012, pp. 551 –556.

[11] O. Vertanen, "Java Type Confusion and Fault Attacks," in *Fault Diagnosis and Tolerance in Cryptography*, ser. Lecture Notes in Computer Science, L. Breveglieri, I. Koren, D. Naccache, and J.-P. Seifert, Eds. Springer Berlin / Heidelberg, 2006, vol. 4236, pp. 237–251.

[12] A. Sere, J. Iguchi-Cartigny, and J.-L. Lanet, "Checking the Paths to Identify Mutant Application on Embedded Systems," in *Future Generation Information Technology*, ser. Lecture Notes in Computer Science, T.-h. Kim, Y.-h. Lee, B.-H. Kang, and D. Slezak, Eds. Springer Berlin / Heidelberg, 2010, vol. 6485, pp. 459–468.

[13] G. Barbu, H. Thiebeauld, and V. Guerin, "Attacks on Java Card 3.0 Combining Fault and Logical Attacks," in *Smart Card Research and Advanced Application*, ser. Lecture Notes in Computer Science, D. Gollmann, J.-L. Lanet, and J. Iguchi-Cartigny, Eds. Springer Berlin Heidelberg, 2010, vol. 6035, pp. 148–163.

[14] G. Bouffard, J. Iguchi-Cartigny, and J.-L. Lanet, "Combined Software and Hardware Attacks on the Java Card Control Flow," in *Smart Card Research and Advanced Applications*, ser. Lecture Notes in Computer Science, E. Prouff, Ed. Springer Berlin Heidelberg, 2011, vol. 7079, pp. 283–296.

[15] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in CryptologyCRYPTO99*. Springer, 1999, pp. 388–397.

[16] J.-J. Quisquater and D. Samyde, "Electromagnetic analysis (ema): Measures and counter-measures for smart cards," in *Smart Card Programming and Security*. Springer, 2001, pp. 200–210.

[17] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The Sorcerer's Apprentice Guide to Fault Attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370 –382, 2006.

[18] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Advances in CryptologyCRYPTO'97*. Springer, 1997, pp. 513–525.

[19] A. Sere, J. Iguchi-Cartigny, and J.-L. Lanet, "Evaluation of Countermeasures Against Fault Attacks on Smart Cards," *International Journal of Security and Its Applications, Vol.5 No.2*, pp. 49–61, 2011.

[20] T. Razafindralambo, G. Bouffard, and J.-L. Lanet, "A Friendly Framework for Hidding fault enabled virus for Java Based Smartcard," in *Data and Applications Security and Privacy XXVI*, ser. Lecture Notes in Computer Science, N. Cuppens-Boulahia, F. Cuppens, and J. Garcia-Alfaro, Eds. Springer Berlin Heidelberg, 2012, vol. 7371, pp. 122–128.

[21] M. Lackner, R. Berlach, J. Loinig, R. Weiss, and C. Steger, "Towards the Hardware Accelerated Defensive Virtual Machine - Type and Bound Protection," in *Smart Card Research and Advanced Applications*, ser. Lecture Notes in Computer Science, S. Mangard, Ed. Springer Berlin Heidelberg, 2013, vol. 7771, pp. 1–15.

[22] G. Barbu, "De la sécurité des plateformes java card face aux attaques matérielles," Ph.D. dissertation, Telecom ParisTech, 2012.

[23] G. Barbu and H. Thiebeauld, "Synchronized Attacks on Multithreaded Systems - Application to Java Card 3.0 -," in *Smart Card Research and Advanced Applications*, ser. Lecture Notes in Computer Science, E. Prouff, Ed. Springer Berlin / Heidelberg, 2011, vol. 7079, pp. 18–33.

[24] Sun Microsystems, "Java Card System Protection Profile," Tech. Rep., April 2010, version 2.6.

[25] J. Iguchi-Cartigny and J.-L. Lanet, "Developing a Trojan applets in a smart card," *Journal in Computer Virology*, vol. 6, pp. 343–351, 2010.

[26] G. Bouffard and J.-L. Lanet, "The Next Smart Card Nightmare," in *Cryptography and Security: From Theory to Applications*, ser. Lecture Notes in Computer Science, D. Naccache, Ed. Springer Berlin / Heidelberg, 2012, vol. 6805, pp. 405–424.

Countering Type Confusion and Buffer Overflow Attacks on Java Smart Cards by Data Type Sensitive Obfuscation

Michael Lackner
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
michael.lackner@tugraz.at

Reinhard Berlach
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
reinhard.berlach@tugraz.at

Reinhold Weiss
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
rweiss@tugraz.at

Christian Steger
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
steger@tugraz.at

ABSTRACT

Java enabled smart cards protect security-related code and data by a sandbox concept. Unfortunately, this sandbox can be bypassed by fault attacks. Therefore, there is a substantial need for transparent, effective, and low-overhead countermeasures. This work demonstrates a new countermeasure against type confusion and buffer overflow attacks. This new countermeasure is based on obfuscating the security critical calculation parts of a virtual machine by secret keys. This countermeasure was integrated into a Java Card virtual machine running on a smart card prototype. New hardware features were added to this prototype to accelerate the obfuscating operation. The execution time overhead of the new countermeasure is demonstrated by performing run-time measurements on the prototype.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Smartcards; D.4.6 [**Operating Systems**]: Security and Protection—*e.g., viruses, worms, Trojan horses*; C.0 [**General**]: Instruction set design (e.g., RISC, CISC, VLIW)

General Terms

Security, Design

Keywords

Java Card, Smart Card, Defensive Virtual Machine, Embedded Security, Hardware Countermeasure, Fault Attack

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CS2 January 20 2014, Vienna, Austria
Copyright 2014 ACM 978-1-4503-2484-7/14/01 ...$15.00.

1. INTRODUCTION

By approximately the end of 2014, the number of people on the planet, currently 7.1 billion, will be smaller than the number of mobile phones. In most of these mobile phones, a secure subscriber identification module (SIM) card is used to authenticate the phone to the mobile phone operator and its network services. Another security component, especially used in smart phones, is the so-called secure element. The secure element enables technologies such as near field communication (NFC) and applications such as digital wallets, electronic passports, transport control, authentication, and data encryption and decryption.

Most SIM cards and secure elements rely on the security features provided by a Java Card. A Java Card consists of a highly secure smart card that can run Java applets executed by a software (SW) Java virtual machine (JVM) [16, 15]. The JVM can execute Java applets and protects the code and data of these applets against illegal operations. These illegal operations are thwarted by a SW firewall and the security concept of the Java sandbox model. This sandbox separates applets from each other so that it is not possible for an applet to access the methods and fields of other applets without explicit permission [15].

Currently, the sandbox concept relies on a static off-card verification process [20]. The verification is performed off-card due to high memory and performance needs. An applet that passes the verification process cannot break the sandbox model of the JVM or obtain illegal access to code or data on the card. Unfortunately, after the verification process, an adversary could change the control and data flow of an applet with a run-time fault attack (FA). Such a FA enables advanced attacks against the card, such as the execution of self-defined code [6], memory dumps [9], manipulation of authentication checks [1], type confusion [2], and buffer overflow attacks [2].

A static applet verification does not counteract FAs. Therefore, additional run-time checks must be integrated into the JVM to obtain a so-called defensive JVM. The selection of appropriate defensive mechanisms is a difficult step during the design phase of a Java Card. Countermeasures should prevent of as many FAs as possible but should not signif-

© 2014 ACM. Reprinted, with permission, from Proceedings of 1st Workshop on Cryptography and Security in Computing Systems (CS2).

icantly increase the execution overhead or chip costs. The chip costs, as well as the security strength and time to market, are very important success factors in the Java Card market.

The contribution of this work is the design and implementation of a new run-time countermeasure to create a defensive JVM. This new countermeasure is based on obfuscating the two most important calculation parts of a JVM: the memory region operand stack (OS) and the local variables (LVs), which are accessed during the execution of almost every bytecode. Based on the Java data type, we use different keys to perform data obfuscation. Due to this obfuscation, an adversary can no longer receive or create useful information or data through a type confusion attack. Furthermore, the obfuscating counteracts buffer overflow attacks on the OS and LVs. With such overflow attacks, the return address of the current Java method could be overwritten with an adversary-defined address. To significantly increase the difficulty encountered by an adversary in determining the obfuscating keys, we propose newly created keys for every invoked Java method.

Section 2 provides an overview of the JVM and state of the art attacks and countermeasures against Java Cards. Section 3 gives insight into the general design concept of the new countermeasure and explains how attacks are thwarted. Section 4 presents the implementation of the countermeasure and its integration into the Java Card prototype. Section 5 analyzes the run-time overhead for our prototype. Finally, conclusions and future studies are presented in Section 6.

2. RELATED WORK

A Java Card is a very resource-constrained embedded system with a single 8/16 bit processor, 10 kB of volatile memory, and 100 kB of non-volatile memory. Due to these very strict hardware (HW) constraints, different functional features are not included such as large-value data types, multi threading or multidimensional arrays. With the help of a specific co-processor, complex cryptographic operations are processed at a satisfactory rate.

During the execution of an applet, the JVM fetches bytecodes from the bytecode area (BA). For every Java method, a new so-called Java frame is created, which primarily consists of the two logical memory parts: the OS and the LVs. Most Java bytecodes push or pop elements from the OS. The LVs are accessible by an index similar to processor registers. The LVs hold, for example, intermediate results of variables and method parameters. Due to heavy usage of the OS, LVs, and BA during applet execution, different attacks on these areas are known, such as type confusion [2, 8, 14], buffer overflow [2], data integrity violations [1], and control flow attacks [5].

The bytecode of a Java Card applet is strongly typed. Based on the Java Card virtual machine specification [16], the data types of operands and results are specified for every bytecode. The first character of a bytecode helps to identify which basic data type is operated by the bytecode. For example, there exist the *sstore* and *astore* bytecodes. Both *store* bytecodes move the top OS element into the indexed LV. *Sstore* is used for all *integralData* type values (e.g., Boolean, byte, short). *Astore* is used for all *reference* type values (object references to arrays or classes).

In fact, FAs against Java Cards are performed by using a laser light at a specific point in time against a specific region on the card. A worthwhile destination for such a FA is the memory. Memory attacks change the read out or written values, for example, to 0x00 or 0xff [1]. By injecting a FA during the JVM fetch phase of the bytecode, the behavior of the bytecode of the executing applet is changed [21]. If an adversary injects 0x00 during a fetch operation, the JVM interprets this as the NOP bytecode. The NOP bytecode performs no operation and skips the normal bytecode. By using a laser beam against digital logic such as the CPU, the control flow can be changed by bypassing instructions [19]. Additionally, the manipulation of computational calculations is a known issue, especially against cryptographic algorithms [4]. By performing a FA during a cryptographic computation, an adversary can receive secret keys [3].

The combination of a FA and a logical attack is called a combined attack (CA). The first hypothetical scenario of such a CA was presented in [21, 14]. CAs can manipulate a previously verified applet in such a way that the now malicious applet can break the sandbox model of the card. In 2010, Barbu [2] presented the first real-world CA with help of a laser beam. In the next years, researchers concentrated on finding different types of new CAs. These CAs can manipulate the run-time behavior of an applet to execute illegal operations or to gain access to secret code or data regions [6, 9, 1].

After concentrating on new FAs against the Java Card platform, current research concentrates on the question of how to secure the Java Card against CAs and FAs. These countermeasures can be classified into two main groups. The first group requires an off-card step to modify or include additional security information into an applet. The additional security information enables run-time verification of the applet. The second group includes countermeasures that do not rely on an off-card step. Our new countermeasure belongs to the second category.

Applet Preprocessing Required: In [18, 5], basic block (BB) checksums are calculated off-card over the bytecode. The checksums are added into an additional applet component. During run-time, the JVM calculates run-time checksums over the current fetched bytecode. A FA is detected when the off-card checksums do not match the run-time checksums. Using off-card calculations of the valid control flow graph for the BBs, invalid jumps between BBs are counteracted [19]. In [18], a more memory-friendly countermeasure is presented, called *field of bit*; this countermeasure verifies whether a FA leads the JVM to illegally interpret bytecode operands as opcodes. An off-card encryption of the method bytecode is proposed in [17]. This bytecode encryption counteracts the security threat of jumping out of the actual method code and executing undefined data. Another countermeasure proposed in [7, 11] counteracts the threat of type confusion between the two main data types *integralData* and *reference*. All elements of one main data type are pushed and popped onto their specific OS. Therefore, type confusion is avoided by data type-based splitting of the OS. This splitting approach requires an off-card step to replace the so-called untyped bytecodes *pop_x*, *dup_x*, *swap_x*. Another countermeasure [13] fingerprints an applet by storing the position at which a critical bytecode occurs in the code. A critical bytecode is a branch, jump, subroutine call, method return, or 0x00 or 0xFF value. During run-time, when a critical bytecode is executed, it is evaluated to de-

termine whether the critical bytecode is in a valid position and whether the OS has the right number of elements.

No Applet Preprocessing Required: In [11, 12], a defensive JVM that stores the used main data type *integralData* or *reference* during run-time for every OS and LV element are proposed. A drawback of this type of storing approach is that it requires additional memory for type storing and additional computational power for the type checks. In [1], three approaches against integrity attacks on the OS are presented. The first approach is a standard double check of the pushed and popped OS value. The second approach reduces the run-time overhead by performing an integrity check only when a firewall check is also performed. In the third approach, all values that are pushed and popped onto the OS are summed by a logical *XOR* operation and are verified when a firewall check or access to a static field is performed.

