Doctoral Thesis

A scaleable multi-application environment for secure tokens

Author(s):
Oestreicher, Marcus

Publication Date:
2003

Permanent Link:
https://doi.org/10.3929/ethz-a-004629441

Rights / License:
In Copyright - Non-Commercial Use Permitted
A Scaleable Multi-Application Environment
For Secure Tokens

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH
(ETH Zurich)

for the degree of
Doctor of Technical Sciences

presented by
Marcus Oestreicher
Dipl. Inf., Friedrich-Alexander-Universität Erlangen-Nürnberg
born 24.09.1971
citizen of Germany

accepted on the recommendation of
Prof. Dr. Friedemann Mattern, examiner
Prof. Dr. Bernhard Plattner, co-examiner
Dr. Peter Buhler, co-examiner

2003
Abstract

Although security has always been an important aspect of computer environments, the establishment of large-scale networks has dramatically increased both the demand for and availability of secured services. In the past, secure tokens based on tamper-evident hardware have been proposed as a method of securing the authentication of their users. They not only protect security-sensitive data from illegal access, but also execute complex computations on behalf of the user. The most dominant secure token is the smartcard, which supports numerous services in different environments, such as subscriber authentication in GSM. Many other types emerged in the past. Although sharing the same basic properties and applications as smartcards, they differ in the form factors and resources they provide. Their use in an interchangeable manner requires a common software environment that hides hardware particularities, establishes a unified view, and facilitates their adaptation to the many available services.

This thesis introduces such a software environment. It is applicable to all secure tokens and fulfills the following specific requirements: (i) seamless plug-in to existing environments with enhanced migration capabilities; (ii) provide a platform that allows the efficient execution of multiple applications on secure tokens; (iii) allow the post issuance installation of applications in the field; (iv) provide convenient, flexible and resource-efficient programming models for fast application development; and especially (iv) address the large discrepancies among secure tokens in terms of memory capacities and processor performance. The system is based on the standard Java environment, but has been significantly adapted to operating-system-, runtime-, and development-oriented aspects.

This thesis proposes modifications necessary to the Java programming language, and details the programming interface design specifically tailored to the characteristics of secure tokens. The servicing runtime- and operating-system structure is presented, its applicability to different token families is discussed, and management of the most critical resources, processor and memory, is detailed. An execution model is introduced, which allows fast code execution, fine-grained API-evolution, and code verification even on resource-constrained secure tokens. Accommodating the unique memory hierarchy found on secure tokens, a model based on persistent systems is presented allowing for safe and convenient memory control by applications as well as the system. It allows the exploitation of the available memory resources on both low-end as well as high-end tokens, and seamlessly integrates transaction processing as a means for updating data reliably.

Diese Arbeit stellt eine solche universelle Softwareplattform für Secure Tokens vor: (i) sie läßt sich in existierende Umgebungen einsetzen, und erlaubt den ökonomisch sinnvollen Austausch bisheriger Systeme; (ii) sie unterstützt die effiziente und sichere Ausführung von mehreren Applikationen; (iii) sie ermöglicht das Laden und Installieren von neuen Anwendungen nach der Ausgabe im Feld; (iv) sie führt flexible Programmiermodelle ein, die eine schnelle und kostengünstige Applikationsentwicklung fördern; und (v) sie erlaubt Applikationen, die divergierenden Speicherkapazitäten und Rechenleistungen der unterschiedlichen Secure Token Familien effektiv zu nutzen.

# Contents

1 Introduction .......................................................... 11

2 Secure Token Fundamentals ............................................. 15
   2.1 Introduction ........................................................... 15
   2.2 Hardware Characteristics .......................................... 16
       2.2.1 Secure Token Families ...................................... 20
       2.2.2 Smartcards .................................................... 21
       2.2.3 Other Mobile Secure Tokens ................................. 23
       2.2.4 Stationary Secure Tokens .................................... 24
       2.2.5 Future Hardware Trends ...................................... 24
   2.3 Secure Token Application Characteristics .......................... 25
       2.3.1 Basic Secure Token Functionality ............................ 25
       2.3.2 Application Areas ............................................. 26
       2.3.3 Environmental Dependencies ................................. 27
   2.4 Secure Token Deployment ............................................ 27
       2.4.1 Token Issuing .................................................. 27
       2.4.2 Static Service Provider Model ............................... 29
       2.4.3 Static Secure Token System Software ......................... 31
       2.4.4 Dynamic Service-Provider Model ............................ 32
       2.4.5 Modern Secure Token Software Environments ................. 35
       2.4.6 Thesis Claims .................................................. 37

3 Secure Token Software Environment .................................. 39
   3.1 Introduction .......................................................... 39
   3.2 Key-Constraints .................................................... 39
   3.3 Feasibility ........................................................... 42
   3.4 Building Blocks .................................................... 43
       3.4.1 Driver Layer ................................................... 44
       3.4.2 System Control ............................................... 44
       3.4.3 Execution Engine ............................................. 45
       3.4.4 Memory Management .......................................... 46
       3.4.5 APIs .......................................................... 47
       3.4.6 System Management .......................................... 48
       3.4.7 System Composition .......................................... 49
   3.5 JavaCard .................................................................. 50
       3.5.1 Standard Java .................................................. 51
CONTENTS

6 Memory Model .......................... 109
6.1 Introduction ............................ 109
6.2 Transient and Persistent Environment .................. 110
   6.2.1 Persistent Systems .................. 110
   6.2.2 General Approaches ................ 112
   6.2.3 JavaCard: Transient Data ............ 114
   6.2.4 Transient and Persistent Environment .... 117
   6.2.5 Implications ....................... 119
6.3 Transaction Support .................... 122
   6.3.1 Goals ............................. 122
   6.3.2 Language Integration ................. 123
   6.3.3 Basic Transactional Semantics ......... 124
   6.3.4 Implementation Strategies .......... 125
   6.3.5 Transactional Object Instantiations ... 129
6.4 Conclusion ............................ 130

7 Evaluation ............................. 133

8 Results ................................ 139
  8.1 Summary ............................. 139
  8.2 Conclusion ........................... 140
  8.3 Outlook ............................. 142
Chapter 1

Introduction

The establishment of global networks, most prominently the Internet, has vastly increased the demand for secure services. Their introduction affects end users and application developers. A way to increase both security and convenience is the integration of secure (access) tokens. They represent fully-programmable and tamper-resistant devices capable of storing and protecting sensitive data from illegal access. Their processing capabilities, memory resources, and network connectivity allow for the execution of complex tasks on behalf of the users. Secure tokens can be deployed in many different scenarios including distributed computations.

The prevalent secure token as of today is the smartcard. The number of smartcards issued increased from 196 Mio. in 1996 to 725 Mio. in 2002 [155, 18]. Smartcards initially authenticated bank customers at ATMs, were later deployed in GSM networks, and are nowadays also integrated in classic computer networks. They are used as electronic wallets, support user logins in LANs, and authenticate users in the Internet. Besides the smartcard, other secure tokens have been developed. They are based on the same hardware foundation—CPU, cryptographic co-processors, memory hierarchy, and communication link—, but differ in form factor, available resources and bandwidth. Despite their common hardware and application characteristics, service development, deployment and management vary greatly among the existing secure tokens. Proprietary, single-function systems are still the norm, and incur slow deployment processes and lack of interoperability. This presents a major obstacle to the adoption of secure tokens on a global scale. As a unifying hardware platform is unlikely to emerge in the near future, a software environment must be established that is adaptable to different infrastructures and is extensible for current and upcoming services.

This thesis presents a general software environment which plugs into current secure token infrastructures, and can cope with the strong assumptions they impose. Issuers can tailor the system towards their environment by adding service-specific applications to the proposed base system. New applications can be downloaded and installed in the field. Tokens can be shared between issuers and third-party service providers by allowing them to utilize their own applications. Modern software development practices are introduced, and applications are distributed and executed in a platform-independent and secure manner. The
resulting system is able to meet existing timing requirements and emulate legacy behavior. It can cope with the unique hardware characteristics, for instance the large discrepancies across secure tokens with regard to memory capacities and processor capabilities.

The software environment encompasses experience gained from current multi-application environments for smartcards, especially JavaCard, and proposes an adoption of Java for secure tokens. As Java originates from powerful, networked and interactive devices, it is heavily modified to satisfy the described goals. In contrast to JavaCard, which restricts smartcard programming to an inconvenient subset of Java, the proposed system delivers all the strengths of Java to the full range of secure tokens.

The key contributions of the thesis are in the areas of overall system architecture, execution and memory model. A novel system architecture for the execution of portable applications on secure tokens is defined. Current smartcard architectures are based on monolithic system blocks offering both operating system and application functionality. They lack the necessary abstractions for a performant and resource-friendly execution of platform-neutral code, and remain dependent on natively implemented services. The proposed system architecture focuses on the operating- and runtime-system services required for the portable applications, and achieves the necessary efficiency. Complex applications as well as system services can be fully implemented in Java.

The thesis presents a novel execution model for secure tokens. In contrast to standard Java, the instruction set, download format, and binding model are applicable to resource-limited smartcards, and allow for a performant and safe interpretation of the secure token applications. Convenient Java features are supported on secure tokens: object-orientation, type-safety, and extensible, standardized APIs. The bytecode verification schemes having been proposed for secure tokens are efficiently supported, and application code can be securely distributed in the existing issuer environments.

A novel model for memory management on secure tokens is introduced. In contrast to file-based systems, the thesis proposes an object-oriented persistent programming and memory model, significantly simplifying the development of secure token applications. The model offers a high degree of programming convenience, and allows applications to control their scarce memory resources in a fine-grained manner. Advanced services like garbage collection and transaction can be efficiently integrated, even on extremely resource limited environments.

The concepts are extensively studied regarding their applicability to secure tokens offering increased memory resources or advanced processing capabilities. Current smartcard systems, including JavaCard, are tuned for limited 8- and 16-bit platforms, and a deployment on larger secure tokens is impractical. The proposed concepts naturally extend to a wide range of secure tokens, and offer superior portability and scalability properties. Applications can be developed which are portable across a wide range of secure tokens, including resource-limited smartcards. In case of larger tokens, the system enables the development of applications which can fully utilize the increased memory resources or advanced capabilities. In both cases, the application developers can rely on the same
underlying system concepts, behavior, and programming models.

The thesis is structured as follows. Chapter 2 categorizes secure token families, details the historical evolution, and presents current multi-application smartcard environments. Chapter 3 introduces the proposed system architecture, and discusses its building blocks. Chapter 4 presents the necessary adaptations to the Java programming language, and drafts an API framework suitable for secure tokens. Chapter 5 introduces the Java based execution model for secure tokens, which allows for fast interpretation, portable applications, and vendor- and token-specific system implementations. It demands few resources, and supports secure download-protocols and code verification. The notion and handling of memory in Java is overhauled in Chapter 6. An object-oriented, persistent and safe programming model supporting garbage collection and transactions is proposed. Chapter 7 evaluates the resulting system. Chapter 8 summarizes the thesis, draws conclusions, and presents future work in this area.
Chapter 2

Secure Token Fundamentals

2.1 Introduction

The term "token" is derived from the Anglo-Saxon word "tacen" originally meaning sign or symbol. It is heavily used in many contexts of computer science, as in case of tokens representing sets of characters in the syntactic analysis or controlling the medium-access in token-ring-nets for example. Ordinarily, they denote pieces of some form and material, and are used to provide their owners the right and access to an associated service. Their appearance, physical properties, validity and value are usually strictly specified, and they are typically produced and issued by service providers. After having been passed to their clients, they have to present them whenever they are about to require the service. Examples include chips or tickets, but also coins or passports. Of course, it is in the major interest of the service provider that any illegal token duplication by other parties is prevented. Although tokens such as notes are produced in a sophisticated manner, faked tokens of minor quality might already be sufficient to allow for illegal use [30]. Furthermore, tokens are inflexible to use as their associated value cannot be altered at a later time.

These deficiencies led to the development of secure tokens where information can be read from and written to in a secure manner. The most widely used representatives are still mag-stripe cards where information is stored in an encrypted form on a magnetic stripe. They are very cheap to produce, but still vulnerable to duplication, and provide only limited storage. More resistant are memory cards where silicon is used to store data in encrypted form and which partly protect the access by simple and fixed authentication schemes.

Better control and much more flexibility are provided by secure tokens featuring a programmable processing unit and protection against physical attacks. Only the token itself is then capable of granting access and allowing for the retrieval and update of information. In addition, the token is able to audit and log the individual transactions.
2.2 Hardware Characteristics

Due to their general applicability, a large number of different secure tokens exist, ranging from general purpose, widely available mobile tokens like smartcards to many specialized secure tokens, such as one-time password generators equipped with small displays and keyboards [133]. Additionally, powerful stationary tokens exist taking over security sensitive operations for large servers. Figure 2.1 shows some of the many secure tokens available as of today. In general, their differences can be categorized as follows:

- **Form factor**: Tokens feature varying sizes and forms where the physical proportions may be exactly defined and restricted as in case of smartcards.

- **Mobile/Stationary**: Mobile tokens are dedicated towards a specific user, whereas stationary tokens act as secured co-processors for associated host machines.

- **Purpose**: The purpose of tokens range from serving a single, fixed function to executing many different functions. Their functionality might be fixed after production or even extensible in the field.

- **Costs**: Secure tokens like smartcards are deployed in very cost-sensitive markets whereas expensive and specialized tokens like secure co-processors are mostly chosen due to their functionality.

- **Resources**: The resources offered by individual tokens vary strongly, especially including processing power and available memory.

- **Connectivity**: More or less all tokens provide a single communication link to connect to remote computers. Latency and bandwidth still differ strongly across secure tokens.

Common Characteristics

The individual properties strongly depend on each other. A token providing plenty of resources will be expensive to produce, while the resources of a mobile token are limited by its size in any case. From the large set of tokens, a standard software environment must define a non-trivial, basic subset. Mag-stripe and memory cards cannot be included as they lack sufficient programmability. Smartcards as the dominating programmable token family must be supported not to loose any practical relevance [164, 26, 3]. They can especially serve as the common denominator of secure tokens as they share the basic characteristics with all other programmable tokens, and their resources are most limited across them. A software environment can then rely on the following token characteristics:

- **Processing resources**: All tokens provide a main processor and optionally one or even more co-processors for cryptographic computations. Speed, complexity and word size may differ significantly, though.
2.2. HARDWARE CHARACTERISTICS

- Memory hierarchy: All secure tokens provide a similar memory hierarchy with different, equally visible areas. Large differences exist in the technology used, access times and sizes.

- Communication capabilities: Most tokens provide a communication link to remote hosts. Physical medium, transmission protocols, and properties like bandwidth vary significantly.

- Tamper-evident or tamper-resistant hardware: A secure token must be expected to be reasonably resistant against physical attacks.

Specialized secure tokens featuring a hardware subset can also benefit from a general software environment, most likely just by picking an according subset of the software environment. These may include tokens without communication links such as a one-time-pad (Figure 2.1). General embedded controllers in contrast are not a primary target for the software environment. On the one hand, they lack hardware protection mechanisms, on the other hand, their application areas are even more diverse and include non-token requirements such as real-time guarantees.

Processing Capabilities

The token hardware is tailored around a main CPU. On the one hand, it is assisted by several, convenient logic components, including memory-controller,
and I/O-subsystem. On the other hand, it can rely on token-specific hardware, like random number generators and co-processors. The latter speed up big-number arithmetic for the fast execution of cryptographic functions based on symmetric or asymmetric cryptography.

In case of smartcards and many other tokens, processors and memory modules are integrated into a single chip. This simplifies manufacturing, reduces costs and allows for a centralized protection scheme against physical attacks. In contrast, more powerful tokens interconnect a number of individual chips on a shared motherboard. They must protect the interconnection/communication as well, by using a physically protected case for example. This adds significant complexity to the design, increases manufacturing costs and opens the door for a number of additional security attacks.

In case of single-chip modules, the CPUs are typically based on standard processor designs, but often extended with token-specific add-ons. In case of smartcards, 8- and 16-bit processor-cores are still the regular case, due to their low transistor count. 32-bit CPUs are still mostly under development, but are also based on well-established processor designs such as the ARM architecture. In contrast, large secure tokens integrate standard 16/32-bit processors in their protected cases, ranging from Intel CISC to ARM RISC processors.

In contrast to general computing environments, the shift from 8-bit to 32-bit processors may not have to lead to the development of new software architectures despite of their significant faster execution speeds and larger address spaces. On the one hand, the most compute intensive operations are the cryptographic functions which always benefit from dedicated co-processors. On the other hand, the physical address spaces on most secure tokens are limited to a fraction of the virtual 32-bit address spaces. They are almost never beyond a few 100KB, even for large tokens, and can still easily be addressed by current 8-bit processors, having been extended by well-known segmentation techniques. Necessary transistor count and power demands remain low, ideal for current single-chip modules. Segmentation still renders the system-software implementation more difficult, and the shift to 32-bit processors is thus still likely to take place in the long term.

Memory Characteristics

All secure tokens provide a storage hierarchy based on silicon, as the current security requirements rule out the use of technologies like magnetic storage. The type and purpose of the different memory areas remain the same across all tokens, only technologies used, their characteristics and sizes differ strongly:

- Transient area: A memory area used for temporary, performance sensitive and short term calculations. Typical sizes range from 512 bytes to 4KB on smartcards, to 64KB on large mobile tokens, to a few mega-bytes on stationary tokens.

- Persistent area: This area stores data which may be updated, but has to outlive individual token sessions. Sizes range from 8 to 64KB on smart-
### 2.2. HARDWARE CHARACTERISTICS

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>RAM</th>
<th>EEPROM</th>
<th>Flash</th>
<th>ROM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number Of Writes</td>
<td>unlimited</td>
<td>10000-100000</td>
<td>100000</td>
<td>0</td>
</tr>
<tr>
<td>Read Access Times</td>
<td>70ns</td>
<td>100ns</td>
<td>100ns</td>
<td>100ns</td>
</tr>
<tr>
<td>Write Access Times</td>
<td>70ns</td>
<td>3-10ms</td>
<td>500ns</td>
<td>-</td>
</tr>
<tr>
<td>Erase Access Times</td>
<td>-</td>
<td>-</td>
<td>500-800ms</td>
<td>-</td>
</tr>
<tr>
<td>Die Size Per Bit</td>
<td>1700</td>
<td>400</td>
<td>200</td>
<td>100</td>
</tr>
<tr>
<td>Addressing Mode</td>
<td>byte</td>
<td>bit or block</td>
<td>block</td>
<td>byte</td>
</tr>
<tr>
<td>Data Retention</td>
<td>power-down</td>
<td>5 years</td>
<td>3-5 years</td>
<td>never</td>
</tr>
</tbody>
</table>

Table 2.1: Memory characteristics

cards, to several hundreds of kilo-bytes on larger mobile tokens, to a few mega-bytes on stationary tokens.

- Persistent, read-only area: This area contains all code and data known initially at production time. Sizes range from 16 to 96KB for smartcards to 128KB for other larger, mobile tokens. The memory technology typically used is ROM.

All tokens deploy either DRAM or more typically SRAM for the transient area. While DRAM is cheaper to produce, SRAM does not have to be periodically refreshed and therefore avoids a source of certain attacks [129]. Both feature fast read and write access times, but loose their contents in case of sudden power loss. Additionally, they take up a lot of die space compared to other memory technologies explaining the limited amounts found on single-chip modules.

Table 2.1 lists the characteristics of the persistent memory technologies in use today: battery-buffered RAM, EEPROM and Flash. Battery-buffered tokens feature fast access time, and can actively defend themselves against physical attacks. However, batteries increase the size, complexity and cost, especially as attacks based on the manipulation of the power supply must be prevented. Thus, most tokens favor either EEPROM or Flash as persistent memory technology. Both are restricted by the limited number of physical writes being possible during the token lifetime. Flash has a number of advantages as it provides much faster write access times, needs less die-size and consumes less power. However, Flash was originally designed for large capacities and as a hard-disk replacement. Therefore it allows only for the access of large blocks at a time making its use in resource-constrained environments difficult. Additionally, blocks have to be erased before they can be written which takes significant time and makes the use of Flash unattractive for high read/write ratios. Thus, Flash is only used so far on large and expensive stationary tokens, and most manufacturers still rely on EEPROM for single-chip modules and mobile tokens. EEPROM allows for a bit- and byte-wise addressing and an efficient use of even limited capacities as in case of smartcards. The performance impact of EEPROM writes must still be taken into account, for instance by exploiting block writes as much as possible. In this case, multiple bytes are written by the hardware at once in parallel, leading to significant performance improvements [110].
Physical Hardware Protection Mechanisms

A secure token must have reasonable hardware support for the detection of and proper countermeasures against illegal physical accesses. The token manufacturers use varying methods and usually do not publish them in detail. They try to make sure that their chips only work in well characterized operating environments. Whenever the chip is pushed outside its normal operating range by atypical power voltage or clock frequency for instance, it is supposed to detect the situation and eventually lock the token from future use. It is also expected to be resistant against various direct physical accesses to the memory through micro-probing, electronic microscopy or x-ray beams. Other attacks are based on observing various physical runtime parameters and concluding therefrom the stored data. One well known, successfully deployed example is the Differential Power Analysis (DPA) on smartcards [102]. In this case, an attacker records a number of samples of the power a smartcard consumes during the repeated execution of a chosen operation. As the power consumption correlates strongly to the operations the chip executes, one may be able to determine from the recorded patterns the instructions and especially the data they operated on, for instance cryptographic keys. Thus, modern chips typically generate random-noise during the execution.

Secure tokens and especially smartcards are not tamper resistant devices. All commercially available secure tokens are expected to be breakable if the attacker has enough knowledge, skills, time, and equipment [4]. The security goal is not to be resistant against any kind of attack, but to increase the effort and costs for the attacker sacrificing any potential gain. On the one hand, systems deploying secure tokens are then built upon different levels of security precautions, starting with hardware, system software and communication protocols, to increase the barrier for a potential attacker. On the other hand, the overall systems are designed to detect successful attacks within a reasonable time. This expects tokens to be audited during transactions, and to be locked from further use after encountering any irregularity in the field. Additionally, expiration dates are enforced for the use of tokens to limit potential token fraud.

2.2.1 Secure Token Families

Mobile tokens are the most common secure tokens. They are typically issued by service providers to their customers. They carry the tokens with them, and connect them to a token-acceptance-device at the time they require the service. The token acceptance device itself is connected to a terminal. As it handles both the interaction with the user and the communication with the token, it fully controls access to the service. Token-acceptance-devices might be integrated into terminals, as in case of mobile phones, or might be separate devices as in case of PC and smartcard-reader. They might be portable or stationary as in case of ATMs. They might act as gateways between token and remote nodes during transactions, or are driven offline.

The crucial limitation of this arrangement results from the fact that a user cannot trust the terminal by default. A user trusts his token, but he cannot
directly interact with it. Any user input must be forwarded by the terminal and thus, can be manipulated. Even when the owner uses the token within a well-known environment like a PC at home, a virus might intercept the communication and steal security sensitive data.

2.2.2 Smartcards

The oldest, most limited, but nevertheless dominating mobile secure token is the smartcard already in use since the seventies. It consists of a piece of plastic—appearance of a credit card is most common—where a single-chip module is embedded into the body with its contacts accessible at the surface. Many specifications exist standardizing the various aspects of smartcards, including mechanical specifications for material, size and thickness [75], and electrical specifications for the operating power and clock of the chip modules [77]. Other standards specify the communication details ranging from the physical transport layer (e.g. connectors and medium to use) [77], the network layer (e.g. the format of the messages to send and receive) [73], to the application layer (e.g. the semantics of individual messages) [71, 74]. The resulting key characteristics are then:

- **Availability**: Smartcards are not only useable at ATMs and point-of-sale-terminals, but can also be easily integrated in PC and network environments. Their installed number surpasses any other token by a factor of thousands.

- **Mobility/Integrability**: The size of the chip module is strictly specified to 25mm$^2$ at most. This allows for easy embedding into different form factors including mobile phones, for example.

- **Resource-restrictions**: Physical size and cost effectiveness led to significant resource limitations. Memory is only available as RAM in the range of 0.5 to 4KB, as EEPROM in the range of 8 to 64KB and as ROM in the range of 16 to 128KB; higher numbers identify high-end cards priced significantly above the economically interesting low-end cards. Processors typically used are still slow 8-bit processors due to their low transistor count, low power needs, and compact machine code.

- **I/O-restrictions**: Smartcards only provide a single serial line for the external communication. Bandwidth is limited (9600 bps are still common) and does not allow for sensibly transferring bulk data.

- **Dependency**: As smartcards lack their own power supply, clock, and sufficient resources, they strongly depend on the terminal for fulfilling complex tasks.

- **Historic burden**: The universal acceptance comes with a price. Many specifications are already outdated today, but must still be supported and hinder possible innovations.
A number of smartcard families with different physical metrics exist. Credit-card-sized smartcards can be used as a replacement for mag-stripe cards [72]. SIM cards define a smaller size to be easier to embed into small devices [72]. Both types belong to the group of contact-based cards where five contacts on the surface are used by card-readers to power, clock, and communicate with them (Figure 2.2). The upcoming contact-less smartcards and card-readers do not need physical contact, and are less vulnerable to wear and tear [79]. Their specifications differ in the distance they allow between card and reader ranging from 1mm for close-coupled cards to 10cm for remote-coupled cards [79]. As power and clock is delivered by the reader through an electro-magnetic field whose strength and stability is limited, contact-less cards provide even less resources than their contact-based counterparts.

**Smartcard Protocols**

Due to the different smartcard types and their technical evolution, a large number of protocols exist. At card reset time, initial handshake protocols setup the communication [73][76]: the card delivers its identity and capabilities to the reader, and both agree on the transmission protocol to use during the session. A number of transmission protocols exist, among them T=0 as the oldest protocol, T=1 as a more modern protocol, and T=CL for contact-less cards. They all share the same base characteristics:

- Optimized for the smartcard restrictions: All protocols are optimized for limited bandwidth and resources. This allows for efficient software implementations, even on 8-bit-based chip modules with limited hardware support.

- Server-driven, synchronous message-passing model: Due to the originally extremely restricted resources, the terminals and card-readers are in full control of the communication. They always send one request at a time, and then wait for a reply. The card can neither initiate a connection nor a request.

- Common network layer: All protocols use the same message format at the network level. A terminal sends Application Protocol Data Units
2.2. HARDWARE CHARACTERISTICS

(APDUs) to the card which replies with a return code and optionally some data. The APDU itself consists of a header describing message type and length, and a data portion.

The transparency of the transmission protocols at the network layer is hindered by the T=0 protocol, a limited byte-based protocol [73]. It defines both the physical transmission of single bytes and the transfer of APDUs by determining the sequences of and timings for the individual bytes. Without the interpretation of its header at the application level, a card cannot correctly receive and respond to messages. This limitation results from the early smartcards having had not enough memory to store the full messages in RAM. Message reception had to be completely controlled by the applications in order to process them on the fly.

The protocol layers are cleanly separated in case of T=1 and T=CL featuring synchronous block-transmission protocols [73, 80]. They allow for the transfer of blocks at a time, the chaining of blocks for the transfer of arbitrarily large messages, and promise a better utilization of the available bandwidth. The protocols are especially independent of the network level and do not need to interpret application-specific information in the APDU header. Despite their technical advantages, they could still not replace the more common T=0 protocol in the large scale.

2.2.3 Other Mobile Secure Tokens

Other mobile secure tokens have mostly been developed to overcome the limited smartcard resources and their dependency on card-readers. A number of tokens thus support more simple connections to terminals like PCs, by just using a different interface, but still relying on smartcard technology. For instance, manufacturers may just package a smartcard chip, a reader and a protocol-adapter into one device as in case of many USB-tokens where the USB-protocol is internally mapped to one of the smartcard protocols. Such tokens are then cheap to produce, but inherit all smartcard deficiencies.

These are overcome by multi-chip tokens connecting to PCMCIA- or USB-interfaces. Examples are Fortezza PCMCIA tokens featuring 32-bit processors, 64KB of RAM and 128KB of EEPROM in tamper-evident cases [128]. Although their resources surpass easily what is found in smartcards, their manufacturing costs currently make their use in the very large scale unattractive. Interfaces like PCMCIA also depend on special terminal hardware, and are still not that common as the card-reader and terminal infrastructure of smartcards.

Their low-level communication protocols are determined by the low-level interfaces used. On top of that, manufacturers use either protocols based on APDUs to be easily pluggable into existing smartcard environments or their own proprietary protocols. In any case, peer-messaging protocols between terminal and token are rare, the server-driven invocation model as in case of smartcards is common.
2.2.4 Stationary Secure Tokens

In contrast to mobile tokens, stationary tokens are not associated with one particular user, but a certain service or host. They are either used for security-sensitive operations in otherwise unsecured environments, or to add another level of security in highly sensitive environments. The deployed hardware then depends on the tasks taken over by the token:

- **Client for one particular service:** An example is the security module in ATMs. An ATM is operated in a public place, might be operated offline, and serves only a limited set of functions. Only the security-sensitive logic and data must be stored in the security module, all other computations can be executed on the ATM itself. Today, security modules are thus very often based on single-chip modules also found in smartcards.

- **General crypto-engine:** These tokens feature large resources to assist host applications in the execution of security-sensitive operations. For instance, the PCI- or SCSI-based crypto-controllers nForce feature 32-bit main processors, several mega-bytes of storage, and crypto engines being able to compute cryptographic operations close to or even faster than the main processor of the associated host, and can thus drastically reduce server workloads [107].

- **Service provider:** The secure token is connected to a network and fulfills a complex service on its own. Examples are key-management systems requiring only minimum assistance by remote nodes and system administrators.

The last category resembles general computers operated in secured environments and are thus out of scope of the described software architecture. The other groups still share the common characteristics and can be targeted by a general software environment as well.

2.2.5 Future Hardware Trends

Any base software environment has to watch out for the advances in the area of the target hardware. In case of tokens, these firstly include the development of their communication capabilities and infrastructure, and secondly the progress of their chip technology. Current secure token infrastructures have been evolved over the last 20 years and are dominated by smartcards as the primary tokens. Any attempt to establish a new communication protocol or interface type demands a completely new infrastructure or extending the current one, both implying huge costs. The only trend in this direction are upcoming contactless/-based cards which might replace pure contact-based cards over time. Full duplex protocols, for instance, having been proposed already years ago, could not gain acceptance due to the necessary changes to the infrastructure [129].

Although smartcard resources increased with the advances in chip technology, the growth advanced more slowly than in other computing environments. In 1994, for instance, 32-bit RISC processors were expected to be the common
case at the end of the nineties, but smartcards still feature 8-bit processors today [121]. This is mostly due to the manufacturers having to set different priorities. As the smartcard market is a highly cost-sensitive market, they first try to decrease production costs, by concentrating heavily on the improvement of yield rates, for example. They squeeze out the maximum out of their current, reliable production technologies, and avoid huge investments into more advanced production lines as long as possible. Although they must still react to the increasing resource demands, they still prefer the extension of the available memory than the inclusion of a sophisticated CPU core.

This dilemma will last as long as the margins for the smartcard production are low, and manufacturers are forced to delay huge investments into new technologies. This might change as soon as the smartcard has found its way into all areas of computing as "the" secure token and is used in more and more applications. For this to happen, many manufacturers still wait for the "killer application" which would necessitate tokens in more or less any kind of security-sensitive computation. Thus, a new software environment must obey the smartcard limitations and provide a base platform for the deployment and exploration of new secure token applications on current hardware. The success of a software environment depending on a sudden and rapid increase of token resources is unlikely.

2.3 Secure Token Application Characteristics

2.3.1 Basic Secure Token Functionality

The hardware characteristics lead naturally to the following abstract functions executable by secure tokens:

- Safe data storage: The data can be specific to the token holder (e.g. private keys), to the token issuer, or specific to third parties. This especially includes shared secrets between token holder and service provider (e.g. symmetric keys).

