# Assessing the Future of Smart Card Operating Systems

Damien Deville[♯] Antoine Galland[‡,*] Gilles Grimaud[♯] Sébastien Jean[†]

[♯]University of Lille I,
LIFL/RD2P, M3/111,
59655 Villeneuve d'Ascq, FRANCE
{deville,grimaud}@lifl.fr

[‡]Gemplus Research Labs,
La Vigie, Avenue du Jujubier, ZI Athelia IV,
13705 La Ciotat Cedex, FRANCE
antoine.galland@gemplus.com

[†]INRIA Rhône-Alpes,
LSR-SARDES, 655 avenue de l'Europe,
Montbonnot Saint Martin,
38334 Saint Ismier, FRANCE
sebastien.jean@inrialpes.fr

**Abstract**. This paper aims at presenting past issues, present work, future challenges and work in progress in smart cards operating system design. Smart cards are built to be tamper-resistant platforms whose goal is to serve their owner and to ensure that all interactions between them and the different information systems they are likely to face with, are as safe as possible. The quest for tamper-resistance nowadays implies to confine the computation part in a very small piece of hardware. Consequently smart cards are very constrained nomadic devices. Despite this, they have become more than security tokens over the last twenty years, providing scaled down capabilities of "classical" computing (*e.g.* databases, virtual machines, ...). Nevertheless, evolutions of smart cards operating systems have not taken into account the fact that classical computing was also evolving. Now, smart cards lack some important features which make them too far from what they should be: extensible, reactive, available, efficient.

## Introduction

Smart card is known to be one of the most constrained nomadic devices. It comes with a relatively slow CPU, low communication bandwidth and no more memory than 80's home computers. These constraints are driven by the need to ensure physical and logical security since the card is and must remain, by essence, tamper-resistant. Since tamper-resistance is for the moment uncompliant with large and open systems, smart cards remain (only) tiny trusted platforms.

Smart cards were considered for a long time as secure tokens. The emergence of nomadic computing at the beginning of the nineties however turned them into "jumping" computing platforms, bringing data and service closer and closer to the final-user.

Consequently, the software in smart cards has radically changed over the last twenty years. This has happened for several reasons, smart card software was initially rigid and monolithic and has now become more flexible with a clear separation between "operating system level" and "application level" parts. What is more, application-level resources are now much more accessible (nearly to end user level). Nevertheless, smart cards have evolved separately from an ever more distributed "outside world".

This paper presents the work we carried out in the area of smart card operating systems over the last two years. It also explains our vision of the future of these particular operating systems. For this purpose, the paper also presents work in progress and our research activities for the next two years.

It begins with a survey of smart card software architecture evolution over the last decades. Section 2 follows with a presentation of work that has been accomplished in designing advanced smart card operating systems. It focuses on two contributions to next-generation smart card operating systems. The first, called Camille, relies on the exo-kernel approach to obtain extensibility, without compromising security, raising making operating systems accessible to application designers. The second, called AWARE, reveals the mismatch

[*]Ph.D. Student at Pierre & Marie Curie University, LIP6 Laboratory.

between the smart card execution model and the role it is expected to play in distributed systems and proposes solutions that allow multi-tasking and reactivity. Section 3 finishes by introducing new challenges beyond these projects, the obstacles that need to be overcome before smart cards can reach what might be the ultimate step in their operating system design, and some current work in these directions.

# 1 Two Decades of Smart Card Software Architecture Evolution

Since smart cards' birth in the early eighties, smart card software architecture has never stopped evolving. During this period, numerous standards have been proposed. They are known under the name ISO7816-x, and rule nearly all smart cards features from physical characteristics to application management. It must be remembered that smart cards and their associated software conform to particular life cycles (see Figure 1).

From the point of view of a smart card, there are five parties involved in the life cycle. *Semiconductor manufacturers* are in charge of chip design and mass production. *Smart card manufacturers* (associated in this paper with smart card software producers) *embed* issuers' requirements. *Card issuers* traditionally have more business/behavioral considerations while deploying and managing smart card-based solutions. *Service providers* design and deploy (under the control of issuers) value-added services and *Users* benefit from those services.

The smart card software life cycle comprises three steps. First, software is produced and loaded (*i.e.* embedded in the chip). Then, depending on its nature (ready-to-run or ready-to-load applications), this software is initialized or instanced. Last, embedded software is operated in client/server applications.

In the following, we explain why and how smart card software architecture has evolved, and what are the features of all these smart card operating systems and runtime environments.

## 1.1 From Monolithic Operating Systems to Open Platforms

This subsection discusses smart card operating systems from the invention of smart cards to the advent of open platforms in the mid nineties.