3. COUNTERMEASURE DESIGN

This section starts with the requirements of our new countermeasure. The requirements are based on the situation in which the countermeasure is ready for industrial usage. Furthermore, this section explains our countermeasure in detail and shows two examples of thwarted FAs.

Transparency: An applet programmer should not worry about how and when to perform FA countermeasures during development. Therefore, JVM countermeasures should be transparent to an applet and its programmer. Furthermore, no off-card applet pre-processing should be needed to enable the countermeasure because industrial applet providers do not want to change their applets in any way; for example, they are not willing to add additional data into an applet as proposed in [13, 7, 19, 18]. Additional data in an applet increase the risk of interoperability issues between different applets and Java Cards. Furthermore, if security features are optional, there is the risk of a downward harmonization of security standards to support every applet on any platform.

Effectiveness: Especially on resource-constrained cards, a countermeasure should counteract as many attack paths as possible. Type confusion between the two main data types *integralData* and *reference* is a required step for different attacks [21, 14, 9, 5, 22]. Therefore, a countermeasure against type confusion is very effective because it counteracts one of the first steps required for different attacks.

Low Overhead: Due to strong cost pressures on the Java Card market, a countermeasure should have a small memory footprint. Furthermore, the execution time overhead should be low due to strict timing constraints for reading out a digital passport or performing an e-banking transaction.

3.1 Data Type Sensitive Obfuscation

For the new countermeasure, we propose obfuscating the values on the OS and LVs by performing a logical *XOR* operation. Based on the Java data type, two different keys are used for the obfuscating operation. For all OS or LV elements with the Java data type *integralData* the secret key $K_{integralData}$ is used.

$$integralData_{hidden} = integralData \oplus K_{integralData}$$

All elements on the OS or LVs of the data type *reference* are obfuscated with the key $K_{reference}$.

$$reference_{hidden} = reference \oplus K_{reference}$$

Every time values are read or written from the OS or LVs, knowledge of the secret key is required. For an adversary, it would be quite easy to find the secret key by a brute force attack. For a 16-bit key, 65536 different key values are possible. To counter such a brute force attack, we propose a dynamic generation of new keys during the run-time. The two secret keys $K_{integralData}$ and $K_{reference}$ are newly generated every time the card is powered on or when a new Java method is executed, which means that the obfuscated values differ for every method on any attacked card. If an adversary discovers the secret key, it will not help him for the next method he invokes on the same card or on another card.

3.2 Countered Attacks

This section demonstrates how type confusion and buffer overflow attacks are countered by our new countermeasure during run-time. The impact of the obfuscating mechanism on the OS and LV values are shown, based on the countered attack examples.

3.2.1 Type Confusion

Figure 1 illustrates a run-time type confusion attack between *integralData* and *reference*. Java code is compiled to bytecode and loaded onto the card. During the fetch process of the *bspush* bytecode, which has the opcode 0x10 and the operand 0x19, the opcode 0x10 is changed to 0x00 (NOP). The NOP instruction performs no operation and causes the JVM to interpret the operand 0x19 as the opcode *aload_1*. The *aload_1* instruction loads the obfuscated array *reference* $reference_{hidden} = reference \oplus K_{reference}$ from the LVs and pushes it onto the OS. The next bytecode *sreturn* causes a type confusion by interpreting the array *reference* from the OS as an *integralData* type. Due to this type confusion the array *reference* $reference_{hidden}$ is unobfuscated with the wrong key $K_{ingetralData}$.

$$reference \neq (reference \oplus K_{reference}) \oplus K_{integralData}$$

For an adversary, the obfuscated array *reference* is worthless because when the actual method returns, two other secret keys are used to obfuscate *integralData* and *references*.

3.2.2 Buffer Overflow

Every time a new Java method is invoked, a Java frame is created. This frame is stored in the volatile memory of the JVM. The frame has a fixed size and contains memory space for the OS, LVs, and specific frame data such as the return address for the former method. By changing the parameters of the bytecodes with a FA, it is possible to perform write operations outside the reserved memory. For example, the EMAN2 attack [5] relies on overflow of the LVs by a malicious *sstore* bytecode. This malicious *sstore* bytecode has an invalid LV index that is higher than the reserved LV memory. Therefore, the malicious *sstore* overflows the LVs and overwrites the top OS element into the frame data, as illustrated in Figure 2. Such overflow attacks on the OS and LVs by malicious bytecodes are now countered by storing obfuscated values into the OS and LVs. An adversary, can no longer predict which value will be produced by an overflow attack of the OS or LV. Therefore, a malicious *sstore* overwrites the return address with obfuscated data. The

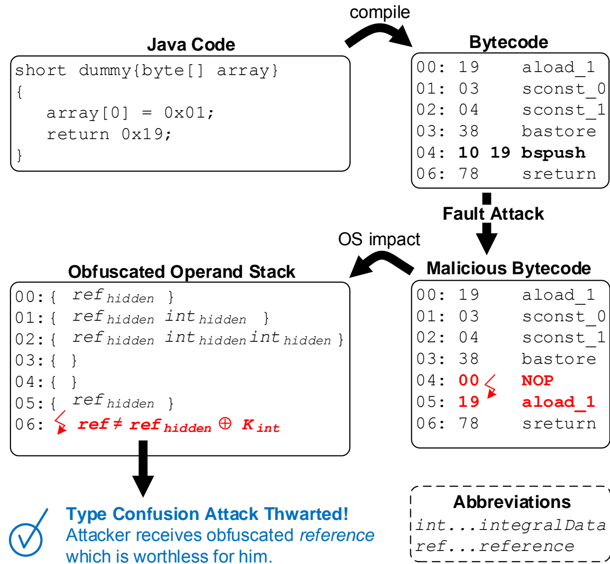


Figure 1: FA example to provoke type confusion between *integralData* and *reference* on the OS. This type confusion is used to illegally return an array *reference* as a return value. With our countermeasure, an adversary receives an obfuscated and therefore unusable array *reference*.

obfuscated data are, for an adversary, random data, which will cause the JVM to return to a random address. This random control flow change will most likely be detected and indicated by other on-card security mechanisms.

3.3 Industrial Usability

Our new countermeasure fulfills all necessary requirements for industrial use. The countermeasure needs no off-card pre-processing of a Java applet, as required by other countermeasures [18, 5, 19, 17, 7, 11, 13]. Furthermore, the countermeasure consumes no additional run-time memory for storing the Java type information during run-time as in [11]. Therefore, no type checks are needed, which would reduce the JVM execution speed. The new countermeasure generates low run-time overhead because it is based on a simple logical *XOR* operation. Note that type confusion

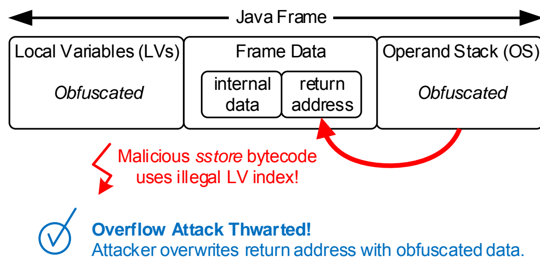


Figure 2: A malicious *sstore* bytecode with an invalid LV index can overwrite the return address of the current active frame. With our countermeasure, the return address will be overwritten with undefined obfuscated data.

inside the two main data types *integralData* and *reference* must be counteracted by additional run-time checks. Such internal type confusion could happen, for example, between a short array object and a byte array object.

4. PROTOTYPE IMPLEMENTATION

Our new countermeasure has been integrated into a prototype Java Card. The JVM was implemented in standard C and assembly language running on a 8051 compatible smart card model. The smart card is an instruction-cycle and memory-accurate SystemC model. To speed up the data obfuscating of the OS and LVs, we moved the *XOR* operation from SW to HW by implementing new defensive assembly instructions into the smart card, as illustrated in Figure 3. Into these defensive instructions are the access direction (read, write) with the memory area (OS, LVs, or BA) and the Java data type (*integralData*, *reference*, *untyped*) encoded. For example a value with the data type *reference* is read from the security critical OS memory area using the defensive instruction MOV_READ_OS_REFERENCE. Every time the defensive instructions access the OS or LVs, the values are automatically obfuscated by the HW data obfuscation unit (DOU) with one of the secret keys $K_{integralData}$ or $K_{reference}$. The values of the two keys are set for every method invocation/return by writing the values into newly added hardware registers.

To increase the overall security against FAs, we integrated two additional HW protection units: a data integrity protection unit (IPU) and a control flow protection unit (FPU). The IPU and FPU are taken from a previous work of us [10].

FPU: The FPU determines whether the current fetched bytecode is inside the Java BA. The lower and upper bound address of the BA must be set inside the FPU. The FPU counteracts attacks that let the applet jump outside the current BA and start executing data instead of bytecodes [6].

IPU: The IPU performs a double check on all values written or read into the OS or LVs during run-time. After reading or writing a *value* to the OS or LVs, the IPU automatically reads the inverted $value_{complement}$ from the same address and creates $\omega = value \oplus value_{complement}$. When $\omega \neq$ 0xff, an integrity attack is recognized by the IPU and results in a security interrupt. The IPU counteracts known integrity attacks on the OS and LVs [1].

Based on the executing bytecode the following operations are performed by the HW protection units (FPU, DOU, IPU) to counteract FAs:

- Verify whether the fetched bytecode is inside the actual BA.
- Obfuscate or unobfuscate values that are written to or read from the OS or LVs.
- Perform an integrity check on every value written to or read from the OS or LVs.

5. RESULTS

This section presents the computational overhead of the countermeasures, compared to an implementation that does not perform these run-time security operations. All security checks performed by the new HW units are, for ease of

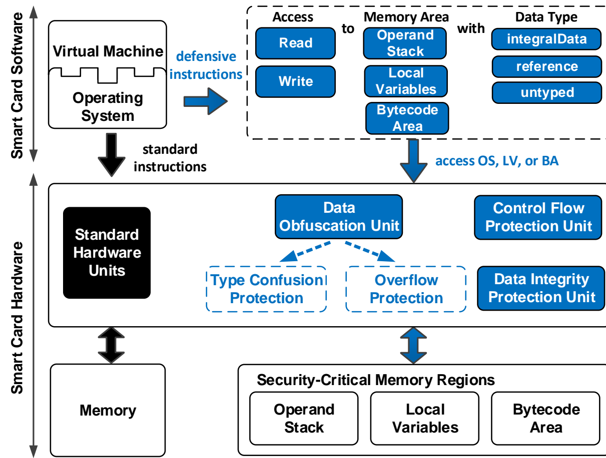


Figure 3: Implementation illustrations for the defensive JVM on a prototype smart card. The Java Card SW uses new defensive assembly instructions to access the security-critical memory regions OS, LVs, and BA. In these critical memory regions, newly added protection units perform HW-accelerated data obfuscating, control flow protection, and data integrity protection.

comparison, also reproduced by the SW checks. The measurements are performed for different groups of bytecodes with micro benchmarks. These micro benchmarks start with an initialization phase in which all needed OS and LV elements are created for the bytecode under test (BUT). Next, the BUT is executed and measured. The following cleaning step removes the outcome of the BUT and returns the card to the baseline condition. The smart card model implements no data or code cache functionality. Therefore, no influences on execution speed caused by caching effects are considered.

The computational overhead caused by the run-time security operations for specific bytecodes is shown in Figure 4. Executing the data obfuscating in the SW adds an execution time overhead of 13% to the *sstore* bytecode. The inclusion of additional SW computations, to perform control flow checks and data integrity checks, results in an additional overall execution time overhead of 107%. When all of these checks are performed by the FPU, DOU, and IPU, the execution time overhead is only approximately 6%.

The overall measurements for the bytecode groups are shown in Table 1. For SW data obfuscating, the overall execution time overhead is approximately 16%. The inclusion of all security operations in the SW increases the execution overhead to 111% while performing all of these checks in the HW decreases the execution time overhead to approximately 5%.

6. CONCLUSIONS AND FUTURE WORK

Our new countermeasure thwarts type confusion and buffer overflow attacks by obfuscating each element on the operand stack (OS) and the local variables (LVs) with a secret key. The secret key depends on the data type of the obfuscated element. For the data type *integralData* the key $K_{integralData}$ is used, and the key $K_{reference}$ is used for the *reference* data type. Both secret keys are generated every time a new method is invoked or when the Java Card powers up. If an adversary performs an overflow attack, obfuscated data are written outside the OS and LV memory area. Furthermore, an adversary performing type confusion will receive or create unusable obfuscated values. The obfuscated values cannot be appropriately used by an adversary for further attacks.

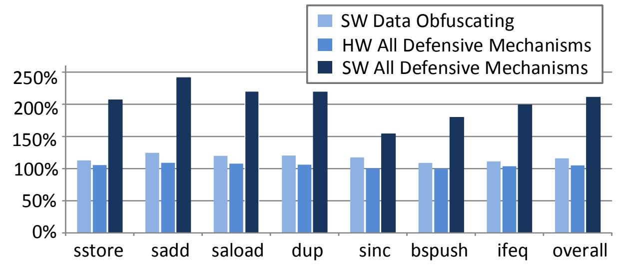


Figure 4: Execution time overhead of specific bytecodes enriched by run-time countermeasures either performed in SW or by HW support.

Table 1: Execution time overhead of bytecode groups enriched by different run-time countermeasures performed with either SW or HW support.