- Access-control: The token controls the access of data by requesting parties. Access might be unrestricted, read-only, or dependent on proper authentication or other environmental dependencies.

- Authentication: A token can play a central role in many authentication services. First, it can confirm the identity of its owner, by verifying a PIN for example. Second, it can authenticate the terminal and other third parties by secure challenge/response protocols. And finally, it can delegate the access rights of the user to remote services, by transparently handling the necessary authentication protocols.

- Cryptographic functions: secure tokens may provide a wide range of cryptographic functions, among them generic services such as signing, verification, random-number generation, cryptographic hashing, and encryption/decryption.
Dependent on the token, applications may rely on different aspects of the base functionality. For instance, host applications especially depend on generic encryption/decryption services of stationary tokens, while terminal applications rely more heavily on the authentication services of mobile tokens.

2.3.2 Application Areas

Mobile tokens are especially used in areas where different applications rely on different token capabilities. Most areas are still exclusively served by smartcards, while other mobile tokens are still used in a limited scale:

- Campus-card: Smartcards are used to grant access to many services provided in a closed environment like a university campus. The concrete behavior of the token might then differ on each service to be accessed [42].

- Health-Care: The card protects security-sensitive data in a large and open environment. Access might be partly or fully granted dependent on the access rights of the requiring party [55].

- Public-Key-Infrastructure: Tokens supporting public-key cryptography may be used to generate keys, store and verify certificates, and implement cryptographic services like signing [136].

- Payment-schemes: Many existing payment schemes rely on smartcards. They may either operate online or offline, and be credit/debit- or electronic-cash-based [15]. The cards are tested to implement the specified protocols correctly, to sufficiently log the transactions, and to support an early detection of fraud [62].

- Network-access: In GSM networks, SIM cards execute multiple functions, from storing address books to supporting the wireless protocol which even expects the cards obeying strict timing guarantees [41]. In computer networks, tokens are used to implement single-sign-on schemes where a token is responsible for verifying the user identity and authenticating him to all remote services available in a computer network [42].

- Public-transport: The use ranges from electronic tickets (i.e. tickets are payed in advance), to fee-collecting schemes where passengers are tracked and charged at a later time. Requirements often include fast transaction times and contact-less operation [164].

Short transaction times are always desirable as they are a key to the necessary convenience for the user. This includes the performance of the back-end and terminal, but also of the token itself. Service downtimes must be avoided as much as possible, and the token should be useable everywhere. Additionally, users tend to expect the security to come for free, and refuse to invest money or time, but expect the issuer to handle all tasks.
2.3.3 Environmental Dependencies

The tasks a token has to fulfill for an application also depend on the environment it is used in. An electronic wallet used offline will look different from one designed for online usage, for example. Thus, not only the applications, but also the environments applications are used in, expect maximum flexibility by secure tokens:

- Physical environment: tokens used in better controlled environments like intranets may utilize more simple implementations in contrast to tokens used in large, open networks.

- Legacy environment: the token is used in an environment where many legacy specifications and applications must be supported.

- Trusted/Non-trusted environment: when the terminal cannot be trusted, additional countermeasures must be implemented. PIN verification and challengeresponse protocols may, for instance, allow for the mutual authentication of card, user and terminal.

- Online/Offline: classic offline environments are ATMs in public places. All transactions are logged on both card and security module inside the ATM. The results will be transmitted eventually to a back-end system and be checked against potential fraud.

- Open/Closed: in an open system, a token allows for the access of multiple services by different service providers, while a closed system only foresees a single service provider.

- User/Issuer token: a user might buy and manage a token on his own. Within PKIs, this might even allow for accessing remote services. However, most services are still based on shared secrets expecting a single token issuer in charge.

2.4 Secure Token Deployment

2.4.1 Token Issuing

The token issuer builds up and manages the token infrastructure, stays in control of the token during its life cycle, and guards the access to its associated services. As the token stores security-sensitive information of multiple parties (e.g., customer and issuer), its life cycle is well-controlled to prevent both information leakage and fraud. Especially, the life cycle of smartcards is subdivided into several stages featuring different security measurements. Other tokens are handled similarly, although their small issued number simplifies their secure management significantly.

In a first stage, the smartcard chip module, including processor and the various memories, are fabricated by the chip manufacturer (Figure 2.3 on the next page). An operating system is placed in ROM at that time as well as an initial
key used to prevent any illegal access already at this stage. The chip module is then delivered to a smartcard manufacturer who produces the final card, i.e. he embeds and glues it in plastic, and connects it to the contacts or antennas necessary for communication. The smartcard manufacturer also personalizes the token, i.e. he initializes it with the customer and issuer information necessary for its use in the field. Typically, he knows the initial key to authenticate to the card, downloads the personalization data, and replaces the initial key with a number of issuer keys which are valid in the field and control access to different system services in a fine-granular manner. After personalization, the card is rolled out and shipped to the customer, and is ready for accessing associated services in the field.

The service delivery strongly differs dependent on type, application and environment, but typically, a customer requests the service by inserting the token into a terminal under control of the issuer. The terminal then executes an application controlling the communication between customer, token and the system delivering the service (which might be the terminal itself in case of offline applications). The service may require mutual authentication of the individual parties, typically based on challenge/response protocols in case of authenticating terminal and back-end system, and on PINs in case of authenticating the user to the token. If customer authentication fails, the token is immediately blocked from any further use. If it succeeds, the service is fulfilled, and the transaction logged. Transaction logs can be used to recover the token state, and especially to compare the recorded state with what the card provides at the time of the service access. In case of an online service, this check is done immediately. In the offline case, this check is delayed until the transactions are transferred from the terminals to the back-end systems. In either case, as soon as fraud is detected, the token is locked from any further use, eventually with a significant delay in case of offline services. Finally, issuers foresee a maximum lifetime for their
issued tokens after which a replacement is necessary. This allows for updating
the keys used in the field, the token hardware and software.

2.4.2 Static Service Provider Model

Single Service-Provider Model

As of today, most smartcard issuers also provide the associated service. Service
provision and token management are very often tightly interconnected. This
is especially true in the very large scale, i.e. in banking environments and
mobile phone networks. The issuer fully controls the smartcards, their life cycle
and management, as well as the terminal applications and back-end systems
responsible for delivering the service. As tokens can only be used with their
particular issuer, customers typically do not accept initial fees for them. Thus,
issuers are often forced to fund the issued smartcards, and are not able to
share costs with the customer. A single service provider and issuer is also most
common in case of other mobile secure tokens. For instance, the US Government
issued more than 10000 PCMCIA tokens to its employees to secure the access to
internal network services. [138]. This number is low compared to the number of
issued smartcards, and the given scenario is much more limited, homogeneous
and easier to control than typical smartcard environments.

The single issuer and service provider model provides a number of strengths.
Only a few parties play a part in the token life cycle, and after personalization,
the issuer is in full control of the token state and all of its activities. Transactions
always succeed under his supervision, and can be easily logged. The deployed
security models are rather simple and well-known, especially their fraud risks
have extensively been studied. For instance, many systems rely on user PIN
authentication and shared secrets for the protected service access. As the issuer
is in full charge of his infrastructure, he is free to adopt the security precautions,
and can in general tailor the whole infrastructure and tokens to his needs. For
instance, he can move as much functionality and knowledge from the token to
the back-end systems, and keep the per-token complexity as low as possible.
This indeed helps to reduce the per-token costs to a minimum. In the banking
environment, chip module prices for smartcards less than $1 are common.

However, the huge quantities of issued smartcards still add up to immense
costs, especially as issuers periodically replace the issued tokens and are unable
to share costs with the customers. Providing new functionality and upgrading
their services might also lead to the need for reissuing all of their smartcards.
This again implies huge costs, but also significant delays. Developing and testing
a new smartcard system is already likely to require several production processes,
a new roll out in the large scale might easily require many months. Even worse,
issuers are in many cases tied to a single smartcard manufacturers and his
proprietary token system. Smartcard development and deployment then depend
on a single source resulting in issuing delays, less innovations and increased costs.

Smartcard manufacturers also suffer from the current situation. Deployment
delays and the need for one smartcard per service reduce customer satisfaction,
and make the use of smartcards for many service providers less attractive. The
growth of the smartcard market therefore remains limited, the future of secure
tokens as the ultimate thin clients taking over all security-related computations
remains uncertain. Chip manufacturers are similarly affected. As they also
suffer from the price pressure passed, they are forced to produce cheap hardware,
and their possible margins remain low. However, as long as issuers cannot shift
the costs for the tokens and their management, the described situation is unlikely
to change.

**Pre-issuance Multiple Service Provider Model**

This model allows for issuers to reduce costs with only a limited adoption of
the single issuer and service provider model (Figure 2.4). The issuer remains in
full control of the token and its infrastructure, but does not only provide access
to his own service, but also to services of other providers. The tokens are sim¬
larly manufactured and personalized as before, but with both the information
for the issuer environment and all the other services. In the field, the differ¬
ten services are again accessed at an issuer terminal. Typically, the terminal
application connects to the issuer back-end which acts as a gateway between
terminal and back-end systems, and forwards the requests on demand to the
responsible providers. The terminal might also directly connect to the service
provider as long as the issuer remains fully aware of all token transactions and
in full charge of his tokens. In any case, the issuer updates his records from the
data he receives from the terminal or service providers, and is able to block the
card in case of potential fraud.

The token issuer is able to share costs without having to carry out any
major changes to his tokens and infrastructure. Service providers can benefit
from an existing and reliable infrastructure, and deploy tokens for securing their service. However, the model requires strong legal assurances and trust relationships, as the issuer gateway has access to provider-related and security-sensitive data. It is thus practically limited to the cooperation of few and large institutions, excluding many interested small service providers. Additionally, other weaknesses of the current single issuer and service provider model remain. With only a few services accessible by a single token, the customer satisfaction and convenience remain low. The dependency on smartcard manufacturers, their proprietary systems and development processes persist.

Clearly, a stricter separation between issuer and service provider, and a direct interaction between token and service provider is required to offer the necessary flexibility. However, this assumes a means of partitioning and sharing the control of the token which has not been possible with the proprietary, still dominating token systems having been developed in the past.

2.4.3 Static Secure Token System Software

The resources of early smartcards were so limited that already the deployment for a single issuer and service was a technical challenge. The whole system software and mostly even the hardware was specifically designed and implemented for the targeted service. Naturally, the use of assembly languages and proprietary vendor tools were mandatory. Worse, a design methodology evolved where the whole system was designed upon the underlying communication protocol. Protocol-design languages were used not only to develop and verify the protocols, but also to assist in structuring and implementing all other aspects of the system [129]. As a result, application-specific logic was interleaved with system functionality, leaving systems without structure and clear boundaries. Consequently, the whole systems had to be re-engineered for every new upcoming service.

The design methodologies were still in use when large issuers like banks started to require more flexibility from smartcards to integrate them into their environments. For instance, smartcards had to support both off-line payment methods like Geldkarte and debit payment schemes like EC card. The smartcard then had to understand several, different application protocols. This led to the introduction of multi-function cards which offered more resources than their ancestors, but which still suffered from the same software design: the logic of the system and the different applications were interleaved, the smartcard represented itself as a single monolithic block, and all the different components had to be developed by a single party. The term “multi-function” already indicates the basic approach: the system was just seen as a handler of multiple protocols (or functions), not as an engine executing different applications.

Of course, the smartcard manufacturers were interested in a flexible base platform being able to support new services without the constant need for re-engineering. Additionally, the platform should support simultaneously multiple, independent services, i.e. it should be partitionable for several parties. With time, the smartcard manufacturers thus introduced generic protocols flexible enough to support different services, the most prominent standardized in the
ISO7816/4 specification. ISO7816/4 specifies APDU-based protocols providing access to a hierarchical file system, user authentication via PIN, remote authentication based on challenge/response protocols as well as a number of cryptographic functions [71]. Flexibility is supposed to be achieved by storing customer, issuer and service provider data in separated trees of the file system, security by the many supported cryptographic functions. However, many of the current smartcard applications require much more flexibility. For instance, typical electronic wallets cannot be implemented on top of the supported interfaces. Additionally, the file systems lack sufficient naming and structuring schemes to allow for a reasonable partitioning in a standardized, inter-operable manner. Finally, the ISO7816/4 specifications lack clarity and rigidity, and incompatibilities across different systems are the common case.

Many proposals in the past tried to address the limitations of the established smartcard systems. They have been motivated by either their inflexible communication protocols or their fixed and insufficient file-oriented abstractions and interfaces. Proposals include more abstract network protocols, peer-instead of client/server-based communication for example. However, they incur practically impossible changes to the infrastructures. Other systems replace antiquated file systems by database models [120, 21]. Although they offer a more flexible handling of data and improved access control, they suffer from the same limitation as file-based cards, fixed interfaces. They are thus only sufficient for the current static issuing models, and do not provide the necessary flexibility for upcoming issuing models promising to remove the current limitations.

2.4.4 Dynamic Service-Provider Model

High issuing costs, vendor-locked deployment and customer dissatisfaction led issuers to think about new service provision and issuing models, most prominent the dynamic service-provider model. Issuers can reduce costs best when sharing them with as many service providers as possible. Attracting them requires a flexible scheme imposing minimum obligations on issuers and service providers. Service providers must be able to receive a high degree of control over token management, and deploy functionality on the tokens rather freely. Additionally, they must be able to setup tokens for their service which are already in the field, and allow for their use only for a limited amount of time. Thus, the dynamic service provider model strongly extends the previous models, and involves the following processes and parties (Figure 2.5 on page 34):

- The card issuer produces and rolls out the cards as usual, the cards get personalized with customer and issuer related data and are ready to access issuer network and services (1, 2, 3).
- The card issuer and a service provider agree upon sharing token and infrastructure (4).
- The service provider is free to partly implement and take over formerly pure issuer tasks: he may install new functionality on the cards in the field, personalize them for his service, track the transactions taking place...
at his terminals and with his back-end, check for fraud, and eventually lock the cards from the further use of his service.

- Independent application providers are foreseen to implement the necessary on-token functionality for service providers (5). As service providers may cooperate with many issuers, they cannot be expected to rely on their individual smartcard manufacturers for application development.

- The token owner applies for the new service (6): the token gets extended in the field with the necessary functionality (7), personalized, and allows for the instantaneous access of the provider network (8).

- The issuer remains in control of the token: the service provider may only control, remove or extend the domain on the token assigned to him, but he cannot interfere with or overrule management decisions of the issuer. He remains responsible for replacements or periodic updates of the token, and is especially able to cancel the rights of the service provider on the token.

The model has not yet been deployed in large scale, as a switch from the previously closed issuer environments to truly open environments requires to develop new or shift current practices in many ways. On the one hand, the success of this model depends on many social, legal and economic factors. Issuers have to adopt their business models. Legal assurances between the different parties must be negotiated, and terms like ownership must be clarified for example. Already the question which and how logos may be placed on future smartcards may significantly hinder a deployment of the described scenario.

On the other hand, the existing proprietary smartcard systems do not provide the necessary technical capabilities. They must make room for standardized, inter-operable, vendor-independent smartcard systems allowing for dynamic extensions, secure execution and management by multiple parties. The platform must reflect the targeted independence and interchangeability of the different participants, and for instance support different underlying chip architectures, provide a standard programming environment to attract application providers, and a standardized, generalized security architecture for the cooperation of service providers and issuers.

The dynamic service provider model has been proposed so far only in the smartcard environment where its business requirements have been met first. However, the model can be easily extended to include all other mobile tokens. As today the different issuer networks grow together into the Internet, services may be accessible by different mobile tokens from different environments. Especially for large and expensive mobile tokens, this might be the first step towards removing their lack of applications, reducing their deployment and development costs, and increasing their market share. As their resources should easily allow for deploying smartcard applications, they might be able to partly replace smartcards in use today, and then be able to market better their additional capabilities, the execution of resource-intensive applications. They might then be able to leave their current niche market.
The dynamic service provider model promises advantages for all involved participants:

- Token issuers: share costs for tokens and infrastructure with many service providers, especially without complex and costly setup processes, and achieve independence from particular smartcard and chip manufacturers.

- Service providers: benefit from a cheap, flexible and feasible way to incorporate secure tokens in their infrastructure. They can expect almost full control over their domains on the rented tokens, and even sharing their space with other parties independently from the issuer, is a natural option.

- Customers: the number of secure tokens they own is reduced, and services relying on secure tokens can be offered instantaneously.

- Chip manufacturers: token sharing requires more resources increasing the demand for more advanced chip modules promising higher margins.

- Application developers: as an independent party could not have been established so far in the secure token market. Their independence from manufacturers, providers and issuers promises more innovation in the area of new applications.

Only the smartcard manufacturers, who so far develop the operating systems and applications, might loose their impact in the market. Clearly, if a standardized software environment is deployed on every smartcard, systems and
applications can be developed by independent parties. This practically reduces
the task of the smartcard manufacturers to embed and personalize the chip.
However, his expertise of the field should allow for him to become an important
application and solution provider, and at the end, benefit from the growth of
the total market.

2.4.5 Modern Secure Token Software Environments

JavaCard, MULTOS, WindowsCard

The topic of this thesis is the design and discussion of the basic concepts of an
on-token software environment suitable for dynamic issuance models. The work
especially proposes an adoption of the Java environment for secure tokens, and
then closely relates to JavaCard. JavaCard is a Java-based standard for smart-
cards, and belongs together with MULTOS and WindowsCard to the group of
smartcard systems having been developed in recent years to achieve the neces¬
sary shift in smartcard deployment [100, 140, 103]. All three systems emerged
at the same time, and share the basic system architecture. Their development
succeeded under strong timing pressure, and especially mostly independent of
each other, due to the involved business parties and market interests.

MULTOS, the eldest of the three systems, has been developed by MASCAO,
a consortium led by MasterCard and Mondex, and a first version has been
released in 1996. Development has mostly succeeded in a closed manner, and
just recently, more detailed information about internals of the system have been
made available publicly. In contrast, JavaCard has been evolved in a rather
open process led by Sun Microsystems and the JavaCard forum, an industry
consortium headed by smartcard manufacturers like Gemplus, Schlumberger,
and IBM. Proposals were firstly privately discussed and evaluated by the group
members, but then publicly presented and reviewed, before they were finally
included in the JavaCard standard. The process thus made it possible for a
number of parties like the IBM Zurich Research Laboratory to significantly
influence the final standards. In contrast, WindowsCard, firstly announced in
1998, was solely developed by Microsoft, and despite a lot of early attention,
could not deliver the initial promises. Especially, Microsoft was forced to open
its development process, and released the system for an adoption of ETSI as an
open standard.

All systems describe smartcard platforms allowing for the execution of multi¬
ple applications in a platform-independent manner. Applications can be down-
loaded, installed, executed, blocked and removed in the field, their code and
data are protected by the system, and they can rely on standardized, hardware-
and platform-independent system interfaces. All systems thus try to achieve ap-
lication portability, provide independence from hardware manufacturers, offer
standardized programming environments to attract application developers, and
enable the partitioning of smartcards for issuers and service providers. Still, all
three systems, including especially the JavaCard, either had to undergo major
revisions until or still do not fully offer all the necessary features for the de-
scribed deployment model. For instance, the support for several, independent
service providers is still not fully supported by WindowsCard.

Figure 2.6 shows the basic system architecture as foreseen by the three competing systems in 1998. Applications are stored in processor-independent bytecode, and are executed by an interpreter. Bytecode characteristics and interpreter are expected to guarantee the strict separation of application code and data in their sandboxes, and to allow for the controlled access of system services by the invocations of standardized APIs. The system services are fulfilled inside a large, monolithic block, provide access to the underlying hardware, communication services, memory management and cryptographic services.

In MULTOS, the monolithic block presents a classic smartcard-oriented operating system. Application code and data are stored in a file system, and protected by similar methods as described in ISO7816/4. MULTOS is thus a natural evolution of the previous smartcard system generations, but then also suffers from similar weaknesses. For instance, relying on an inflexible file system with limited naming capabilities made the introduction of multiple service providers and/or code sharing difficult, and the chosen abstractions seem inapplicable for larger mobile tokens.

WindowsCard resembles MULTOS in its basic design. A monolithic system block offers well-known smartcard functionality, i.e. file system and associated cryptographic services, and optionally an interpreter capable of executing applications programmed in Basic. No means of managing multiple service providers has been foreseen at the beginning, as a small and efficient implementation was the initial goal. This led to a number of inflexible, highly smartcard-centric approaches being less applicable to larger tokens.

Initially, JavaCard assumed an existing, underlying operating systems to deliver the necessary base services for the Java environment built on top, similar as in case of standard Java. With time however, it became clear that an adoption of Java to smartcards requires an overhaul of most concepts found in
standard Java (Section 3.5.2). Modern and efficient JavaCard implementations then follow the system structure proposed in this thesis, for example.

2.4.6 Thesis Claims

The thesis proposes the necessary Java adaptations to provide a resource-friendly, scaleable, and secure execution platform for secure token applications. Results are based on experiences gathered during the engagement of the author in the JavaCard specification process since 1997, and the implementation of many Java-based prototypes and commercial JavaCard systems. On the one hand, the discussions give an insight into the reasoning behind current JavaCard standards, as the thesis presents proposals having been adopted and rejected during the specification process. On the other hand, superior alternatives are presented. They improve the JavaCard regarding resource friendliness, security, performance and development convenience. Additionally, they allow for applications exploiting all secure tokens, including large mobile tokens and future smartcards. The thesis details the Java based software environment in the following manner:

It presents a new architecture for the execution platform of portable secure token applications:

The thesis claims that it is possible to build a smartcard system which does not depend on native and smartcard-centric services tightly integrated into the base system. Instead, portable applications can be executed by an interpreter to implement all token services, including platform-dependent and time-critical applications. Even on resource constrained platforms, multiple applications can be executed strictly isolated from the runtime-environment and base system in a secure manner. The achieved execution performance and resource-friendliness is similar as in case of proprietary smarcard systems. The thesis claims that the designs of proprietary systems—based on the separation of operating system, native services, and runtime-environment—lead to many inefficient system layers. The thesis presents a design tightly integrating operating-system and run-time services to optimally support the portable applications. This allows for an efficient integration of Java as application and system programming language. Additionally, the design is fully applicable to the hardware and application characteristics of larger secure tokens.

It details the necessary adaptations of the Java language and APIs:

The thesis claims that at the syntactic level, the Java programming language can be strictly subsetted for the development of secure token applications, and language extensions can be avoided. At the semantic level, standard Java features have to be removed accordingly. Additionally, fundamental properties of Java have to be redefined, for instance the notion of object lifetime. In case of APIs, the thesis categorizes the necessary base services for token applications. Platform-independent APIs are differentiated from token-specific and legacy-constraint extensions. The thesis claims that applications do not have
to rely on static system call interfaces, but can be embedded in convenient API frameworks, even on resource-constrained smartcards. They have to differ significantly from their standard Java counterparts, and can only scale across tokens if the underlying execution model offers the necessary binary compatibility.

It presents a new Java based execution model for secure tokens:

The thesis shows that the standard Java model of distributing, downloading, and executing code can not be deployed on secure tokens. It relies on concepts inapplicable for secure tokens, places high resource demands, and lacks the necessary execution performance. Instruction set, download format, code linking and verification must be newly designed. The thesis claims that an instruction set can be derived from the standard Java bytecode which offers good code density, performant interpretation, and low resource needs for execution. The thesis designs two new download formats. The first one reaches minimum code sizes and fast downloads. The second format demands slightly more resources, but increases machine independence and binary compatibility. Applications can be optimally downloaded on 8-/16-/32-bit based tokens providing vendor-specific system implementations and token-specific API extensions. Convenient standard Java features such as type-safety, object-orientation, application sandboxes are preserved. The code verification schemes having been proposed for secure tokens are applicable, and the proposed bytecode can be securely distributed in the existing issuer environments.

It presents a new Java based memory model for secure tokens:

The thesis claims that file-systems are insufficient abstractions for the storage and retrieval of persistent information by secure token applications. The thesis presents a novel object-oriented and persistent programming model which is optimized for the transient and persistent memory technologies available on secure tokens. Objects can be placed - independent of type - into the transient or persistent store, and seamlessly manipulated within language expressions. The model is simple to implement, and allows for an efficient integration of advanced features like garbage collection and transactions. The thesis claims that the model on the one hand offers the programming convenience of large-scale persistent systems. On the other hand, applications can make efficient use of the limited memory resources on smartcards, and exploit the increased memory resources on larger tokens.

The thesis especially claims that the proposed concepts lead to a platform which is not tight to 8-/16-bit based hardware and/or smartcards as in case of JavaCard. The execution platform is scaleable as it is deployable on small and large secure tokens. Applications are scaleable as they can target a wide range of secure tokens. They can rely on the same system concepts, behavior, and programming interfaces independent of the memory resources and processing capabilities provided by the tokens.
Chapter 3

Secure Token Software Environment

3.1 Introduction

The system environment must meet the business requirements imposed by the dynamic issuance model discussed in the previous chapter. Their technical equivalents can be easily deduced. The existing price pressure demands the software environments to be optimized for the limited resources of current hardware. The system must be sufficiently cheap to make the replacement of previous generations attractive. It has to be deployed in existing markets and infrastructures, and must offer excellent support for legacy applications and protocols. It must provide a convenient development platform to attract independent application providers. The system must be scaleable to be deployed on larger tokens. It must be trustworthy for the involved business parties, and meet the necessary security guarantees.

This chapter presents an overview of the technical requirements, briefly studies the overall feasibility of a scalable software environment, and presents the structure of the system as it was successfully designed and implemented at the IBM Research Laboratory Zurich. After that, Java and JavaCard are introduced, and the consequences for the further thesis presented.

3.2 Key-Constraints

Security

A multi-application environment on secure tokens must be designed to offer both safety as well as security. Safety includes the requirement that the programming environment should exclude typical programming bugs as much as possible. Security demands for the guarded installation and execution of code. Applications may only access their own data, must be strictly separated from each other in sandboxes, and may access system resources only after a controlled migration into the system space. Thus, a software environment must feature sufficient security properties for its virtual processor, based on a software-MMU
or type-safe code for example.

Additionally, an issuer relies on cryptographic measures to achieve the full control of his tokens in the field (Section 2.4.1), and will insist on guaranteeing the authenticity, integrity and/or privacy of downloaded applications by the use of encryption and/or signing techniques for the application code and communication protocols. In case of the dynamic issuing model, he must offer similar capabilities for the service providers for their application management.

The management information exported by tokens will likely be a superset of the information issuers gather from current single-function tokens (Section 2.4.1). Of course, the issuer has to track additional information like number, type and origin of installed applications. However, he will still be interested to base the management on his current, reliable infrastructure.

Legacy Support

Any new token system must integrate into its current infrastructure as seamlessly as possible, especially in case of smartcards (Section 2.2.2). However, the more legacy support is included in the base system, the less innovations are possible, and more and more time for the design are devoted to achieve compliance to existing standards. For operating systems on large computers, estimates exist that 90 percent of the time from the design stage to the implementation have to be spent with fulfilling existing standards [122]. The situation is even worse in case of smartcards, as so many legacy standards exist, and the limited resources give less room for complex adaptation schemes.

Smartcard systems have to live with limitations in the transmission protocols making transparent communications by all applications impossible [129] and hindering the deployment of abstract programming interfaces (Section 4.3.2). Similarly, many legacy applications rely on incompatible file system abstractions which do not reflect the new environments accordingly and require expensive emulations. Finally, the tokens must integrate into the token issuer environment determining secure-messaging protocols, authentication schemes, management interfaces, and personalization schemes to use.

Development Experience

The necessary developer audience can only be attracted by a well-known development process. On the one hand, this requires access to the full range of appropriate development tools including compilers, integrated development environments, but also source-level debuggers and simulation environments. On the other hand, the programming convenience is determined by the deployed programming language and APIs. Standard programming languages providing sufficient safety features must be favored as token-specific languages require additional education and hinder a large-scale adoption.

In contrast, on-token APIs must significantly differ from the APIs provided on standard computing environments due to the different nature of secure tokens. They might share certain design patterns to reduce the initial complexity and learning effort for the developer, but this must not lead to inefficiencies in
3.2. KEY-CONSTRAINTS

the base token system.

Resource Utilization

Any software environment for secure tokens must define a basic target profile, the minimum amount of resources expected for a reasonable implementation. The concentration on a target hardware with extremely limited resources will immediately lead to increased hardware dependencies and lack of functionality. For example, WindowsCard fully focuses on low-end smartcards, and defines an abstract machine based on a limited virtual 8-bit processor, immediately raising scalability concerns [60].

Closely related to the proper compromise between functionality and resource savings are two other items: performance and portability. In contrast to previous smartcard software implemented in assembler, applications must now be executed by an interpreter whose performance impact must be acceptable even on low-end tokens. The virtual processor may not only allow for an efficient use of physical processors on current smartcards, but should also be portable among and make efficient use of the many different, partly larger and also future tokens.

Scaleability

Scaleability concerns have only been a minor issue for new smartcard architectures. With the many different tokens and the advances in chip technology however, a new software environment must be able to adapt to future improvements and provide a well-defined degree of scaleability. Scaleability expresses the possible adaptation of system software and applications to changes of the underlying hardware and its environment. Environmental changes include the introduction of advanced transmission mechanisms and communication protocols, and are likely to succeed over several generations of the system software (Section 2.2.5).

The hardware differences across secure tokens require the system not only providing application portability, but achieving more advanced goals. In the best case, an application is only developed once, can be executed on all different tokens, can live with the minimum resources on low-end tokens, and can at the same time fully exploit larger processing and memory resources on future and high-end tokens. In practice, specialized applications for either exploiting extremely limited or high-end tokens might still be necessary, but their implementation should still be able to rely on the same principles regarding programming models, APIs and system mechanisms as the range of applications deployable on all tokens.

Scaleability is also tightly connected with the possibilities for system adaptations and evolutions in the future. These should take place as much as possible without incurring changes of application sources or their executable contents. Most implementations adhere to strictly specified standards and have to pass expensive certification processes. Later changes just lead to costly repetitions of the certifications. Thus, binary compatibility is especially important for the
evolution of secure tokens.

3.3 Feasibility

A general software environment can be derived from current multi-application smartcard architectures. On the one hand, most tokens conform to their basic hardware characteristics (Section 2.2), and thus already find a good coverage in the smartcard environment. On the other hand, smartcard applications build the largest group of token applications, and are executed on the platform with the largest resource constraints. They can then immediately serve as the group of applications executable on all tokens, and fill the current lack of standardized applications on larger tokens. No add-ons seem to be needed when many standardized smartcard applications such as certificate databases or electronic wallets are moved on larger, mobile or stationary, tokens. Thus, smartcards can serve as the common denominator on the hardware and on the software side.

Smartcard applications could be supported on larger tokens by porting a smartcard system directly to their hardware. The token can then act immediately as a smartcard replacement as long as it can be treated as such within its environment. For instance, a JavaCard-port on the IBM 4758 token kept the communication model based on APDUs [69], but had to tunnel them through its platform-specific transmission protocol. Such ports requiring explicit support by terminals are only feasible as long as the token environment can be adopted to obey the standards set by smartcards. Legacy smartcard limitations are still spread to the new environment, and new applications might not benefit from advanced hardware.

In general computing environments, increasing hardware resources often led to major shifts of the software environments, and quickly to incompatibilities between different generations. A few observations vote for a single, scaleable software environment where a large, non-trivial subset of functionality can already be implemented on low-end tokens:

- fixed hardware: Token-hardware is fixed after production, and is mostly tailored towards a number of predetermined cryptographic services. This allows for many simplifications as the system must not support dynamic system extensions, and can concentrate on providing lightweight access to the functionality predetermined by its hardware.