### 1.1.1 Monolithic OS

So called *first-generation* smart card software architecture is a monolith given by smart card manufacturers to semiconductor manufacturer that meets client requirements and chip specifications. We will not go into detailed discussion of *first-generation* systems (see [23] for more details). It is however useful to note that the key factor, when upgrading to *second-generation* systems, is time. Smart card manufacturers (associated with software producers) quickly realized that, as chips benefited from *market* standardization, more and more pieces of code (mainly related to hardware management) were simply *copied*. Software reuse that was supposed to save time and reduce increasingly critical *time-to-market* gave rise to the *second generation*, that is characterized by a *three-part monolith*:

- a set of *hardware management modules*,
- a set of application level modules, well-known and commonly used application-related features such as PIN[1] code management, defining a kind of incomplete application class (which will be further called for that reason application Class- - in Figure 2),
- the code meeting the requirements, with regards to the targeted application, that have not yet been fulfilled by the previous pre-written modules.

In *second-generation* technologies, the first two parts are a response to the software producer's question: "what modules can be reused", the rest is custom designed. In the first two generations of smart card software, no changes could be made after the chip had left the factory.

### 1.1.2 Dedicated Runtime Environments

*Third-generation* software emerged not by considering time but cost. Smart card manufacturers quickly understood that the applications they had to design could be nearly entirely (90%) derived from limited platforms such as file systems or database systems. In this case, software producers no longer ask "which modules can be reused" but rather "which platform best fits the application requirements".

At this point, smart card software was split into distinct parts and became more adaptable. Standard platforms (which can be considered as a combination of hardware management modules and

[1] Personal Identification Number.

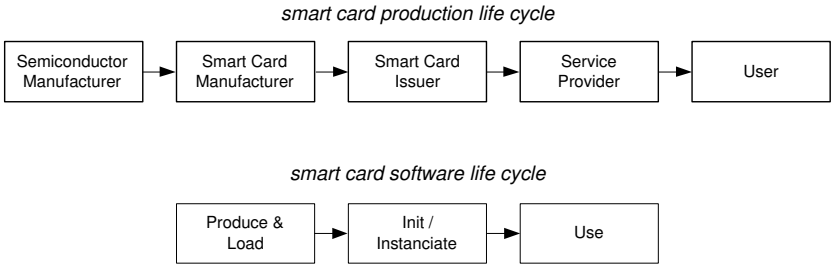


Figure 1: Smart Card Life Cycles

application classes) are embedded, and can further be specialized by smart card manufacturers by *combining* additional components called *filters* (which are functions that have not yet bee incorporated into the standard platform, for example special PIN code management). The application itself is then added by defining data structures and filling them with adequate data. This final step can be performed by service providers even if usually they leave it in the hands of smart card manufacturers.

### 1.1.3 Open Platforms

The transition between the third and the fourth generation is no longer linked to the way the software is produced but to the way the card is used. *Time-to-market* remains a key factor but in this case it involves many more service providers. Indeed, the faster they react, the faster the providers will gain (or retain) customers. Service providers no longer want to *re-burn* a new card when they choose to change the way they serve their clients (especially in mobile communication applications).

Another factor in this change is that service providers learned how to benefit from cooperation (through data sharing). This allows them to build value-added services, which must be enhanced (with code sharing) as the applications become increasingly complex.

In response to these new requirements, smart card manufacturers radically changed the way they design smart card software [22]. Modern smart card platforms offer the opportunity, also known as *post issuance*, to download code into the card while it is in the user's possession. For this to become possible, smart card manufacturers started delivering a *ready-to-load applications* platform resulting from adding a framework (that can be understood as a set of application programming interfaces). This platform usually provides tools that enable data or code sharing[2] and enables upper-layer users (services providers, users) to design and load the applications they want whenever they want. Unfortunately, the most common platforms lack card management, making them *less flexible* (this problem has however been studied and solutions were proposed in [13]). *Fourth-generation* platforms usually rely on a virtual machine [20], both for portability (a single application can be loaded into several different cards, *i.e.* relying on different hardware, without needing to be modified) and for security (it is usually easier to prove or ensure the safety with intermediate codes).

## 1.2 The Evolution of Smart Card Software

The four generations of smart card software architecture described in the previous sections are shown side-by-side on Figure 2 that focuses on the places where the software is managed (either entirely or partially). To be more precise, the first *generation* starts around 1981 with memory and credit cards [23], the second in 1985 with health care cards, the third around 1992 with CQL [21] and SIM[3] cards, and finally the fourth around 1996 with open smart cards like Multos[4] and Java Card [2][5].

By showing smart card operating systems over time, we can see two important facts. An immediate consideration is that, from monolithic model of the early eighties to open platforms model of the late nineties, changes in software architecture have been characterized by a progressive separation be-

[2] One exception is the BasicCard, a commercial product from ZeitControl. It is an open platform that can accept only one application at a time.

[3] Subscriber Identity Module used in Global System for Mobile Communications (GSM networks also know as wireless).