	Data Type Sensitive Obfuscating	*All Defensive Mechanisms*	
Bytecode Groups	*SW*	*HW*	*SW*
Arithmetic/Logic	+21%	+8%	+130%
LV Access	+15%	+5%	+126%
OS Manipulation	+16%	+4%	+108%
Control Transfer	+14%	+5%	+105%
Array Access	+15%	+6%	+100%
Overall	+16%	+5%	+111%

The new countermeasure was integrated into a Java Card prototype and was either operated in SW or accelerated with HW mechanisms. An important advantage of the new countermeasure is that no off-card processing of the applet is required. Furthermore, the run-time memory requirements are quite low because there is no need to store data types to thwart type confusion. Due to the low performance impact of an additional 5% when performed with HW support, the new countermeasure fulfills all requirements for a fault attack (FA) countermeasure in industrial usage. New industrially usable FA countermeasures are necessary because FAs cause a Java Card virtual machine to execute malicious bytecodes that change the valid data and control flow of the executing applet. Such increased Java Card applet security is needed for new applications such as near field communication and to protect secure elements in smart phones.

Future work can focus on evaluating our new fault attack countermeasures in terms of HW overhead. Such an evaluation is necessary because the gate-size is very constrained on smart cards. Furthermore, an evaluation of the countermeasures is needed in terms of side channel attacks. The newly added countermeasures should not open new security breaches which can be abused by adversaries.

Acknowledgments

The authors would like to thank the Austrian Federal Ministry for Transport, Innovation, and Technology, which funded the CoCoon project under the FIT-IT contract FFG 830601. We would also like to thank our project partner NXP Semiconductors Austria GmbH.

7. REFERENCES

[1] G. Barbu, G. Duc, and P. Hoogvorst. Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures. In E. Prouff, editor, *Smart Card Research and Advanced Applications*, volume 7079 of *Lecture Notes in Computer Science*, pages 297–313. Springer Berlin Heidelberg, 2011.

[2] G. Barbu, H. Thiebeauld, and V. Guerin. Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In D. Gollmann, J.-L. Lanet, and J. Iguchi-Cartigny, editors, *Smart Card Research and Advanced Application*, volume 6035 of *Lecture Notes in Computer Science*, pages 148–163. Springer Berlin Heidelberg, 2010.

[3] A. Barenghi, G. Bertoni, L. Breveglieri, M. Pellicioli, and G. Pelosi. Fault Attack on AES with Single-Bit Induced Faults. In *Information Assurance and Security (IAS), 2010 Sixth International Conference on*, pages 167–172, 2010.

[4] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache. Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

[5] G. Bouffard, J. Iguchi-Cartigny, and J.-L. Lanet. Combined Software and Hardware Attacks on the Java Card Control Flow. In E. Prouff, editor, *Smart Card Research and Advanced Applications*, volume 7079 of *Lecture Notes in Computer Science*, pages 283–296. Springer Berlin Heidelberg, 2011.

[6] G. Bouffard and J.-L. Lanet. The Next Smart Card Nightmare. In D. Naccache, editor, *Cryptography and Security: From Theory to Applications*, volume 6805 of *Lecture Notes in Computer Science*, pages 405–424. Springer Berlin / Heidelberg, 2012.

[7] J. Dubreuil, G. Bouffard, B. N. Thampi, and J.-L. Lanet. Mitigating Type Confusion on Java Card. *International Journal of Secure Software Engineering (IJSSE)*, 4(2):19–39, 2013.

[8] S. Hamadouche, G. Bouffard, J.-L. Lanet, B. Dorsemaine, B. Nouhant, A. Magloire, and A. Reygnaud. Subverting Byte Code Linker service to characterize Java Card API. Proceedings of the 7th Conference on Network and Information Systems Security (SAR-SSI), pages 122–128. 2012.

[9] J. Iguchi-Cartigny and J.-L. Lanet. Developing a Trojan applets in a smart card. *Journal in Computer Virology*, 6:343–351, 2010.

[10] M. Lackner, R. Berlach, M. Hraschan, R. Weiss, and C. Steger. A Defensive Java Card Virtual Machine to Thwart Fault Attacks by Microarchitectural Support. In *International Conference on Risks and Security of Internet and Systems (CRiSIS)*. in press, 2013.

[11] M. Lackner, R. Berlach, J. Loinig, R. Weiss, and C. Steger. Towards the Hardware Accelerated Defensive Virtual Machine - Type and Bound Protection. In S. Mangard, editor, *Smart Card Research and Advanced Applications (CARDIS)*, volume 7771 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2013.

[12] M. Lackner, R. Berlach, W. Raschke, R. Weiss, and C. Steger. A Defensive Virtual Machine Layer to Counteract Fault Attacks on Java Cards. In *Information Security Theory and Practice. Security of Mobile and Cyber-Physical Systems*, pages 82–97. Springer, 2013.

[13] G. Morana, E. Tramontana, and D. Zito. Detecting Attacks on Java Cards by Fingerprinting Applets. *2012 IEEE 21st International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 0:359–364, 2013.

[14] W. Mostowski and E. Poll. Malicious Code on Java Card Smartcards: Attacks and Countermeasures. In G. Grimaud and F.-X. Standaert, editors, *Smart Card Research and Advanced Applications*, volume 5189 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin / Heidelberg, 2008.

[15] Oracle. *Runtime Environment Specification*. Java Card Platform, Version 3.0.4, Classic Edition, 2011.

[16] Oracle. *Virtual Machine Specification*. Java Card Platform, Version 3.0.4, Classic Edition, 2011.

[17] T. Razafindralambo, G. Bouffard, B. Thampi, and J.-L. Lanet. A Dynamic Syntax Interpretation for Java Based Smart Card to Mitigate Logical Attacks. In *Recent Trends in Computer Networks and Distributed Systems Security*, volume 335 of *Communications in Computer and Information Science*, pages 185–194. Springer Berlin Heidelberg, 2012.

[18] A. Sere, J. Iguchi-Cartigny, and J.-L. Lanet. Checking the Paths to Identify Mutant Application on Embedded Systems. In T.-h. Kim, Y.-h. Lee, B.-H. Kang, and D. Slezak, editors, *Future Generation Information Technology*, volume 6485 of *Lecture Notes in Computer Science*, pages 459–468. Springer Berlin / Heidelberg, 2010.

[19] A. Sere, J. Iguchi-Cartigny, and J.-L. Lanet. Evaluation of Countermeasures Against Fault Attacks on Smart Cards. *International Journal of Security and Its Applications, Vol.5 No.2*, pages 49–61, April 2011.

[20] Sun Microsystems Inc. Java Card 2.2 Off-card Verifier. *White Paper*, June 2002.

[21] O. Vertanen. Java Type Confusion and Fault Attacks. In L. Breveglieri, I. Koren, D. Naccache, and J.-P. Seifert, editors, *Fault Diagnosis and Tolerance in Cryptography*, volume 4236 of *Lecture Notes in Computer Science*, pages 237–251. Springer Berlin / Heidelberg, 2006.

[22] E. Vetillard and A. Ferrari. Combined Attacks and Countermeasures. In D. Gollmann, J.-L. Lanet, and J. Iguchi-Cartigny, editors, *Smart Card Research and Advanced Application*, volume 6035 of *Lecture Notes in Computer Science*, pages 133–147. Springer Berlin Heidelberg, 2010.

Memory-Efficient On-Card Byte Code Verification for Java Cards

Reinhard Berlach, Michael Lackner and Christian Steger
Institute for Technical Informatics
Graz University of Technology
{reinhard.berlach, michael.lackner, steger}@tugraz.at

Johannes Loinig and Ernst Haselsteiner
NXP Semiconductors
{johannes.loinig, ernst.haselsteiner}@nxp.com

ABSTRACT

Java enabled smart cards are widely used to store confidential information in a trusted and secure way in an untrusted and insecure environment, for example the credit card in your briefcase. In this environment the owner of the card can install and run any applet on his card, such as the loyalty application of your favorite store. However, every applet that runs on a trusted card has to be verified. On-card Bytecode Verification is a crucial step towards creating a trusted environment on the smart cards. The innovative verification method presented in this work comes without any additional off-card component and uses nearly the same amount of memory as the execution of the applet uses. The usage of a Control Flow Graph and Basic Blocks and the implementation of a temporary transformation of the methods reduces the complexity of this new verifier. We will show a detailed analysis of the implemented algorithm and preliminary tests of a prototype on a Java Card.

1. INTRODUCTION

Smart cards are used in a wide range of applications (e.g. digital wallets, transport tickets, credit cards) to store security-critical code, data and cryptographic keys. Today's smart cards are more than a simple plastic card. They are used as a secure element in NFC enabled smart phones. With the help of these secure elements the NFC-phone is able to emulate a smart card.

Smart cards are resource-constrained devices that include an 8- or 16-bit processor, up to 4kB of volatile memory and hundreds of kB of persistent memory and sometimes a cryptographic co-processor.

Java enabled smart cards (Java Cards for short) allow small Java applications, called applets, to run on such a smart card. The *write-once, run-everywhere* approach of Java is the primary purpose of implementing a Java Virtual Machine (JVM) on a smart card. Another advantage of the JVM is the sandbox concept. This *box* separates the different application that are on one card and protects sensitive data.

In the actual system the user gets a different smart card for each application. So every one of us has a number of different smart cards in our wallets. For example, a credit card, bank card, several loyalty cards and maybe even a card to access your workplace. This is called Issuer Centric Smart Card Ownership Model (ICOM) [1]. In the ICOM the card issuer controls the applets that are installed on the card.

Since NFC enabled smart phones are widely used today, some users want to shift the functionality of their smart cards to their smart phone. Java Card enables the users to install as many applications on their "smart cards" as they want. The users are in complete control over which applications are installed on their cards. This is called the User Centric Smart Card Ownership Model (UCOM). An illustration of the UCOM we see in Figure 1. One of the major issues of the UCOM is that applets from an untrusted source are able to break the sandbox of the JVM. In the security concept of the JVM, the Bytecode Verification (BCV) is a crucial part. The BCV checks the correctness of the strong typing used in the Java programing language.

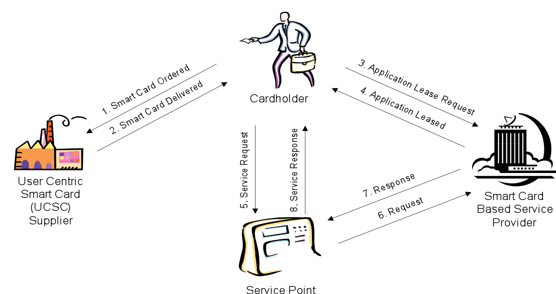


Figure 1: Relationships in the User Centric Smart Card Ownership Model [1] between card supplier, card user, application issuer and the service point

The composition of this work is as follows: We present a innovative on-card byte code verification method, we show a proof-of-concept implementation that is able to run on a smart card and we will give a analytical review of the memory consumption of this algorithm.

The paper is organized as follows: Section 2 relates the work regarding the on-card verifier. Section 3 focuses on the concept and design of this method. In Section 4 we discuss the analytical review, Section 5 shows the first results of the implemented prototype and Section 6 discusses the results and provides some potential for future work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CS2 January 20 2014, Vienna, Austria
Copyright 2014 ACM 978-1-4503-2484-7/14/01 ...$15.00.

© *2014 ACM. Reprinted, with permission, from Proceedings of 1st Workshop on Cryptography and Security in Computing Systems (CS2).*

(a) off-card BCV (b) on-card BCV

Figure 2: Trusted environment with off-card and on-card BCV.

2. RELATED WORK

This section is split into related work about the security model, the BCV algorithm and on-card BCV methods.

2.1 Java Card Security Model

Witteman [10] discussed the key characteristics of the security model in Java Cards. According to the specification [7], Witteman identified four main aspects of the Java Card security concept. These points are:

- Bytecode Verification
- Secure Loading
- Applet Isolation and Object Sharing
- Atomic Operations

The first two points and the last point are mandatory. Secure loading is needed to implement the safe path between the off-card BCV and the on-card installer, as seen in Figure 2a.

To implement this secure loading, today's off-card loader calculates a cryptographic signature for the verified application and then transmits it to the card. There the signature is verified and if it is correct, the application is installed. To verify the signature on-card an exchange of keys is needed. To prevent misuse a single source of trust has to be implemented. In the ICOM [1] this is the card issuer.

However, the paradigm shift in the ownership model [1] makes it impossible to implement a single source of trust. Moving the BCV to the card renders this trusting source obsolete.

2.2 Bytecode Verification

The original BCV was presented in 1995 by Goslin [4] as a part of the low level security of the JVM. The BCV has to check every non abstract method in all classes of an application. Therefore, each Bytecode instruction will have to be interpreted by an abstract JVM.

As an example the `SADD` instruction has to add two shorts. The two shorts have to be at the Operand Stack (OS) and they are taken from there and the result is pushed back to the OS. Consequently, the abstract JVM has to check the OS to see if there are two shorts; if so, it takes them from the OS and pushes one short back to the OS.

As we see this abstract interpretation has to be done on the Memory Frame (MF), which contains the OS and the Local Variable (LV) Registers, and each Bytecode is "*executed*" regarding to the types of the instruction. The verification fails when the OS does not contain the right types for an instruction.

Branching instructions are of special interest. Here the program flow forks. In such a fork the state of the MF in the JVM has to be forwarded to each branch. Therefore the Suns BCV [4] saves a 'dictionary'. Later, when they are joined together each of the states of the MF from the two branches has to be merged. The dictionary contains the states and the offset of the instructions and has to be saved until the method is verified.

This dictionary has to save the states of the MF for each fork and for each join in a method. The saved MFs can be changed very often during the verification. The size of such a dictionary in the memory can be given by $O(B \times (S + L))$, where B is the number of branches and exception handlers in the method, S the maximum height of the OS and L the number of LV used. This will get for a moderate complex method in an applet with 50 branches, 5 words maximal on the stack and 15 words for LV about 3500 bytes for the dictionary [3].