- application patterns: The system cannot, but also need not allow for the implementation of processing-intensive applications as they tend to show the behavior of protocol-handlers, glue-logic or scripts. They typically read data from the memory areas and I/O-channel, distribute it among the cryptographic engines, and write it back to memory or I/O-channel. Most "scripts" can thus be expected to be executable on all tokens. Additionally, many may also be able to fully utilize larger tokens, by benefiting from large address-spaces for example.

- external dependency: The more services can be moved to remote nodes, the more resources are saved on the tokens themselves. In the best case,
3.4 Building Blocks

The general software environment must thus leverage the existing smartcard systems, replace or adopt insufficient abstractions and especially avoid their basic design mistakes. A monolithic approach of as in case of WindowsCard and MULTOS must make place for an operating system and run-time environment broken up in components which leave space for extensions on larger platforms. The system should not incorporate former smartcard-centric design methodologies (Section 2.4.3), and avoid the dependency on a file system for system activities, for example. In contrast to JavaCard, the system must be complete and incorporate operating system and management services. The resulting global system view helps in reducing the number of necessary abstractions and reusing them in different areas.

Figure 3.1 shows in detail the composition of an abstract and general multi-application secure token system. The system is split into individual layers, starting with token hardware up to applications making use of the available system interfaces. The most complex block of functionality is provided by the system engine and management instance. The other layers include hardware drivers, APIs, and application sandboxes which may be grouped into a domain managed by a third-party instance, the service-provider representative.

functionality may be implemented on the token or the remote node transparently for the application (i.e. without impacting the application implementation itself). Examples include placement of code verifiers, code linkers and/or of code optimizers and compilers.
The software-environment especially does not rely on an underlying operating system, but delivers the services of both operating system and run-time environment (Section 3.4.7). These services are grouped in the system engine, and concentrate on fulfilling application and management requests. Especially, they do not support functionality accessible by remote nodes such as file systems and databases in proprietary smartcard systems for example.

3.4.1 Driver Layer

The driver layer provides the first step to abstract from the concrete token hardware. The necessary drivers and the minimum required functionality are predetermined by the underlying hardware features. A first driver must allow for the access of persistent memory as it is not transparently supported by the hardware in case of EEPROM or Flash [112]. A second driver must include implementations of cryptographic functions, most of the time by making use of an underlying cryptographic co-processor, but stand-alone implementations for algorithms such as DES are also common. Finally, the driver layer must cover the physical I/O interface. With single, dedicated I/O lines and point-to-point connections to terminals, the transmission protocols of secure token are likely to be already fully implemented at this level.

Smartcard Caveats

In case of smartcards, the highly platform-dependent drivers are typically implemented in assembler, either for timing, space and/or performance reasons. Especially, the implementation of the various cryptographic algorithms and the protocol stack require significant resources, optimizations and skills. In contrast to general computing environments, the driver layer may not offer fixed interfaces across different smartcards or co-processors, but is likely to define a layer where the implementation can switch to another programming language such as C. At least, the individual drivers incur few system dependencies, implement only mechanisms and can be fully controlled and exploited through simple interfaces by the system engine. As long as internal interfaces need not be exported by the system implementor, implementations on larger tokens can succeed in the same way.

3.4.2 System Control

The system engine carries out the services requested by API invocations. Many functions can already be fulfilled in the application domain itself, others can be mapped directly and thus simply on the underlying driver layer. Many operations still have to be checked for permission by the requesting application. In case of secure tokens, these impact cryptographic operations, due to export-control reasons for example, and memory reservation. The system control must forward these security-sensitive requests to the management instance for a final decisions as they depend on the issuer-policy. Other security-sensitive operations as for instance PIN-handling can be transparently handled directly be-
3.4. BUILDING BLOCKS

tween applications and management instance and transparently for the system engine.

For management decisions, the system control must record information about downloaded code and installed applications, and keep track of the currently executing application and its context. It eventually exports this information to the management instance which can rely on system hooks to carry out its decisions, for instance installation, personalization, blocking or deletion of an application. These functions can also partly be exported to applications in general, but can always only be fulfilled after verification by the management instance.

The system control partly undertakes operations transparently for applications and management instance. These include operations necessary at reset time as for instance aborting pending transactions, but especially control of the external communication. The system control is responsible for routing messages from terminal to applications and vice versa, and correctly handling connection setups and shutdowns. In most environments, only a single connection between terminal and application is supported at a time, but support for multiple communication paths is more a question of available resources than of system complexity (Section 4.2.26.2.5).

Smartcard Caveats

Most smartcard systems integrate the management instance tightly into the operating system and heavily depend on file systems for checking access rights. With their incompatibilities and insufficient abstractions (Section 2.4.3), they have to be moved to the application space, and should only be supported in legacy scenarios. In the best case, they can be implemented by two parts: the externally visible storage-functionality is implemented by a regular application on top of the system, the protocol for access control and application management is handled in the management instance.

The JavaCard lacks a file system, i.e., it lacks a standardized API covering a system-global file system. It also lacks management-specific APIs, and—as a specification of a run-time environment—it fully leaves out the protocol between operating system, run-time environment and management instance. Both incompatibilities as well as security weaknesses across different implementation are then likely.

3.4.3 Execution Engine

Execution control is simplified by the virtual processor the system has to provide for the portable execution of applications. Nevertheless, the provision of a sufficient virtual processor remains the most difficult task. With the resource constraints of smartcards, an efficient execution by an interpreter, deployment on 8-bit-based hardware, and a space-efficient instruction set are basic requirements (Section 5.4). Additionally, the virtual processor may not only support the common set of applications across secure tokens, but allow for dedicated applications targeting legacy behavior or special capabilities. These include the access of wider address ranges and data types on 16-/32-bit-based hardware
(Section 5.2.2), but also the use of system and API extensions (Section 4.3.1). The latter requires flexible download schemes as any extension must not affect the portability of the common set of applications only relying on base APIs. Nevertheless, download size and installation complexity must still be as low as possible in case of smartcards. Finally, the code format and instruction set must provide certain security guarantees. At the minimum, this includes the detection and prevention of insecure operations at run-time. Especially in case of secure tokens, an analysis of code before its execution is preferable, and favors the deployment of type-safe code as in case of Java and JavaCard.

Smartcard Caveats

All current smartcard systems lack sufficient virtual processor abstractions. Both WindowsCard and MULTOS do not only rely on software-MMU models, but especially on limited instruction sets and code formats. WindowsCard features an overly restricted 8-bit processor, MULTOS an inconvenient, programming-unfriendly block-based instruction set [60, 99]. Both lack sufficient information in executables for later inspection and analysis, and only foresee fixed system interfaces with limited means for system evolution. In comparison, the JavaCard can benefit from the basic properties of its ancestor, but fails to adopt them sufficiently, especially in the area of its binding and verification model (Section 5.5.7).

3.4.4 Memory Management

In case of memory management, it is difficult to benefit from solutions deployed in ancestor systems as the memory hierarchy, ratios of transient and persistent space and application patterns on secure tokens are unique: (i) applications are mostly interested in manipulating their persistent data surviving individual sessions with the secure token, but (ii) have to intervene them with manipulations of short-lived data, either for security or performance reasons (Section 6). The handling of both transient and persistent data must thus be conveniently integrated into the programming model, i.e. it should build on the same principles and succeed transparently as possible for the programmer (Section 6.2.4). With tight resources on many secure tokens, the system must allow for fine-grained and predictable memory-control, but without hurting security. This implies a safe programming model not being based on manual memory control for example, but relying on garbage collection even on extremely restricted tokens (Section 6.2.5). On larger tokens, wider address spaces should not only benefit the system by being able to install more applications for example, but also the applications themselves by transparently storing and manipulating more data.

The requirements are increased by the need for updates being not only persistent, but also reliable. The integration of transactions must succeed in a similar way, resource-efficient, convenient and secure (Section 6.3).
Smartcard Caveats

Most smartcard systems including Windows and MULTOS are based on conventional programming models for handling memory and storage. Applications are started at the beginning of a session, have access to multiple segments in the memory hierarchy, and mostly rely on the externally visible file system for the long-term-storage of data. Applications either suffer from static memory-reservations or from the need for manual memory control. Resulting programming models remain unsafe as in many larger, conventional computing environments.

JavaCard lacks a file system, and thus foresees the transparent usage of the persistent space by allowing for object placement and manipulations in it. The underlying philosophy is that smartcards, their applications and data are supposed to live and be active forever, even between different card sessions [146]. The importance of transient data is neglected, the resulting model suffers from the same limitations as the other smartcard systems such as static memory-reservation for example [117].

3.4.5 APIs

The APIs establish the access points where applications acquire all system services. Simple services may already be fulfilled in a library embedded in the application sandbox, e.g. helper functions such as copying memory areas. Security-sensitive services lead to crossing the sandbox boundary, and are fulfilled under the control of the system engine. The base services to be published by APIs are fixed due to the closed hardware concept and its intended purpose, and include:

- **Communication services** allow for the pickup of and response to terminal requests independent of the underlying protocol, and inquiries about the current protocol and state.

- **Crypto services** allow for applying cryptographic algorithms on application data and might be checked by the system for the confirmation to export-control restrictions.

- **Interapplication communication** must provide a mechanism for publishing services, looking up services, and passing control and data for the remote execution.

- **Management services** allow inquiries about installed applications and manipulations of their states.

- **Memory services** allow for the exploitation of transient and persistent memory and transactions

Smartcard Caveats

With the common hardware and application characteristics, the smartcard architectures already address the necessary base API areas, but suffer from many
platform dependencies. Their APIs must be split in platform-independent parts applicable to all secure tokens and platform-dependent parts covering either legacy services or advanced capabilities. Where functionality is so far only available in a platform-dependent manner on all secure tokens, abstract layers must be introduced, providing an uniform view on all secure tokens.

As additional layers usually incur costs, their inclusion must always be carefully considered. For instance, the APDU-based smartcard communication is inflexible and platform dependent, and should thus be abstracted properly. However, many smartcard specifications still expect the applications to have close control over their APDU traffic. Thus, a new abstract communication layer can only be offered in parallel to the legacy APDU-based interface, and only few applications can benefit from the new model and its portability bonus. As the communication model depends on the environment and legacy infrastructure, it can only evolve with time hindering a fast and successful deployment of a general architecture.

The strict partitioning of the services might partly succeed with minimum costs. In the smartcard environment, for instance, applications are named by AIDs where the format and usage of AIDs is surely platform dependent [70]. However, the typical operations in the context of application names are inquiries and manipulations of application states which can be fulfilled without the knowledge of the concrete data representation of application names. Access to this information can then be offered in the platform dependent part of the API.

All three methods, the partitioning of APIs, the introduction of abstract layers and the offering of advanced, platform-specific APIs, require flexible system invocation and modularization facilities. Both WindowsCard and MULTOS rely on a fixed system call interface, and are oriented towards a monolithic, closed and static base system. Although JavaCard tries to foresee the flexible packaging mechanisms of its ancestor, its restrictions on executable contents prevent the design of APIs being sufficient for both low-end as well as high-end tokens (Section 5).

3.4.6 System Management

Global Management Instance

Management services are offered by the management instance which represents the issuer on the token and is responsible for enforcing its policy after personalization and issuance. On the token, its execution is triggered by the system engine during the verification of an API invocation and/or explicitly on an application request. Dependent on the issuer policy, the request may be fulfilled or lead to a management action such as blocking an application from further use. Other than manipulating the life cycle of an application, the management instance might export additional functionality to applications like terminal-authentication or secure-messaging facilities. The card management instance can also be contacted and explicitly controlled by an external terminal application or remote issuer back-end. Possible actions are inquiries about installed
applications, manipulations of application states, removal of applications, or secure downloads of new applications.

**Provider Security Domains**

A simple management instance may only represent and support the token administration by the token issuer. Advanced specifications, targeting dynamic issuing models (Section 2.4.4), foresee the grouping of applications managed and controlled by different service providers [116, 161]. They allow a service provider to install a representative on a token and to participate in the management of its domains. The global management instance partitions the token into the different provider domains and controls the actions undertaken in each domain. Whenever a management decision is due for a specific domain, it will ask the responsible provider representative for his decision. Only if both the service provider and the issuer representative agree, the task is fulfilled. The tasks may include downloading, installing, blocking and removing applications, and may be triggered by installed applications or by external, remote requests of service providers or issuer.

**Smartcard Caveats**

In case of smartcards, the management instance does only depend minor on the hardware and resources of the card, but strongly on the issuer and environment it is used in. For example, management instances according to the Visa Open Platform specification are typical in the banking sector while implementations of the ETSI SIM-Toolkit specification are required in GSM networks [41]. Although larger tokens provide the resources to host multiple management instances at the same time, a single management instance representing the token issuer (e.g. bank, company or government) must be in full charge of the token. With similar functions found in the various standards, one could, however, think of having a single instance implementing multiple management protocols and supporting different environments at the same time. The different parties in each environment can then be integrated similarly to the service providers described before.

In any case, most applications only require few management-specific functions on the token which should be possible to realize within all current management standards. This subset could then be offered on every secure token in a platform-independent way. The additional, standard-specific functions should again be strictly isolated. This leaves room for the system implementor to equip the token with the management instance required in his target environment.

**3.4.7 System Composition**

The software environment does not prescribe strictly separated system components. For the applications, the interface between system engine, drivers and management instance is completely hidden behind available APIs. Thus, the system implementor is absolutely free to decide upon the system design and
structure. In contrast to large computing environments, a replacement of system components adhering to strictly specified interfaces incurs too much overhead on a smartcard to be implemented efficiently. The system implementor will of course use internal interfaces to structure his system for the ease of the implementation and for the replacement and extension of internal components. For instance, he is likely to foresee the incorporation of management instances adhering to different standards in order to serve different markets with the same system core. He is still free in deciding by which means components are replaced (e.g. at the level of source code, object modules or libraries). Without such freedom, an implementation of any software environment for smartcards under the current resource and portability constraints is economically just infeasible.

The more resources a token offers, the more choices remain for structuring the system. A system might be based on an already existing operating system as it is the case for many embedded Java implementations of today [118]. However, mapping of API services on base operating system services might either not be possible or at least very expensive in terms of performance and memory footprint. This behavior has for instance been shown by a number of Java-Card implementations where JavaCard run-time environments were implemented on top of existing smartcard operating-systems [116]. The barrier between operating system and run-time environment especially hinders the implementation of services which are difficult to assign (e.g. the transaction service). These are typically implemented twice, in the operating system for the native applications, and in the run-time environment for the portable applications.

The performance and memory impact just leads to the ongoing development of applications for the underlying operating system and slows down the acceptance of the new, generic software token environment. Successful and efficient implementations are thus likely to follow the described system composition, and directly target the hardware of secure tokens. Their closed and fixed hardware concept make such an approach much more feasible than in large computing environments. They are characterized by extensible hardware concepts and many available devices hindering a successful introduction of a new operating system.

With increasing resources in the future, system-internal interfaces may be standardized to allow for the inter-operable replacement of system-inherent services, similar to the JNI specification in Java for example [141]. However, the impact will still differ strongly from standard computing environments. In case of secure tokens, the system software is always placed in ROM at production time, is fully composed by the system implementor, and systems extensions in the field are hardly acceptable due to security implications.

3.5 J avaCard

The software environment has been described so far independent from a concrete programming language and environment. Clearly, as soon as system-internals are discussed, they have to be determined, and then heavily impact the design and immediately out-rule certain choices. The thesis examines the adoption of the Java language and environment for the detailed reasons (Section 2.4.6).
Nevertheless, the proposed system structure applies to secure token systems integrating different programming languages and run-time environments.

3.5.1 Standard Java

The standard Java environment consists of the definition of the Java programming language [81], the Java virtual machine specification [96], and a set of standardized run-time libraries, and allows for the portable, safe, and secure execution of applications. First, applications can be executed on all systems providing the necessary run-time services. Second, the system cannot be crashed by any kinds of programming errors and finally, the system allows a secure execution by rejecting malformed code and the verification of access rights. The overall acceptance of Java in various environments of today is due to its reasonable mix of preexisting features and its explicit orientation towards networked environments.

The Java source language is an object-oriented language based on well-known constructs. It is statically typed, supports single implementation inheritance, and multiple, abstract views on classes via interfaces. The grouping of classes into different namespaces—packages—allows for component-oriented programming where package- internals are hidden by fine-grained access specifications of classes and their members. The language demands on the one hand a number of expensive features such as garbage collection. On the other hand, a number of features such as built-in primitive types promise an efficient use of underlying machines.

Source code is compiled into intermediate, strongly-typed bytecode. It allows for a simple and efficient interpretation [54], but its compilation to machine code at run-time is the regular case on larger platforms. Bytecode is distributed in class-files where individual classes may be loaded in a lazy fashion at the time they are needed for a first time during the execution. At run-time, applications have access to a large number of standardized APIs optimized for workstation and server scenarios [19].

Secure execution of applications is based on three concepts: verification, sand-boxing and access-control. The verifier checks the correctness of the class-file, and enforces the type constraints expressed in the source language. Most of the necessary safety checks can already succeed at this stage, only a limited set of operations must be further verified at run-time. The execution of the application then takes place within a sandbox. The application can only leave its sandbox at service access points where the system is able to check the access rights of the caller before it fulfills the request.

Despite proper basic concepts, the standard Java environment was never meant to be deployed on extremely resource-limited devices. Its instruction-set and resource-intensive execution-, binding- and verification-model do not only hinder resource-efficient implementations, but also do not fully address the needs for secure tokens, no support for persistence or transactional computations for example [112]. Java-based secure token environments may provide the convenient development experience of Java, but cannot comply to the “write once, run anywhere” strategy once touted. This is true for other embedded architec-
tures as well where the shift from one fixed, general standard Java environment towards a number of architecture-specific Java environments took place much later. This resulted in a whole family of related, but partly very different specifications such as for embedded Java, personal Java, or connection-limited devices [145, 147].

3.5.2 Specification Process

The first Java-based smartcard system was proposed by Schlumberger and is described in [61]. It extended a proprietary operating system by a very small interpreter for the execution of a subset of the Java instruction-set. The system used a conversion step to transform the standard class-files of the application into a smaller format suitable for the download and execution on the card. The run-time system only offered very limited functionality and was tailored completely towards the underlying, proprietary operating system.

With the prototype showing the feasibility, Sun Microsystems announced an official JavaCard specification process and the JavaCard forum was established by a number of smartcard manufacturers like Schlumberger, GemPlus, and IBM to assist in the standardization process [82]. The first JavaCard specification defined the language subset having to be supported by a JavaCard, but only outlined the remaining system architecture. The code conversion step and virtual machine (VM) specification were left unspecified, and the APIs and run-time services were only marginally described. Applications on the card should be embedded in a framework hiding the communication details, and should have access to a number of outlined services like cryptographic functions [142].

The JavaCard specification 2.0, released in 1997, finally defined for the first time the standard system classes including cryptographic, file system and run-time-specific services, and discussed in more detail the communication model [144, 143]. However, the specification process was mostly engaged with the discussion of the basic memory model. A number of different proposals were discussed at that time and finally a limited compromise accepted [117].

The conversion step, code format, and execution were still unspecified and supposed to be the main open issue for the JavaCard 2.1 specification. However, a number of weaknesses in the previous specification led to a major redesign of the APIs. The cryptographic services were heavily reorganized, and the file system was removed completely. The agreement on the executable content was still the biggest obstacle where the final compromise was more a result of time pressure than of general acceptance.

The weaknesses in the final 2.1 specifications resulted from the tight and fixed schedule in the JavaCard specification process. Proposals were partly not backed up by full implementations, but were still accepted due to the lack of alternatives. Discovered weaknesses in draft or previous specifications just led to ad-hoc fixes in the following specifications, the concepts itself were not sufficiently reconsidered. The lack of few, clear concepts and various proposals enforced by the different vendors hurt especially in a resource-constrained environment like the smartcard. The original hardware target of smartcards with 512 bytes of RAM and 16KB of ROM had then to be given up for smartcards
3.5.3 Main JavaCard Properties

The concentration on the language and APIs right from the beginning stresses the main motivations of JavaCard, to establish a comfortable development environment for smartcards. JavaCard applications, named applets, are developed within the standard Java environment and can be initially tested within a simulation environment based on standard Java [20]. JavaCard thus limits itself to a strict subset of the language, with language features like floating point arithmetic being removed. Any proprietary language extension is avoided.

For the final tests and deployment, the standard class-files have to be converted to a format suitable for the card. The converter takes all the classes of an applet and groups them together into one JavaCard executable, a “converted applet” or cap-file. The static, package-centric distribution and execution model replaces the dynamic and class-centric model of standard Java, and allows for a number of size optimizations [116]. Although verification of cap-files is still possible, the JavaCard environment does not foresee the verification on the card itself, but only off-card before the code is downloaded [158].

At installation time, the cap-file is downloaded into the persistent card memory, an applet instance created, and finally registered at the system under a specified name. The registered applet instance is notified whenever an external communication request is received. The system then forwards all incoming APDUs by method invocations on to the applet instance, waits for its completion and sends the response back to the external terminal application. The session with the applet ends as soon as the terminal or user selects another application or finishes the communication with the card.

The JavaCard architecture specifies a number of APIs, but contains only the base infrastructure for application management like binding, naming and invoking applications. The advanced and security-sensitive operations are only outlined and left for accompanying management specifications specifically tailored towards the different card environments [41, 161]. For instance, the OpenPlatform specification envisioned by Visa offers the additional services necessary for a secure token environment. It offers a set of on-card management functions and general secure communication protocols for use by applications, and especially defines the contract for service provider representatives to participate in the management of their application domain. Additionally, it specifies the protocols and functionality necessary for remote management. These include the mutual authentication between card and remote systems, code signatures, key management and the secure download, installation, removal and blocking of applets and their associated code.

3.6 Conclusion

The chapter shortly introduced the main technical constraints as they result from the business requirements of a dynamic issuing model. Legacy support, security, resource efficiency and development convenience have already been partly
CHAPTER 3. SECURE TOKEN SOFTWARE ENVIRONMENT

Targeted by the recent smartcard systems JavaCard, MULTOS and WindowsCard, whereas scaleability has only been a minor issue for them. A general software environment must address all secure tokens, and avoid their smartcard-centric design methods and abstractions.

The chapter presents the building blocks of a new software environment for multiple applications on secure tokens, and outlines the necessary operating-system- and run-time-specific services. Their applicability to smartcards are discussed, and existing solutions in the smartcard environments are analyzed whether they can be sufficiently abstracted for all secure tokens. Especially, their communication- and management-related services cannot be overhauled without uneconomic changes to their infrastructures. Their implementations still depend heavily on the design of two less legacy-restricted, but nevertheless critical system aspects: (i) virtual processor and execution engine, (ii) memory management and its programming model. On the one hand, hardware characteristics, resource constraints and application patterns require a scaleable execution model allowing for the flexible download and resource-efficient interpretation of the portable application code. On the other hand, the limited memory models in use today must be replaced by a system integrating both persistent and transient stores in a convenient, resource-efficient and safe programming model.

The chapter introduces its choice of Java as the programming language for secure token applications. Although it represents the de-facto-standard for safe and secure programming in the Internet, its adoption by JavaCard has immediately raised concerns regarding sufficient resource effectiveness. A comparison to other proposed languages shows its general applicability, and lead to a proper adaptation for secure tokens. After that, the APIs required for secure tokens are designed. Programming language and APIs deliver the final requirements for the underlying memory management and execution model.
Chapter 4

Application View

4.1 Introduction

Many languages have been proposed for programming secure token applications in the past. Nevertheless, supporting a variety of languages at the same time has never been a major goal for smartcard systems. On the one hand, the scope of applications on secure tokens is limited. On the other hand, a single language simplifies the necessary tool chain, and allows for optimizations across the whole system.

The Java environment also places a strong emphasis on a single language, but it has been criticized for its huge resource demands. Even the introduction of sophisticated APIs and run-time environments could not reduce the resource footprint of today’s Java applications on standard platforms. Resource-friendliness was thus not the main reason for its adoption in case of JavaCard, but its security model and development convenience. Systems lacking a convenient programming language such as MULTOS have not been accepted in a large scale so far.

The choice of Java in case of JavaCard was initially justified with remarks of its origin from the embedded space. Nevertheless, the resources on platforms initially targeted by Java easily surpass most secure tokens of today, and a sufficient adaptation of Java is then still uncertain. An adaptation itself is not a major problem as other languages, even C, also have to be altered in case of secure tokens [87]. The difficulty is to identify the language elements which are expensive and and can be removed or modified without affecting the original spirit of the language. Modifications must especially exclude language extensions, as they require additional programmer education.

APIs must be designed from scratch, and thus always require a learning process. Their categories only depend on the hardware and application characteristics of secure tokens, and can be discussed language-independent. Their programming convenience and resource demands depend on language expressiveness and feature-set. These result in design guidelines reaching the original goals: a general software environment must be able to specify base APIs applicable to all secure tokens and be extensible for add-ons on specialized and larger tokens.
Both programming language as well as APIs are especially interesting regarding their demands on the underlying software environment. Removing all resource-intensive elements from Java keeps them low, but results in the loss of the Java spirit. Keeping many of them might make a deployment on resource-restricted smartcards impossible.

4.2 Programming Language

4.2.1 Feature Sets

The languages, having been proposed for multi-application smartcards, include Forth, C, Basic, and Java, and already differ strongly in their base characteristics. Properties like readability might be a matter of personal taste, but other features can be discussed more precisely, including especially type and module concept, safety and security features, implementation complexity, and machine dependency.

Type Model

With a look at smartcard application patterns, a script language originating from a general computing environment such as Perl or Python seems a valid option. However, they typically include platform-specific functions and a large number of built-in types, difficult to subset and incurring significant costs. Additionally, they are dynamically typed, making a static analysis before an execution difficult. Statically typed languages are better suited and for instance allow for a better error-detection during the compilation step.

Statically typed, procedural languages like C concentrate on the application algorithms and lack the proper hiding of data representations [88]. With the need for strict API barriers, the shift to an object-based or object-oriented (OO) language seems natural, due to their strong encapsulation of code and data. Additionally, polymorphism and inheritance allow for the fine-grained sharing and export of code and data, and offer a lot of programming convenience. They still hinder a simple and efficient static analysis of programs and incur run-time costs (e.g. increased resource usage and run-time complexity).

OO-languages like Smalltalk, featuring uniform object representations, are not affordable in any case. Although they can be built very compact, the handling of simple arithmetic and block data types is cumbersome and inefficient [92]. Especially, these types must be efficiently supported as token applications tend to move raw data and to manipulate simple state during a session.

Advanced language features, such as reflection, may not only be omitted due to resource constraints. In many cases, they are used to allow for queries and actions by applications having to adapt dynamically to their environment, a need mostly not known in secure token and other embedded environments. It could already be argued whether the support for multiple inheritance or multiple views on a class via interfaces as in Java is really needed in closed environments like secure tokens.
Module Concept

In any case, the language must offer a strong packaging and module concept. Applications must be strictly separated from each other, and be implementable independent of different system-data representations and API implementations. Especially, with APIs and applications distributed globally in case of smart-cards, programmers must be able to register and use unique, strictly separated namespaces for them. In case of languages such as C, this can only be achieved by additional support in the development tools and system. First, they lack the proper information hiding of implementation details and second, offer only a global namespace under which libraries can export access points.

Module-oriented languages like Limbo offer module-local namespaces and a service-oriented programming concept. Modules are separated from each other, can specify variables and functions to be externally accessible, and may import other namespaces and their exported items. The model is simple to understand, and allows for good source and binary compatibility schemes, but only allows for a coarse-grained sharing of modules.

OO-languages with strong namespace concepts such as C++ or Java allow for the fine-grained encapsulation and access specification of class and instance members. This involves more complex binding and binary compatibility schemes, but allows for framework-oriented programming models. Frameworks promise to save resources, and simplify the development, as they embed applications and implement large parts of their functionality [43].

Implementation Complexity

The complexity of run-time support, compilers and tools also vary between the proposed languages. In case of C, the minimum dependency on the run-time environment was one of the key factors for its huge acceptance. The language could be supported easily on more or less every computing platform [88]. If the language does not make strong environmental assumptions, the target architecture must specify all system aspects such as execution and memory model. This might provide maximum freedom for system architects, but might limit the programming convenience. In contrast to the language, all tools, APIs, and services might be inconvenient for the programmer.

The target assumptions of languages such as Limbo, Forth or Java begin with an abstract virtual processor, and range to expensive services like garbage collectors or run-time type-information [36, 96]. Although implementations can be arbitrarily complex, the tight connection between language and system promises consistency, potentially allowing for a model simpler to understand and easier to analyze.

Machine Dependency

Every programming language targeting portability tries to abstract as much as possible from the executing machine including its word size, instruction-set, byte-order and memory architecture. Languages like C make only minimal premises and leave it to the compiler to enforce the machine-specific parameters
at compilation time. For instance, it maps the sizes of primitive types and addresses to the preferred values of the underlying processor, ranging from 8 to 32 bit in case of secure tokens. Indeed, this allows for the adaptation of C to a wide range of processors, but can easily lead to subtle underflow and overflow errors on certain targets. Thus, environments like Java and Inferno fully define the abstract processor, including primitive types and word sizes. They then can easily be adhered to by the programmer.

Languages also differ in the memory architecture they expect from the underlying system. Languages like C expect linear, byte addressable memory areas, typically code, stack and heap space [88]. Forth typically features a stack architecture with separate call and expression stacks, but also fixed data segments. Pointer size and address space width is then a major portability concern, reduced in case of platforms like Java or Inferno. They manipulate objects and leave the mapping to the memory representation up to the VM. The addressable memory is then not determined as rigid as in C-based systems [36, 96].

Safety and Security Properties

The type model strongly influences the safety and security model a language offers. A weakly typed language like C, allowing unsafe casts, can only be securely executed within an environment based on a hardware or software MMU. Removal of unsafe casts is the first step towards a safe programming language guaranteeing the referential and type integrity on each object access. Additionally, the support of manual memory control must be removed, due to potentially dangling references. Finally, the compiler must check for all other type-related, safety-critical operations such as assignments or sub-class relationships. At runtime, the system must still ensure the proper code security. The model itself may be independent of the language, either verification- or MMU-based.

The explicit support for error-handling can also reduce typical programming mistakes. Modern exception-handling facilities allow for an efficient signaling of errors to clients without the need for the programmer to manually propagate an error through the whole call chain. This is especially useful in case of secure tokens, as the dependency on external nodes and the lack of interaction with the user do not allow for the token-local handling of errors, but require most of the time the immediate forwarding of the error to the terminal.

4.2.2 Case Study: JavaCard

Java’s safety features and well established programmer community make it a first candidate for an adoption by secure tokens. However, Java supports expensive features tailored towards the needs of network applications which have to adopt dynamically to the environment they are downloaded into. Java also foresees powerful 32-bit CPUs for its deployment. At least, the basic Java philosophy promises a possible subset, even suitable for smartcards: Java itself is just a mixture of many features found in previous languages [54], leaving a good chance for fine-grained removals.

The dynamic download of classes and resulting open-package concept has to
be removed in any case. The lazy or application-initiated download of classes is only possible with a constant network connection, incurs security risks and hinders a static analysis of applications. The JavaCard replaces the class-centric with a package-centric model, where class packages act as the base distribution entity. This incurs a number of significant changes, as code binding, initialization, and versioning is now specified at the level of class packages (Section 5). The JavaCard then anticipated a similar development having taken place in standard Java. The introduction of jar-files as distribution entities and package-versioning mechanisms show the need for separating classes, representing the implementation of types, from packages, acting as distribution units. Despite the heavy architectural changes, the average programmer may only be slightly affected, as language and API abstractions may not be concerned.

The removal of closely related dynamic features like reflection have been introduced with the JavaCard, but are also optional in a number of other Java standards, due to resource constraints and limited use [145]. Other dynamic features regarding the type and object model have been left untouched. These include instance methods being virtual by default, the support of interfaces and the late binding of object layouts. The dynamic dispatch by default hinders the static analysis of Java programs, incurs performance penalties and must always be considered explicitly for security-sensitive API definitions. A model similar to C++ with both virtual and static instance methods is beneficial, but incurs changes to the language and additional programmer education.

Interfaces as a means of emulating multiple inheritance might also be questionable, as large and general APIs have been easily designed with languages offering only single inheritance like Objective-C [29]. Indeed, a look at the JavaCard specifications reveals that an API specification without their use would have been possible, delegating and abstract classes could have been used instead. At least, the JavaCard specification foresees a very space efficient representation and implementation of interfaces and their invocation [150].

The determination of the object-layout, including field- and method-table offsets, at link-time is one of the key advantages of Java compared to C++, and must be supported in case of JavaCard as well. For instance, it avoids the fragile base-class problem found in C++ where any change to a super-class, like the addition of a private field, breaks binary compatibility with sub-classes compiled against a previous super-class version. The system must allow for these class variations as different vendor implementations of the same API on many platforms may exist. The JavaCard then allows for portability across systems of the same class, but fails to deliver all important binary compatibility rules of standard Java necessary for API extensions on larger tokens (Section 4.3.3 and 5.5.3).

The removal of threading and garbage collection seem both natural choices in case of JavaCard, due to resource constraints. For a general architecture, a means of threading should at least be optional on larger tokens. The handling of multiple simultaneous communication streams and the simultaneous selection of multiple applications on the token by a terminal are there valid scenarios. These can still be supported by API and system extensions, and do not strictly depend on the use of Java’s monitor concept, being questioned of satisfying the
needs of safe APIs [63]. In contrast, the removal of garbage collection in case of JavaCard already affects the smartcard programmer. Its memory model expects all applications to allocate all of their resources at installation time, and to get along with them during their life cycle. This static reservation model leads to an inconvenient programming model, restricts all applications and gives a false sense of portability (Section 6).

Java supports explicitly primitive and array types for the efficient handling of arithmetic expressions and raw data. JavaCard had to restrict the supported number, as an efficient floating point and 64-bit integer arithmetic may not be needed on secure tokens for a long time. 32-bit integer arithmetic can also not be efficiently fulfilled on low-end smartcards, and is only optional in the JavaCard standard. A programmer can only rely on 8-bit and 16-bit arithmetic to be available on all JavaCards. This complies with the described general architecture philosophy that the base platform should only address the common applications, and lack all advanced features which can only be executed reasonable on advanced tokens. Indeed, smartcard applications rarely depend on 32-bit arithmetic. Most calculations involve cryptographic operations, and are fulfilled within libraries and by special hardware. Larger tokens still have to offer extended APIs and extended instruction sets for specialized applications (Section 5.4). The resulting need for token identification can be handled by the off-token environments (Section 5.2.2).

4.2.3 Comparison

The tables 4.1 on the facing page and 4.2 present the discussed features among languages having proposed as or having certain advantages in smartcard or token systems. The languages compared are then:

- **Java** features a central class concept, and is statically and strongly typed. A module concept exists, but packages are open for the dynamic download of classes. Garbage collection, referential integrity, the support of primitive types, and bounded arrays allow for a safe and efficient programming. Portable applications are possible due to the definition of a stack-based VM, strictly defining word size, object heap characteristics etc. Adoption as-is by smartcards is rather difficult.

- **C** is an excellent system programming language, but not suitable for the development of applications, having to be safely programmed and securely executed. Weak module concept and unsafe pointers must be compensated for by the target system. Machine dependent parameters like word size are determined at compilation time allowing source portability across different platforms, but only with careful use. Binary portability is again a matter of the target platform.

- **Forth** has the advantage that its VM is very small, and can be easily supported on smartcards. However, Forth is nowadays limited to a small developer community, and its global symbol dictionary hinders a strict
4.2. PROGRAMMING LANGUAGE

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>Forth</th>
<th>Java</th>
<th>Limbo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Model</td>
<td>static/weak</td>
<td>untyped</td>
<td>static/strong</td>
<td>static/strong</td>
</tr>
<tr>
<td>Packaging</td>
<td>weak/global</td>
<td>weak/global</td>
<td>weak/class</td>
<td>strong/module</td>
</tr>
<tr>
<td>Primitives</td>
<td>all/compilation</td>
<td>all/compilation</td>
<td>all/fixed</td>
<td>all/fixed</td>
</tr>
<tr>
<td>MM</td>
<td>manual</td>
<td>manual</td>
<td>GC</td>
<td>GC</td>
</tr>
<tr>
<td>VM</td>
<td>x</td>
<td>stack/heap</td>
<td>stack/object</td>
<td>register/object</td>
</tr>
<tr>
<td>Adoption</td>
<td>simple</td>
<td>simple</td>
<td>difficult</td>
<td>difficult</td>
</tr>
<tr>
<td>Prog. Safety</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Convenience</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>o</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison of language-related features

<table>
<thead>
<tr>
<th></th>
<th>JavaCard</th>
<th>Basic</th>
<th>MEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Model</td>
<td>static/strong</td>
<td>dynamic</td>
<td>untyped</td>
</tr>
<tr>
<td>Packaging</td>
<td>strong/package</td>
<td>weak/global</td>
<td>weak/global</td>
</tr>
<tr>
<td>Primitives</td>
<td>restricted/fixed</td>
<td>all/fixed</td>
<td>byte/blocks</td>
</tr>
<tr>
<td>MM</td>
<td>static</td>
<td>manual</td>
<td>manual</td>
</tr>
<tr>
<td>VM</td>
<td>stack/object</td>
<td>-</td>
<td>stack/MMU</td>
</tr>
<tr>
<td>Adoption</td>
<td>x</td>
<td>difficult</td>
<td>x</td>
</tr>
<tr>
<td>Prog. Safety</td>
<td>+</td>
<td>o</td>
<td>-</td>
</tr>
<tr>
<td>Convenience</td>
<td>+</td>
<td>o</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4.2: Comparison of language-related features (2)

...packaging concept [94]. Additionally, some VM parameters are not strictly specified, and it allows for manual memory control similar to C. Still, the Java stack processor inherits some features from Forth, such as the multiple stack cell concept to support larger primitive data types.

- **Limbo**: is the language of choice for the Inferno environment [36]. Its basic architecture and goals are similar to Java, but Inferno features a module- and procedure-oriented language, lacking OO-features, and a VM, based on a register machine. Similar to Java, the underlying machine is strictly defined, and supports automatic memory control. In contrast to Java, Limbo features a strict packaging concept right from the beginning.

- **JavaCard**: the subset of Java is package-based and removes many dynamic concepts of Java. It tries to keep the programming convenience of Java, but only offers a rigid and inflexible memory model.

- **VisualBasic** has been adopted to be the programming language of choice for the WindowsCard. As ancestor from various Basic dialects, it lacks clear concepts, and adopting in form of a strict subset is difficult. Thus, original VisualBasic features have been removed (e.g. the class concept) whereas other language types have been introduced to cope with the memory hierarchy [103].

- **MEL** names not only the bytecode used in the MULTOS system, but also...
its assembly language [98]. The languages maps directly to the properties of the underlying code and VM. Memory segments are directly addressable, modules are only visible at the system level. The instruction-set only offers types and arithmetics based on bytes or variable-length blocks.

4.3 Application Programming Interfaces

4.3.1 Considerations

Programming languages significantly influence possible API specifications. Their topics, functionality, and design methodologies can still be independently studied, and should address the different overall secure token constraints:

Resource Constraints

APIs must avoid resource intensive system-activities, taking place without notice of applications and especially including hidden memory allocations. Otherwise, neither the applications nor development tools can make reasonable assumptions about resource utilization and execution costs, especially with the variety of tokens and system providers. Hidden system-activities contrast the advantage of an abstract view of the hardware by a VM. In case of segment-based systems such as MULTOS, a developer can easily make detailed assumptions about the memory behavior of applications as long as APIs and system behave nicely. The same is true for object-based VM abstractions as in case of Java-Card, although a developer has to take more VM characteristics into account such as its object-layout.

Portability, Scaleability and Evolution

APIs should allow for different vendor-implementations on varying hardware, be extensible for different tokens, and adoptable to future requirements. This calls, on the one hand, for a strictly defined VM, guaranteeing the portable execution of code, on the other hand, for the establishment and enforcement of strict source- and binary-compatibility rules. In case of a module-based system, these rules are easy to setup and simple to understand. The addition of functions and variables keeps source- and binary-compatibility, the removal of or changes to function and variable declarations break them. OO-systems require more careful treatment, adding instance fields already lead to source and/or binary incompatibilities in a number of systems [104]. APIs must then be structured in such a way that their implementation and evolution can take place with minimum breakage of compatibility. This is especially important in case of secure tokens, where the system software is shipped in ROM, and cannot be upgraded once in the field.

API extensions are typically introduced by two ways. First, novel token services are offered in additional modules which are strictly separated from the base APIs, and are thus added in a coarse-grained manner. Second, extensions to already available services are integrated into existing packages in a fine-grained
manner, either due to implementation dependencies or programmer convenience. Already, range and offset parameters used in APIs on restricted smartcard platforms must be expanded on larger platforms. In case of JavaCard for instance, parameters are restricted to 16-bit values. Java’s polymorphism allows for a convenient way of parameter extensions as multiple method declarations may just differ in their signatures. These compatibility and portability constraints must then be reflected in the underlying execution model and run-time system (Section 5.5).

Security

A secure token system must support the strict separation between system and applications. Although API barriers can be easily controlled by the system engine, the passing and manipulation of arguments can arbitrarily weaken the sandbox concept. A few guidelines for the API design can help to increase the overall security of the system and to simplify its implementation:

- Avoid scenarios where applications can instantiate objects which can be passed to and used internal to the system. Even when the object classes themselves are fully implemented by the system, attacks exist, based on the prevention of properly constructed instances [47]. This makes careful checks within the system necessary, and leads easily to security leaks by already minor software bugs.
- Avoid mutable system objects: as soon as mutable system objects are exported to and accessible by applications, their access must be multiplexed, and thus incur additional costs.
- Avoid re-implementable classes or interfaces in the system: this might, for instance, lead to the illegal replacement of system functionality by applications, and in general to a less controlled system view. Similarly, virtual-method invocations and application callbacks should only succeed after careful consideration.

These restrictions are unlikely to be enforceable in large computing environments, but are affordable for the common base of secure tokens as the experience with smartcards show. These guidelines are then not only applicable for the system APIs, but should also be followed by applications exporting services to clients on the token.

4.3.2 API categories

Application Framework

Execution should be tailored around the communication-centric nature of token applications, and its embedding in a framework shows a number of advantages. In case of JavaCard, it completely takes over the application prologue and epilogue (e.g. connection setup, APDU reception and reply). Whenever an APDU
is received, the system signals the connected application by an method invocation, the response is sent back after its completion. Application complexity and size is reduced, and execution state is minimal when the system waits for new incoming messages. The JavaCard still fails to fully exploit its framework, for instance for efficient garbage collection schemes and resource-friendly handling of multiple APDU streams (Section 6.2.5).

Communication

The described framework leads to a finite-state-machine programming model. Dependent on its communication state, an application will handle incoming messages differently, and is likely to adjust and mark the state as having been progressed, before it sends back a response. Although this model is not that common in large computing environments and its complexity tends to increase heavily with sophisticated protocols, it should still be easy to handle for most token applications.

More abstract models like RMI can be implemented on top of the APDU-based facility, and can then also be used on larger tokens. More generic protocol stacks will be built on many different layers, including transmission, network, and transport layer, offering a port concept for multiple simultaneous connections. Another layer will be necessary to secure the communication (e.g. authenticating the involved entities, and encrypting the messages). On top, RMI or RPC facilities may be provided, which invoke the methods with the arguments, deserialized from the messages. The deserialization might either be fulfilled in stubs shipped with the application, or might be performed by the system with the help of run-time type-information [157].

Legacy dependencies and limited resources currently do not allow for the deployment of such a generic protocol stack on smartcards. Proposals for APDU-based RMI mechanisms at least exist [158], but still lack a sufficient adaptation to the smartcard environment. For instance, they do not foresee a lifetime specification for method arguments, thus being stored in RAM on each method invocation. However, they might not fit the available RAM, or they are meant to be stored persistently, and must then be copied manually by the applications. At least, many interface description languages used for RMI or RPC mechanisms allow for the specification of additional argument properties [44]. These must be incorporated to allow for the transparent placement of arguments into the transient and/or persistent store. In any case, efficient argument-marshaling places significant constraints on the underlying memory model, and requires a proper redesign of current smartcard architectures.

Interapplication Communication

For consistency reasons, the chosen invocation model should be deployed for the interapplication communication as well. MULTOS uses a message passing model for both the terminal as well as the interapplication communication. An application can temporarily relinquish control, and send an APDU to a target application. The system will start the target application, wait for its completion,
4.3. APPLICATION PROGRAMMING INTERFACES

and return response and control back to the caller. The calling chain need not be limited to two, but can involve an arbitrary number of applications. Although the approach looks simple, it involves a full startup of each involved application, and thus can incur high costs.

A remote-procedure like approach has the advantage that the called application is likely to be able to satisfy the request without requiring the full state as in case of an external request. However, safe RPC mechanisms typically involve marshaling of arguments, a mechanism which might not be affordable under resource constraints.

Cryptographic Services

The base cryptographic services, required on all tokens, are the common cryptographic algorithms, where a reasonable implementation partly depends on proper hardware support. Algorithms, based on symmetric cryptography such as DES, can already be implemented on low-end smartcards, lacking dedicated accelerator hardware. These and closely related functions such as hashing algorithms can then be offered on all secure token platforms. Services such as encryption, decryption, signing, and secure communication can thus already be fully supported, with the limitations and properties implied by the use of symmetric cryptography [136].

Public-key based algorithms like DSA or RSA depend on dedicated and/or powerful token hardware. With the importance of public-key cryptography and the large base of smartcards featuring the necessary hardware, the accompanying algorithms must still be included in the standard APIs. The off-token environment must then make sure that their applications only get installed on properly equipped tokens. Dependent on the token environment, additional algorithms may have to be supported, for instance ICOMP128 in case of GSM [41]. Thus, APIs must allow for the add-on of cryptographic services, and be ready for future evolutions, for instance elliptic-curve-based algorithms [59].

The system does not only have to cope with optional algorithms, but also with weak or strong versions of the different algorithms, due to export-control reasons. Some specifications foresee applications, which can cope with different cryptographic strengths, and adapt at run-time to the restrictions enforced by the token. Others have strict requirements on the cryptographic strengths, and can only be executed on compliant tokens. Furthermore, they might have to adhere to export regulations only allowing for strong cryptography in case of certain operations. For instance, full-strength signing operations, but only weak ciphering operations may be permitted. Typically, the off-token environment is responsible for correctly identifying the application needs and token capabilities, and managing the application download and installation accordingly (Section 5.2.2).

Other Services

The management functions, applications typically access, have already been discussed (Section 3.4.6): lookup of applications, and inquiries and manipulations
of their states. Possible states are similarly defined in most management specifications [41, 161]: downloaded after the application has been loaded on a token, installed after it has been registered at the system, personalized after it has been fed with user-related data, active as long as terminals can communicate with it, deactivated as soon as it is blocked from any communication and destroyed after it has been deleted from the token.

Closely related is the PIN handling or more generally the support for user and terminal authentication. In many cases, the central management instance will take over completely the user authentication, for instance by managing a global PIN [161] or biometric methods [109]. Any system will still export an API to applications requiring additional user authentication, for instance a private PIN. The terminal will typically be authenticated by a challenge/response protocol which depends on the token environment, and is thus mostly handled by the management instance.

The described services should cover the base APIs required, additional services should not be necessary for a wide range of applications. They especially exclude a file system most smartcard systems offer. With their inherent limitations, for instance regarding file-names, they are not adequate for larger tokens. Additionally, the different file types required by ISO7816/4 seem rather outdated in respect to the file system abstractions in use on large computing platforms today [71].

4.3.3 Case study: JavaCard

A file system specification was removed from the JavaCard specifications, not due to limited scalability, but due to incompatibilities across existing standards. A number of other areas are addressed efficiently and without hurting generality. For instance, the necessary classes for the language support in the java.lang package are kept to a minimum, but offer everything even needed on larger platforms as the embedded Java specification shows [145]. The JavaCard also successfully embraces a communication framework hiding complexity and saving resources.

Limitations result, on the one hand, from the JavaCard being fully oriented towards smartcards, for instance in case of the APDU-based communication. Abstracting RMI proposals exist, but incur two problems. First, the associated protocols are difficult to implement on non JavaCards, and do not comply to many existing application specifications. Second, the current memory model of JavaCard does not allow for a resource-efficient implementation (6.2.3).

The static memory model is also one of the reasons for the unique adaptation of RMI for interapplication communication on the token. Arguments are never marshaled and un-marshaled (i.e. transferred by copying), but are always passed by reference. They are not only protected by Java's safety rules, but also by a firewall introduced in JavaCard [105]. Only objects implementing a certain interface can be accessed by remote applications, other objects can only be used by the applications having created them. The model has a number of limitations, such as a limited possibility for transferring bulk data [105], and must be revisited for the deployment on large tokens in any case.
Crypto services have been modeled following standard Java APIs [19] which are extensible by user-supplied algorithms, and allow for applications to cope with export-control regulations. Their downsizing keeps a convenient programming experience, but incurs significant drawbacks. The required hidden-memory allocations together with the static reservation model force the system to worst-case memory allocations in advance, even in case of very simple cryptographic operations. This especially restricts the flexibility and capacities of applications offering general crypto services like PKCS#15 [135][134]. The price must then be paid without being able to benefit from original standard Java goals. User-supplied cryptographic algorithms are not an option on tokens, as they have to be executed in machine code for a sufficient performance. Even in case of large computing platforms, algorithms implemented in Java and compiled to machine code by the run-time can by far not achieve the speed of optimized native implementations [125]. The remaining enforcement of export-control regulations cannot justify the significant number of necessary classes and their impact on code size and memory footprint, as shown by other proposals [116].

Many API classes rely heavily on the use of instance and virtual methods instead of class and static methods. This impacts possible API evolutions, as the JavaCard executable format does not allow for an addition of virtual methods without breaking binary compatibility [150], a severe restriction compared to standard Java. Thus, classes cannot be extended to provide additional functionality on larger tokens without breaking binary compatibility and application portability with smaller platforms. This for instance excludes range and offset parameter extensions being absolutely necessary for the larger memory spaces on many tokens (Section 4.3.1).

The heritage of APIs from standard Java was decided at a time when many people still believed that one or at least a close subset of standard Java could target all platforms, as for instance indicated in the initial Embedded Java specifications of that time [145]. In the meantime, the development of many different Java-enabled devices led to a shift in designing APIs tailored towards specific groups of devices. An example is the CLDC and MLPD specification for connection-limited devices such as phones whose user-interface model strongly differs from standard Java [147]. Although target devices are expected to feature up to 512KB of RAM in case of CLDC, the specification is kept very simple, and the pure number of classes is low, even compared to the JavaCard.

## 4.4 Conclusions

Clearly, it would be surprising if the Java programming language does not provide enough features for programming secure token applications in a convenient manner. With its machine independence, namespace concept, support for threading and component-based programming, it is after all the language of choice for web-oriented and server-side applications. However, the chapter points out the features necessary for the application development for secure tokens: strictly separated primitive types, built-in block types, resource-friendly single-class hierarchy and global namespaces. Many language features must still
be reasonable adapted, and modifications can benefit from Java being a mix of many different, mostly strictly separated, partly cheap and partly expensive features which allow for an easy subsetting.

The chosen subset by the JavaCard must be overhauled for a general software environment. A few minor weaknesses in case of secure tokens (e.g. exported instance methods being always virtual) must be tolerated in any case not to impact the experience of standard Java programmers. Other limitations result from the underlying secure token hardware, and include limited primitive types on smaller and the need for additional primitive types on larger platforms. Their use implies platform awareness by the application developer and target environment (Section 5.3.2), and a properly architected execution engine resolving a number of JavaCard limitations (Section 5.5.3). Additionally, the JavaCard language subset suffers from its memory model, sacrificing programming convenience, limiting address-spaces and introducing an insufficient object model (Section 6).

The chapter outlines the base APIs the software environment has to provide, and presents a number of important API patterns designers should follow. The JavaCard API is discussed regarding these rules, violations are noted, and necessary API reorganizations are introduced. The APIs are specified for the common hardware and application platform, but leave room for extensions on legacy-dependent or larger tokens. These API extensions may not only succeed in a coarse-grained, but also in a fine-grained manner (Section 4.3.1). In contrast to JavaCard, the execution model for a general software environment must support both: (i) base APIs and virtual processor on low-end tokens, (ii) both extended APIs and processor on large tokens. Additionally, the proposed APIs also expect a better control over memory by the applications as in case of JavaCard.
Chapter 5

Execution Model

5.1 Introduction

A scaleable secure token application environment places huge requirements on the execution model. Applications must, on the one hand, be portable across all token families, and, on the other hand, be able to benefit from advanced tokens either by exploiting additional APIs and hardware capabilities or profiting from additional system guarantees such as code verification. Portability must be supported in the strictest sense as modifications to certified secure token applications incur significant costs. This must not affect the secure download and execution of application. Additionally, both security and portability requirements must not lead to significant performance reductions, as many specifications expect fast response times from their implementations.

Application-portability has been achieved by VMs for a long time where many have suffered from significant performance drawbacks [92]. These were addressed by either narrowing the scope of the target system, relying on expensive hardware support or implementing complex compilation schemes, which are all not directly applicable to secure tokens. At least, the execution model can be designed rather unconstrained from previous, proprietary secure token systems, and may include code generation, packaging, download, binding, execution and security.

However, existing token environments assume certain procedures for distributing code ranging from offline- and online- to broadcast-scenarios. These must harmonize with the previously discussed requirements of Java-based language and APIs (Section 4.2): both API as well as virtual processor must be extensible for legacy systems and/or advanced token hardware. Standard Java has been designed for applications executed on powerful, networked, interactive devices in dynamic environments, and all aspects of its execution model, including instruction set and code format, remain insufficient.

In the following, a new Java-based instruction set is proposed, its characteristics are studied, and its consequences on the run-time environment are discussed. The instructions must be packaged in a space-efficient format allowing for a fast download and installation process. Two new code formats and binding processes are then proposed. The first one targets minimum run-time
and code size, but falls short on possible scaleability and verification. The second one addresses reasonably equipped tokens, and fully addresses all described portability and scaleability requirements. It does not inhibit address-space limitations, supports fine-grained API-extensions and allows for code verification even on resource-limited secure tokens.

5.2 Basic Properties

5.2.1 Packaging

With secure tokens being deployed in the very large scale, programmers should be able to associate their application sources and binaries with unique identifiers. In standard Java, package names are based on the global domain names used in the Internet. After registering a domain name, a programmer is supposed to use it freely as the prefix for all of its packages. In the smartcard world, developers must register at a central instance to reserve a globally namespace for their exclusive use [70]. In both cases, the proper handling of the registered namespaces is up to the designated parties.

An issuer will only download and install content he can trust. Dependent on his policy, he might demand the inspection of the application source code and its proper behavior regarding specified test suites. He is likely to insist on secure downloads and installations under his control, including signed/encrypted code and the transfer over a secured channel. Additionally, he might require certain on-token measures, a static code verification for example. Although he might not insist on all of the described measures (e.g. allowing the download of signed code over an insecure channel), he is unlikely to ever allow for the anonymous download of untrusted code in the large scale.

Other environmental dependencies influence executable format, download and execution as well. First, downloads may either be performed during an online connection from the issuer back-end, or offline from a terminal. Second, important network parameters differ in current issuer networks. Downloads may succeed over reliable and fast connections (e.g. LAN in case of PCMCIA tokens), or under limited connection speed in case of smartcards. In case of GSM networks, the basic transport protocols are unreliable [41], and support for broadcast scenarios is required. As the installation process may also suffer from on-token resource constraints, network bandwidth cannot always be traded for processing power.

5.2.2 Portability, Scaleability

The assumptions the execution model can make about the underlying processing resources must be based on the common, low-end smartcard hardware (e.g. low processing speed, limited word size, and limited memory sizes). This must then lead to a space-efficient executable format, a binding process with minimum temporary RAM needs, a compact run-time representation, and good execution performance. Three basic scenarios for the distribution of portable code are conceivable:
5.2. BASIC PROPERTIES

Figure 5.1: Off-token VM-model

(1) Off-token Execution Model

Language and APIs are strictly specified, but code generation and distribution is only specified for the shipment of code to an issuer and its back-end system. The execution engine consists of an off-token part, executed on the issuer back-end (responsible for receiving, preparing, and downloading code), and an on-token part for executing it (Figure 5.1). With the protocol being an internal detail, the system implementor is free to fully optimize the download and installation process to the token and network characteristics, ranging from the code being completely pre-linked on the remote node to a self-describing, high level format suitable for on-token verification. Code might be transferred in an intermediate representation or even compiled to machine code.

An off-token engine can be successfully deployed in environments where only tokens of the same type are used, or can always be properly identified and served on-the-fly, for instance tokens used in LANs. In large smartcard environments, tokens of different vendors must be deployed, and an on-the-fly processing adds significant management costs. Additionally, the off-token engine fails in offline and broadcast scenarios as in the banking and GSM sector. In the offline case, it cannot be expected that a terminal can handle all vendor-specific tokens. In the GSM case, it cannot be expected that a different code representation for each token can be broadcast over the air.

(2) Fixed, On-token Execution Model

Minimum administration costs and simple distribution are the advantages of a fixed, on-token execution model. In this case, APIs and execution engine are fully specified at the level of the token, and all tokens have to comply to a single specification. Thus, all applications can be installed on every token, no client identification is needed, and management overhead is kept minimal. However, only the common denominator can be covered by the specification, and all advanced tokens and their applications fall out of it.

(3) Extensible, On-token Execution Model

This model conforms to the previous language and API considerations. Base, on-token APIs and execution-engine are strictly specified for the common hardware
denominator. Additional, but optional elements may be supported on advanced tokens, and be exploited by certain, token-specific applications. Only if an environment foresees the deployment of advanced applications, the management system must foresee the identification of the client tokens at installation time, and be able to cope with different application sets across tokens. The number of optional elements in the specifications must still be carefully considered (Section 5.3.2).

5.2.3 Execution Engine: Interpreter

The resource constraints on low-end tokens must lead to a compact run-time representation of the executable code. This includes a space-efficient storage of run-time type-information (RTTI) and a good density of the instructions. The necessary code density will immediately lead to a design based on a stack processor instead of a register machine. Instructions of stack processors always address their arguments on the operand stack implicitly, while instructions of register machines denote them explicitly by register numbers [36]. The implicit addressing does not only save space, but also leads to faster decoding and better performing interpretation. An interpreter can be implemented space efficient, and almost constant size across different platforms. As the intermediate code is portable and strictly specified, necessary run-time sizes for applications can be predicted easily and mostly exact for different platforms.

Interpreter execution is still slow compared to machine code, on large Java systems known to be up to 50 times slower [126]. Although interpretation is the only reasonable choice on smartcards, the code should still allow for a compilation to machine code on very large tokens. Additional resources are still required when an intermediate format, based on a stack processor and optimized for interpretation, is compiled to machine code of decent quality [36]. At least, the performance study shows that the typical application patterns of secure token applications only require compilation schemes in rare cases (Section 5.4.5).

Executable content should allow for quantitative and qualitative statements by a static analysis before its download. It is able to cover all possible instruction and/or data states of an application, while observations based on the dynamic tracing of code may leave certain application behavior unrecognized, even in case of large test suites. Important methods include correctness-proofs of applications against formal specifications [123, 85], data-flow analysis for inquiries of allocation and data-usage patterns [106], program-flow analysis for inquiries of execution behaviors and code optimizations [106], and safety analysis for the verification of code obeying rules set by a type-safe language or system [119].

5.2.4 Security Model

Code Protection

Code modeled after a type-safe language allows to verify code access patterns statically. Many expensive run-time checks are unnecessary, and the possible
performance is increased. Many unsafe operations – permitted in other environments and often leading to subtle program errors – are detected before they are executed, and affected applications can already be rejected at install time. In contrast, MMU-based systems are based on data protection at runtime. In MULTOS and WindowsCard for instance, each application is assigned to different, linear memory segments (e.g. two segments in RAM for stack and temporary heap, and one segment in EEPROM for persistent data). Each attempt to access data outside will be detected, and lead to an error. The model is easy to understand and allows for good formalizations [60]. However, in case of stack processors and their implicit addressing, every instruction addresses at least one segment, and incurs at least one boundary check during its execution. The resulting performance impact led to the hardware realization of MMUs on larger systems. However, as abstract processors for secure tokens are designed for efficient software implementations in the first place, their implementations in hardware can be expected to remain unattractive [91].

Protection by MMU is still optional in case of a system based on type-safe code, where instructions do not operate directly on memory cells, but on language entities. At run-time, the access of language entities is mapped to the access of memory cells, which can be guarded by an MMU for additional security [156].

**Code Signing**

Secure execution based on a software-MMU or code verification is an important, additional security measure, but is unlikely to replace all other measures deployed in secure token systems, especially not code signing. Code verification ensures consistent code, but does not authenticate code, and only authenticated code can be permitted to access security-sensitive system services on issued tokens. Code signatures additionally prevent malicious applications of being able to exploit well-known security holes in system implementations. It might be possible to expect the update of run-time environments in case of security weaknesses on large computer systems, but is extremely difficult in case of smartcards, where the system resides in ROM.

However, relying exclusively on properly signed code also incurs a significant security problem, the dependency on a single, trusted instance, its security precautions and its well-operated environment. Code signing thus complements the security measurements of the execution environment deployed on the token.

### 5.3 Standard Java

**5.3.1 Deficiencies**

Java features stack processor and type-safe instruction-set, and has foreseen code interpretation and verification right from the beginning. However, the resource needs of standard Java exceed by far what is available on even large mobile tokens. The initial orientation towards powerful, networked and interactive devices led to totally different assumptions about target hardware and
applications. Many unnecessary or expensive features must thus be removed or adopted for secure tokens.

Class-File-Format

Java supports a class-centric execution model with the lazy download of new classes at execution time. Every class of an application must contain the necessary binding information which is thus shared among the classes referencing each other. The degree of redundancy depends on the concrete application, but it is not unlikely that 50% of the binding information is shared by all class-files in an application or package [131].

Additionally, the binding information itself in form of symbolic names and signatures is stored space-inefficient in the class-files. For instance, the name of the package, an application resides in, is likely to be stored fully and multiple times in a number of method signatures. Dependent on the package, the symbolic information alone can take up to 30% of a class-file [64].

Due to an open-package concept and class-files lacking a proper organization, an implementation is forced to keep all binding and symbolic information at runtime. With the binding process implied by standard Java, it is impossible to strip down a class-file just to the bare minimum required for execution.

Binding Costs

Standard Java foresees the dynamic and lazy binding and initialization of classes. Code is not downloaded and installed statically, but is supposed to be bound and initialized in the order of its execution. The model is even questioned in case of interactive and networked environments [46] [45] as the VM must keep additional state during the execution, has to keep all binding information and must write the locations to be bound multiple times (especially expensive when code is stored in EEPROM as in case of secure tokens, Section 2.2). It must be noted that standard Java allows the download and binding of all classes referring statically to each other at once, but only as long as the application behavior is unaffected. This implies that errors encountered during the bind process must still be delivered at run-time, making a static binding process unattractive [96].

The initialization of applications always succeeds at run-time during the execution of class initializers [96]. Even predetermined data is not included in the class-file, but built up by the explicit class initialization code. This avoids assumptions in the code about the layout of data, but incurs significant processing and storage overhead. To describe a single byte of a preinitialized array, a class-file contains at least five bytes initialization code.