2.3 On-card Verifier

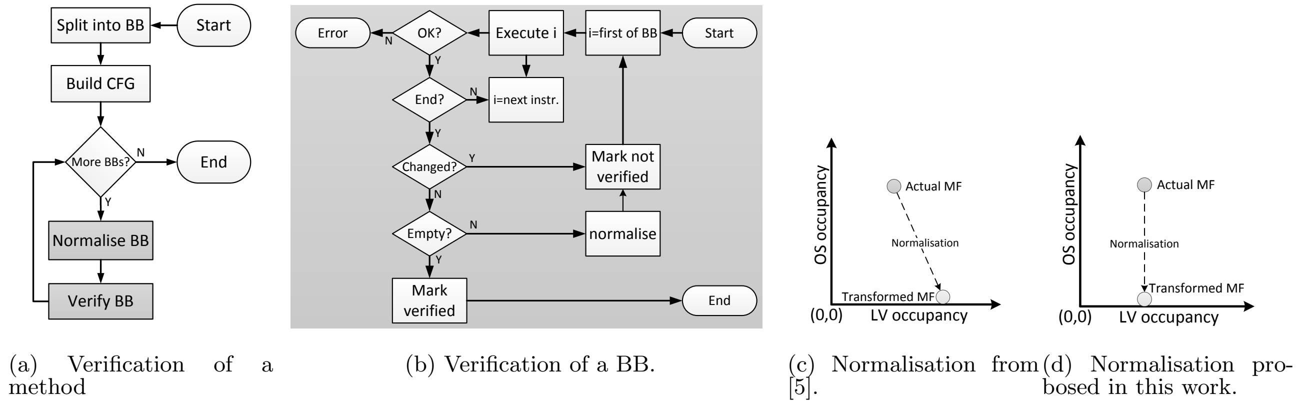
Soon after the initial presentation of Java Card the scientific community began with the research for a possible solution of an on-card BCV. The main problem with the existing algorithm is the memory consumption. For a smart card with only hundreds of byte of RAM or tenth of kilobytes of EEPROM/Flash it is not possible to store the whole dictionary. Even if it fits into the persistent memory, the amount of write access to the memory will stress it. Several authors proposed methods to overcome these problems. We will discuss two of them in the following section.

2.3.1 *Reducing the dictionary*

The first idea was to minimize the memory usage of the dictionary on the card. We will discuss two approaches forthwith.

Naccache et al. [6] proposed the use of BBs in their work. For each BB, Naccache et al. calculated the elements used in LV and OS. In the dictionary they only saved the elements used for each BB and so can save up to 90% of the size of the dictionary. The major drawback of this algorithm was the complex calculation of the elements used, and their representation in the dictionary.

The second approach was brought by Bernardeschi et al. [2]. In a similar way to Naccache et al. [6], they used the BB, or regions as they called them, to determine which entries in the dictionary can be deleted. They deleted an entry in the dictionary, if there was no path from the actual BB to the BB corresponding to that entry. Therefore, this approach only saved the reachable entries in the dictionary and reduced the memory usage of the dictionary by 75%.



(a) Verification of a method (b) Verification of a BB. (c) Normalisation from [5]. (d) Normalisation probosed in this work.

Figure 3: Verification of a Method, of a BB and the normalisation.

2.3.2 *Adding off-card components*

The second idea was to use some off-card components to modify the applet so that it can be processed using an on-card verification in a more efficient way.

Rose & Rose [9] used the off-card verifier to add Proof Carrying Code (PCC) to the applet. The PCC enabled the on-card verifier to verify the code in a single pass and the memory usage was reduced to the same amount that the execution of the method needed.

Leroy [5] used an off-card normalization for the applet to reduce the memory consumption. This normalization ensured that at each jump in the code, the OS of the JVM was empty. Overall he reported of additional 5% applet size and 2% more RAM space needed for the applets.

These two approaches reduced the memory consumption to $O(S + L)$, but both needed an off-card component to do their work.

3. CONCEPT OF THE NEW VERIFICATION METHOD

For all applets the security of the card has to be ensured in the UCOM. A crucial step therefore is to verify the correctness of the applet in respect to the types. At this point we introduce requirements for the on-card BCV:

- The BCV shall fulfill the given specification from Sun [8]
- The BCV shall run with respect to the resource constraints of the card, e.g. small RAM (hundreds bytes), usage of EEPROM (slow write access, tearing)
- The BCV shall run without any additional off-card components

Using these requirements, we propose a method that will verify an applet on the card. This method will be based on the previous work of Naccache [6], Bernardeschi [2] and Leroy [5].

Our proposed verifier will also use the abstract interpretation as it was introduced by Goslin [4]. As proposed by Naccache [6] and Bernardeschi [2] our algorithm splits each non abstract method of the application into BBs. The next step in our verification is the normalisation and the abstract interpretation. The normalisation is used, as discussed at Leroy [5], to make the process that verifies the method stateless. This can be seen in Figure 3a

This stateless process is archived in a similar manner as Leroy [5], but contrary to him we move the normalisation onto the card. Therfore, our system will not change the code that will be executed later on the card. Consequently our BCV has to store the normalising function temporarily, i.e. it stores the results of the normalisation in the transient memory.

3.1 Normalization

The normalization process proposed by Leroy has to ensure that the OS is empty at the beginning and end of each of these BBs. In our new verification method this precondition of each BB will be met through the usage of a temporary normalization. *Temporary* means that the normalized code will not be executed later on the JVM. It only is needed for the verification.

Each BB that has a non-empty OS at the end, has to be normalized. A normalization can be seen as an transformation of the MF from the actual state to a predefined. In our case this predefined state is a MF with an empty OS. We call the predefined state of the MF MF_{def}. The state at the end of a BB we call MF_{BB}. Then we can define $N_{BB} : MF_{BB} \mapsto MF_{def}$, where N_{BB} is the transformation function of the BB. The differences between Leroy's function and our proposed system can be seen in Figure 3c and 3d.

3.2 Verification

The verification of a BB starts with the abstract interpretation of the instructions as it is shown in Figure 3b. This works in the same way as the original BCV [4]. If an error occurs while interpreting the instructions, the system goes to an `Error`-state. When the last instruction from the BB is correctly interpreted, the MF is checked to see if there were any type changes in the LV. If an element of the LV has changed, the verification of the BB starts again at the beginning, and also all following BBs have to be marked as `not verified`. If not, the status of the OS is checked. If the stack is empty, the actual BB is marked as `verified` and the next BB in the queue has to be verified. If the stack is not empty after the last instruction from the BB, the BB has to be normalized and reverified.

4. IMPLEMENTATION

Our first prototype is implemented completely in Java and runs on a Java Card. The required queue and BBs are implemented as objects and therefore stored in the persistent memory. Since the BBs are only written a few times in one verification and the queue and its elements are mostly used statically, this decision will not result in a tearing problem in the persistent memory. Also runtime drawbacks only come with the write-access of the EEPROM.

The MF is implemented as a static object. The LV and OS are put into the RAM and can only be accessed through methods of the MF. The MF is also responsible for calculating the function N and its inverse.

Normalized BB will hold the normalization function N, or its inverse, in the transient memory. The inverse transform function will be interpreted before the first Bytecode and the function itself will be interpreted after the last Bytecode. This normalization functions can be discharged, when the verification of the method is finished.

A rather simple method is used in this proof-of-concept implementation to save the normalizing function N. We merely save the elements of the OS in a transient array. Since in 95% of the normalizations only one element (most of the time a short) is on the stack, this simple method does not use to much of the valuable RAM.

5. RESULTS

From the work of Leroy [5] we have seen that only up to 5% of BBs have to be normalized. Also we have seen that the normalization function N only needs less than 5 byte for each BBs.

So when we take the moderate complex method from Section 2.2 with a maximal stack size of 5 words, 15 words of LV and 50 branches, we will get around 70 to 75 BBs. Four of these BBs need normalization, which will use additional 5 bytes each. In this example the usage of RAM will be about 80 bytes, $4 \times 5 = 20$ bytes for the transformation functions and $(5 + 15) \times 3 = 60$ bytes for the MF.

The verification of an applet from our industrial partner shows that the BCV does not need more than 100 bytes of RAM, when working on an industrial Java Card applet.

First tests with the implementation have shown that all explained algorithms (Figure 3) are capable of running on a commercial Java Card, which was provided by our industrial partner.

6. CONCLUSION AND FUTURE WORK

In this work we have shown that it is possible to implement and use an on-card BCV on a low-cost Java Card. This verification is a first crucial step forward to a trusted environment that lies completely on the card.
The proposed BCV can verify an uploaded applet without any off-card preprocessing. Also the BCV runs with less memory consumption and a similar runtime behavior. The proposed BCV will not extend the memory consumption of the methods, when they are executed, as Leroy's BCV does. A further advantage of the shown algorithm is, that it won't revert the optimizations of the compiler or programmer.

For the moderate complex method (50 Branches, 5 words OS and 15 words LV) our BCV uses 80 bytes of RAM. The original BCV [4] uses, for the same method, 3500 bytes. The verifier using the PCC will only use 60 bytes for the MF. Leroy's BCV [5] needs 66 bytes, because of the expansion of the MF for the normalization. These two verifiers are better than the proposed one but they need some off-card components.
The verifiers that are not using an off-card component need more than four times the memory than our BCV needs. In the case of Naccache et al. [6] it is 10% of the memory consumption of the original BCV, also 350 bytes. The verifier of Bernardeschi et al. [2] needs 25% of memory as the original BCV. This gives a memory consumption of 875 bytes. This shows that our proposed BCV is capable of running even in the most constrained environment like a smart card.

A possible field of further development could be the implementation of transient objects in the Java Card Runtime Environment. This could further optimize access to objects that are only created for the verification, such as the BB or the Control Flow Graph.

Acknowledgments

This work is part of the CoCoon Project. This project is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FIT-IT contract FFG 830601. The authors would like to thank the project partners, NXP Semiconductors Austria GmbH and Graz University of Technology, for their support and encouragement.

7. REFERENCES

[1] R. Akram, K. Markantonakis, and K. Mayes. A Paradigm Shift in Smart Card Ownership Model. In *International Conference on Computational Science and Its Applications (ICCSA), 2010*, March 2010.

[2] C. Bernardeschi, L. Martini, and P. Masci. Java bytecode verification with dynamic structures. In *International Conference on Software Engineering and Applications (SEA)*, Cambridge, MA, USA, 2004.

[3] D. Deville and G. Grimaud. Building an "impossible" verifier on a java card. In *Proceedings of the 2nd conference on Industrial Experiences with Systems Software - Volume 2*, Berkeley, CA, USA, 2002. USENIX Association.

[4] J. Gosling. Java intermediate bytecodes. *ACM SIGPLAN workshop on intermediate representations (IR'95)*, 30(3):111–118, 1995.

[5] X. Leroy. Bytecode verification on Java smart cards. *Software: Practice and Experience*, 32(4):319–340, 2002.

[6] D. Naccache, A. Tchoulkine, C. Tymen, and E. Trichina. Reducing the memory complexity of type-inference algorithms. In *Information and Communications Security*, volume 2513 of *Lecture Notes in Computer Science*, pages 109–121. Springer Berlin / Heidelberg, 2002.

[7] Oracle. *Virtual Machine Specification*. Java Card Platform, Version 3.0.4, Classic Edition, 2011.

[8] Oracle. Java card 3 platform off-card verification tool specification, classic edition. Beta Draft Version 1.0, Oracle, February 2012.

[9] E. Rose and K. H. Rose. Lightweight Bytecode Verification. *Journal of Automated Reasoning*, 31:303–334, 2003.

[10] M. Witteman. Java Card Security. Information Security Bulletin, July 2003.

A Fault Attack Emulation Environment to Evaluate Java Card Virtual-Machine Security

Michael Lackner, Reinhard Berlach, Michael Hraschan, Reinhold Weiss and Christian Steger
Institute for Technical Informatics
Graz University of Technology
Inffeldgasse 16/1, A-8010 Graz, Austria
{michael.lackner, reinhard.berlach, rweiss, steger}@tugraz.at
michael.hraschan@student.tugraz.at

***Abstract*—Java-enabled smart cards are used in different fields of application, such as access control, electronic banking, and passports. On these cards, a standardized virtual machine runs, which protects the security-critical code and data using a sandbox model. Unfortunately, this sandbox can be circumvented by fault attacks, which corrupt the data on which the virtual machine operates.**

The fault emulation environment of this work enables the user to configure faults at definable Java applet code locations. The user specifies which Java code she wants to attack but does not need to provide any information on where these data are placed in the memory and when the memory is accessed. To enable this approach, our environment monitors the virtual machine during the applet execution to receive the information which Java code is currently executed and which security-critical memory regions are in use. Then, the faults are injected using a bus saboteur at the correct clock cycle and memory location. This generic high-level approach provides the environment user an abstraction of the internal states of the virtual machine and the emulated hardware. Therefore, our environment enables the recreation of currently known attacks and allows us to study the effects of fault attacks on the virtual-machine behavior. The concept was successfully evaluated by a Java wallet case study. This case study shows a speedup of 6,600 compared to a simulation.

I. Introduction

The current Java Card security relies on the concept of a sandbox model, where no applet can perform illegal memory access operations. This sandbox relies on the fact that every applet and its bytecodes fulfill the Java Card specification [1], [2]. The fulfillment of this specification is currently verified by a static applet verification process, which is executed one time either on-card or off-card [3]. Unfortunately, researchers have found that this static verification does not protect against fault attacks during run-time [4].