Execution Costs

The instruction-set of Java is oriented towards 32-bit processors, and is difficult to support on 8-bit and 16-bit controllers on smartcards. Although primitive types can be stored space-efficiently in objects, operands on the stack are always expected to be at least 32-bits wide. This keeps the interpreter simple and reduces the number of necessary instructions. However, the necessary stack size...
is heavily increased, and the performance on low-end processors is significantly reduced. This contrasts to other complex CISC instruction-sets where arithmetic operations on all different kind of data types are supported, and clearly shows the orientation of Java towards VM implementations in software [54].

Good performance on low-end systems is also hindered by the strong object-orientation and dynamic nature of Java. Most optimizations like method inlining are expected to be applied by the VM at run-time and not by the source-code compiler at bytecode generation time [54]. These run-time optimizations require resources which are not available on secure tokens.

Verification Costs

Static code verification is always expensive, but a number of constructs in the Java bytecode hinder especially the deployment on low-end tokens. The need for a typeflow analysis following the control-flow of an application, and the complex semantics of a number of instructions incur high resource demands or processing needs. Simplified semantics and code conversion can decrease the resource needs dramatically (Section 5.7).

5.3.2 Adaptation Strategy

Two-Stage Code-Distribution

The class-file format of standard Java still provides additional flexibility for distributing executable content. Instead of compiling source code to final binaries as in case of WindowsCard or MULTOS, a Java-based environment can offer a two-stage code-distribution model for secure tokens. Application sources can be compiled to and shipped in form of class-files to service providers and token issuers. They allow for verification and analysis without the need for source inspection, and can be effectively used for an additional post-processing and translation step to generate the final token code.

The two-stage distribution model allows for the possible deployment of all three base execution models (Section 5.2.2). In case of an off-token model, class-files are shipped to the issuer, and are downloaded and converted on the fly to the format specific to the connected token. In case of an on-token model, the code format for the secure token is strictly specified. Dependent on the policy of the issuer, the application provider may have to ship the code in the final format or in class-files. In the latter case, the issuer may analyze the code with the many tools available for Java before converting and download it on the tokens.

Virtual-Processor Flexibility

A general environment has to foresee an extensible, on-token model where optional elements are added in a coarse-grained manner. Most environments can then be operated without a need for token identification. At least, a single class of tokens, i.e. smartcards, is currently used in environments where token
identification is difficult. Multiple classes are only deployed in networks where identification is simple as for instance in case of LANs. Furthermore, the larger the Java subset supported on low-end tokens, the less extensions are required on larger platforms. The memory model proposed later satisfies both low-end as well as high-end tokens (Section 6). Language elements or API constructs like multi-threading or reflection have been discussed to be unnecessary or highly specialized (Section 4.2). Absolutely required by language and APIs is only the support of additional primitive types on larger tokens (Section 5.4). Other possible extensions of the execution engine may succeed transparently, code compilation or verification for example. Code compilation is rarely necessary due to the performance characteristics of secure token applications (Section 5.4.5). Code verification is critical, but difficult to achieve on resource-constrained platforms (Section 5.7).

The virtual processor has to support a high degree of application portability on all tokens. The on-token execution model relies on a strictly specified format for the code downloaded on all secure tokens. Independent of fine-grained API extensions on a particular token platform, the binding process is expected to guarantee of the portability of the base applications.

Benign Transformations

The model of the executable content should be derived from and close to standard Java. On the one hand, this allows for increasing the trust into the newly established software environment. On the other hand, chances are high for being able to convert not only class-files into token executables, but also the other way around. This might then not only allow for the reuse of many existing Java tools like verifiers or debuggers [116], but also for a comfortable execution of the token format on a standard VM in the future.

An approach based on standard Java will be sufficient as soon as the resulting performance and code size can even satisfy low-end hardware needs. At least, the achievable run-time performance is mostly determined by the instruction-set chosen for the secure token VM.

5.4 General Java-based Instruction-Set

5.4.1 Design Steps

An instruction-set suitable for smaller platforms can be easily derived from standard Java without sacrificing its original goals. This results from the fact that its instruction-set is strictly grouped into a number of categories (e.g. arithmetic instructions) where each category expects certain argument types at execution time. Indeed, typed instructions are key to good interpreter performance and code verification [54], but also simplify deriving a suitable instruction-set by removing or slightly adopting the different instruction categories. The individual categories are:
• Removal of unnecessary instructions: Standard Java supports separate instruction-sets for each primitive type (i.e. integer, float, long and double). With only integer arithmetic being feasible on secure tokens, all instructions operating on different types should be removed in a first step, including logic instructions like `lxor`, arithmetic instructions like `ladd`, conversion instructions like `l2i`, array-load/store instructions like `laload`, local-variable access instructions like `lload`, and a number of load-constant instructions like `lconst0`. The removal can basically be performed by just looking at instruction opcodes, and does not involve complex decisions. The only remaining primitive type, supported in the instruction-set, is now the 32-bit integer-type.

• Introduction of the base 16-bit-integer-based instruction-set: With the need for efficient interpretation on smaller chips, the original instruction-set based on 32-bit integer-arithmetic must be mapped to a 16-bit-based set. With mapping the 32-bit cells on the operand stack to 16-bit cells and the renaming of the opcodes of the affected instructions for clarity, most of the work is already done. The instructions renamed are then again arithmetic, logic, conversion, array, local-variable, load-constant and comparison instructions. Additionally, control-flow instructions like `tableswitch` operating on 32-bit integers have to be adopted.

• Support of optional 32-bit arithmetic-instructions: Other arithmetic types can always be supported by additional APIs. Variable-sized integers can be efficiently supported by classes implementing the arithmetic operations natively, for example [19]. The overhead of necessary object invocations is negligible compared to the computation to be performed. In contrast, efficient interpretation of 32-bit arithmetic requires an explicit support in the instruction-set. The 32-bit integer-type can be added in the same way as the 64-bit integer-type in standard Java. A second set of arithmetic, logic, conversion, comparison, array, local-variable and load-constant instructions is added, where each instruction expects its 32-bit arguments in two 16-bit cells on the stack. As long as no 32-bit numbers are used within an application, the necessary stack size remains unchanged. In case of secure tokens, the results of arithmetic expressions rarely exceed the 16-bit barrier as heavy computations are likely to be performed within the cryptographic libraries, even on large tokens. More important is the support for array ranges larger than 16-bits (Section 4.3.2). As arrays are accessed by primitive types, this naturally leads to the native support of 32-bit integers on larger tokens.

• Object-related instructions: With the support of the OO features of Java, the set of type-related instructions (e.g. `new`, `checkcast`, and `instanceof`) remain similar to standard Java. Standard Java interpreters translate field- and method-related instructions at execution time to more specific, but proprietary instructions signaling additional information about the target as for instance in case of the `invokespecial-instruction` [54]. With preferred static properties in case of secure tokens, the instruction-
set should feature separate field-instructions for each supported primitive type, and allow for different invoke-opcodes signaling specific method-invocation types.

- Remaining instructions: The stack instructions-reordering operands on the stack—may be left as is. While they operate on single/double 32-bit-cells in standard Java, they now operate on single/double 16-bit-cells. The only critical instruction left is then the jsr-instruction used for the invocation of method subroutines. With its rare occurrence in token applications and its expensive verification-semantics, the jsr-instruction may be removed (Section 5.7).

- Newly introduced instructions: In order to reduce size and increase performance, new instructions may be introduced. They should fit the current bytecode properties to keep the trust into and not hinder its verification-model. The simplest way is to map sequences occurring very often in typical JavaCard applications into single instructions. The single instructions behave in terms of execution and verification exactly like the original sequences, and are thus easy to support in the tool chain. Table 5.1 shows a number of possible instruction sequences, their length and number of occurrences in a sample system-classes implementation of JavaCard. The instruction sequences differ in complexity and number of occurrences across different applications. In the example, the first and forth sequence are most interesting as they strongly correlate to language expression heavily used in applications. To exploit application-specific instructions, other approaches must be deployed, as for instance described in section 5.8.

### Table 5.1: Sample occurrences of instruction sequences.

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Occurrences</th>
<th>Length</th>
<th>Arguments</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>aload0, getfield</code></td>
<td>186</td>
<td>3</td>
<td>1</td>
<td>this.f</td>
</tr>
<tr>
<td><code>aload0, aaload1, sipush</code></td>
<td>72</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><code>invokestatic, pop</code></td>
<td>80</td>
<td>4</td>
<td>1</td>
<td>(void)m(...)</td>
</tr>
<tr>
<td><code>baload, sipush, iand</code></td>
<td>57</td>
<td>5</td>
<td>1</td>
<td>b &amp; 0xff</td>
</tr>
</tbody>
</table>

An instruction-set is not fully orthogonal to the encapsulating format, but places certain restrictions on it. A Java-based instruction-set optimized for secure tokens expects from the binding step to allow for small RTTI and minimum execution contexts at run-time.

#### Small RTTI

The instructions can cope with only a minimum of available RTTI, especially no symbolic information is needed just for the purpose of execution. Classes, fields and methods may not be addressed by name, but by run-time identifiers.
5.4. GENERAL JAVA-BASED INSTRUCTION-SET

<table>
<thead>
<tr>
<th>Type</th>
<th>Covers</th>
<th>Size in Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class-Descriptor</td>
<td>super class, instance-layout description, references to tables</td>
<td>8</td>
</tr>
<tr>
<td>Interfaces-Table</td>
<td>one per implemented interface with address and m method-references</td>
<td>(n \times (2 + 2 \times m))</td>
</tr>
<tr>
<td>Method-Table</td>
<td>element count + single entry per method</td>
<td>(2 + 2 \times n)</td>
</tr>
<tr>
<td>Method-Descriptor</td>
<td>number of arguments, locals and stack-operands</td>
<td>2</td>
</tr>
<tr>
<td>Field-Descriptor</td>
<td>field offset and size (fully encoded in field-instructions)</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.2: Typical run-time type-sizes in case of a 16-bit-based platform

such as offsets, indices or addresses. Especially, the lookup of class or interface methods may not be performed by a double-/triple-indirect name lookup, but by looking up class addresses, method tables and indices [150].

Table 5.2 shows typical run-time type-sizes one can reach on a Java-based secure token VM assuming an underlying 8/16-bit-based hardware. Addressing items internally by 16-bit and adopting standard Java limits (e.g. the maximum number of method arguments) help to achieve these small numbers. Although the given numbers may vary across different implementations, they already show that the type information required for the execution of a class can be less than the space needed to just store its symbolic name.

Minimum Execution Context

In case of Java, a stack frame consists, on the one hand, of the method arguments, the local-variables and the operand stack. Their sizes depend on the currently executed method, are precomputed by the compiler, and are thus predetermined for an interpreter [54]. On the other hand, an interpreter has to reserve space on the frame for saving the current execution state whenever a new method is called. In order to reduce stack size and increase performance, the execution state should be kept as minimal as possible.

Any Java-based interpreter has to save the method-local execution context (i.e. program counter, location of local-variables and current stack operands) in case of an invocation [96]. Standard Java interpreters also have to save context information for the current method (e.g. current class and constant pool). This context information is needed whenever exceptions have to be caught, binding actions have to take place, and security decisions have to be fulfilled [53].

In case of secure tokens, applications are shipped in a closed-package, and can be stored and statically linked in a contiguous area on the token. The required context information can then be reduced, and fully derived from the stored program counter in the best case.
5.4.3 Code Generation

A conversion from standard bytecode to the described instruction-set is straightforward. First, a converter will load an application class, parse it and decode its instructions. Second, it checks for all illegal opcodes not to occur. Finally, it maps the 32-bit-based integer-instructions to the 16/32-bit-based instructions defined for the target. The mapping is possible as the Java bytecode is strongly-typed, and allows for the prediction of the exact operand types at every instruction by a typeflow analysis [81].

In the simple case, the target is the common denominator token, and only allows 16-bit arithmetic. With a typeflow analysis, the converter can verify that only 16-bit operands (i.e. Java shorts) are manipulated by the original 32-bit integer instructions, and can then map all 32-bit integer instructions straight to the 16-bit pendants. In the mixed code case, the converter will use the typeflow analysis to identify the integer instructions operating on 32-bit numbers (e.g. Java integers) and on 16-bit numbers. The identified and categorized instructions are then either mapped on the 32-bit or 16-bit instructions in the target set.

It must be noted that programmers tend to use a single integer type for most of the declarations and expressions in their applications, and immediately switch to the 32-bit integer type by default, as soon as it is supported by the system. In this case, the converter has to perform a dataflow analysis to optimize the integer usage [33]. This increases the complexity of the converter implementation, although the outcome is still questionable. The state of a secure token application depends on the external input, and is thus difficult to predict with a static dataflow analysis only. Previous systems have always shown significant resource increases when switching from 16-bit to 32-bit environments [12].

5.4.4 Low-End Case Study: JavaCard

The JavaCard instruction-set mostly follows the assumptions and approaches described. It features a 16-bit-based instruction-set and optionally an instruction-set operating on 32-bit integer-types (each covering two 16-bit cells on the stack). Control-flow, conversion, logic and local-variable instructions are then offered for both types. Other primitive types like variable-sized integers are expected to be offered by APIs.

Instructions related directly to classes are organized as in standard Java, the number of instructions for accessing class members have grown as described: field-related opcodes have been added for the fine-grained specification of field type, and additional invoke-opcodes provide details about target method and invocation type. Important differences to standard Java are due to file format and binding process, as for instance fixed-sized fields and fixed method-table layouts. They do not significantly impact the basic execution engine, but significantly the portability and scaleability of the system (Section 5.5.3).

Many standard Java instructions get minor updates. For instance, 32-bit alignment-constraints are removed, some instructions are compactified by reducing the possible argument space, the stack-instruction family (i.e. \texttt{dup}, \texttt{dupsx1},
5.4. GENERAL JAVA-BASED INSTRUCTION-SET

5.4.5 Performance

The JavaCard instruction-set already gives a good hint on what performance can be expected by a general Java-based system. As long as systems are based on the above assumptions and mixed 16/32-bit instruction-sets, performance patterns will be similar even across different tokens. Systems may still feature different executable formats which finally determine how instance fields are accessed, methods are invoked, and static or class members are manipulated. This then decides upon scaleability and verification costs, but is unlikely to significantly impact the performance (Section 5.5).

Figure 5.2 shows a performance comparison of VisaCash operations on three different smartcard systems, one native VisaCash implementation on a proprietary system versus an implementation in bytecode on one highly- and one less-
optimized JavaCard. The base hardware of the measured JavaCards is the same in both cases, while the native implementation features a processor of a preceding generation being 2 to 4 times slower on typical arithmetic operations. The examined VisaCash application is based on a non trivial electronic wallet specification similar in functionality to the Geldkarte, and involves around 4.5KB of JavaCard bytecode instructions [159]. The benchmark shows the timing results for three typical operations: select the VisaCash application on the card, load some value on and deduct some value from it. The first operation does not involve too much application code, and should thus show similar performance on all platforms. In contrast, operation two and three involve a lot of application logic, and should show significant differences between the three platforms. However, the timing results on the native and highly-optimized interpreter platform are still close to each other. The operation times are dominated by the cryptographic operations, executed natively in both cases, and slow EEPROM write-accesses, fulfilled by the hardware. Thus, interpretation overhead is extremely small compared to large environments, and the chosen instruction-set shows satisfying performance characteristics on smartcards. The numbers of the straightforward implementation still show that a JavaCard implementation must be heavily optimized to be comparable with native implementations. Optimizations must then cover native functionality (e.g. cryptographic services), but also the interpreter itself [39].

Applications are likely to show the same patterns on large tokens as on smartcards. They will also depend on cryptographic functions protecting data read from and written to the permanent storage. Interpreter performance is then still likely to be sufficient for the high-level application-“scripts”. If performance is critical on a certain token, bytecode can still be compiled to machine code. With the instruction-set based on standard Java, compilation might be performed with similar success and under similar trade-offs [68]. Modern, high-performance compilation-schemes relying on run-time profiling and recompilation still seem inappropriate due to their high costs and dynamic nature. In case of secure tokens, ahead-of-time compilation-schemes (i.e. the bytecode is fully compiled at installation time) seem better suited [40, 126]. Static, hybrid execution-engines, where only performance-sensitive parts of applications are compiled, and remaining code is interpreted, might provide a good trade-off between space and size, and might fit secure tokens even better [111]. The part having to be compiled must then be specified externally, and marked in the executable content [84].

5.5 General Java-based Link-Format

5.5.1 Basic Considerations

The goals for the link-format are determined by the resource and network constraints of the many different secure tokens:

- Resource-friendly binding-process: The download process must efficiently use the available RAM and EEPROM on smartcards, i.e. the required
temporary state must be kept minimal, and expensive EEPROM writes must be avoided.

- Minimum run-time size: Code and RTTI remaining after the binding process should be as close as possible to the minimum the instruction-set requires.

- Small download-size: The executable content having to be transferred during the download should also be as small as possible, due to limited communication-speeds as for instance in case of GSM networks.

- Orthogonality to code signing: The executable content should not enforce specific cipher or code signing schemes.

- Binary compatibility: For portability and scaleability reasons, the executable content should allow for flexible API specifications and implementations not breaking binary compatibility across different platforms (Section 4.3.3).

The link-format must contain enough information to allow for the system binding the application code, i.e. to replace certain references within the code with run-time-specific identifiers. A link-format must then encapsulate the following:

- Instruction-set and RTTI: This data is required for execution, and references items either located within or external to the code. These references are replaced by run-time-specific identifiers like addresses during installation.

- Export information: The link-format includes information about which items are accessible by other code packages, and where to find or how to address them at run-time.

- Relocation information: The link-format has to allow for identifying the locations having to be bound and by what means.

- Naming information: Items exported by and referenced within the code must be named to be bound during the download.

Many different link-formats have been proposed in the past. Machine-code formats like ELF foresee a simple and fast, but very machine-specific link-format [154]. They typically consist of code, data, and symbol section, and relocation table. At execution time, the individual sections are loaded, and the relocation table is scanned (Figure 5.3). Each entry contains the offset, where the relocation has to take place, and the name of the link target. If the link target is internal, it is named by section and offset, a fix-up like its address is calculated and applied at the affected location. If the referenced item is external, its name is located in the symbol table, and is looked up in one of the system export tables. The associated value is then taken from the export table and applied at the affected location.
In contrast, a standard Java class-file has to be parsed and strongly adopted by the VM to be ready for execution. Interpreter-based systems typically keep the instructions in their original form, but may differ strongly in the layout of RTTI like class, field and method structures. The linking and binding itself takes place in a number of steps and is expected to succeed in the order of execution. The whole process is very expensive to execute, but still allows for optimizing the layout of run-time structures to the underlying machine regarding word size and address width.

Clearly, a format for secure tokens should share properties with both formats, the simplicity and speed of machine-code file-formats, and the machine independence of class-files. However, both formats were not designed with the memory hierarchy and RAM/EEPROM characteristics in mind. These require that the download and binding should be processed as much as possible on-the-fly, the necessary resources should already be known in advance, and temporary information should be kept in RAM for its later release. The persistent code information should only have to be written once, multiple EEPROM writes to the same location should be avoided during the download and binding process. Streaming based formats similar in spirit have also been proposed in other networked environments to optimize the download process [17].

5.5.2 Format Options

With different goals such as compactness and verifiability, many options for the link-format exist, especially in the areas packaging, instruction layout, naming, binding and export information:
5.5. **GENERAL JAVA-BASED LINK-FORMAT**

**Package Concept**

Grouping and merging all classes of an application into single packages helps to decrease code size dramatically. For instance, already merging the constant pools of class-files reduces significantly the size of standard Java packages [14]. An efficient format must thus incorporate all classes of an application or library, must give up the dynamic binding process of standard Java, and prefer static binding at installation time.

Executable content can then provide global information about package, contents and link requirements. For instance, it can signal whether this code package acts as a library and link target at a later time, or contains an independent application. Especially, it can describe its structures and code in a way that it can be efficiently streamed in and space-efficiently stored in one contiguous area on the token.

**Naming Information**

The executable content has to name the items it has to get bound to. All locations referring to named items get then patched with their run-time-specific identifiers. The naming scheme deployed may differ across different link-formats:

- **Symbolic/offset-based linking:** Machine-code formats typically reference external items by symbolic names and internal items by offsets. While a symbolic name always has to be looked up in an external dictionary, an offset can be directly used to determine a fix-up.

- **Inherent symbolic-naming:** Systems such as Smalltalk and standard Java only foresee the replacement of symbolic references by run-time-specific identifiers, and especially expect symbolic information to be run-time-inherent. All or at least most of the symbolic information cannot be discarded after download.

- **Structured symbolic-naming:** All items are again referenced symbolically, but symbolic information identifying externally or internally visible items are differentiated in the link-format, and can then be fully or partly deleted.

- **Compressed symbolic-naming:** Symbolic information can be compressed either at code-generation time or at link-time. Examples for compression during code-generation are name-mangling for signatures in C++, or name-remapping by Java obfuscators [52, 162]. The compression scheme may either be used globally as in C++, where all symbols across systems and applications are generated and stored in a compressed manner, or may be used package-locally, where only package-internal symbols are compressed, and package-external symbols remain in their original form. Resulting compression ratios can still be impressive as Java obfuscators have shown [162]. Inferno uses a link-time based compression scheme [36]. Names and signatures are replaced at link-time by an MD5-based hash reducing the size for
names and signatures to a constant value. Whenever a symbol has to be found, the binder computes and looks up its hash in an export table. The hashing function used must still guarantee that either the correct symbol is bound or an error is signaled.

- Token-based naming: symbolic information can be completely avoided if the symbols are mapped at code generation time to fixed, small integers or tokens. The chosen token mapping for a library must be preserved and used whenever a client application is compiled. The client application can then be equipped with the proper tokens to identify the referenced items in the library. The most space-efficient representations can be achieved when run-time-specific identifiers are mapped to tokens as for instance offsets in case of instance fields. The token can then be directly used at run-time, no additional mapping or lookup of the token must take place on the secure token. Other tokens and their values (e.g. class number and address) must still have to be stored explicitly in an export table.

**Instruction and RTTI Layout**

On-the-fly processing requires the code package obeying a number of ordering constraints. For instance, interfaces should be transferred before classes and consequently, super-classes should always occur before their sub-classes. This allows for the on-the-fly binding of the class, method and field descriptors and for the on-the-fly determination of object-layouts and method-tables.

On the one hand, these structures and parameters may be heavily precomputed and transferred in a form suitable for an immediate and in-place execution by the VM. Many VM parameters are then fixed naturally leading to scalability problems. On the other hand, the structures may be transferred in high-level form as in case of standard Java, and must then be transformed on the token into a format suitable for execution by the underlying engine.

In standard Java, instructions occur alternating with RTTI, and cannot be bound on-the-fly as they can always reference items not having been loaded so far. Instructions must thus occur either at the end, or a different naming scheme (i.e. token-based naming) must be used to allow for an on-the-fly process.

**Binding Schemes**

**Most Space-efficient Binding-Scheme: Marker-based Format**

If the naming scheme is based on tokens and offsets, all binding information can be easily fully processed on-the-fly. Additionally, it can be stored most space-efficiently, as the binding information can be directly injected at the locations requiring relocation [114]. The locations to be bound can be detected by two methods. First, the format is fully parse-able and all affected locations are implicitly detected by the linker during the download. Second, special markers are injected into the code stream, explicitly signaling the locations to be bound, and avoiding a complex parsing scheme (Figure 5.4). Experiments show that a typical JavaCard application like VisaCash needs a relocation at about every
ninth byte, and that a careful design can reduce the overhead for the injected markers to one byte per location, resulting in a total increase of the executable size by circa 10%. For comparison, the JavaCard 2.1 file-format needs around 20% relocation information in case of VisaCash, and incurs an additional, single EEPROM write per instruction having to be relocated. Additionally, the marker-based linking scheme can already be efficiently implemented on smart-cards with 256 to 512 bytes of RAM [83].

A format, based on preprocessing, streaming, markers, offsets, and tokens, is the most efficient format regarding download size, run-time size, and binding performance. Especially, the code size at run-time can remain as minimal as possible, as markers do not have to be stored persistently. The format additionally achieves the same properties for binary compatibility and machine dependency as JavaCard 2.1 (Section 5.5.3).

Verifiable Binding-Schemes

However, offset-based schemes hinder code verification. All items referenced by an offset have to be verified to be valid items and to be type compliant with the location referring to them. This basically implies associating items with offsets, and consequently parsing and transforming the code into a representation more suitable for verification. As the code has been designed for optimum run-time size and in-place execution, all verification information must additionally be stored separately in the code file. This results in a significant code increase and resource-intensive parsing process. Both execution and verification information must be read, transformed and merged to provide the full information about all run-time items such as classes and methods. On-the-fly processing is then impossible, and the verification process is totally different from the binding and execution process.

An offset-based naming scheme is thus difficult to support, and the switch to a symbol-based scheme seems likely. Although many items can still be identified
space-efficiently by indices during the link-process, e.g. a field by its number within a class, most items must be named symbolically for a code verification to take place. Especially, symbolic information cannot be stored within the instruction stream anymore, due to its size.

Machine code formats use relocation tables to identify locations to be bound and separate symbol tables to name the referenced items. In standard Java, the symbol table is called constant pool, and contains names, signatures and types for referenced and exported items (Figure 5.5). In contrast to machine-code formats, parse-able class-files lack relocation tables, but directly contain indices into the constant pool at affected locations. At link and execution time, each referenced items is looked up, and a fix-up is applied at each source location.

Export Information

Items referenced by offsets or tokens being applicable at run-time can be bound directly. To hide implementation, the run-time values for many tokens still have to be stored and looked up in export tables [150]. They must be kept for all libraries on the token, and are then stored separate from the instructions and run-time descriptors.

Symbolic references must always be looked up before a necessary fix-up can be applied. In case of machine-code formats, symbols are typically stored separately from and are referenced by indices within the export and relocation tables. In standard Java, the run-time descriptors like classes, methods and fields are linked with each other, are directly associated with their symbolic information, and are thus traversable (Figure 5.5). This speeds up the necessary lookups, but increases RTTI size.

In any case, binding and symbolic information should be discarded as much
as possible after the download. In case of libraries, only symbolic information of items being externally visible should be kept. In case of applications, symbolic information should be fully discarded.

5.5.3 Case Study: JavaCard

Properties

JavaCard targeted a format applicable to smartcards with 1KB of RAM. Verification should be possible off-card today, but on-card only on future smartcards. The resulting cap-file format is based on the following design:

- Closed-package concept: A cap-file contains either an application or a library, and encapsulates all of its classes.

- Token- and offset-based: A cap-file uses tokens to identify cap-file-external items and offsets to identify cap-file-internal items. The token mapping is stored at conversion time in export files. In case of libraries, they must be shipped with the APIs to allow for the proper conversion of client applications. The token namespace is thus not global, but local to packages.

- Section-based: A cap-file is separated into individual components including the header component for package-global information like name, version, and referenced packages, the executable components including RTTI and instructions, the binding components including relocation information, the verification components including necessary type information for code verification, and the export component.

- Export information: The export component must only reside on the token for libraries, and can be removed for applications. The export component contains the values for the tokens which could not be mapped directly to run-time identifiers (e.g. the address of a class at run-time).

- Run-time descriptors and instructions: RTTI and instructions are pre-linked as much as possible, and are suited for an execution with only a minimum amount of preparation.

- Binding information: As the format is not fully parse-able, the cap-file contains a relocation table identifying affected locations within the instruction section. The affected locations use indices into a constant pool as in standard Java to name the referenced items. The names used are then tokens or offsets.

- Verification information: The verification information is kept completely separated from the other components, and especially describes all RTTI items by location, size, name and signature. Additionally, a type pool describes all the signatures associated with the different constant-pool items. Names and signatures used are still token-based, and do not use the original symbolic names.
The JavaCard cap-file format allows for a compact code representation at run-time and a sufficient execution performance (Section 5.4.5). Although large parts of the run-time information are pre-linked, the binding mechanism is still flexible enough to hide most of the API implementation details from the client applications. Additionally, the cap-file and export files contain enough information for allowing a back conversion into class-files and/or an off-card verification of cap-files [116].

Weaknesses

The JavaCard format tries to address two contrary goals, minimum code-size and simple binding-scheme on the one hand, maximum flexibility and code verification on the other hand. The necessary compromises must then incur a number of weaknesses:

- **Machine dependency**: Most descriptors are fully pre-linked and predetermined, including classes and method-tables. All items are especially assumed to be addressable by 16-bit, hindering scaleable implementations. Proponents have argued that handles can be used to indirect the 16-bit-based access on large platforms, but they are typically used to refer to language objects for simple and efficient garbage collection schemes [86], and not to object-internal items as for instance a class descriptor within a package. As the package is loaded into one contiguous area, its internal items are not independent and valid heap items making their associations with handles inappropriate.

- **Code size and performance**: Run-time size and execution performance are sufficient (Section 5.4.5), but binding performance and final code-size could be better. Even if the verification components are left out at download time, the binding information in form of relocation tables and constant pool takes up more space than markers in case of a marker-based format. The constant pool must especially be temporarily stored which might be difficult on low-end smartcards. And finally, only the relocation tables can be processed on-the-fly, resulting in an additional single EEPROM write per location to be bound. Even worse, no management specification foresees the download of cap-files on low-end tokens without the inclusion of the verification information. In case of the VisaCash application, the verification components increase the minimum cap-file size by another 18% (Section 5.5.2). During the download, the additional information is then just discarded, as on-card verification is not supported.

- **Binary compatibility**: Mapping symbolic information to tokens is not loss-free. With names and signatures of virtual methods mapped to single table-indices, the JavaCard introduces a subtle binary compatibility problem, i.e. the extension of a class by a virtual method prevents an execution of all previously existing client applications. The implications on scaleable API-design have already been discussed (Section 4.3.3).
• Development process: The conversion process is inconvenient for issuers and programmers due to the necessity of export files. With different API versions, issuers have to take care to use correct export files when creating new cap-files, as irregular cap-files may lead to unpredictable code bindings during their download. With different export files already in use for the JavaCard 2.1 and 2.1.1 standards, this problem might even increase in the future.

5.5.4 Proposed Format

Assumptions

The JavaCard file-format lacks the required scaleability and portability properties. With verification information shipped in any case, it can be inter-weaved more directly with the remaining code. Additionally, offset- and token-based schemes should be given up in favor of a naming scheme closer to standard Java. The assumptions a new format can be built on are:

• ROM is cheap: The linker code is placed in ROM which is large and cheap. A more complex linking scheme can then be afforded as long as applications, finally residing in EEPROM, can be small.

• EEPROM writes are expensive: The linker can trade computation for memory as soon as the memory operations involve EEPROM writes.

• Applications do not cover many classes: The JavaCard system classes consist of many classes and interfaces, but stay in ROM. Thus, they can be optimized heavily during the ROM-image generation (Section 5.8). Downloaded applications typically consist just of one or two and rarely more classes, and the amount of necessary RTTI is then relatively small compared to the number of instructions.

• Application downloads are the normal case: The system should optimize for the download of applications where all symbolic information can be discarded afterwards. The download of libraries in the field is rare, and may then perform less efficient.

Details

A code format based on symbolic information and suitable for secure tokens should then consist of the following parts:

• Header: It names the package or application, and contains all required global information about the application and/or package code: version information, number of classes, number of static fields (primitive and reference type), number of exception handlers, and number of absolute references needed during execution. The last item allows to indirect the constant-pool references in the code to entries in a space-efficient execution pool which may contain run-time-specific references of arbitrary address width. In standard Java, these
run-time references are stored in the constant-pool itself, and the memory footprint is increased, due to many unused entries during the execution. After receiving the header, the linker must be able to preallocate all global structures (e.g. execution pool or package-global exception-table), which are filled in on-the-fly during the binding process. Additionally, the linker can setup a proper context-object in RAM, keeping the necessary temporary state during the download.

- Constant pool: The incoming constant-pool is processed on the fly and stored temporarily. Names and signatures of exported items should have been marked especially to directly store them in EEPROM, all other symbols remain in RAM, and are discarded afterwards. Names and signatures can be compressed or obfuscated, and/or individual constant pool entries can be optimized to keep their sizes down [131, 127, 162].

- RTTI: The following class, method and field descriptors are parsed, and partly distributed into either RAM or EEPROM where only the information required at execution time must be stored persistently. In case of an application download, a number of descriptors (e.g. private instance fields) and many associations between descriptors and their symbolic information can be kept temporarily in RAM, and be discarded after an installation. In case of a library, the exported descriptors are directly and persistently associated with their symbolic information which has already been loaded persistently with the constant pool. Libraries can then still act as link targets, while applications can rely on an amount of RTTI close to the bare minimum (Section 5.4.2). For instance, no field descriptors must be stored, and class and method descriptors must only be slightly increased to allow for a later verification and/or their lookup during the binding process. The run-time descriptors must be loaded in a strict order: interfaces before classes, super-classes before sub-classes, and especially field and method descriptors before their class descriptors. This allows for an on-the-fly binding of all descriptors and an on-the-fly construction of the run-time structures including all necessary method-tables. During the process, EEPROM locations must only be written once per associated code location. This especially includes the locations having to be bound during the link-process.

- Bytecode: The bytecode, i.e. instructions and exception handlers, must occur at the end. As all run-time descriptors have already been received, the instructions can be processed and bound on-the-fly. With the exception handler information in the package-header, all exception-tables in the package can be grouped together in one table allowing for similar code savings as in the JavaCard cap-file format [150].
5.5. GENERAL JAVA-BASED LINK-FORMAT

Discussion

The format places only minimal assumptions about the concrete target machine, all internal class, method, field and table descriptors can be chosen freely by the VM. Especially, the file-format does not assume a fixed address-width, the optional execution pool allows a space efficient, natural and well-performing deployment on high-end tokens. They are also properly addressed by the fact that all standard binary-compatibility rules of Java can be easily obeyed. Nevertheless, all binding and symbolic information can still be discarded in case of applications, and the final executable content on the token then may not differ significantly in size from the JavaCard. Of course, binding by symbolic names incurs additional run-time costs, especially as a linker on a low-end token will have to trade computation for memory, and will thus lack efficient symbol-hashing schemes. However, as long as APIs are not too complex, the number of necessary symbolic comparisons is still low. Additionally, the on-the-fly processing avoids any unnecessary EEPROM writes. Finally, the format places minimal constraints on the heap management, as both persistent and temporary data can be streamed into linearly growing heap segments, and the finally executed content can be kept in a single heap-unit in EEPROM.

Table 5.3 on the next page presents the costs which must be expected by a binding process with the given characteristics in comparison with one based on a JavaCard cap-file. The example is again the VisaCash application whose implementation consists of a single class in case of the IBM implementation [159]. Although the standard Java class-file does not comply with the described format, it can be used for gathering the necessary memory sizes. In a first step, all internally-used symbols in the VisaCash class-file have been compressed by an obfuscator. The original class-file size of 16406 bytes then already shrunk to 9655 bytes which is already close to the cap-file size of 7986 bytes. Especially, the class-file-format needs 960 bytes to store the static field values in the class initializer code, while the JavaCard cap-file needs only 201 bytes to present the data statically. With a static representation of the data, an optimized class-file already comes close in size to the JavaCard cap-file-format.

The persistent code size increased from 4587 bytes to 5481 bytes, an increase in the order of 20%. However, this is not due to the size increase of the runtime descriptors which is negligible as VisaCash is implemented in a single class. The VisaCash class-file contains the standard 32-bit-based instructions of standard Java compared to the compact 16-bit-based instructions of JavaCard in the cap-file. Its instruction-set not only increases performance, but also saves significantly space.

The proposed format requires more temporary resources than in case of JavaCard. In case of cap-files, only the constant-pool has to be stored temporarily, the relocation table is applied on-the-fly. In case of the proposed format, constant-pool, symbolic information, and additional class-, field-, and method-related information has to be stored temporarily, and is thus kept in RAM. The simulation assumes obfuscated symbols in an UTF8-representation, a constant-pool with fixed-sized entries, and results in the binding process temporarily requiring 1975 bytes. A lot more than necessary for cap-files, but still
Available on current high-end smartcards. Of course, a final format specification can foresee a better symbol compression scheme and save additional space [131, 127], but this might increase the computation needs even further. It might then be easier to just store the constant pool temporarily in EEPROM instead of RAM, slowing down the binding process, but allowing for significant resource savings in RAM.

### 5.6 Related Work

The described work is closely related to JavaCard whose instruction-set- and code-format deficiencies have been discussed throughout the chapter. Other smartcard systems rely either on unportable machine-code formats, or on insufficient virtual processors as in case of WindowsCard or MULTOS. WindowsCard suffers from an instruction-set based on a limited 8-bit-based processor and inhibits serious address-space limitations [60], MULTOS provides both an inconvenient block-based instruction-set and protection based on a software-MMU which must lead to performance drawbacks in case of interpretation [98]. Especially, their code-formats do not provide sufficient RTTI limiting any analysis of the executable content before its download. Deployment on larger platforms is hindered by their limited system-call interfaces not allowing for modularization and fine-grained extensions of system services.

Compact code-sizes have also been an important topic in other embedded environments adopting standard Java, and either compactifying the instruction-stream and/or the file-format has been proposed in the past. The first approach concentrates on compact code-sizes at run-time, and is based on keeping compressed application code in memory and just decompressing it at execution-time [13]. However, significant resource and computations needs makes software implementations less attractive, and clearly favor hardware implementations [163]. In general, compactifying established instruction-sets, as in case of the Thumb-instruction-set of ARM-processors for example, seem to be more preferred than introducing general compression schemes for code sequences [5, 95]. Compactifying instruction-sequences can still be interesting in case of secure tokens, and appropriate methods are introduced in Section 5.8.

Modifications or compressions of the standard Java file-format have mostly targeted faster download times, and approaches range from general compression schemes like Lempel-Ziv as in standard Java to class-file-specific compression schemes with higher compression rates [14, 64]. They all trade network band-
5.7. CODE VERIFICATION

width for client resources, where code-decompression requires temporary storage and especially significant processing costs on the client [95]. Other than that, they do not touch the expensive binding and execution process of standard Java (Section 5.3.1) making their deployment on secure tokens unattractive. Their concepts can still be partly reused for further refinements of the proposed file-format, a more resource-friendly representation of symbolic information for example (Section 5.5.2).

5.7 Code Verification

5.7.1 Basic Steps

Code verification incurs significant costs in a standard Java run-time environment [119], regarding both necessary processing as well as memory resources. The individual verification steps still differ in their complexity:

- Structural verification: This ensures that executable content is consistent and all of its elements are structured according to the file-format specification. In case of the proposed format, the necessary checks can be easily integrated into the parsing step required at download time. In case of JavaCard, they require significant computing and memory resources, due to the re-association of verification and code information and the resolution and verification of locations addressed by offsets.

- Descriptor checks: These ensure that all run-time descriptors such as classes, methods, and fields obey the language semantics, no class is permitted to inherit from a super-class declared final for example. These checks may involve significant computations, but only negligible additional memory.

- Binding checks: These verify that all named items, referenced within the code, are existent, and are of the correct type. These checks again involve significant computations, but no additional memory. Many lookup operations have to be performed whose performance depend on the layout the system chooses for the executable content. In case of JavaCard, the layout is mostly predetermined and not optimized for a fast lookup of descriptors by name.

- Instruction verification: This last step requires the largest portion of the necessary processing and memory resources. All instructions have to be checked to obey the safety rules implied by the language and bytecode. This step is based on an abstract code interpretation on types (i.e. a typeflow analysis) for each method [28].

5.7.2 Typeflow Analysis

The verifier must ensure that instructions always operate on the types they are specified for, independent of how they may be reached at run-time [54]. The
first rule ensures the referential integrity of the code, e.g., no integer can ever be treated as an object reference. The second rule implies that local variables and operand stack at each instruction have a predetermined, fixed and well-known layout and size. Again, the bytecode enforces a run-time property of the language where the number and types of the variables and the expression depth is always fixed [54].

The verification of a method starts with an initial state, traces all instructions on all possible control-flows, and its termination is ensured by certain break-conditions:

- Setup the verification: The verifier initializes the initial trace-state, i.e., program counter (i.e. zero) and type-frame. Type-frames store the types of the local variables and items on the operand stack during the analysis. At method entry, the initial type-frame is preset with the argument types of the method.

- For each instruction then do:
  - Check the type constraints of the current instruction, and apply its effects on the current type-frame. An instruction like `aload0`, for instance, asserts the first local-variable in the type-frame to be of a reference type, and then loads this type on the operand stack.
  - If the analysis did not reach this instruction so far, save the current type-frame, and pick the next instruction to trace. If the current instruction leaves the method, break from the current execution path. If it is a control-flow instruction like `if`, follow all possible execution paths in parallel. Otherwise, take the instruction immediately following the current instruction.
  - If the analysis reached this instruction before, check and unify the current state with the previously saved state.\(^1\) If the current state is already fully contained in the previously saved state, terminate the analysis of this execution path. If not, repeat the analysis of the path after this instruction, but with the unified type-frame as starting point.

- the verification terminates if an error occurred or all paths have been traced successfully.

Figure 5.6 presents an example for a simple typeflow analysis. All instructions operate on integer types, either on the operand stack or in the local variable table. The `if_ icmplt` instruction leads to the trace of two execution paths, the `ireturn` instructions to the termination of the analysis. As the different execution paths do not merge, individual type frames do not have to be merged, and no execution path must be traced multiple times.

\(^1\)The unified type-state adds the types of the previously saved state to the types of the current type-frame. Assuming, a local variable is of type A in the previous frame, and of type B in the current frame, both types, A and B, are saved in the unified frame for the given local variable. Of course, if type A and B do not properly relate to each other, the repeated verification of the path will fail.
5.7. CODE VERIFICATION

Figure 5.6: Sample typeflow analysis for a simple method. Its source code, bytecode as well as the resulting type-frames are given.

Figure 5.7 presents a code sample which details the need for type unification at branch targets and repeated tracing of execution paths. If \( B \) is assumed to be a subtype of \( A \) in the example, block 3 may have to be traced twice dependent on by which execution path it has been reached first:

- block 2 is traced before block 1: after tracing block 2, the initial type-frame of block 3 contains the type \( B \) for the variable \( a \). After tracing block 1, its last frame records the type \( A \) for the variable \( a \). Even when block 3 has been verified before to be type-compliant with \( a \) being of type \( B \), e.g. all accessed fields are defined in class \( B \), it does not guarantee that the same is true for \( a \) being of type \( A \). Thus, block 3 must be traced a second time, this time assuming \( a \) being of type \( A \).

- block 1 is traced before block 2: after tracing block 1, the initial type-frame of block 3 contains the type \( A \) for the variable \( a \). After tracing block 2, its last frame records the type \( B \) for the variable \( a \). In this case, the frame unification does not have to enforce a second trace of block 3. As \( B \) is a subtype of \( A \), any \( a \) of type \( B \) conforms to the type constraints of block 3 having been checked before for an \( a \) being of type \( A \).

Note that Figure 5.7 presents Java source code instead of bytecode. Of course, java source compilers expect all variables to be declared before they are used the first time, and the type of \( a \) is thus always well-defined in the example. In
case 5: // code block 1
a = new A();
break;
case 10: // code block 2
a = new B();
break;
...
// code block 3
a.doIt();
...

Figure 5.7: Example for necessary unification of type-frames

case of bytecode, the types of local variables and intermediate operands on the operand stack are not pregiven, but must be fully recovered by the typeflow-analysis. This for instance allows for reusing local variables for different types, and does not require to name temporary operands.

Although the type-frame transformations for the most common bytecode instructions can be implemented rather simply and efficiently, the total typeflow-analysis requires significant computing and memory resources due to succeeding in execution order, type-frame unification, recursion and repetition. First, the verifier has to follow all possible execution paths at branches in parallel, i.e. it has to keep a list of all outstanding paths not having been traced yet. A single tableswitch instruction and/or nested control-flow instruction can easily lead to a rather large history having to be maintained. Additionally, the verifier has to remember the type-frames for all branch targets that have been computed before. It cannot discard them during the analysis as any trace of an execution path may have to be repeated at a later time. In the worst case, a branch instruction at the end of a method can lead to a complete re-computation of all other, previously analyzed execution paths. Thus, both the required computing as well as necessary memory resources are unknown at the beginning of code verification.

In case of the previously described VisaCash application, the most complex method contains 82 labels, and keeps at the maximum 9 local variables and 10 stack operands. Even if an extremely space-efficient and compute-intensive implementation with only 3 bytes per type-frame entry is used, the temporary information required to save all type-frames needs at least 4674 bytes (82 * 19 * 3). Additionally, one program counter per label and the information about outstanding paths must be stored. A successful verification of VisaCash thus seems unlikely even on next generation smartcards, more complex applications are unlikely to be verifiable on-card even on large, future smartcards.

5.7.3 Verification-friendly Bytecode Conversions

Processing and memory footprint can be optimized by placing stronger semantics on the bytecode and its instructions regarding code verification. Especially, code conversion can be used to rearrange and insert instructions allowing for
tightening verification semantics and allowing for a more efficient deployment on the token. A reasonable scheme can be based on the following optimizations:

1. Simplify type-frame transformations of individual instructions

These especially cover the `jsr` and `new` instruction which do not fit the previously described instruction-local transformations, but require additional state and processing by the verifier. The instruction `jsr` is used to call bytecode subroutines which are translated from `finally-/synchronized`-blocks in Java source code. To save code space, subroutines can be called from different locations with different type constraints. The constraints regarding branch verification are then weakened: type-frames of calling instructions do not have to match, but can differ under certain conditions [139]. This has significant consequences on the verification complexity [119]. Larger systems propose to duplicate the subroutine at every caller site, and thus fully remove them at download time [1]. In case of secure tokens, the converter can fulfill this task before download, especially as the resulting increase in code size is negligible. Although `finally-/synchronized`-blocks are used frequently for error handling in many Java environments [48], they rarely occur on secure tokens as their applications typically do not try to recover from errors, but just notify the terminal.

The converter can also simplify the compliance with standard Java expecting the first call on a newly created instance to be a constructor. This guarantees the proper initialization of objects, and simplifies the safe programming of classes [49]. Between allocation and constructor invocation, a more or less arbitrary sequence of instructions is allowed. Thus, the verifier has to keep state about the newly created object, and check any access accordingly. If the system does not internally access objects allocated by applications (Section 4.3.2), this check can be completely skipped by the verifier. However, the converter can also group the separate allocation and invocation instructions into a single instruction which creates an instance and immediately calls the specified constructor with the arguments pushed previously on the stack. No additional state has to be kept during the verification, and the type-frame transformation associated with the `new` instruction can be performed in a single step.

2. Optimize type-frame sizes

The verifier has to keep all type-frames at branch targets during the analysis where a type-frame consists of both local variable and operand stack types. Java source compilers typically do not optimize the allocation of the stack operands, and especially generate code mostly leaving the operand stack empty at branch times. If this can be enforced in general for the bytecode, verification gets simplified and memory is saved. In case of VisaCash for example, the most complex method is annotated in the class file to require 9 local variables and 10 stack operands at the maximum whereas only a few branch instructions exist expecting any operand on the stack.

The converter can easily transform standard bytecode to satisfy the more strict condition. At each branch source, it can insert instructions saving any
existing stack operand in a local variable, and at the branch target, it can insert instructions loading the associated local variables back on the stack. In case of the standard Java compiler javac, only translations of the \(?\)-operator (i.e. cond ? expr1 : expr2) lead to the described bytecode pattern, and then only with one stack operand pushed at branch time. As the \(?\)-operator occurs rather rarely in application sources, the increase of the final code size is negligible in any case.

(3) Optimize type-frame unifications

Inserting additional instructions can also be used to prevent the repeated analysis of execution paths. This is necessary whenever a type-frame at a branch destination changes due to its unification with a type-frame from a branch source. As shown in the previous example (Figure 5.7), this especially results from lacking type declarations for stack operands and local variables. As the previous optimization prevents any live stack operand at branch time, only differing local variable types in different execution paths may enforce a type-frame unification and repeated analysis. In standard Java environments, instructions have to be traced about two times on average, and repetition is thus quite common. In case of secure tokens, the number of available types and complexity of type hierarchies are much more limited. A repeated analysis of an execution path is then much more unlikely, in case of the VisaCash application for example, almost all execution paths have to be traced only once.

If all local variables, their types and ranges are declared in the bytecode as in the original source, unifications at branch targets can be avoided, and a simple comparison of the frame at the caller and callee site can verify the validity of the code. However, this information takes up a significant amount of space and is also not fully required. Most local variable types are already correctly recovered during the first pass of most execution paths. Only the execution paths must be annotated which otherwise wrongly identify the type of a local variable before a branch. A converter can for instance insert special instructions at the affected locations signaling the expected type or subtype relationship to the verifier.

In case of the presented example (5.7), block 2 must be annotated with an instruction casting a from type B to A which is expected by the successor block 3. If this instruction is missing, the frames resulting from tracing block 1 and block 2 differ, and the verification fails. If the code tries to cheat with the type of a, the verification of block 3 fails. If block 3 expects a being of type A, verification only succeeds if both block 1 and block 2 result in a local variable a of type A.

(4) Optimize number of type frames

Type-frames now only have to be compared at branch targets, type-frame unification is avoided and thus, no paths have to be analyzed repeatedly. Especially, the type-frame of a branch target does not have to be saved during the whole analysis anymore. Instead, it can be discarded as soon as all of its branch sources have been reached, and their type-frames have been successfully checked.

This may for instance be exploited by the following two-pass implementation
of the verifier. In the first pass, the verifier records all branch targets and their reference count, i.e. how often they are called within the method. In the second pass, the typeflow analysis is fulfilled. Whenever the verifier reaches a branch target, it verifies target and source frame, and decrements its reference count. As soon as it drops to zero, the frame can be discarded or reused.

The number of live branch targets at a time is rather low. In case of the VisaCash applet for example, there do not have to be more than 16 type-frames active at the same time. The exact number depends on which execution paths the verifier follows whenever it reaches a branch. As the bytecode structure closely relates to the original Java source code structure, the verifier should try to follow the paths which are close to the paths it has already traced [1]. This increases the chance to discard type-frames as early as possible. Most branches and branch targets especially result from simple if statements which can then be easily resolved.

VisaCash Sample Study

In case of the VisaCash application, the verifier has to keep a reference table for 82 branch targets in total. Additionally, it has to keep the stack of outstanding paths during the analysis. For each path, it has to remember its program counter, and for all paths originating from the same source, it has to save a copy of its initial type-frame. Furthermore, it must save the type-frames of all live branch targets, at the maximum 16, and a single type-frame for the transformation and inspection of the instructions on the current trace path.

The reference table can be stored in 246 bytes (82 * 3), a single type-frame with 9 local variable types in 27 bytes. Even if we assume an implementation where each outstanding path saves its own copy of the associated type-frame, the verifier must not save more 20 + 16 + 1 = 37 type-frames or 999 bytes during the analysis. The verification in total then requires around 1.2KB, far less than the originally estimated 4.6KB.

5.7.4 Proof-Carrying-Code (PCC)

Code verification based on PCC simplify the task of the on-token verifier [108], and has also been proposed in the context of JavaCard [132, 58]. A typeflow analysis is already performed off-token at compilation time, and the precomputed, unified type-frames for all branch targets are shipped together with the bytecode to the token. The bytecode may remain unchanged, and is only annotated with local variable and stack operand types. They must then not be recovered, but only checked on the token. Neither cheating on the code nor on the type annotations allow for circumventing the standard Java code security.

PCC especially allows for tracing the individual blocks independently from each other. Consequently, the bytecode may not be traced in execution order, but in linear order, simplifying the task of the verifier significantly. The verifier then starts with a single, initial type-frame at the first instruction, steps linearly though the bytecode, applies the transformation on the type-frame dependent
on the current instruction, and eventually compares it with a precomputed type-
frame shipped for a branch target.

5.7.5 Discussion

The implementation of both the off-token as well as on-token part is much simpler in case of PCC. Off-token, only a conventional typeflow analysis has to be performed whereas in case of the described on-token scheme, the locations for instructions annotating local variable types additionally have to be searched. Furthermore, its on-token verifier still has to trace the bytecode in execution order. This is necessary as the bytecode allows the reuse of slots in the local variable table within a method. If the described on-token verifier does not obey the execution order, it cannot correctly identify changes in the assignment of local variable types. The converter could rename and reassign local variables to ensure their uniqueness, but then increases the sizes of the local variable tables. Dependent on the invocation depth at run-time, this might have severe impact on the memory footprint of the interpreter.

PCC significantly increases the code size to be transferred at download and verification time. In case of VisaCash for example, the type-frames for 180 branch targets with 10 slots in average have to be transmitted. Assuming 2 bytes per item plus 2 bytes per program counter, this accumulates to 3960 bytes into total. In the simplest case, the precomputed type-frames are downloaded at once with the bytecode. Due to their size, they have to be stored in EEPROM during the verification. The performance advantage of PCC might then get lost due to the increased code size, slower download time, and EEPROM write penalty.

Performance can be improved by exploiting the fact that methods can be verified independently from each other. In case of the described file-format, the bytecode instructions of a method can be interleaved with their type-frames, and their verification can then succeed on the fly. The remaining increase of communication costs should easily be outweighed by the computation and memory savings. This scheme might still not be applicable for all download scenarios as verifying a method still takes a certain processing time. In case of smartcards for instance, broadcast environments place certain timing constraints which then might not allow for an on-the-fly verification. Without changes of the communication protocols, code verification must then succeed after the download in a separate pass.

5.8 Code Optimizations

Code conversion also allows for reducing the size of executable contents and/or increasing the execution speed. In case of stack processors, optimizations resulting in smaller code size are very likely to increase the execution performance as well as shorter instruction sequences result in less decoding effort during the interpretation. The possible optimizations and their impact depend on the target, either a ROM-image or the portable file-format. At ROM-image generation time, the converter will take a number of code libraries and applications, and
pre-link them completely for the execution-in-place after the token production. The converter has thus a global view on the system, can tailor the image specifically to the execution engine, and must not obey the portability and verification constraints of the standard file-format. In this case, the converter cannot make assumptions about the implementation of the referenced APIs, and possible code savings and performance improvements are thus more limited. In contrast, a ROM image generator can even help to structure the system efficiently.

**System-Packages Aggregation**

Any Java-based specification will separate individual system services into different code packages, and will foresee additional packages for system add-ons or extensions. While the package interfaces are specified for the needs of the applications, their implementation and internal structure is up to the system implementor where the different, especially the core system packages are likely to depend on common functionality and/or common state. On larger systems, a system implementor is likely to provide the common functionality in a separate package invisible for applications and internal to the system. The system then requires additional classes and incurs indirections on many accesses.

However, the additional step of a ROM-image generation allows for an adaptation of an optimized system implementation to the API interfaces it has to offer at run-time. The whole system and all API classes can be implemented in one single package where all items can be accessed directly and dependent on the needs of the system implementor. The ROM-image generator can reorganize the single package into the different API packages, and thus enforce their interfaces and access rules for all client applications. The efficient and compact internal structure and implementation remains. The main task of the converter is then mostly based on the renaming of the individual package items such as classes, fields and methods [93] and on the proper layout of the final ROM structures. The result is maximum freedom for the system implementor and compact system representation at run-time.

**Package-Global Optimizations**

Many other optimizations can be built upon two inherent system features: the global view on all classes due to the closed-package concept allows for a class-hierarchy analysis [32], and the typeflow analysis allows for the observation of the program-flow. The information can then be used for global optimizations, such as optimizing virtual method invocations, in-lining of methods, and removal of unreached and in-lined methods.

Table 5.4 on the following page shows sizes and possible savings for different code packages. The package java.lang is an example for an API implementation with many classes, but few bytecode instructions. Second, the VisaCash application shows the typical ROM properties of smartcard applications with only few classes and large portions of instructions. The third column presents numbers for the IBM JavaCard implementation [83], including APIs, system, and Visa Open Platform management services. Finally, the fourth column shows the
numbers for a final ROM-image, including four typical applications, VisaCash, PSE, VSDC and PKCS#15 [159, 38, 160, 135]. The first rows show the number of classes and methods in the packages, the original class-file sizes, and their final sizes in ROM, separated for RTTI and instructions. The ROM-image sizes are always small compared to the original class-files, due to the removal of binding information and the huge redundancy across the class-files in a package. For instance, the class-files in the java.lang package are three times larger than the single VisaCash class, despite their few instructions. Inside the ROM-image, the VisaCash package takes up much more space, due to the many more instructions. In general, the ratio of RTTI and code instructions decreases significantly with more applications added to the ROM-image and/or large parts of the system implemented in Java.

Optimizing virtual method invocations also helps to decrease the necessary run-time information. The converter can determine which virtual methods are run-time final, i.e. do not override any other method, and can replace their virtual invocation by a static one. The converter can then compress the class structures and/or method-tables where optimized virtual methods do not have to be referenced anymore. In case of java.lang this helps to save 22 bytes, i.e. around 20% of the final run-time descriptors. Of course, as soon as applications like VisaCash are examined, these savings are negligible. However, they can still benefit from the performance improvements as 87% of the virtual method invocations in the final image can be replaced by static method invocations. This number is much larger than the average number observed for standard Java applications [152] which results from the fact that secure token APIs and applications are less complex, and do rarely depend on the reimplementation of virtual methods. A similar observation can be revealed by a look at the methods in-lined and/or removed. In case of the IBM JavaCard implementation,
most methods being in-lined are constructors and a few delegating methods (i.e. methods immediately forwarding their invocation). Otherwise, the observed applications implement their functionality in few classes and few large methods, and do thus not provide a good base for successful method in-lining. The results again differ from the higher numbers measured for such optimizations in standard Java environments [130, 34].

Of course, the numbers always depend on programming style and skills of the application developer. The given JavaCard system and applications have been manually optimized a lot, and provide limited room for the described optimizations. In any case, the global class and program flow analysis allows for the computation of a number of important statistics, including the call graph of an application and a precalculation of the worst-case stack usage by an application.

**Package-Internal, Instruction-Sequence Optimizations**

All optimizations so far depend on an inter-procedural analysis [2]. An intra-procedural analysis can directly address the occurrences and ordering of instructions to achieve a better code density. A number of ad-hoc optimizations are simple to implement, and allow for better code quality with minimum effort. They include for instance the introduction of new instructions specific to ROM images, the removal of instructions only required for platform-independent verification, and cleanups of the original bytecode generated by the Java compiler, for instance by peephole optimizations [101]. They make use of fixed bytecode patterns, and allow for code savings in the range of a few percent. In contrast to register machines, general and efficient heuristics for an optimized instruction scheduling regarding performance and/or size are still unknown for stack processors [16, 97]. Algorithms for the block-local scheduling of instructions exist, but their impact especially for interpreters is limited [90, 124], more efficient algorithms operating on whole methods are questioned [90].

More promising is a brute-force analysis for macros, i.e. instruction sequences shared across all methods [23]. The sequence may only be stored once in the final ROM image, and all original occurrences may be replaced by a special instruction indicating a call of the referenced macro. As a macro might be called from any method, the standard jsr instruction to invoke method-internal subroutines cannot be used [96]. Instead, the macro invocation and especially the tracking of the return address must be supported explicitly by the run-time. Table 5.5 on the next page shows a few of the sequences occurring most often in the previous ROM-image consisting of system and four applications. The maximum savings reached with 47 macros in total is then around 1.6KB or 5% of the final image size which is much less than claimed in the literature [24]. The reason is, on the one hand, that the JavaCard instruction-set is more compact than the standard Java instruction-set reducing the possible savings by macros. On the other hand, the described macro analysis looks only at the portable class-file instructions, and not at the instructions having to be shared at run-time. For instance, the first macro in the table is based on an invoke instruction addressing a specific slot and not a specific name and signature. The given instruction is then actually more likely to be sharable across many
sequences than in the platform-neutral representation.

The referenced publication correctly states that only basic blocks are worth to be scanned for macros (i.e. no control-flow instructions must be checked for macro inclusions), and that parametrized macros (i.e. macros where each caller can pass arguments to the macro) are too complex to be efficiently implemented. Indeed, proper exception handling and additional frame state required for macro invocactions make implementations on 8-bit smartcard chips already difficult enough.

Table 5.5: Sample macros, their occurrences and savings per macro. The numbers assume a call-macro instruction with three bytes in size and an additional trailing byte per macro.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Occurrences</th>
<th>Savings</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>aload0, aload1, sipush 26368, invoke #3</td>
<td>17</td>
<td>76</td>
<td>8</td>
</tr>
<tr>
<td>invokestatic arraycopy(), pop</td>
<td>70</td>
<td>65</td>
<td>4</td>
</tr>
<tr>
<td>iload2, invokestatic throwIt()</td>
<td>26</td>
<td>46</td>
<td>5</td>
</tr>
<tr>
<td>baload, sipush 255, iand</td>
<td>26</td>
<td>46</td>
<td>5</td>
</tr>
</tbody>
</table>

5.9 Conclusion

The chapter defines a number of basic execution models applicable for secure tokens, and shows the clear need for an on-token, extensible model. On the one hand, code linking must be fully performed on the token and independently of the remote host the code is downloaded from. This allows for application deployment in all different token environments including broadcast networks, and emphasizes the fundamental independence of secure tokens despite all of their constraints. On the other hand, the execution model must offer flexibility regarding virtual processor and APIs. It must allow for coarse-grained virtual-processor extensions such as 32-bit-integer arithmetic and code verification on larger tokens, and fine-grained API extensions without sacrificing portability.

The thesis ruled out execution models incorporating hardware/software MMUs, but also an unmodified deployment of standard Java. Its dynamic execution model and open-package concept do not fit, and additionally incur significant trade-offs regarding code size, execution performance and resources needs. At least, its bytecode contains a high degree of type information, and can be effectively used as a source for conversion to other formats.

Although deficiencies of Java’s class-file format have led to many new proposals in other environments [89, 131, 14], adaptations have been refused so far to avoid in-portabilities across different Java environments. It has always been argued that all applications targeting smaller controllers and devices should still be directly executable on larger tokens. However, it is already questionable whether an application optimized for a small screen on a mobile phone will be sufficient and not replaced on a large desktop system. Especially, security-sensitive software written for non-interactive deployment on secure tokens will never be effectively useable on standard Java environments. Thus, code conver-
sion is natural, and should then foresee a format fully satisfying a wide range of secure tokens.

The chapter presented the properties of a new Java-based instruction-set applicable for secure tokens, and has shown its performance characteristics. The downscaling of the standard Java instruction-set to the targeted 8-bit, 16-bit and 32-bit processors can be easily fulfilled, and the performance impact by interpretation is low. This contrasts larger platforms where code must be compiled, and results from the unique application and hardware characteristics of secure tokens. JavaCard shares the presented instruction-set philosophy, but fails to deliver all other goals of the proposed model. Although JavaCard follows an extensible execution model with optional 32-bit arithmetic on larger smartcards, it does not further exploit its base concept. Neither does it fully address the need for API extensions and evolution, nor for code verification. Its file-format limits machine independence and binary compatibility. Applications are hardly portable across smartcards offering the base APIs and larger tokens additionally featuring API extensions.