To counteract fault attacks against the virtual machine (VM), new software (SW) checks [5], [6] and hardware (HW) checks [7], [8] were added to the Java Card. For example, HW-accelerated Java data-type checks [7] are integrated to counteract type-confusion attacks among incompatible data types. Testing and evaluating SW and HW security features are challenging tasks during the design process of a card. To support these tasks, a new tool is required at an early design stage.

A main drawback of the currently proposed fault injection environments for Java Cards is that the HW checks cannot be verified in detail because of their high HW abstraction [9], [10], [11]. Otherwise, waiting for the final silicon product, as proposed in [12], significantly slows down the overall design process. Another approach is to simulate the entire card and its HW mechanisms at the register transfer level. Unfortunately, a simulation of the entire Java VM at this notable low abstraction level implies a significant increase in the simulation time. To overcome these disadvantages, in this work, we emulate the HW in a field-programmable gate array (FPGA) instead of using the computationally expensive simulation.

Our proposed environment, which is shown in Figure 1, helps the test engineers and security evaluators to perform fault attacks on Java VMs to bypass the Java sandbox model. To create such attacks, our environment parses the Java applet (class and debug files) on a host personal computer (PC), shows the corresponding bytecodes, and enables the test engineer or security evaluator to inject faults into memory regions that are affected by the bytecode. For example, to create these malicious applets, a fault can be injected when the VM fetches the bytecode of a Java applet. By manipulating the fetched data, it is possible to change the control and data flow of the applet.

Therefore, in this work, we propose a fault emulation platform that can inject faults into the security-critical memory regions of the VM. The user can select the bytecodes and security-critical memory regions that she wants to attack. Then, the environment automatically injects the appropriate faults into the bus at the correct moment when the VM accesses the memory to execute the bytecodes under attack. The contributions of this paper to circumvent the drawbacks of current fault injection environments for Java Cards are as follows:

- The platform increases the environment execution speed by emulating the system on an FPGA to inject faults at a detailed hardware model with a high execution speed.
- The environment enables the evaluation of SW and HW checks against fault attacks.
- The fault configuration and location are defined at a high abstraction level and automatically injected during run-time.
- A scalable framework design that supports a variable number of faults per applet and per Java code location.

© 2014 IEEE. Reprinted, with permission, from Proceedings of 17th Euromicro Conference on Digital Systems Design (DSD).

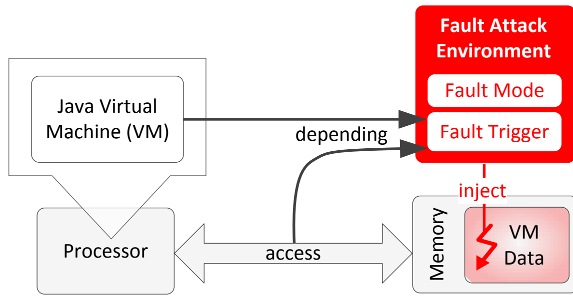


Fig. 1. Schematic view of the main functionalities of the emulation-based fault attack environment of this work. The user selects the faults from a high abstraction level (bytecode of the applet), and the environment automatically triggers the faults at the right moment during the applet execution. The fault trigger depends on the state of the VM and the state of the bus.

Section II provides an overview of related work for Java Card security, the currently proposed fault injection environments, and the fault emulation. Section III provides insight into the design of the fault emulation environment of this work and its main parts. Section IV describes the tools to implement the prototype. Furthermore, a case study is shown on how to use the environment to manipulate a personal identification number (PIN) check of a Java wallet applet. Then, measurements of the performance results and HW overhead are provided based on an FPGA prototype. Finally, conclusions and future studies are presented in Section V.

II. Related Work

In this section, an overview of Java Card security is provided with a focus on fault attacks. Furthermore, the memory regions of the VM that can be attacked by our environment are explained. Then, previous works on fault attack simulation environments for Java Cards and smart cards are summarized, and their drawbacks are discussed. Finally, an introduction to fault emulation is provided.

A. Java Card Security

A Java Card can execute different applets, which are protected from each other by a sandbox concept. The sandbox protects the security-critical code and the data of all installed applets from illegal accesses. This sandbox is only effective when every applet and its bytecode fulfill the Java Card specification [1], [2]. The specification fulfillment is verified by an applet verification process [3], which has high memory and performance requirements. Therefore, among the current Java Cards, this verification is performed off-card in a secure environment. It is possible to perform the post-issuance installation of applets by signing the verified applet with a secret key.

B. Fault Attacks

It is possible to change the execution behavior of a silicon chip by operating it beyond its standard technical specification, such as the allowed temperature range, the allowed supply voltage amplitude, or the maximum/minimum clock frequency. Another destination of fault attacks is the firing of high-energy optical pulses (laser attack) or X-rays (electromagnetic radiation) onto the security-critical HW components of the chip. Such a fault attack during the execution of cryptographic operations can extract the secret cryptographic key [13], [14].

Java Card Attacks: Further targets of fault attacks include the memory chip and the system bus, which can change the data that are read out of or written into the memory [15]. The adversary that changes the data in the security-critical memory regions of the VM can create a so-called malicious Java applet. Such a malicious applet can break out of the sandbox model and illegally access the code and data of other applets or the Java Card environment [16], [17]. For Java VMs, the main targets of such memory attacks are the following VM memory regions:

- **Bytecode Area (BA):** The BA contains the Java bytecodes of an installed applet. A bytecode consists of an opcode and optional operand [1]. An attacker, which changes the values that are fetched during the execution of an applet from the BA, can change the control and data flow of an applet. For example, it is possible to jump outside the BA and illegally execute a data array that is filled with malicious bytecodes by changing the operand of a *goto* bytecode [17]. Another threat is the skipping of bytecodes (e.g., PIN check, control flow check) to circumvent security checks. Such security checks can be skipped by manipulating the fetched bytecodes using a fault attack to 0x00, which is interpreted by the VM as the *no operation* (NOP) bytecode [16].
- **Operand Stack (OS) and Local Variables (LVs):** For every invoked Java method, a new Java frame is created which consists of the fixed-sized OS, LVs, and general frame data. The OS can be viewed as a standard first-in-first-out data structure, which pushes and pops data onto the top element. The LVs are similar to the registers of a processor and can be freely accessed. Most bytecodes rely their operations on the data integrity of the OS and LVs. For example, the bytecode *ifeq* performs a jump operation when the top value on the OS is zero; otherwise, no jump is performed. An adversary that manipulates the data integrity of the OS can manipulate the outcome of the *ifeq* execution [18] and can therefore change the control flow of a Java applet to an illegal behavior.
- **Java Heap (JH):** The JH contains the object instances of classes, which are created by the Java Card VM. In other words, the JH contains the general object description and the data of the object members. Therefore, the data on the heap is another profitable memory region for a fault attack.
- **Java Program Counter (JPC):** The JPC indicates which bytecode is currently executed in the Java applet. The JPC is updated after every executed bytecode to point to the next bytecode to be executed. An adversary can repeatedly execute bytecodes by preventing the JPC updating mechanism. Furthermore, an adversary can manipulate

the control flow of the applet, by changing the JPC value. This action can cause the threat of jumping out of the BA and illegally executing a data array instead of a valid bytecode, as shown by the authors in [17].

During its execution, almost every bytecode accesses the security-critical memory regions of the VM (BA, OS, LVs, JH, and JPC) and relies on the data integrity of these regions. A fault attack on these regions can execute an adversary-defined code [16], perform memory dumps of security-critical data [19], jump over security checks [18], and attack the Java type system [20]. In our environment, it is possible to recreate and evaluate such fault attacks to perform a design space exploration of the appropriate SW or HW countermeasures.

C. Fault Injection Environments

Most currently proposed fault injection environments for Java Cards are based on SW fault injection. In [10]], the bytecode inside the BA is manipulated to create a malicious applet. Then, this malicious applet is uploaded into a simulator and evaluated. In this environment, it is not possible to inject faults during run-time into the main memory where the VM contains the OS, LVs, JH, and JPC. In contrast to their work, our environment can inject faults into these main memory regions.

Another SW-based Java Card fault injection environment was proposed by the authors in [9]. They used the freely available binary file of the Java Card simulator cref, which runs on a desktop computer, as a Java Card environment. They wrote a plugin for cref to access and manipulate the data inside the simulated non-volatile memory. The non-volatile memory contains the BA. Therefore, this tool cannot inject faults into the main memory that is used by the VM.

Another idea was to directly embed the fault injection unit as an SW module into the final smart-card product [12]. No SW simulator is required because the SW fault injection runs directly on silicon. The big drawback is that they can use this fault injection tool only when the first silicon prototype remains available. At this late stage in the project, any design change, particularly on the HW, is extremely expensive.

The authors in [11] proposed a high-level fault injection environment for smart cards that are written at the system level. The drawback here is that the HW is highly abstracted from the real implementation. Furthermore, in their work, there is no link between the Java VM and the fault injection environment to automatically inject faults at this point in time when a specific bytecode is executed.

Advantages of Our Fault Attack Environment: The previously proposed Java Card and smart card fault injection tools are based on SW fault injection. The drawbacks of this technique include the high abstraction of the HW, the impossibility of injecting faults into the main memory [10], [9], [11] and the late usability in the design process [12]. To overcome these drawbacks, our environment emulates the actual hardware at the register transfer level on an FPGA board. This emulation indicates that our environment can be used during the HW/SW design process and enables a high execution performance of the VM. Furthermore, it is possible to inject faults into the main memory where the security-critical VM data are located. Therefore, in contrast to the previous works, our environment can be used during the design process of the chip to test and verify SW checks and, in particular, HW-accelerated security checks of the Java VM as proposed in [7], [8].

D. FPGA Fault Emulation Techniques

FPGA platforms are used to overcome the long time required to simulate HW on a processor. The digital HW components are described in an HW description language (e.g., VHDL, Verilog), synthesized to a netlist and uploaded onto the FPGA. This simulation of the hardware on the FPGA is called emulation. Then, the emulated HW can execute programs with a high execution speed and clock frequency. Different concepts of how a fault can be injected into the emulated HW on an FPGA are as follows [21]:

- **FPGA Reconfiguration:** The netlist on the FPGA is changed during the run-time to simulate fault attacks [22]. A drawback is that this reconfiguration feature is only available for specific FPGAs.
- **Mutant:** An emulated HW module is exchanged during the run-time with another module that simulates the attack [23].
- **Saboteur:** It is possible to change the values of a series of signals by placing saboteurs units in the series [24]. For our fault attack environment, we integrated saboteur units into the data signals of the bus between the emulated processor and memory. With this saboteur approach, we achieve a higher flexibility and adaptability than the FPGA reconfiguration or additional mutant units.

III. Design of the Fault Emulation Environment

This section summarizes the main features and design requirements of the fault emulation environment of this work. An overview of the main components of our environment is shown in Figure 2. The *fault injection tool* runs on the host PC, which is controlled using a graphical user interface (GUI). Using this tool, the fault attacks are configured for different bytecode points in the Java applet. It is possible to configure more faults during the execution of one bytecode. Then, these fault configurations are loaded to the *fault injection controller*. The controller runs on the emulated processor on the FPGA board and is aware of the currently executed bytecode and currently used memory regions of the VM. The memory bounds of these regions are received from the *data provider* unit, which is also added to the VM. When a bytecode for an attack is reached, the *bus saboteur* unit is configured immediately before this to-be-attacked bytecode is executed. Based on this configuration, the bus saboteur changes the data on the bus when the access to the security-critical VM data occurs. The design of the entire environment and the main components are based on the following requirements:

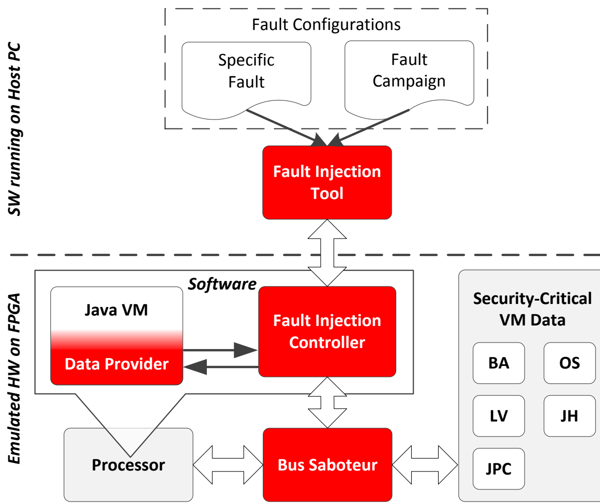


Fig. 2. Detailed overview of the proposed fault emulation environment of this work. The interface to the user is the fault injection tool, which runs on a host PC. Using this tool, the user configures specific faults or the automatically performed fault campaigns onto different bytecodes. The fault injection controller hides the required low-level knowledge of the internal state of the VM and configures the bus saboteur unit.

- Easy interchangeability of different environment components to support different VMs or different communication interfaces from the FPGA to the host PC.
- Scalable design to inject multiple faults for several attacked bytecodes.
- Online re-configurability of the saboteur units to re-use a previously triggered saboteur to inject faults during other bytecode executions.
- Support of fault campaign scripts, which enable an automatic creation and evaluation of the fault attack runs.
- Faults can be injected into the BA, OS, LVs, JH, and JPC memory regions of the VM.

Fault injections are only performed in specific security-critical memory regions of the VM. Therefore, the fault injection controller, which runs in the SW on the same processor as the VM, will never corrupt its own data in the memory. Such self-injection would lead to unpredictable side effects.