The chapter proposed a new file-format offering machine independence, improved binding properties and a clear path towards code verification and deployment on 32-bit platforms with larger address-spaces. It is based on a closed-package concept as in case of JavaCard, but reintroduces symbolic information to achieve its goals. With compressed symbolic information, strict ordering, execution pool and optimized EEPROM handling, it can still provide compact code/run-time sizes and fast download times. Nevertheless, it does not only provide application portability across different vendor-specific API implementations, but also supports fine-grained, token-specific API extensions on larger or legacy-restricted tokens. Finally, both instruction-set and link-format allow for code verification on larger secure tokens or next generation smartcards. The integration of proof-carrying-code (PCC) has been discussed as well as a number of code conversions have been presented to optimize a standard bytecode verification scheme. PCC trades bandwidth for computation, significantly simplifies the on-token verifier and results in large resource savings. The described adaptation of the standard verification scheme also results in resource savings, although by far not at the same level as in case of PCC.

JavaCard tries to deliver a portable executable format executable on low-end smartcards and verifiable on next generation smartcards. On the one hand, this significantly increases resource needs compared to the proposed marker-based format (Section 5.5.2). On the other hand, the offset-based format and its standard Java verification model hinder verification even on large mobile tokens. This especially hurts as the download of JavaCard applications on extremely resource-limited smartcards still does not take place in environments where a portable and verifiable file-format is needed. These low-end tokens can be more efficiently supported by an off-token execution model and/or the described marker-based file-format. All current state-of-the-art smartcards, future smartcards and other large mobile tokens can be supported by the proposed format.
Chapter 6

Memory Model

6.1 Introduction

The memory model for secure tokens must fit the characteristics of the unique memory hierarchy available on secure tokens (Section 2.2). The available ROM can be used in a well-known manner, and store the operating system including device drivers, execution engine, APIs, and eventually a number of preinstalled applications and their read-only data. The persistent store in form of memory like EEPROM and its access characteristics of fast reads and slow writes differ from the persistent storage technologies used in other environments. Additionally, both ratio and limited sizes of persistent and transient memory significantly contrast general computing environments.

The system must allow for a convenient and efficient manipulation of both stores by applications. On the one hand, they have to be able to use the available RAM for security-sensitive, temporary data and performance-critical computations. On the other hand, they have to manipulate their long-term data in the persistent store. Updates of persistent data must succeed reliably, even in case of a sudden power-loss. Especially in case of mobile tokens, a sudden power-loss may occur regularly, which must immediately lead to systems and applications being able to rely on transaction mechanisms when updating persistent data.

The application needs must be satisfied within the general secure token constraints (Section 3.2). First, the memory model must be fully exploitable by a simple and convenient programming model. Second, it must scale across different applications, i.e. allow for applications being able to use effectively the limited memory resources on low-end tokens or to exploit all of the available resources on current high-end and future tokens. Third, the proposed memory and transaction model must be simple and resource efficiently to implement, especially on low-end tokens. Finally, the model must allow for a safe implementation and secure execution of applications.

In case of Java-based tokens, the safe, but expensive Java programming model must be adopted without affecting the original Java philosophy and programmer feel: the notion of persistent data and transactional guarantees must be introduced in the Java environment, but all safety features (e.g. automatic memory control in form of garbage collection) must be retained even on ex-
tremely resource limited smartcards.

6.2 Transient and Persistent Environment

6.2.1 Persistent Systems

Object Lifetimes

On almost every computing platform, applications manipulate data with different lifetimes. Short lifetimes of items like local-variables are convenient, intentional, and typically not a major concern for programmers, in contrast to the data which has to survive individual application executions. On larger platforms, a non-trivial part of an application may be engaged with serializing long-term data before exiting and deserializing the persistent state after an application restart. In case of secure tokens, an application is invoked after its installation and during its communication with external terminal applications (Section 4.3.2). On the one hand, it has to manipulate short-term and temporary data only required during and for the communication. On the other hand, it can be expected to update data which has to survive individual communications and might even have to persist across the whole application life cycle. In general, the lifetime of objects fall among the following categories [10]:

1. Transient (temporary) results in expression evaluations and local-variables in procedure activations: Data in this category resides in the individual interpreted bytecode frames, and only lives as long as the enclosed method is activated. The individual items are primitive entities in the proposed instruction set (Section 5.4). In contrast, some languages foresee complex objects explicitly allocated on the stack. However, resource constraints, limited stack sizes, and security and safety implications [51] make their use in case of secure tokens unattractive.

2. Instance variables, class variables and heap items whose extent is different from their scope: In case of secure tokens, they include the items stored in the persistent store, but also objects residing temporarily in RAM. In contrast to the persistent items, they are only accessible during a communication and may not survive a power loss. They typically encapsulate the state of the communication, and include session keys or the communication buffer for example.

3. Data that exists between two program executions: In case of secure tokens, the persistent set of objects must survive individual token sessions, and must be available the next time a communication is setup. The necessary transient data may be allocated as soon as an application is invoked, and may persist as long as the application session takes place. Dependent on the system, it might even survive several, distinct communication sessions with an application during a single token session.

4. Data that exists between different versions of a program or data that outlives the program: Some persistent systems foresee mechanisms like
schema evolution for the explicit support of program updates and data migration across different program versions [11]. With the limited resources of secure tokens, the system may just rely on the upgrade-strategy typically deployed on larger systems: if a token application is deleted and its associated data must be preserved, a terminal should set up a special communication with it, archive its data before it is destroyed, and use its data for the initialization of the new version.

Object Placement

A look at the practices on larger systems leads to three initial approaches for the placement and management of objects:

1. objects are instantiated in RAM and are serialized into EEPROM.
2. object instantiation always in EEPROM.
3. object instantiation in RAM and EEPROM.

The first possibility resembles the model in the regular Java environment. The application freely instantiates objects in RAM, and stores their data with the help of specific API functions into the long-term storage, for instance a file system. The serialization and deserialization of its state is fully implemented by the application itself. A deployment on tokens seems unattractive due to two reasons. First, the working set of an application could be larger than the available RAM, especially on low-end tokens. Second, the underlying operating system must provide additional means for managing the long-term storage, i.e. it must support the binding of names with serialized state and an enforcement of access rights.

If objects are instantiated in EEPROM, the language protection rules can be used to enforce the integrity of application and system objects in the persistent heap. If all entities are encapsulated in regular objects, a uniform view can be established, and the number of system mechanisms can be reduced. Furthermore, as secure token applications are mostly interested in manipulating persistent data, its first-class representation in terms of language entities allows for a convenient programming model.

However, performing all allocations and deallocations in EEPROM will never be feasible: write operations to EEPROM are just too slow, their number is physically limited, and it must be possible to place security-sensitive data in RAM for an immediate destruction in case of power-loss. The additional instantiation of objects in RAM then saves space in EEPROM, increases performance, and allows for a fine-grained control of security-sensitive objects as session keys. Other temporary data include execution contexts, stack frames, and system-internal data which must in any case be stored in RAM for sufficient performance.
Persistent System Properties

For programming convenience, resource utilization and safety reasons, the token environment must thus support both the manipulation of transient and persistent objects with Java language expressions. Such transparent access is one of the main reasons for the development of persistent programming languages and environments [10] which try to simplify the handling of persistent data as much as possible. Instead of forcing applications to spend significant efforts on the input and output of persistent data, applications should concentrate on their computations and internal data representations, and should be able to identify and update persistent data with simple language expressions. The persistent store should then be managed by the system as transparent as possible to the applications.

The many proposed persistent systems can be classified regarding the programming comfort they provide [8]. First, they offer different degrees of transparency, e.g. how similar language expressions manipulating persistent data appear to expressions operating on data with shorter lifetimes. Second, they may differ in the degree of orthogonality: full orthogonality expresses the demand that any instance can be persistent independent of its type, and that its lifetime may not be expressed at instantiation time. Finally, all systems must make sure that the lifetime of data can be easily expressed by the programmer, and persistent data can be easily identified by a simple and consistent mechanism (persistence identification).

6.2.2 General Approaches

Orthogonal Persistent Systems

An orthogonal persistent systems offers the highest degree of transparency and orthogonality, and is chosen as the basic architecture for adding persistence to Java in the PJama project [6]. PJama identifies persistent objects by their reachability from a persistent root. New objects are always instantiated independent of their types in RAM, and can then be dynamically marked persistent or transient by either assigning them to or removing them from the tree of persistent objects. Every now and then, the persistent tree is lazily stabilized into the persistent store [9].

The approach incurs a number of implementation problems, especially on low-end tokens. With the working set of applications (i.e. its persistent and transient data) easily exceeding the total RAM, the available transient space can only act as a cache for persistent objects which must be managed completely and transparently by the system. An efficient caching scheme satisfying all application patterns is already difficult to implement on large systems [27], on small tokens it is practically impossible. The implementation complexity is additionally increased by the fact that any lazy update of the persistent store must occur transactional to guarantee its consistency in case of sudden power-losses.

The dynamic model of PJama fits the dynamic execution model of Java [7], but then also requires the large resources of standard Java environments. The
implied system activities make predictions and strong assumptions by applications difficult. Especially, programmers lose the fine-grained control over the transient and persistent store questioning the whole implementation effort in the first place.

To counter inefficiency and provide more control to applications, other persistent systems limit the degree of one or more goals of persistent systems, e.g. restrict the available transparency or orthogonality. Instead of assigning persistence dynamically, most persistent systems foresee a static and final persistence specification of data either at declaration or instantiation time [37]. The proposed models then still differ strongly, ranging from well-known type-based persistent systems to the inconvenient transient-data model in JavaCard.

Type-based Persistence

A common approach to introduce long-term persistence in statically typed languages is to tightly connect the persistence of objects to their types [22]. Long-living objects have to be instances of classes inheriting from a specific super-class or implementing a specific interface. Indeed, the original proposal for JavaCard advocated a Persistence interface where instances of implementing classes would be allocated in EEPROM, instances of all other classes in RAM. A similar approach proposed the introduction of a Local interface for marking class instances to be allocated in RAM.

However, type-based persistence incurs two general problems. First, it allows for persistent objects freely referencing transient objects and vice versa. This then leads to the problem of dangling pointers in case of a sudden power-loss. Although references to transient objects still exist in persistent objects, the transient objects themselves disappear at the time of a power-loss. The next time the token is used, the system must thus scan the persistent objects for the affected locations and reset them. With the performance penalty of the necessary EEPROM writes, this cannot be afforded, especially not on smartcards with their strict timing constraints at reset time [73]. The second problem arises from the fact that the connection between type and persistence must lead to a duplication of the class hierarchy. Every type having to support both lifetimes must be implemented twice, both for transient and persistent instances. Even worse, important built-in types like arrays are not extensible, can thus only support one lifetime at all, and are then suddenly of limited use.

Many proposals extend the language with additional rules or keywords to avoid the described problems and/or to allow for efficient compilation schemes to machine code [67, 22, 137]. However, language extensions and add-ons require additional programmer education having to be avoided in case of Java-based systems [37]. Changes to the type system must additionally be propagated to the final executable code [66] which might then impact significantly the execution engine and verification process. In the worst case, instructions may place not only type but also persistence constraints on their arguments, making an already complex code verification even more difficult.

The Java programming language only provides the keyword transient for declaring fields not to be part of the serialized state of their encapsulating ob-
jects. PJama makes use of this definition by expecting transient fields not to be part of the persistent state of objects. In case of JavaCard, it has been proposed that objects referenced by transient fields should be transient, all others persistent. The transient keyword is thus used as a hint for the system which objects to keep in RAM and which not. However, this does not avoid the need for scanning all persistent objects at reset time. Additionally, the transient keyword lacks expressiveness. When an object is referenced by both a transient and persistent field at run-time, its lifetime becomes unclear.

6.2.3 JavaCard: Transient Data

Approach

The memory model described in the JavaCard 2.1 specification tries to avoid the described deficiencies and is based on the following goals and assumptions:

- Object allocations are by default persistent: An application allocates its root-instance and all other objects in the persistent store at installation time. The root instance is signaled by a method invocation of every message being received during the communication with a terminal.

- No changes to the notion of lifetime in standard Java: An object is alive as long as it is reachable by a garbage collector [81]. Although garbage collection is only optional in the JavaCard specification (Section 6.2.5), changes to the Java semantics are avoided for future compatibility.

- Static reservation model: As garbage collection cannot be expected and manual memory-release affects the system and application safety, applications are considered to allocate their objects at installation time and to reuse them throughout their lifetime. As objects cannot be expected to be reclaimed, further allocations just fill up the available EEPROM.

- Objects are persistent, only data may be transient: all objects are allocated at installation time and are persistent. Consequently, the only way to allow for the exploitation of the transient space is to associate persistent objects with transient data: persistent objects may reserve transient space for their data, and each data access is then guarded accordingly by the system. As many persistent reservations may quickly fill up the available transient space, the JavaCard specification allows for a limited reuse of the transient space by foreseeing different, fixed lifetimes for transient data.

Details

The new-instruction will always lead to a default object allocation in the persistent store. The reservation of transient space is offered by a number of factory methods creating persistent arrays associated with transient data, (e.g. `makeTransientBooleanArray`, `makeTransientByteArray` in the `JCSystem` class). They reserve a given amount of storage in the transient space, and allocate an array
in the persistent store which identifies location and size of the reserved RAM area.

As all applications tend to require transient data, the available transient space fills quickly up as soon as a number of applications are installed. If the lifetime for the reservation of the transient data can be significantly reduced, different applications can share parts of the transient space. The JavaCard specification thus foresees two lifetimes for transient arrays, clear-on-deselect and clear-on-reset. The lifetime is specified at instantiation time, and the array is then allocated in one of the two different transient spaces as depicted in Figure 6.1. Ranges in the clear-on-reset space are reserved persistently: their data will be valid as long as a token is powered, and only get lost in case of power loss. The ranges in the clear-on-deselect space are only reserved for the duration of an application session. As soon as the remote terminal finishes a communication with an application and selects a different one on the card, the reserved space gets reassigned to the newly selected application, and the data associated with the previous one gets discarded. Thus, if an application gets reselected during a token session, it can only rely on the contents of transient data it has stored in the clear-on-reset space.

Figure 6.1 also shows the typical implementation of the different transient spaces. Both heaps grow with every allocation towards each other allowing for some flexibility in the distribution of the clear-on-deselect and clear-on-reset space. On every allocation, the system must still ensure that both areas never overlap. This, for instance, expects the system to keep track of the largest clear-on-deselect space whenever a new clear-on-reset array is allocated by an application.
Deficiencies

Although an adaptation to secure tokens requires the removal of many dynamic elements in Java, the memory model proposed by the JavaCard is too rigid, and incurs too many deficiencies regarding performance, resource utilization, scaleability and programming convenience:

- Performance: Although one of the main reasons for transient data is the performance improvement, every access of transient data incurs one indirection by the persistent header. Additionally, slow EEPROM writes are necessary at instantiation time, and the “clear-on-select” space must be zeroed out whenever a new application gets selected. Dependent on the size of transient space, the necessary time can be significant on an 8-bit-based smartcard.

- Resource utilization: The static lifetime description leads to problems as soon as an application depends on interapplication communication. The “clear-on-select” space can only be valid for one of the participating applications, the second application must either avoid the use of transient data at all, or, more likely, must allocate its data in the clear-on-reset space. The pressure on the clear-on-reset space is furthermore increased by the static reservation model: all transient and persistent data necessary in case of worst-case execution has to be allocated at once at installation time.

- Scaleability: It is argued that allocating all objects at installation time guarantees the possible execution of an application at later times. However, execution can still fail due to other memory demands, as for instance for a sufficient transaction buffer or execution stack. In general, a system should not expect applications to limit themselves, but should enforce given limits on the applications. For many applications, it makes totally sense to instantiate all objects at installation time, but many other applications like certificate databases depend on being able to increase their storage on demand [134]. The system should then limit their resource needs just regarding to the user and/or issuer policy. The static reservation model also does not allow for applications to keep a set of transient objects optimized for a specific communication state and its reorganization in case of a state change. With an efficient garbage collection scheme (Section 6.2.5), this does not have to incur any resource increase, to the contrary, the optimum adaptation of the object state to the communication state allows for an efficient use of the available resources and a convenient programming model.

- Programming convenience: The compliance to the standard Java specification by the shift from temporary objects to temporary data has resulted in an inconvenient programming model. On the one hand, only array data and no object data can be allocated in RAM leading to an inflexible environment, especially for an OO language. On the other hand, the programmer has to cope with inconvenient lifetime specifications. Lifetimes
need not only specified when programmers create new arrays, but must also be passed to all APIs allocating objects on behalf of an application. The memory model then even clutters the APIs, and hinders their future evolution.

6.2.4 Transient and Persistent Environment

Basic Approach

With a look at the described deficiencies and the needs of secure token applications, a sufficient model must address the basic issues as follows:

- **Point-in-time for persistence specification:** Both the dynamic persistence specification by assignment and the static specification by type incur too many problems. Lifetime specification at instantiation time is easy to implement, can succeed independent of type, and is especially not restricted to arrays as in case of JavaCard.

- **Connection between persistent and transient objects:** as soon as connections between persistent and transient objects can occur freely, potentially dangling references cannot be handled efficiently by the system anymore. Thus, the system must not permit the arbitrary storage of transient references into persistent objects, but must foresee a more tightly controlled access to transient objects.

- **Transfer between persistent and transient data:** only orthogonal persistent systems allow for the system transparently moving objects from transient to persistent stores and vice versa. All other systems rely on the applications manually copying data between the stores. This fits nicely the secure token application patterns as transient and persistent objects typically serve different purposes, i.e. RAM objects covering communication state and EEPROM objects being target for input and output. Thus, objects must rarely be moved fully between the two stores. Instead, applications most of the time copy raw or primitive data between transient and persistent array ranges. This must then be controlled and initiated by the applications anyway.

Details

The resulting solution lies in introducing a mechanism that is symmetrical to the persistent environment. The persistent environment is spanned by the application instance as root object and the set of objects reachable therefrom and also allocated in the persistent store. Analogously, the transient environment is also formed by a tree of objects with the difference that all objects reside in RAM. Both environments are separated as far as references to transient objects are forbidden in the persistent set [113].

The location of an object is specified at instantiation time with the help of an API that can be used to toggle the allocation mode (Figure 6.3). Allocations between `beginTransience()` and `endTransience()` end up in RAM, otherwise in
Figure 6.2: Separated, symmetric transient and persistent environment

EEPROM. This also allows to signal factory methods in the APIs the allocation mode to use on object instantiations. A RAM overflow is signaled as in case of an overflow of the persistent store in JavaCard, by an exception.

The allocated transient objects can be interconnected freely to form the transient environment whose root can be registered at the run-time environment by invoking the setTransientEnvironment() method. The transient environment can be retrieved freely during the applet execution by an explicit invocation of the getTransientEnvironment() method. In contrast, the persistent environment is implicitly passed on each method invocation of the application instance. This then results in a slight, but correct emphasis on secure token applications manipulating persistent data in the first place. The given code example is oriented towards the communication APIs used in JavaCard, but is applicable to other environments as well as long as the application invocation model is embedded in a communication-oriented framework (Section 4.3.2).

Transient references may not be stored in persistent objects leading to the transient and persistent environment being strictly separated. The retrieval of the transient environment is always controlled at a single point (i.e. getTransientEnvironment()), and allows for accessing all connected, transient objects by following their path in the object tree as depicted in Figure 6.2. At the same time, the assignments of transient references to persistent fields are prohibited to avoid dangling pointers in case of a sudden power loss. The VM must check possible assignments accordingly, and possibly throw an exception. The overhead incurred is still negligible as the proposed bytecode instruction-set uses different instructions to store fields of primitive and reference type (Section 5.4). Only the must be checked, and occur rarely enough to have no impact on the overall performance. Assignments in the opposite directions need not be checked, as they do not impact the integrity of the persistent store.

Advantages

The transient environment fits naturally the execution model of token applications. It is typically built up at the beginning of a session, and its transient
6.2. TRANSIENT AND PERSISTENT ENVIRONMENT

// Sample Applet DummyApplet
class DummyApplet extends Applet{
    ...
    short select(){
        // instantiate transient object.
        JCSysmtem.beginTransience();
        Object o = new Object();
        JCSysmtem.endTransience();

        // register transient environment
        // for later during the session.
        JCSysm tem.setTransientEnvironment(o);
    }
    ...
    short process(APDU apdu){
        // reuse transient environment on each
        // incoming message during the session.
        o = JCSysmtem.getTransientEnvironment();
        ...

    }

Figure 6.3: Code Example for Transient Environments

objects either cover information about the communication state and/or contain
data whose access is performance or security-sensitive. If transient data must
survive the session, it may be copied manually to the persistent store by the
application. After the session, the transient environment is discarded, and the
whole transient space is fully available for the application selected next. The
simple approach leads to transient and persistent objects independent of type
and thus to a convenient programming model. It results in better performance
and more flexibility, and can be expected to lead to more compact code. The
analogy between transient and persistent environment also simplifies the system
where the uniform object model can lead to uniform system implementations,
for instance the same object-layout and heap implementation can be used for
both the transient and persistent store.

The only limitation introduced by transient environments is the restriction
on assignment. However, systems may implement more flexible schemes on
larger tokens without preventing the execution of applications having been de-
signed and implemented for the default memory model. This fully satisfies the
requirements of the general token architecture: applications targeting the com-
mon denominator should still be deployable on large tokens.

6.2.5 Implications

Memory Reclamation

Memory reclamation is an extremely important issue in the resource-limited
smartcard environment. Surprisingly, it is not sufficiently addressed in the Java-
Card 2.1.1 specification [149]. Manual memory reclamation has been judged to be unsafe and inconvenient, garbage collection schemes to be too expensive. With the different memory characteristics of RAM and EEPROM, possible memory reclamation schemes must first be discussed separately before a final conclusion can be drawn.

Indeed, the necessary EEPROM writes limit the possibilities for a general garbage collection of the persistent store on low-end token. For instance, a simple mark-and-sweep collector has to annotate in a first pass all reachable objects, and has to scan in a second pass the heap to free all unreferenced objects [86]. In case of limited RAM sizes, both passes (i.e. annotating the mark bits and resetting the state of heap items) involve many EEPROM writes. The performance penalty makes it impossible to execute a garbage collection at arbitrary points or to interrupt an application as soon as memory is scarce. However, the necessary code size for a garbage collector is so small that it should be offered in any case, if not at arbitrary times, at least at fixed points in time. Garbage collection might be signaled by an application for having to take place after its session has finished, or is fulfilled by the system after an explicit request by a terminal. This already fits the typical application behavior. Most applications can be expected to allocate their persistent set at installation time, and to limit changes to data updates. Other applications might rarely replace and/or instantiate new objects, but usually perform this at specific, well-controlled points. Necessary garbage collections can then be easily signaled either on- or off-token [115].

The available RAM is much more limited than the persistent store, and can thus fill up very quickly, especially when an application allocates and uses temporary objects without connecting them to the transient environment for later use. Although high allocation rates are rare in case of secure tokens, temporary one-time object allocations should still be possible for their applications. Indeed, they can already be efficiently supported on low-end smartcards assuming the use of the transient and persistent environment as described:

1. The transient environment can be garbage collected completely separated from the persistent set. Persistent objects need not be scanned as their fields must not reference transient objects.

2. The VM must not preempt the execution of an applet. The VM can delay the garbage collection until the application returns from its invocation. No application method is then active, and, thus, no references on the execution stack have to be included in the root set of the garbage collector. This makes garbage collection simple to implement and efficient, even on low-end 8-bit platforms with only several hundreds of bytes of RAM. The system can thus again benefit from the application being embedded in a communication framework.

An implementation on a large token might interrupt the applet execution and perform the garbage collection more often to allow for high allocation rates and to be able to react to immediate memory pressure. The applications targeting the common denominator hardware are still not affected.
With the proposed uniform object model, a system can reuse a single implementation to garbage collect both stores. In case of JavaCard, an EEPROM garbage collector may comply with the model described, but memory reclamation in the transient space is difficult to achieve. The system can try to reuse the space for globally allocated *clear-on-reset* arrays after their applets have been deleted. It can also allow for the increase of the *clear-on-reset* space after the applet with the highest *clear-on-deselect* usage has been deleted. A compactification of the *clear-on-reset* might also be an option, but requires expensive write operations to the persistent object headers in EEPROM. In any case, the added complexity always contrasts the low amounts of reclaimed memory and programming flexibility by JavaCard.

**Environment Multiplexing**

The transient environment even fits the needs of extremely limited environments with RAM resources up to 512 bytes [116]. In case of smartcards, one single application is selected by a terminal at a time, and, thus, is in full control of the communication channel. With the persistent and transient environment, it also controls the available memory resources as much as possible. It can fully exploit the available transient space, can seamlessly benefit from the garbage collection of RAM, and can fully control garbage collection of EEPROM. It may allocate additional objects in EEPROM, but may then have to cope with a quota imposed by the system.

On larger and future tokens, a terminal is likely to setup communications with several applications, and to keep multiple channels open at the same time. Each incoming request on a channel must then be routed to the responsible application. If the system supports only a single transient environment, the different applications must share it (i.e. agree on the set of nodes they have exclusive access to). This assumes the cooperation of all participating applications,
and cannot be expected in general. However, the system can easily multiplex
the transient environment at its single access point (i.e. `getTransientEnvironment()`), and associate each channel with its own environment on large tokens
as depicted in Figure 6.4. All applications only supporting a single connection
at a time can then be deployed on low-end and high-end tokens. Applications
supporting multiple simultaneous connections must incur additional logic, and
have to rely on system extensions for proper synchronization.

6.3 Transaction Support

6.3.1 Goals

Convenient manipulation of objects is one important requirement for the soft¬
ware environment, support for the reliable update of objects another. If a
terminal only reads data from an application, it can deliver the requested data
without any precaution. If it has to create or update data during the commu¬
nication, care must be taken that the integrity of its data is preserved. Either
all updates take place, or the data on the card is reverted to its initial state
in case of an interrupted execution. The underlying system must thus provide
a proper transaction mechanism which ensures the correct transition between
consistent states of applications, and offer its functionality to all applications.
The task of the system is then twofold. First, the system is required to ensure
that all updates by an application are performed atomically; second, it must
perform crash recovery to provide stability: the system must recover the system
and application state to a consistent state if a transactional computation fails.

An observation of application behavior can already narrow the scope for the
necessary transaction support. The transaction model must only support user¬
level transactions in the traditional sense [57]. Basic transactions take place
locally on the token, can be assumed to begin and end within the communic¬
ation with a terminal application, are thus short lived, and need not be split
in multiple sub-transactions even if multiple applications cooperate together.
Advanced transactional semantics can be offered by applications relying on the
base mechanism [78, 21, 120].

Additionally, the RAM contents such as transient objects and execution con¬
texts need not be target of transactional updates, either due to performance or
security reasons. Transactional guarantees must thus only cover manipulations
of persistent data. Explicit hardware-support cannot be expected as EEPROM
does not even guarantee atomic writes of single bytes, and the system can only
rely on the outcome of single-bit writes. The transaction model must then also
meet the general requirements:

- Resource efficiency: Transactions must be supported efficiently, even on
  low-end tokens. With most security-sensitive updates requiring transac¬
tional guarantees, the transaction mechanism places an important aspect
  on the overall useability. The mechanism must even be fully applicable
  in scenarios where contact-less smartcards with limited performance and
  bandwidth are used in services requiring short transaction times.
• Scaleability: Transactions must be fully exploitable not only by applications, but also by the system itself which currently places the highest demands. Downloading new applications incur heavy-weight transactions due to the large number of newly allocated objects and system-internal structures. Exporting these capabilities to applications allows for the implementation of large portions of the system in the application programming language simplifying its adaptation to larger and future tokens. Additionally, chances are increased that all current and future applications can be sufficiently served.

• Programming convenience: as transactional updates of persistent data are inherent in secure token systems, a convenient integration into the application development is extremely important for a secure token environment.

6.3.2 Language Integration

In case of general computing environments, transactional computations are typically separated from all other application computations and occur isolated in the application code. They might be able to rely on transactional services offered by a system, for instance a transaction framework [153, 35], but an application is then forced to structure its logic accordingly. The inherent transaction needs of secure token applications would force most system and application classes to comply to the given transaction framework. The resulting lack of flexibility and implementation complexity thus enforces a transparent integration of transactional computations into the language.

The JavaCard API for specifying transaction boundaries already fulfills the basic requirements having been proposed by persistent systems [31], and matches the properties of the proposed memory model. The type of an object is not only orthogonal to its persistence or transience, but its update can also be transactional independent of its type. Additionally, the transaction scheme satisfies the following requirements:

1. Persistent updates are independent of transactional updates: Changes to objects residing in EEPROM persist even when occurring outside of transaction boundaries. A single write to an object field in EEPROM has to be atomic to guarantee the object integrity in any case [149]. However, multiple writes to persistent fields inside or outside a transaction may differ in their behavior.

2. Transactional independence: Source code executed inside or outside a transaction can look exactly the same.

3. Execution within transactions do not compromise the security of Java: No changes have to be applied to the language or to the instruction-set. Additionally, the recording of state changes is invisible and unaccessible to the executing applet.

Figure 6.5 shows the use of the current API in the JavaCard specification for initiating, committing and aborting transactions. The control of transactions by
Figure 6.5: Transaction API in JavaCard

static methods has still one disadvantage. The begin and end of a transaction are not connected to each other, neither in the program text nor at run-time. As a result, the execution state cannot be reset exactly (i.e. the beginTransaction() invocation) when a transaction is aborted by an abortTransaction() invocation. Instead, execution continues right after the abortTransaction() call or not at all.

Transactional systems like Transaction-C extend the language by constructs allowing the linguistic connection between begin-, commit- and abort-blocks [153, 50], and execution continues at well defined locations in case of abort or commit. PJama achieves a similar effect by expecting the transaction to be coded within a single instance method [31]. The system will execute the given method within a transaction, and return in any case (i.e. commit or abort) from its invocation. Such a mechanism adds an overhead of one temporary instance per transaction which might still be acceptable even on resource-constrained smartcards.

However, a method encapsulation interferes with the communication-oriented framework as proposed (Section 4.3.2). It is not possible to extend the lifetime of a transaction across multiple invocations of an application as soon as the transaction is encapsulated within a single method invocation. A transaction could then span only a single message cycle between application and terminal, the download and installation of applications could not be fulfilled in a single transaction, for example. The transaction API therefore favors flexibility and resource friendliness, although the missing linguistic connection is partly responsible for the deficiencies in JavaCard described in Section 6.3.5.

6.3.3 Basic Transactional Semantics

As soon as a transaction is started, the system must keep track of the changes to the persistent environment. The system must at least record the state before and the most current value for any given element during the transaction. The updates are most likely to be logged at the granularity of single access. Only very large systems with proper hardware support may group objects in pages, manipulate them in RAM during the transaction, and log their changes lazily at the granularity of a page into stable storage [65].

The system must provide two guarantees. If a system commits a transaction
on request by an application, it must guarantee that the changes to the persistent set are applied in any case. Any necessary commit information must be stored persistently at commit time to allow for the restart of the commit process in case of sudden power-loss. If the commit process succeeds without a crash, execution continues after the return from the commit.

Whenever a transactional computation aborts, the system must be able to restore the state at its beginning. The reason for an abort first includes system crashes, e.g. sudden power-losses, or system-initiated aborts of application computations: the system throws an exception in case of any irregularity detected during the transaction (for instance a transaction buffer overflow), and aborts the application in case of the exception not being handled. In any case, the system recovers the previous, persistent application state. The recovery information has to be stored persistently to be able to restart the recovery process in case of a sudden power-loss. The system is then free either to deselect the current application, or to let it continue in its current session.

An application can always explicitly request an abort by an `abortTransaction()` invocation, for instance on request by a terminal. The application must still remain active afterwards, and be able to further communicate with the terminal application.

### 6.3.4 Implementation Strategies

During the execution, updates or writes to the persistent set only occur on the access of persistent fields and arrays. The transaction implementation must thus touch only a minor fraction of the chosen instruction-set, all other instructions will be executed and perform as usual. This especially includes all computations operating on transient data during the transaction. Persistent updates can also succeed in native, system-internal methods which must then use special accessor functions to not bypass the transaction mechanism. In case of JavaCard, this, for instance, includes the native `Util.arraycopy()` method allowing for the transactional update of a number of array elements at once [148].

For the logging of write accesses during a transaction, two schemes are well-known, either new value or old value logging [56]. In case of old value logging, the update of a location during the transaction occurs in place, i.e. directly at the referenced location. The general properties of old value logging are:

- fast read accesses as the up-to-date values are always written to the referenced location.
- the original value for a given location must be saved in a transaction buffer, typically once at the time of the first write access to the location.
- committing a transaction is cheap as all new values are already in place.
- aborting a transaction is expensive as all saved values have to be written back to the original locations.
In case of new value logging, each value for a store operation to a given location is saved in the transaction buffer while the original value remains at the affected location. The general properties are here:

- slow read accesses as the up-to-date value for a location must be searched in the buffer.
- write operations always have to affect the buffer as any new store operation has to be recorded there.
- committing transactions is expensive as all new values have to be written to their target locations.
- aborting transactions is cheap as all original values are still in place.

Although the advantages and disadvantages still apply in general in case of secure tokens, the degree depends on the exploitation of their memory characteristics. The critical performance aspect on low-end tokens mostly depends on the number of necessary single or block EEPROM writes whereas RAM accesses are negligible to a large extent [110, 129]. One might also take the typical application patterns into account where writes to the same location during a transaction are usually rare. So what are reasonable implementations and the achievable performance for both schemes on a low-end token like a smartcard?

**Old Value Logging**

Read performance always remains excellent in case of old value logging. In case of a write, the referenced location has to be checked for having already been saved. A reasonable implementation on smartcards scans the transaction buffer linearly for the given location, and, if found, just writes the new value to the target location. As multiple updates of the same location are rare, this case—a single EEPROM write for the update—does not occur too often. If the original value of the given location has not been saved so far, a new entry consisting of location and previous value must be added persistently to the transaction buffer to support its recovery in case of a sudden power-loss. Two schemes are conceivable, a marker- or counter-based transaction buffer. The latter one adds first the new entry to the buffer, and then increments the entry counter of the buffer. The counter must be incremented atomically, for instance with the help of a shadow counter and a flag indicating which counter is currently valid. Thus, three EEPROM writes are necessary. One block write for the new entry, one write for the incremented shadow counter, and one write for flipping the counter flag. Performance can be increased with a marker-based scheme where a flag after the last entry in the buffer indicates its end. Entries are added to the buffer by first appending a new entry with a new marker in a single block write and then clearing the marker of the previous entry with a second single EEPROM write. The number of EEPROM accesses is reduced to two, while the entry size is increased by an additional byte.

Table 6.1 summarizes the properties of an old value logging scheme with a marker-based implementation. Appending a new entry needs two EEPROM
 writes. In case of commit, the expected total number of EEPROM writes per location is then expected to consist typically of three assuming multiple updates of the same location are rare: two for adding an entry, one for updating the target location.

In case of commit, the transaction buffer must just be marked invalid to complete the transaction. In case of abort, the saved values in the buffer have to be written back to their former locations (Figure 6.6). After a sudden power loss, the write process may just be restarted from the beginning of the buffer, as buffer entries remain constant and can thus be rewritten as often as possible (although the number is actually limited by the physical limit of EEPROM writes).

### New Value Logging

Similar overall performance can be achieved in the new value logging scheme dependent on its implementation and available resources. Read performance lags always behind, as the transaction buffer must be scanned—typically linearly—

---

![Diagram](image)

**Figure 6.6: Abort in case of Old Value Logging**

<table>
<thead>
<tr>
<th>Logging Strategy</th>
<th>New Value</th>
<th>Old Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commit Costs</td>
<td>High</td>
<td>Minimal</td>
</tr>
<tr>
<td>Abort Costs</td>
<td>Minimal</td>
<td>High</td>
</tr>
<tr>
<td>Minimum E2 Accesses for Logging</td>
<td>1</td>
<td>2/Log Entry</td>
</tr>
<tr>
<td>Maximum E2 Accesses for Logging</td>
<td>1/Store</td>
<td>2/Log Entry</td>
</tr>
<tr>
<td>Expected E2 Accesses per Committed Store</td>
<td>1 + 1/Store</td>
<td>3/Store</td>
</tr>
<tr>
<td>Expected E2 Accesses per Aborted Store</td>
<td>1</td>
<td>3/Store</td>
</tr>
<tr>
<td>Writes per Log Entry on Abort</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 6.1: Comparison of Logging-schemes**
JCSystem.beginTransaction();

obj.i = 0x20;

JCSystem.commitTransaction();

Figure 6.7: Commit in case of a RAM cache based Transaction Buffer

for a formerly written value. The situation can be better in case of the much more expensive write operations. A straightforward implementation will scan the transaction buffer for a formerly written entry for the target location, and replace its value with the new value in a single EEPROM write operation. If the location is accessed the first time—the most common case for typical JavaCard applications—a new entry must be added to the buffer non atomically, e.g. with one single EEPROM block write.

The performance can thus be increased significantly by caching the transaction buffer in RAM and writing it out lazily to EEPROM on overflow (Figure 6.7). If RAM resources are not too limited and the transaction does not involve too many write operations, all updates can be logged in the RAM cache, and are only written to EEPROM in one single EEPROM block write at commit time. The cache must be flushed persistently as the system must be able to fulfill the commit even in case of a sudden power loss. After that, the run-time environment can scan through the transaction buffer, and apply the stored values to the given locations. Aborts are again free in the sense that the contents of the transaction buffer can just be discarded.

Table 6.1 summarizes the properties of a new value logging scheme with RAM caching. Best commit performance can be achieved if all log entries can be cached in RAM and are saved in EEPROM at commit time with one single block write. The value in each entry must then be flushed to its target location with another EEPROM write. In the worst case however, EEPROM has to be
6.3. TRANSACTION SUPPORT

accessed on each log operation, for instance, if log entries are reused and a RAM overflow led to the cache already flushed to EEPROM during the transaction.

The described cache-based scheme cleverly utilizes EEPROM block writes to achieve excellent performance in the common case. It also optimizes for aborting an application during its transactional processing. This especially benefits scenarios where the processing time of an application outweighs the processing time of the system and an abort during the application execution is quite likely as in case of contact-less cards for example. However, the increased memory footprint and its complex implementation might still favor a more conventional implementation in case of low-end tokens.

6.3.5 Transactional Object Instantiations

Although performance can be sufficient, two issues regarding generality and scaleability remain with current JavaCard specifications. On the one hand, the possible duration for a transaction is limited to one message cycle which can be corrected quite easily [112]. On the other hand, the specification does not foresee object allocations with transactional guarantees. Objects are supposed to be allocated only at application installation time, and are then expected to be guarded by a specialized, unspecified system mechanism. Other object allocations may not have to be supported at all, and even if supported, the JavaCard specification states that object allocations within transactions may fail and any allocated space may get lost forever in case of an abort [149]. Clearly, this does not conform to proper transactional semantics where the system and application state is expected to be exactly the same as before the transaction in case of abort, and thus any allocated space in between is released.

With the transient environment, temporary object allocations are supported in a flexible manner on all secure tokens. The same can and must be achieved for persistent object allocations. A system is even able to support the allocation and manipulation of large objects during a transaction on low-end tokens. For instance, the IBM Open Platform implementation relies completely on the regular transaction mechanism to download and install new applications. During the transaction, a new array is created, and the executable content is downloaded and stored in it. If the transaction fails, the transaction mechanism ensures that the newly allocated object will go away during the abort process.

The newly allocated array can be arbitrarily huge in the given example, and, as any write to the array incurs an additional entry in the transaction buffer so far, the buffer is likely to overflow during the transaction. However, each access to a newly created object can be easily detected within the system, and directly forwarded to the contents of the newly created array. In the object-aware instruction-set [25], objects are always addressed by reference and field-offset. As the object header is logged at instantiation time, the transaction mechanism can decide on each access whether a given object already existed before or has been allocated during the transaction. The referenced object is just searched in the buffer, and, if not found, the object existed before, and the store operation is regularly logged. If found, it has been newly allocated, and the value can commence directly to the newly allocated region at the given offset.
In case of commit, the object header and its heap management information is written permanently, and the object thus becomes allocated persistently. In case of an abort, it is automatically released.

This still does not explain the statement about potentially lost space in the JavaCard specification. This results from the fact that the transaction mechanism logs only writes to persistent fields, and references stored in RAM may still refer to newly instantiated objects after an abort. Figure 6.8 shows a code example where $f$ is a local, transient variable which references an object instantiated during the transaction and which is still accessible after its abort.

In case of the JavaCard specification, the persistent state must be reverted back immediately at the time of abort. Consequently, if the system wants to release all newly allocated objects, it has to find and reset immediately all now dangling references. They may either reside in transient objects or directly on the stack, where especially detecting the latter ones incurs significant overhead [112]. The JavaCard specification prefers to lose memory in such a case and allows for the newly allocated objects to just remain valid and lost after abort.

The application patterns, their invocation model and the use of the transient environment allows for a simple modification of current specifications: the abort of the persistent state should be delayed to the return of the application from its current invocation. The execution stack is then empty, and dangling references stored in the transient environment can be easily reset during the garbage collection [112]. The simple update allows for fully transactional object allocations, and naturally completes the proposed memory model.

6.4 Conclusion

The proposed model can satisfy all secure token needs, ranging from low-end tokens to large mobile or stationary tokens. Its basic properties are in line with the goals of persistent systems having been proposed in the past. Persistence
independent of type and transparent manipulations of objects independent of their lifetime within language expressions are goals which are not even fully reached in a number of large persistent systems [22]. The proposed model is then clean, comfortable and easy to understand: objects are placed at instantiation time in either the transient store or persistent store, and can then be freely manipulated. They can be interconnected to form the persistent and transient tree of objects (i.e. the persistent and transient environment), and are then both accessible and reusable on each invocation of the application during a communication. The necessary transaction support can be seamlessly integrated for the update of persistent data, and the memory model then allows not only the manipulation of persistent and transient objects, but also to encapsulate any sequence of persistent updates within transactional boundaries.

The memory model is lightweight to implement, even on extremely resource-limited tokens [117], and allows for all applications to fully exploit the available resources. Applications are in full control of the transient resources, and can always lay out the transient space specific to the communication state they are currently in. They can even instantiate temporarily objects for a one-time use, as the proposed garbage collection scheme allows for an efficient reuse of memory. The garbage collection and heap implementation of the transient store can additionally be reused by the system for the management of the persistent store allowing for space-efficient implementations and garbage collecting the persistent store on explicit request. In contrast, the memory model implied by current JavaCard specifications is static, resource-demanding and inconvenient and already remains insufficient for smartcards.

The proposed model also allows for the full exploitation of larger tokens and their resources by advanced applications. High allocation rates, access to a large tree of transient objects is a matter of the underlying token resources and application needs, but not of the proposed memory model. With the simple multiplexing of the transient environments per communication channel, the concept can also be adopted to large or future environments where multiple simultaneous communication channels may have to be supported.

The transaction model is also applicable to both low-end as well as advanced token. It allows for very efficient implementations with a series of persistent updates being faster inside of transactional boundaries than outside. With its performance and convenient language integration, transactional programming becomes simple. With removing the restrictions imposed by the underlying JavaCard standard, it additionally becomes flexible enough to even satisfy the needs on larger secure tokens.

The proposed model fully complies to the original Java philosophy, and provides not only an efficient, resource friendly, and secure, but also a programmer friendly and safe model. Typical programming errors by manual memory allocation are avoided. In contrast, all systems based on software MMUs such as MULTOS and WindowsCard allow for the arbitrary access of the reserved areas by the applications, and foresee the applications to be fully responsible for their heap management. Manual memory control with the implied risks is for sure not the first choice in a security- and integrity-sensitive environment like secure tokens.
Chapter 7

Evaluation

The presented system properties sum up to a base software environment supporting dynamic issuing models (Section 2.4.4). Its business requirements - low cost, increased independence of manufacturers and application providers, customer satisfaction, trustworthiness - translate to the need for sufficient security, development convenience, and real-world applicability. The system must offer sufficient performance, resource friendliness and legacy support on existing smartcards.

A Java-based system can satisfy these goals as long as core standard Java abstractions are adopted. Although modifications seem natural considering the specialized nature of secure tokens, they easily affect Java’s generality and portability. Before JavaCard, Java was only strictly subsetted for resource constrained platforms as elder Embedded and Personal Java specifications show. JavaCard represents the first device category where platform-specific modifications and APIs were embraced. This development led to the many environment- and device-specific Java specifications of today [147, 151]. Having been the first device-specific platform, JavaCard still suffers from an unnecessary compliance to standard Java. Problematic standard Java constructs were adopted unmodified, for instance certain instruction semantics (Section 5.4). JavaCard only half-heartedly adopted superior proposals suggesting more advanced Java adaptations. For instance, the JavaCard 2.1.1 standard introduced temporary objects to limit certain security risks [146]. They originate from the proposed transient objects (Section 6.2.4), but without exploiting all other advantages.

Security

In case of security, the JavaCard offers the necessary base guarantees for the secure and reliable execution of multiple applications on the token. The Java namespace and sandbox model is adopted in a manner to allow for the separation of system and application data, and for the secure execution of applications. The system is resistant against application crashes, and applications are protected against the interference of each other. Furthermore, JavaCard incorporates a lightweight transaction mechanism for the reliable execution of applications. The thesis shows that implementations can succeed in such an optimized manner that applications can almost unobjectionable make use of it
(Section 6.3), simplifying the task of the programmer significantly.

The base services of JavaCard can be complemented by an implementation of the available management specifications such as VISA OpenPlatform to fully support a dynamic issuing model. These specifications describe secure communication protocols for partitioning the JavaCard into strictly separated domains. Each domain is assigned to a service providers which can freely download and install applications into it. The management specifications depend on cryptographic measures such as code signing or encrypted communication channels to ensure the authenticity and integrity of the code. This relies from the fact that JavaCard only foresees off-card bytecode verification. On-card verification in case of its instruction-set and link-format is too resource-intensive, and necessitates approaches as described in this thesis (Section 5.7). They allow for practical on-card verification on next-generation smartcards, and increase the overall security of the system. For instance, issuers cannot crash their tokens by code having been improperly generated by accident. Additionally, it enriches the possible business models for Java-based multi-application tokens. It represents a big step towards secure tokens solely manageable by their users, as it provides the necessary resistance against viruses and/or faulty operation by the user. The thesis then closes the gap to secure token systems based on defensive virtual machines. They check most instructions for being safe to execute at run-time, and achieve similar security properties, but dramatically suffer from decreased performance.

The proposed memory model also increases the overall system security. JavaCard keeps the lifetime notion of standard Java: objects are alive as long as they are reached. In contrast to the simplicity and elegance of its ancestor system, it is unnecessarily complex and inconvenient (Section 6.2.3). The proposed memory model foresees a strict distinction between transient and persistent memory. It naturally extends the base philosophy of partitioning the token, and allows for a high degree of control by token applications over resource- and security-relevant actions. It enables garbage collection as a safe means for memory reclamation. It supports transactions for reliable updates by both applications and system [115]. JavaCard relies on a number of ad-hoc mechanisms to reclaim memory and/or utilize transactions. Furthermore, it complicates application and system development provoking programming bugs and security leaks.

Development Convenience

The proposed memory model significantly increases the development convenience (Section 6.2.4). It enriches the typical programming style for secure token applications - communication driven, resource sensitive - with well-known and dynamic Java features such as garbage collection. JavaCard already surpasses the competing smartcard systems MULTOS and WindowsCard regarding the programming convenience, although it mostly relies on static Java language features. It introduces object orientation, namespaces, and static typing in the smartcard environment. JavaCard offers flexible API frameworks where systems like MULTOS and WindowsCard provide limited and fixed system calls. Fast application development is possible, especially as many mature standard Java
Table 7.1: Code sizes for the system modules for JCOP10 (implementing a DES-only JavaCard with VOP), and JCOP21sim (implementing JavaCard, VOP, SIM, and public-key cryptography) on an 8051 based 8-bit processor. Sizes are given in bytes.

<table>
<thead>
<tr>
<th>System Module</th>
<th>Language</th>
<th>JCOP10</th>
<th>JCOP21sim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low-level Code</td>
<td>Assembler</td>
<td>1145</td>
<td>2007</td>
</tr>
<tr>
<td>Crypto Drivers</td>
<td>Assembler</td>
<td>538</td>
<td>10615</td>
</tr>
<tr>
<td>Protocol Stack</td>
<td>Assembler</td>
<td>1940</td>
<td>1232</td>
</tr>
<tr>
<td>Engine, Interpreter, MM</td>
<td>Assembler/C</td>
<td>12045</td>
<td>12091</td>
</tr>
<tr>
<td>Management Instance, APIs</td>
<td>Java</td>
<td>11380</td>
<td>30388</td>
</tr>
<tr>
<td>VSDC</td>
<td>Java</td>
<td>8616</td>
<td>do.</td>
</tr>
<tr>
<td>VisaCash</td>
<td>Java</td>
<td>4681</td>
<td>do.</td>
</tr>
<tr>
<td>PSE</td>
<td>Java</td>
<td>817</td>
<td>do.</td>
</tr>
</tbody>
</table>

tools can be reused during the development process.

API evolutions and extensions are still limited as JavaCard’s code format and on-card linking process lacks the necessary flexibility (Section 5.5.3). Extending an API by new virtual methods breaks binary compatibility with elder applications, for example. The proposed code format and link process lifts these subtle restrictions, and supports all standard Java compatibility rules (Section 5.5.4). APIs can be designed similar to their standard Java ancestors, and the same principles can be applied to the development of token, terminal or back-end applications. This especially attracts application developers as they typically do not only implement the on-token functionality, but the total solution, including terminal and back-end applications. In contrast, WindowsCard and MULTOS significantly contrast the programming experiences of typical desktop and/or server programming environments.

Real-World Applicability

Programming convenience helps to keep down the development costs for applications. These are still negligible compared to the deployment costs of secure tokens, especially in case of smartcards. However, the proposed system offers excellent real-world applicability to current token infrastructures and environments. It can replace the proprietary smartcard systems deployed as of today: it can be implemented on the cheap, currently available mass-market hardware, and is able to support the existing legacy communication and application protocols. Transaction times can be kept at the same level as with existing proprietary smartcard systems (Section 5.4.5, 6.3.4). The additional system resources required to support all the new features can be kept at a low level (Section 5.4.2, 3.4.7). Switching to a Java-based system software platform is feasible for card issuers. Their necessary initial investments are reduced to an acceptable level. The investments are quickly amortized even when only a few service providers - which is likely in case of a newly introduced business model - are attracted at the beginning.
The described Java-based system achieves an outstanding performance. Although the limited smartcard resources only allow for code interpretation, the applications stored as bytecode can service existing smartcard protocols as fast as proprietary systems executing them in machine code on smartcard hardware of previous generations (Section 5.4.5). One reason for the excellent performance is the chosen instruction set. Similarly to Java and JavaCard, it avoids the concept of a defensive virtual machine and extensive run-time checks (Section 5.2.3). It features instructions operating on built-in primitive and block types, and allows for fast object access (Section 5.4). It allows for a number of possible optimizations at code generation time (Section 5.8). The Java-based instruction set for secure tokens benefits from the fact that Java was originally already designed with code interpretation in mind, in contrast to environments like .NET for example.

Token applications still spend only a limited amount of their time in executing their own code, but invoking system services, especially EEPROM write operations and/or cryptographic services. Thus, the proposed system structure (Section 3.4.7), memory model (Section 6.2.4), and API design (Section 4.3.1) allow for a very lightweight and performant access of hardware and system resources. Additionally, a number of system optimizations increase the performance by exploiting the memory hierarchy on secure tokens. For instance, the proposed transaction model minimizes the necessary number of expensive EEPROM writes (Section 6.3.4). The code format allows for downloading and relocating the code on-the-fly in RAM before writing it to EEPROM (Section 5.5).

A Java-based secure token system, including JavaCard, must make efficient use of the processor, but also of the available memory resources. Regarding code size, the proposed instruction set and JavaCard achieve good code density, and allow for a compact code representation of applications. Additionally, the size of system classes and code in ROM can be significantly optimized (Section 5.8). The compact code, excellent execution performance, and feature set of Java (Section 4.2) allow to implement a large fraction of the system itself in Java (Table 7.2). Maintenance, porting efforts and future extensions are significantly simplified. This flexibility cannot be emphasized enough, especially as secure tokens feature a large variety of processors—from awkward 8-bit based to modern 32-bit based processors.

A final JavaCard implementation as in case of the IBM Zurich Research Laboratory (ZRL) can implement in 48KB of ROM a smartcard system featuring all the services required in existing banking environments (Table 7.1). This contains the native system, including communication, memory, execution and cryptographic services, the APIs, the management instance adhering to the Visa OpenPlatform standard (VOP), and a number of banking-related applications, VisaCash, VSDC and PSE for credit-, debit-, and wallet-based payment applications [159]. Similarly, a JavaCard SIM implementation at the IBM ZRL provides all the functionality required for its use in mobile phone networks in 64KB of ROM. Table 7.1 and 7.2 show that almost all of the required SIM functionality could be implemented almost fully in Java, and on top of the same system core.
Table 7.2: Number of lines of code in Assembler, C, and Java

<table>
<thead>
<tr>
<th>Language/System</th>
<th>JCOP10</th>
<th>JCOP21sim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler</td>
<td>8357</td>
<td>11120</td>
</tr>
<tr>
<td>C</td>
<td>18296</td>
<td>20403</td>
</tr>
<tr>
<td>Java</td>
<td>8449</td>
<td>16182</td>
</tr>
</tbody>
</table>

The proposed system also keeps the resources required for the system state and data extremely low as indicated throughout the thesis. Small execution context, closed package concept (Sections 5.4.2 and 5.5.2), finite state machine programming model (Section 4.3.2), memory layout (Section 6.2.4), and transaction implementation (Section 6.3.4) do not result in the same resource needs as in case of other Java based environments. For instance, the IBM ZRL implementation only reserves a few hundred bytes of the persistent and transient memory for system purposes, the remaining memory resources remain free for applications. In case of a SIM card, which requires large portions of EEPROM to save user related data, the overhead of the system becomes negligible.

JavaCard still leaves room for performance and resource optimizations, as previous Java-based prototypes at the IBM ZRL have shown [116]. These include the size of the code having to be transferred at installation time (Section 5.5.2), the design of the APIs (Section 4.3.3), and the inefficient management of the transient RAM resources. They can be easily locked by a few applications, even when there is still plenty of ROM and EEPROM space available for new applications (Section 6.2.3).

In general, the JavaCard standardization committee was in favor of rapidly approving proposals as long as they promised to be in line with the targeted smartcard applications and resources. Flexibility and generality were not crucial factors. For instance, the limited JavaCard memory model allows for implementing the kind of applications the attendees of the committee had in mind, electronic wallets for example, but the diverse needs of the application developers have been vastly underestimated. The first Java-based prototype at the IBM ZRL already showed that a superior flexibility regarding language, APIs, memory model and system design does not have to result in large resource needs, but can easily undercut existing JavaCard systems [116]. Its execution model was still similarly limited as in case of JavaCard. Although download performance was superior (Section 5.5.2), API extensions and code verification were only limited possible. These limitations can be removed by the proposed download format, although larger resource needs have to be accepted (Section 5.3). It is still fully applicable to existing and next generation smartcards which are also the first generations of Java-based smartcards having a large impact on current markets.
Chapter 8

Results

8.1 Summary

This thesis categorizes existing secure tokens, pointing out the common hardware characteristics—communication interface, memory hierarchy, and hardware protection—and the differences regarding processing and memory resources. The evolution of the prevalent secure token, the smartcard, is described, and the need for new issuing models and software environments is presented. These must allow for accessing new services in the field by applications being securely downloaded and executed on demand in a platform-independent manner.

This thesis presents a generalized Java-based multi-application software environment for secure tokens. The system is adaptable to existing legacy infrastructures. Applications can live with the limited resources of smartcards, and exploit large tokens, without jeopardizing the portability of base applications. All capabilities are offered in a programmer-friendly and secure manner.

This thesis presents the architecture of current smartcard systems such as JavaCard, and details their dependency on natively implemented services. A new system architecture is introduced which enables portable applications to deliver all token services. The system architecture is applicable to smartcards and large tokens, and can be tailored for different application programming languages and environments. Java can be efficiently integrated, but requires significant adaptations for secure tokens.

The necessary adaptations of the Java programming language and APIs are presented. At the syntactic level, the Java language is strictly subsetted. At the semantic level, Java features are removed, and concepts like the notion of object lifetime are redefined. The standard Java APIs are replaced by a framework suitable for secure token applications. It offers the necessary base services for portable applications, and allows for token-specific and legacy-constrained extensions.

This thesis proposes a novel Java-based execution model. It guarantees the application portability without the assistance of the remote host the code is downloaded from. Additionally, it supports fine-grained API- and coarse-grained system-extensions on larger platforms. The model is implemented by (1) designing a new instruction set, (2) proposing a new download format, and (3)
integrating code-verification schemes. The presented instruction set is derived from the standard Java bytecode, and allows for a satisfactory interpretation performance and compact code size. Possible optimizations at code-generation time are presented and measured.

The download format of JavaCard is based on a closed-package concept, replacing symbolic information by short identifiers, and pre-linking run-time-type descriptors. It is limited to 16-bit platforms, and restricts possible API designs and code verification. A derivative marker-based format is presented, which suffers from similar limitations, but demands significantly less resources.

A new format is introduced, which features a closed-package concept as in the case of JavaCard, but a linking scheme based on parsing and symbolic naming as in the case of Java. All standard Java binary-compatibility rules are preserved. The executable content can be optimally layed out on any 8/16/32-bit-based token. Fast downloads are possible by the exploitation of the memory characteristics of secure tokens (e.g. minimum number of EEPROM writes). The format is applicable to current low-end tokens, and allows code verification to be deployed on next-generation smartcards. Both adaptations of the standard verification scheme as well as proof-carrying-code (PCC) are studied.

The memory model of JavaCard lacks programming convenience, limits memory reclamation, and restricts scalability. This thesis examines persistent systems of the past, and introduces a novel object-oriented persistent memory model for secure tokens, the concept of a transient and persistent environment [113]. Any object—indepedent of its type—can be allocated in the transient and persistent store. The available memories of both RAM and EEPROM can be exploited in a fine-grained manner on low-end and high-end tokens. Garbage collection and transactions can be efficiently integrated. The model achieves excellent performance by cleverly exploiting the hardware characteristics of secure tokens (e.g. EEPROM characteristics).

8.2 Conclusion

The proposed Java based system, its novel system architecture, execution and memory model constitute a step forward for the deployment of secure tokens. It establishes a modern platform in the legacy constrained smartcard environment which is able to fully replace the existing proprietary systems. They tightly integrate native applications into their base systems. They are characterized by many system layers, and only offer limited virtual execution environments on top of them. Inadequate performance of the interpreted applications is the consequence, and the need for native and platform-dependent service implementations remain.

The proposed system architecture and execution model allows for a performant and resource-friendly execution of portable applications. Both complex applications as well as system services can be fully implemented in Java. Applications can be quickly deployed on vendor-specific platforms, and the system implementations can be easily ported to new secure token hardware. This contrasts the deployment of standard Java on desktop platforms. Its high resource
8.2. CONCLUSION

Demands and insufficient execution performance were two major factors preventing the successful introduction of Java based operating systems.

The proposed system architecture reduces the number of necessary system layers. It allows to highly optimize the system services required for the applications, and to rely on them for complex system-internal tasks. The novel execution model enables compact code, performant execution and low resource needs at download and execution time. Convenient Java features such as the sandbox model are preserved. The resulting system is able to serve legacy applications as efficient as proprietary systems on similar base hardware. For the first time, smartcard issuers are able to replace the existing proprietary systems by a modern and extensible platform in a cost effective manner.

The presented system supports the new issuing models having been proposed for secure tokens. Issuers can download and install applications in the field, offer new services on demand, and share their tokens with third party service providers. The novel system architecture and execution model provide the necessary execution performance and security guarantees. Additionally, the system demands few resources: the memory resources are mostly at the disposal of the issuer for a later deployment of additional applications and services.

The development of secure token software has so far been characterized by awkward programming environments, platform particularities, and time consuming test cycles. Implementations mostly had to be carried out by the smartcard manufacturers and token vendors themselves. The proposed Java based system offers a fast and cost-effective development process, and allows for the development of applications by independent application providers. The proposed concepts, especially the bytecode characteristics and persistent programming model, establish a new level of programming convenience on secure tokens. Features like type-safety, object-orientation, and garbage collection lead to a development experience close to what has been accomplished on much larger computing platforms so far. Programming smartcards, terminals and back-end systems finally converge, significantly simplifying the task of issuers to extend their infrastructures and introduce new services.

The proposed concepts are fully applicable to larger secure tokens and future smartcard hardware. Proprietary smartcard systems rely on legacy and smartcard-centric abstractions which are unsuitable for other tokens. Similarly to JavaCard, these systems are tuned for 8- and 16-bit based hardware platforms. This for instance limits the address space available for applications on larger platforms. The proposed Java based execution and memory model removes these restrictions. On the one hand, applications can exploit resource-constrained platforms and secure tokens with increased processing and memory resources by relying on the same basic concepts, system behavior, and programming models. On the other hand, a wide range of secure token applications seems possible which is portable across a large group of secure tokens. This enables issuers and service providers to extend their security infrastructures by seamlessly incorporating secure tokens of different form factors, capabilities and price. This represents another important step towards the different secure token infrastructures growing together in the future.
8.3 Outlook

The JavaCard standard moves ahead, and the ongoing specification process for JavaCard 3.0 promises significant and incompatible modifications. The concepts proposed in this thesis represent valuable input for the JavaCard, especially the execution and memory model. The time-line for JavaCard 3.0 is not as tight as for previous releases, and might allow for a reasonable experimentation with new concepts, and a coherent system architecture.

The presented Java based concepts have been mostly implemented on smartcards and only partially on larger tokens like the PCI IBM 4758 token [69]. In a next step, further results can be collected from implementations for other mobile secure tokens such as PCMCIA or USB devices. They offer similar amounts of resources as devices deployed in other embedded environments where different Java specifications exist for [147, 151]. Especially for card readers and low-end smartcard terminals, a software environment based on the presented token system is a natural choice.

The system can benefit from studying other embedded environments, especially regarding proper support of inter-application communication and multiple simultaneous connections. Their resource-friendly, safe and efficient integration constitute another significant step. They affect the base programming convenience and system, and are much more difficult to integrate than other advanced services. Features such as the support for biometric authentication methods can be supported by extensions of existing APIs.

The more the individual issuer environments—Internet, mobile-phone and banking networks—grow together, the more the various transmission, communication and management protocols of secure tokens are harmonized. Whether for instance TCP/IP will be the transmission protocol of choice in all smartcard environments, is not only a matter of technical feasibility, but also a political, economical and social question. The more generality the base system provides, the more flexible future improvements can be integrated.
Bibliography


[122] Rob Pike. Systems software research is irrelevant. online, February 2000. Talk at Lucent Technologies, Seattle, USA.


Curriculum Vitae

PERSONAL DATA

Name: Marcus Oestreicher
Date Of Birth: September 24th, 1971
Place Of Birth: Werneck, Germany
Nationality: German

EDUCATION

1977–1981: Grundschule Eßleben (primary school)
Major: Computer Science
Minor: Philosophy
1993: Vordiplom (Intermediate graduation)
1998: Diplom (Graduation with distinction)

SOCIAL SERVICE

1990–1991: Gemeindekrankenhaus Werneck

PROFESSIONAL SERVICES

since 1997: Research Staff Member at the IBM Research Laboratory Zürich