A. Fault Injection Tool on the Host PC

The fault injection tool provides an easy-to-use user interface to simplify the creation of fault attacks into Java applets as much as possible. Injecting a fault must be as easy as writing the Java applet itself. The fault injection tool runs on a host PC and enables the user to inject faults from a high level of abstraction into the security-critical memory regions of the VM. To enable this high level of abstraction, the Java class and debug files are parsed after the compilation of the Java applet. Based on these parsed data, the Java code and corresponding bytecode of the Java applet are shown to the user. Then, the user can configure specific faults or create fault campaigns. For the specific faults, the user uses a GUI to select the bytecodes that should be attacked and configures different faults that will be performed during the bytecode execution. The fault campaigns are programmed with a script language. Based on this script, different fault attacks are automatically created with different variations based on when, where, and what fault is injected.

Attackable Memory Accesses: In this section, we provide examples of which memory regions are attackable during different bytecode executions. These information were extracted from the Java Card VM specification [1] where for every bytecode a textual description of the operation and the needed memory accesses is given. Note that these memory accesses can differ between different VM implementations.

For example, the *sstore* bytecode is used by the Java compiler to store the result of an arithmetic operation into a local variable. During the *sstore* execution, the top element is read from the OS and subsequently written into the LV at a specific index. The *sstore* bytecode consists of one opcode byte (0x29) and one index byte that points into the LVs [1]. Both bytes are stored inside the BA. Therefore, a user who selects the *sstore* bytecode can inject faults during the fetching of two bytes from the BA, two reads from the OS, and two writes into the LVs. In Figure 3, we present the vulnerable memory regions for the *sstore* and further bytecodes. We did not illustrate the updating of the Java program counter to point to the next bytecode. For example, for the *sstore* bytecode, the JPC is updated to $JPC = JPC + 2$.

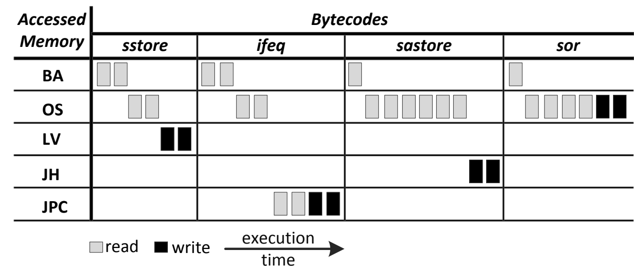


Fig. 3. Based on the Java Card VM specification [1], different bytecodes and their required byte accesses to the VM memory are shown. Our fault attack environment enables the injection of faults during the accesses into these memory regions. The *ifeq* example illustrates a successful branch execution.

B. Fault Injection Controller

The fault injection controller, which is implemented in the SW, is shown in Figure 4. This controller communicates with the host PC, bus saboteur, and data provider. The controller can send and receive data from the host PC to receive the fault configurations and break points for the next test run.

After every executed bytecode, the fault attack controller is called in the SW by the VM. The controller checks whether the next executed bytecode should be attacked or whether a break point is reached. If the current bytecode is a break point, then the Java applet is halted, and the host PC is notified by a message. The host PC can then send requests to obtain internal information from the VM or applet. This information can be the values of the method variables, object fields, or

Java arrays. When the next bytecode should be attacked, the controller configures the bus saboteur unit based on the following mandatory points:

- **Fault Mode:** The transient fault mode provides the bus saboteur with the information on how to manipulate the data on the bus. Different supported fault modes are shown in Table I. With the bit-precise mode, every data bit on the bus can be independently manipulated to be ”0” or ”1”. Current attackers cannot inject such bit-precise faults, but this situation may change in the future. The current realistic fault model is that an adversary can change all data bits to either ”0” or ”1” [18], [5]. This realistic model is represented by two fault modes: set-all-one and set-all-zero. Further supported modes are the indetermination, inverting, and delay-transfer of the bus data.
- **Fault Bit Mask:** The fault bit mask defines exactly which bits of the bus data signals are affected by the fault. Only these bits are changed based on the configured fault mode.
- **Memory Region:** For every fault configuration, the attacked security-critical memory region of the VM must be defined. The start and end addresses of these regions partially change for every executed method and cannot be statically calculated without deep insight into the VM. Therefore, in our approach, the fault controller obtains the memory range information over an SW interface from the data provider unit that is added into the VM.
- **Access Direction:** The access direction specifies whether the fault is triggered when data are read or written from the memory.
- **Access Counter:** The access counter number is set by the host PC program and determines how many accesses to a specific memory region are required to trigger the fault.

After handling the fault attack or break point, the controller gives the control of the processor back to the VM, which begins executing the next bytecode.

TABLE I
SUPPORTED TRANSIENT FAULT MODES OF THE BUS SABOTEUR UNIT. BASED ON THESE FAULT MODES AND A FAULT BIT MASK, THE DATA ON THE BUS ARE MANIPULATED DURING ONE BUS TRANSFER.

Fault Mode	Description
Bit-Precise	Data are overwritten at the bit level
Set-All-One	All data are set to ”1”
Set-All-Zero	All data are set to ”0”
Indetermination	Data are set to a random value
Inverting	Data change from ”1” → ”0” and ”0” → ”1”
Delay-Transfer	Data of the previous bus transfer are used

C. Data Provider

Using the data provider, the fault injection controller has a generic SW interface to obtain internal data from the VM. This ability indicates that only the internal functionality of the data provider must be adapted when another VM is used. The data provider provides the following listed SW interfaces and must implement their functionality:

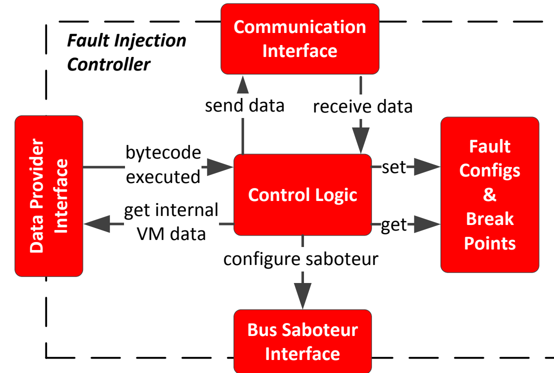


Fig. 4. Overview of the main parts of the fault injection controller, which communicates with the host PC over the communication interface. The control logic is informed by the VM for each executed bytecode and configures the bus saboteur based on the corresponding fault configurations.

- **getCurrentMemoryRegions:** This interface returns the upper and lower memory bound addresses (BA, OS, LVs, JH, or JPC) of the currently used security-critical memory regions of the VM.
- **inspectVariable:** This interface returns the value of a local or static Java variable.
- **inspectField:** The values of a Java object field can be inspected using this method.
- **inspectArray:** This method returns the values of a Java array object.

The functionality to obtain the currently used memory regions of the VM using the *getCurrentMemoryRegions* interface is required because these memory regions change during the run-time. Without this interface, the memory bounds must be manually analyzed and updated for every VM or applet change. Examples of memory regions that often change are the OS and LVs, which must be updated for every Java method invocation or return. When method_1 invokes method_2, the virtual machine switches its operations from OS_1 and LV_1 to OS_2 and LV_2.

The functionality to inspect the values of variables, fields, and arrays is particularly used during the evaluation of multiple automatically performed faults during a fault campaign run. The values are used to indicate whether a fault attack was successful and could manipulate a value at the Java layer. Successful attacks are logged and can be subsequently analyzed in detail by a security or test engineer.

D. Bus Saboteur

The proposed bus saboteur unit is particularly described and designed for the AMBA High-performance Bus (AHB). We chose the AHB because of its wide usage in different applications. Nevertheless, the general design concepts can be transferred to other buses.

The bus saboteur unit is placed into the AHB between the processor and the memory. All data that are sent from the

processor to the memory and vice versa are routed through the bus saboteur unit. The bus saboteur is more complex than the classical saboteur units, which are described in [21], [24]. The bus saboteur analyzes the data transfer based on the control and address signals. The saboteur can count the number of memory accesses at the byte level to the security-critical memory regions of the VM. To enable this byte counting, the bus saboteur understands advanced AHB data transfer features (e.g., burst transfers and split transfers).

An overview of the bus saboteur unit is shown in Figure 5. The bus saboteur consists of trigger evaluation units and data saboteur units. Both units are configured using the fault injection controller. The trigger evaluation units analyze the AHB control and address signals and trigger the corresponding data saboteurs. The data saboteur units manipulate the AHB data signals depending on the configured fault model and fault bit mask. To inject different faults during one bytecode execution, the bus saboteur has a generic design to integrate several trigger evaluation and data saboteur units. The trigger evaluation units analyze the AHB control and address signals in parallel. The data saboteurs are placed in series into the AHB data signals. The evaluation and manipulation are performed using a digital combinatorial logic with zero clock-cycle delay. In our prototype implementation, even 12 data saboteurs are feasible and obtain a valid system, which does not violate any data path timing requirements of the AHB. With these 12 data saboteurs, we are able to manipulate 12 memory accesses on the bus during one bytecode execution.

IV. Prototype Results

In this section, first, we present the tools that we use for our prototype implementation. Based on this implementation, we present a case study of how a PIN check method can be disturbed to accept any entered PIN. Furthermore, we provide measurements of the HW and the execution time of our prototype.

A. Used Tools

To implement the prototype, we use SimpleRTJ[1] as a Java VM. This VM is particularly designed to run on resource-constrained embedded systems and is freely available for non-commercial purpose. SimpleRTJ is written in C and can be easily configured to run on a wide range of HW platforms. As a target platform, we use an open-source Leon3 Sparc V8 processor implementation from Aeroflex Gaisler[2]. The basically configured Leon3 processor is synthesized on a Xilinx Spartan-3 FPGA development board, as listed in Table II. The FPGA board communicates with the host PC over a serial communication interface. On the host PC, the fault injection client was programmed with Eclipse 4 RCP.

B. Case Study - Manipulate PIN Check

For this case study, we use a self-programmed Java wallet applet without special software security hardening. The wallet applet is only an aid to demonstrate that our concepts are able to recreate fault attacks on the OS as described in [18]. It is not our aim to show a new attack against Java Cards.

[1] web.archive.org/web/20130803012237/www.rtjcom.com

[2] www.gaisler.com/index.php/products/processors/leon3

TABLE II
Configuration of the LEON3 processor that was used for the run-time performance of all case studies and HW overhead measurements.

Operating Frequency	40 Mhz
#Leon3 Cores	1 Core
Instruction Cache	Deactivated
Data Cache	Deactivated

Before performing the money transfer functions, the wallet applet user must enter a PIN. In our wallet applet, the PIN is validated by the method *validatePin()*, which is shown in Listing 1. When a valid PIN is entered, *validatePin()* executes *return true;* otherwise, it executes *return false;*. The goal of our fault attack is that *validatePin()* also returns *true* when the pin is invalid. To reach this goal, we analyze the applet over the host tool on the PC and inspect the *return false;* code segment. The tool, shown in Figure 6, indicates that *return true;* is compiled to the bytecodes *iconst_0* and *ireturn*. The bytecode *iconst_0* pushes 0x00 on the OS, and *ireturn* transfers this OS element to the previous method. Using a set-all-one fault attack during the execution of the *iconst_0* bytecode, the data that were written into the OS are manipulated from 0x00 to 0xff. Through this manipulation, the Java code *return false* logically changed to *return true*. Then, the fault configuration is uploaded to the fault injection controller on the FPGA board, and the applet is started. As a result of our fault injection, we can perform money transfer functions without knowing the valid wallet PIN.

```
1  public boolean validatePin(int pin) {
2    if (pin == secret_pin)
3      return true;
4    return false;
5  }
```

Listing 1. This *validatePin* method is attacked by our environment to also return *true* when the entered PIN is not valid. This simplified wallet example is programmed without any special security hardening or cryptographic support.

Background Information: During the Java bytecode execution, the OS memory region is used to push the results of comparison operations onto it. When the comparison was successful, the value 0x00 is pushed onto the OS. When the comparison was not successful, a value unequal to 0x00 is pushed onto it. In [18], it was shown that it was possible to inject a fault into the OS and change the value that was written or read to an adversary defined value, such as 0x00 or 0xff. Therefore, an adversary can change the control flow of an applet and, for example, change the PIN comparison operation to her favor. In this case study, we demonstrate that our environment can reproduce such fault attacks on the security-critical memory regions of the VM, as shown in related works [18], [16], [17].

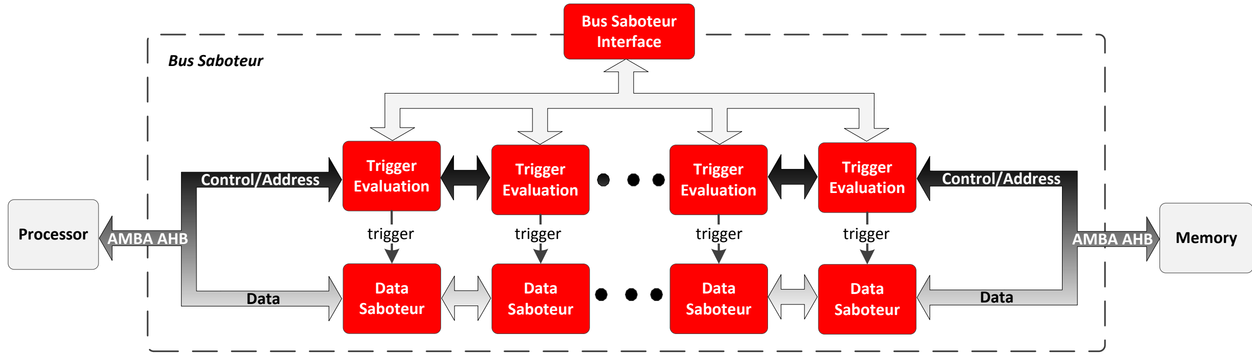


Fig. 5. Implementation overview of the generic bus saboteur unit. The bus saboteur obtains the actual fault configuration from the fault injection controller and sets the corresponding trigger and evaluation units. These trigger units evaluate the control and address signals of the AHB. When a trigger point is reached, such as the first write access to the OS, then the specific data saboteur is activated. These data saboteur changes the AHB data signals. The evaluations of the AHB control and address signals and the manipulation of the AHB data signals are performed inside of one clock cycle.

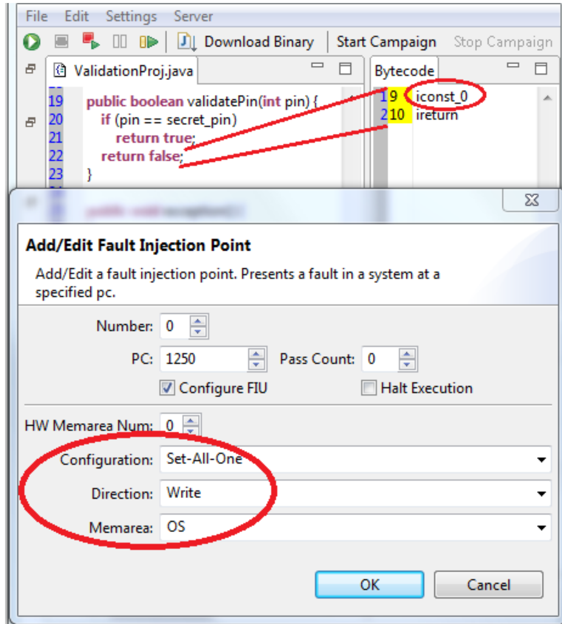


Fig. 6. To manipulate the PIN verification process, the *return false;* statement in the *validatePin* method is selected for a fault attack. The *return false;* code is performed when the entered PIN is not valid, and this Java code is equivalent to the Java bytecodes *iconst_0* and *ireturn*. During the *iconst_0* execution, a set-all-one fault is injected to manipulate the first byte, which is written into the OS to 0xff. Through this attack, the *validatePin* method logically returns true also when the entered PIN is not valid.

C. Hardware Overhead

The bus saboteur unit is highly generic and can be synthesized to support several fault injections during one bytecode execution. An overview of the HW overhead for different bus saboteur configurations are shown in Table III and compared to a Leon3 configuration without our additional HW units. The smallest tested configuration with one monitored memory region can inject two faults into this region. In other words, $1*2 = 2$ different faults can be injected during the execution of one bytecode. This small configuration adds an HW overhead of +14.5% look-up-tables and +13% slices. We used this configuration for the previously shown PIN check case study.

The biggest tested configuration can inject four different faults into three memory regions. In other words, $3 * 4 = 12$ different faults can be injected during one bytecode execution. The HW overhead increases by +85.5% look-up-tables and +68.9% slices. The additional HW overhead is only required during the design phase of the product and will not increase the chip costs of the final product in silicon.

TABLE III
OVERALL HW OVERHEAD ON THE FPGA COMPARED TO A SYNTHESIS WITHOUT OUR ENVIRONMENT.

Monitored Memory Regions	Fault Configurations per Memory Region	FPGA Look-Up-Tables	FPGA Slices
1	2	+14.5%	+13%
1	4	+28.3%	+24.7%
2	2	+28.4%	+27.1%
3	2	+46.8%	+41.7%
2	4	+64.2%	+57.7%
3	4	+85.5%	+68.9%

D. Performance Results

In this chapter, we compare the speedup of our emulation approach against ModelSim simulations. The simulations run on a quad-core 3.4 GHz Intel i5 processor with 16 GB of RAM and Win7 64-bit. The performance overhead for different test cases is shown in Table IV. Before each test run, the fault configurations are uploaded from the host PC onto the FPGA board. In our current implementation, this upload is particularly slow because it is performed over a serial cable. This start-up phase adds a linear overhead for each test run on the FPGA. By running the Java wallet case study on the FPGA, a speedup of 6,600 is obtained compared to a simulation. Furthermore, a fault campaign that was executed with 500 test runs achieves a speedup of 1,107. One different fault was injected into each fault campaign run.

TABLE IV
EXECUTION SPEEDUP ACHIEVED BY THE EMULATION APPROACH COMPARED TO A SIMULATION.

Fault Injection Scenario	Simulation Time	Emulation Time	Achieved Speedup
Manipulate PIN Check	1 hrs 50 min	$<$1 s	6,600
Fault Campaign (500 Runs)	2 hrs	6.5 s	1,107

V. CONCLUSIONS AND FUTURE WORK

This work presents an emulation environment to inject faults into different security-critical memory regions that are used by the Java virtual machine (VM). These regions are the bytecode area (BA), operand stack (OS), local variables (LVs), Java heap (JH), and Java program counter (JPC). A case study illustrates how to use the environment to illegally access the functionality of a Java wallet applet by a fault attack. Furthermore, the hardware (HW) overhead and performance results of our framework are presented. For the wallet example our environment adds an HW overhead of +14.5% look-up-tables and +13% slices. Furthermore, for the wallet example we achieve a speedup of 6,600.

The sandbox concept of the Java VM can be circumvented by fault attacks. Currently proposed fault injection frameworks lack the possibility to inject faults into the run-time data of the VM (OS, LVs, JH, JPC). Furthermore, they are programmed at a high abstraction level. With our proposed emulation framework, we circumvent these drawbacks by emulating the hardware at a low abstraction level (register transfer level) and can inject faults into different security-critical memory regions of the VM. This framework enables the evaluation and testing of security mechanisms that are programmed in the software (SW) or HW by a security engineer or test engineer during the design phase. All SW and HW components of our environment can be easily removed from the final product.

In future work, to increase the overall speed of the system, we will shift the fault injection controller from SW to HW or to a dedicated processor. Additional speed improvements could be reached by increasing the communication speed between the host computer and field-programmable gate array board.

ACKNOWLEDGMENTS

The authors would like to thank the Austrian Federal Ministry for Transport, Innovation, and Technology, which funded the CoCoon project under the FIT-IT contract FFG 830601. We would also like to thank our project partner NXP Semiconductors Austria GmbH.

REFERENCES

[1] Oracle, *Virtual Machine Specification*. Java Card Platform, Version 3.0.4, Classic Edition, 2011.

[2] Oracle, *Runtime Environment Specification*. Java Card Platform, Version 3.0.4, Classic Edition, 2011.

[3] Sun Microsystems Inc., “Java Card 2.2 Off-card Verifier,” *White Paper*, June 2002.

[4] W. Mostowski and E. Poll, “Malicious Code on Java Card Smartcards: Attacks and Countermeasures,” in *Smart Card Research and Advanced Applications*, ser. Lecture Notes in Computer Science, G. Grimaud and F.-X. Standaert, Eds. Springer Berlin / Heidelberg, 2008, vol. 5189, pp. 1–16.

[5] A. Sere, J. Iguchi-Cartigny, and J.-L. Lanet, “Evaluation of Countermeasures Against Fault Attacks on Smart Cards,” *International Journal of Security and Its Applications, Vol.5 No.2*, pp. 49–61, April 2011.

[6] G. Bouffard, B. Thampi, and J.-L. Lanet, “Vulnerability analysis on smart cards using fault tree,” in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science, F. Bitsch, J. Guiochet, and M. Kaniche, Eds. Springer Berlin Heidelberg, 2013, vol. 8153, pp. 82–93.

[7] M. Lackner, R. Berlach, J. Loinig, R. Weiss, and C. Steger, “Towards the Hardware Accelerated Defensive Virtual Machine - Type and Bound Protection,” in *Smart Card Research and Advanced Applications (CARDIS)*, ser. Lecture Notes in Computer Science, S. Mangard, Ed. Springer Berlin Heidelberg, 2013, vol. 7771, pp. 1–15.

[8] M. Lackner, R. Berlach, M. Hraschan, R. Weiss, and C. Steger, “A defensive Java Card virtual machine to thwart fault attacks by microarchitectural support,” in *International Conference on Risks and Security of Internet and Systems (CRiSIS)*, Oct 2013, pp. 1–8.

[9] J. Chorko, T. Kobylarz, and P. Nazimek, “Testing fault susceptibility of Java Cards,” *Przeglad Elektrotechniczny*, vol. R.90. NR 2, 2014.

[10] J.-B. Machemie, C. Mazin, J.-L. Lanet, and J. Cartigny, “SmartCM A Smart Card Fault Injection Simulator,” *IEEE Intl. Workshop on Information Forensics and Security*, December 2011.

[11] 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*, Nov 2004, pp. 118–121.

[12] M. Berthier, J. Bringer, H. Chabanne, T.-H. Le, L. Rivière, and V. Servant, “Idea: Embedded fault injection simulator on smartcard,” in *Engineering Secure Software and Systems*. Springer, 2014, pp. 222–229.

[13] E. Biham and A. Shamir, “Differential fault analysis of secret key cryptosystems,” in *Advances in CryptologyCRYPTO 97*. Springer, 1997, pp. 513–525.

[14] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, “The Sorcerer’s Apprentice Guide to Fault Attacks,” *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370 –382, 2006.

[15] M. Witteman, “Advances in Smartcard Security,” *Information Security Bulletin*, July 2002.

[16] G. Bouffard and J.-L. Lanet, “The Next Smart Card Nightmare,” in *Cryptography and Security: From Theory to Applications*, ser. Lecture Notes in Computer Science, D. Naccache, Ed. Springer Berlin / Heidelberg, 2012, vol. 6805, pp. 405–424.

[17] G. Bouffard, J. Iguchi-Cartigny, and J.-L. Lanet, “Combined Software and Hardware Attacks on the Java Card Control Flow,” in *Smart Card Research and Advanced Applications*, ser. Lecture Notes in Computer Science, E. Prouff, Ed. Springer Berlin Heidelberg, 2011, vol. 7079, pp. 283–296.

[18] G. Barbu, G. Duc, and P. Hoogvorst, “Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures,” in *Smart Card Research and Advanced Applications*, ser. Lecture Notes in Computer Science, E. Prouff, Ed. Springer Berlin Heidelberg, 2011, vol. 7079, pp. 297–313.

[19] J. Iguchi-Cartigny and J.-L. Lanet, “Developing a Trojan applets in a smart card,” *Journal in Computer Virology*, vol. 6, pp. 343–351, 2010.

[20] J. Dubreuil, G. Bouffard, B. N. Thampi, and J.-L. Lanet, “Mitigating Type Confusion on Java Card,” *International Journal of Secure Software Engineering (IJSSE)*, vol. 4, no. 2, pp. 19–39, 2013.

[21] J. Grinschgl, A. Krieg, C. Steger, R. Weiss, H. Bock, and J. Haid, “Efficient fault emulation using automatic pre-injection memory access analysis,” in *SOC Conference (SOCC), 2012 IEEE International*, Sept 2012, pp. 277–282.

[22] U. Legat, A. Biasizzo, and F. Novak, “Automated SEU fault emulation using partial FPGA reconfiguration,” in *Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2010 IEEE 13th International Symposium on*, April 2010, pp. 24–27.

[23] P. Ellervee, J. Raik, K. Tammemae, and R.-J. Ubar, “FPGA-based fault emulation of synchronous sequential circuits,” *Computers Digital Techniques, IET*, vol. 1, no. 2, pp. 70–76, March 2007.

[24] N. Druml, M. Menghin, D. Kroisleitner, C. Steger, R. Weiss, A. Krieg, H. Bock, and J. Haid, “Emulation-Based Fault Effect Analysis for Resource Constrained, Secure, and Dependable Systems,” in *Digital System Design (DSD), 2013 Euromicro Conference on*, Sept 2013, pp. 337–344.

Bibliography

[1] J. Teich, "Hardware/software codesign: The past, the present, and predicting the future," *Proceedings of the IEEE*, vol. 100, pp. 1411–1430, May 2012.

[2] S. Becker, W. Hasselbring, A. Paul, M. Boskovic, H. Koziolek, J. Ploski, A. Dhama, H. Lipskoch, M. Rohr, D. Winteler, S. Giesecke, R. Meyer, M. Swaminathan, J. Happe, M. Muhle, and T. Warns, "Trustworthy software systems: A discussion of basic concepts and terminology," *SIGSOFT Softw. Eng. Notes*, vol. 31, pp. 1–18, Nov. 2006.

[3] C. H. Kim and J.-J. Quisquater, "Faults, injection methods, and fault attacks," *Design Test of Computers, IEEE*, vol. 24, pp. 544–545, Nov 2007.

[4] C. P. Pfleeger, *Security in Computing*. Prentice Hall; 4th edition, October 2006.

[5] Ericsson, *Ericsson Mobility Report*. November 2013.

[6] Eurosmart, "Contactless Smart Secure Device Shipment Estimates," 2013. https://web.archive.org/web/20140424170736/http://www.eurosmart.com/publications/market-overview [Online; accessed 30-April-2014].

[7] Eurosmart, *Security IC Platform Protection Profile, Version 1.0*. European Smart Card Industry Association, 2007.

[8] M. Witteman, "Advances in Smartcard Security," *Information Security Bulletin*, July 2002.

[9] Oracle, *Virtual Machine Specification*. Java Card Platform, Version 3.0.4, Classic Edition, 2011.

[10] Oracle, *Runtime Environment Specification*. Java Card Platform, Version 3.0.4, Classic Edition, 2011.

[11] Eurosmart, "Security of Mobile Devices, Applications and Transactions," June 2012. http://www.eurosmart.com/images/doc/Papers/security_of_mobile_devices_applications_and_transactions_20_june_2012_eurosmart.pdf [Online; accessed 21-July-2014].

[12] J. Staunstrup and W. Wolf, *Hardware/Software Co-Design*. Springer US, ISBN: 1441950184, September 2010.

[13] M. Platzner and L. Thiele, *Hardware/Software Codesign - lecture notes.* Institut für Technische Informatik und Kommunikationsnetze, ETH Zürich, 2006.

[14] *CoCoon - Codesign for Countermeasures against Malicious Applications on Java Cards.* FIT-IT Project Proposal Cooperative Research Projects, Graz University of Technology, January 2012.

[15] P. Kocher, R. Lee, G. McGraw, and A. Raghunathan, "Security as a new dimension in embedded system design," in *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, (New York, NY, USA), pp. 753–760, ACM, 2004. Moderator-Ravi, Srivaths.

[16] O. Kömmerling and M. G. Kuhn, "Design principles for tamper-resistant smart-card processors," in *Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology*, WOST'99, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 1999.

[17] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Advances in Cryptology—CRYPTO'99*, pp. 388–397, Springer, 1999.

[18] S. Hamadouche and J.-L. Lanet, "Virus in a smart card: Myth or reality?," *Journal of Information Security and Applications*, vol. 18, no. 2–3, pp. 130 – 137, 2013. Smart Card and {RFID} Security.

[19] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The Sorcerer's Apprentice Guide to Fault Attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370 –382, 2006.

[20] J.-J. Quisquater and D. Samyde, "Electromagnetic analysis (ema): Measures and counter-measures for smart cards," in *Smart Card Programming and Security*, pp. 200–210, Springer, 2001.

[21] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi, "The em side—channel (s)," in *Cryptographic Hardware and Embedded Systems-CHES 2002*, pp. 29–45, Springer, 2003.

[22] S. Mangard, E. Oswald, and T. Popp, *Power analysis attacks: Revealing the secrets of smart cards.* Springer Publishing Company, Incorporated, 2010.

[23] G. Bouffard, T. Khefif, J.-L. Lanet, I. Kane, and S. Casanova Salvia, "Accessing secure information using export file fraudulence," in *Risks and Security of Internet and Systems (CRiSIS), 2013 International Conference on*, pp. 1–5, Oct 2013.

[24] G. Bouffard, J. Iguchi-Cartigny, and J.-L. Lanet, "Combined Software and Hardware Attacks on the Java Card Control Flow," in *Smart Card Research and Advanced Applications* (E. Prouff, ed.), vol. 7079 of *Lecture Notes in Computer Science*, pp. 283–296, Springer Berlin Heidelberg, 2011.

[25] E. Faugeron, "Manipulating the Frame Information with an Underflow Attack," in *Smart Card Research and Advanced Applications (CARDIS)*, to be published, 2014.

[26] M. Lackner, R. Berlach, J. Loinig, R. Weiss, and C. Steger, "Towards the Hardware Accelerated Defensive Virtual Machine - Type and Bound Protection," in *Smart Card Research and Advanced Applications (CARDIS)* (S. Mangard, ed.), vol. 7771 of *Lecture Notes in Computer Science*, pp. 1–15, Springer Berlin Heidelberg, 2013.

[27] G. Bouffard and J.-L. Lanet, "The Next Smart Card Nightmare," in *Cryptography and Security: From Theory to Applications* (D. Naccache, ed.), vol. 6805 of *Lecture Notes in Computer Science*, pp. 405–424, Springer Berlin / Heidelberg, 2012.

[28] M. Lackner, R. Berlach, W. Raschke, R. Weiss, and C. Steger, "A Defensive Virtual Machine Layer to Counteract Fault Attacks on Java Cards," in *Information Security Theory and Practice. Security of Mobile and Cyber-Physical Systems*, pp. 82–97, Springer, 2013.

[29] G. Barbu, H. Thiebeauld, and V. Guerin, "Attacks on Java Card 3.0 Combining Fault and Logical Attacks," in *Smart Card Research and Advanced Application* (D. Gollmann, J.-L. Lanet, and J. Iguchi-Cartigny, eds.), vol. 6035 of *Lecture Notes in Computer Science*, pp. 148–163, Springer Berlin Heidelberg, 2010.

[30] S. Hamadouche, G. Bouffard, J.-L. Lanet, B. Dorsemaine, B. Nouhant, A. Magloire, and A. Reygnaud, "Subverting Byte Code Linker service to characterize Java Card API," Proceedings of the 7th Conference on Network and Information Systems Security (SAR-SSI), pp. 122–128, 2012.

[31] W. Mostowski and E. Poll, "Malicious Code on Java Card Smartcards: Attacks and Countermeasures," in *Smart Card Research and Advanced Applications* (G. Grimaud and F.-X. Standaert, eds.), vol. 5189 of *Lecture Notes in Computer Science*, pp. 1–16, Springer Berlin / Heidelberg, 2008.

[32] O. Vertanen, "Java Type Confusion and Fault Attacks," in *Fault Diagnosis and Tolerance in Cryptography* (L. Breveglieri, I. Koren, D. Naccache, and J.-P. Seifert, eds.), vol. 4236 of *Lecture Notes in Computer Science*, pp. 237–251, Springer Berlin / Heidelberg, 2006.

[33] J. Iguchi-Cartigny and J.-L. Lanet, "Developing a Trojan applets in a smart card," *Journal in Computer Virology*, vol. 6, pp. 343–351, 2010.

[34] E. Vetillard and A. Ferrari, "Combined Attacks and Countermeasures," in *Smart Card Research and Advanced Application* (D. Gollmann, J.-L. Lanet, and J. Iguchi-Cartigny, eds.), vol. 6035 of *Lecture Notes in Computer Science*, pp. 133–147, Springer Berlin Heidelberg, 2010.

[35] G. Barbu, G. Duc, and P. Hoogvorst, "Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures," in *Smart Card Research and Advanced Applications* (E. Prouff, ed.), vol. 7079 of *Lecture Notes in Computer Science*, pp. 297–313, Springer Berlin Heidelberg, 2011.

[36] S. P. Skorobogatov and R. J. Anderson, "Optical fault induction attacks," in *Cryptographic Hardware and Embedded Systems-CHES 2002*, pp. 2–12, Springer, 2003.

[37] A. Krieg, J. Grinschgl, C. Steger, R. Weiss, and J. Haid, “A Side Channel Attack Countermeasure using System-On-Chip Power Profile Scrambling,” in *On-Line Testing Symposium (IOLTS), 2011 IEEE 17th International*, pp. 222 –227, July 2011.

[38] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, “Examining smart-card security under the threat of power analysis attacks,” *Computers, IEEE Transactions on*, vol. 51, no. 5, pp. 541–552, 2002.

[39] A. Sere, J. Iguchi-Cartigny, and J.-L. Lanet, “Automatic detection of fault attack and countermeasures,” in *Proceedings of the 4th Workshop on Embedded Systems Security*, WESS ’09, (New York, NY, USA), pp. 7:1–7:7, ACM, 2009.

[40] A. Sere, J. Iguchi-Cartigny, and J.-L. Lanet, “Checking the Paths to Identify Mutant Application on Embedded Systems,” in *Future Generation Information Technology* (T.-h. Kim, Y.-h. Lee, B.-H. Kang, and D. Slezak, eds.), vol. 6485 of *Lecture Notes in Computer Science*, pp. 459–468, Springer Berlin / Heidelberg, 2010.

[41] A. Sere, J. Iguchi-Cartigny, and J.-L. Lanet, “Evaluation of Countermeasures Against Fault Attacks on Smart Cards,” *International Journal of Security and Its Applications, Vol.5 No.2*, pp. 49–61, April 2011.

[42] T. Razafindralambo, G. Bouffard, B. Thampi, and J.-L. Lanet, “A Dynamic Syntax Interpretation for Java Based Smart Card to Mitigate Logical Attacks,” in *Recent Trends in Computer Networks and Distributed Systems Security*, vol. 335 of *Communications in Computer and Information Science*, pp. 185–194, Springer Berlin Heidelberg, 2012.

[43] J. Dubreuil, G. Bouffard, J.-L. Lanet, and J. Cartigny, “Type Classification against Fault Enabled Mutant in Java Based Smart Card,” in *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*, pp. 551 –556, aug. 2012.

[44] J. Dubreuil, G. Bouffard, B. N. Thampi, and J.-L. Lanet, “Mitigating Type Confusion on Java Card,” *International Journal of Secure Software Engineering (IJSSE)*, vol. 4, no. 2, pp. 19–39, 2013.

[45] G. Morana, E. Tramontana, and D. Zito, “Detecting Attacks on Java Cards by Fingerprinting Applets,” *2012 IEEE 21st International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, vol. 0, pp. 359–364, 2013.

[46] G. Bouffard, B. Thampi, and J.-L. Lanet, “Detecting laser fault injection for smart cards using security automata,” in *Security in Computing and Communications* (S. Thampi, P. Atrey, C.-I. Fan, and G. Perez, eds.), vol. 377 of *Communications in Computer and Information Science*, pp. 18–29, Springer Berlin Heidelberg, 2013.

[47] D. Karaklajic, J.-M. Schmidt, and I. Verbauwhede, “Hardware designer’s guide to fault attacks,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 21, pp. 2295–2306, Dec 2013.

[48] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni, “Low-cost software countermeasures against fault attacks: Implementation and performances trade offs,” 2012.

[49] J. van Woudenberg, M. Witteman, and F. Menarini, "Practical optical fault injection on secure microcontrollers," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on*, pp. 91–99, Sept 2011.

[50] Sun Microsystems, "Java Card System Protection Profile," tech. rep., April 2010. Version 2.6.

[51] M. Lackner, R. Berlach, M. Hraschan, R. Weiss, and C. Steger, "A defensive Java Card virtual machine to thwart fault attacks by microarchitectural support," in *International Conference on Risks and Security of Internet and Systems (CRiSIS)*, pp. 1–8, Oct 2013.

[52] M. Lackner, R. Berlach, R. Weiss, and C. Steger, "Countering Type Confusion and Buffer Overflow Attacks on Java Smart Cards by Data Type Sensitive Obfuscation," in *Proceedings of the First Workshop on Cryptography and Security in Computing Systems*, pp. 19–24, ACM, 2014.

[53] R. Berlach, M. Lackner, C. Steger, J. Loinig, and E. Haselsteiner, "Memory-efficient on-card byte code verification for java cards," in *Proceedings of the First Workshop on Cryptography and Security in Computing Systems*, CS2 '14, (New York, NY, USA), pp. 37–40, ACM, 2014.

[54] R. G. Ragel and S. Parameswaran, *Microarchitectural Support for Security and Reliabiliy - An Embedded Systems Perspective.* VDM Verlag Dr. Mueller Aktiengesellschaft Co. KG, Dudweiler Landstr. 125a, Saarbrücken, 2008.

[55] M. Lackner, R. Berlach, M. Hraschan, R. Weiss, and C. Steger, "A Fault Attack Emulation Environment to Evaluate Java Card Virtual-Machine Security," in *17th Euromicro Conference on Digital System Design (DSD)*, 2014 - in press.

[56] Intel, *MCS 51 Microcontroller Family User's Manual.* February 1994.

[57] IEEE, *Open SystemC Language Reference Manual IEEE Std 1666-2005, IEEE.*

[58] M. Lackner, M. Lang, and P. Randeu, "Architectural Design Document - 8051 SystemC Project Team," Institute for Technical Informatics, Graz University of Technology, May 2009.

[59] U. S. o. A. Department of Defense, *MVery High Speed Integrated Circuits - VHSIC.* Final Program Report 1980 - 1990, September 1990.

[60] D. . C. G. Oregano Systems, *MC8051 IP Core, Synthesizeable VHDL Microcontroller IP-Core, User Guide.* Version 1.1, June 2002.

[61] S. Oberauer, "CAP File Enhancement to Enable a Defensive Virtual Machine," *Institut for Technical Informatics, Graz University of Technology*, 2013.

[62] A. F. Rodriguez, L. H. Encinas, A. M. Munoz, and B. Alarcos, "A toolbox for dpa attacks to smart cards," in *International Joint Conference SOCO'13-CISIS'13-ICEUTE'13* (l. Herrero, B. Baruque, F. Klett, A. Abraham, V. SnÃ¡Å¡el, A. C.

Carvalho, P. G. Bringas, I. Zelinka, H. QuintiÃ¡n, and E. Corchado, eds.), vol. 239 of *Advances in Intelligent Systems and Computing*, pp. 399–408, Springer International Publishing, 2014.

[63] M. Hutter and J.-M. Schmidt, "The temperature side channel and heating fault attacks," *Smart Card Research and Advanced Application-CARDIS 2013*, 2013.

[64] K. K. Gunnam, "Hardware countermeasure against cryptographic attack," Jan. 7 2014. US Patent 8,627,131.

[65] L. Zussa, A. Dehbaoui, K. Tobich, J.-M. Dutertre, P. Maurine, L. Guillaume-Sage, J. Clediere, and A. Tria, "Efficiency of a glitch detector against electromagnetic fault injection," in *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, (3001 Leuven, Belgium, Belgium), pp. 203:1–203:6, European Design and Automation Association, 2014.

[66] N. Theissing, D. Merli, M. Smola, F. Stumpf, and G. Sigl, "Comprehensive analysis of software countermeasures against fault attacks," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pp. 404–409, March 2013.

[67] N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson, "Formal verification of a software countermeasure against instruction skip attacks," *Journal of Cryptographic Engineering*, pp. 1–12, 2014.