[4] Maosco Ltd., `http://www.multos.com`.

[5] Note that the new generation did not eliminate earlier ones. Current dedicated applications, such as health care cards, conform to the second generation model.

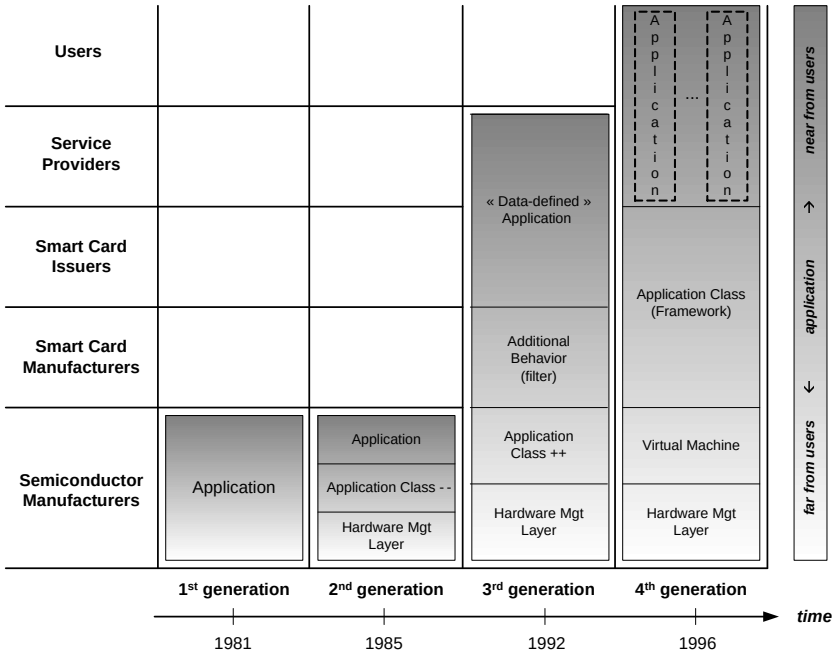


Figure 2: Smart Card Software Architecture Evolution

tween the operating system and applications, placing applications nearer users. Although originally dictated by the needs of smart card manufacturers, this separation ultimately turned out to be very beneficial to the providers. Indeed, application-level and operating system-level parts were no longer managed together, meaning that providers could manage the software life cycle themselves. If the operating system is loaded and initialized at the lowest level, the loading and instantiation of application-level software is moved up to user level. Nevertheless, the hardware management layer is still hidden below application runtime. Furthermore, this layer defines only a few abstractions. It seems utopian to hope that a *super-open-platform* might fulfill all application needs by providing designers with an exhaustive set of abstractions, meaning that application designers are forced to develop the abstractions they need with all immediate and harmful consequences (we have pointed out this in [12] by measuring the performance of a database management on top of the Java Card [2] platform). Smart card software architecture has also evolved following a sequence of steps motivated by *card-centric* considerations, *i.e.* related to the way that an object is handled without much consideration of how the object will be handled in a more general context. Meanwhile, the *outside world* evolved toward distributed computing, creating a gap between the card and its users. As we entered the era of web services, open platforms were found to lack the flexibility inherited from post-issuance by focusing security management exclusively on certificate-checking techniques.

# 2 Recent Advances in Smart Card Operating System Design

The research presented in this paper has been motivated by two main considerations.

First, we need to make smart cards accessible at very low levels to application designers. To prevent drops in performance which occur when too many abstractions are combined, the operating system needs to be raised to the highest level by enabling service providers to design the abstractions closer to the hardware: in other words, the operating system must be extensible.

Secondly, we must bring smart cards closer to the requirements of modern distributed computing. The evolution of smart card platforms must not be driven by current customers needs alone. We must think about further use (mainly as a means of mobility within distributed applications). Smart cards have become multi-application exe-

cution platforms, and application design models have given rise to new requirements that influence card software in several ways, namely administration, interaction, and execution.

In the next two subsections, we will present the research activities that are directly concerned by these considerations. First, we will present results of the Camille project whose goal is to provide an extensible operating system for open smart cards, focusing on the notion that operating system code should not be considered differently from the application code. We then present system issues in the AWARE software architecture that focuses on the way smart card is used in distributed applications, proposing appropriate system-level and middleware-level components.

## 2.1 Earning Extensibility Without Loosing Safety

Camille, designed jointly by the University of Lille [10] and Gemplus' Research Labs, is the first experiment in open operating system design for smart cards.

Camille aims at supporting the various hardware resources used in smart cards. Memory pages, numerical values, microprocessor, and native code can easily be manipulated by applications while ensuring a high level of security. The Camille design approach can be compared to the MIT *Exo-Kernel* [9] with identical principles and concepts (*i.e.* no hardware abstractions but a secure (de)multiplexed access). Embedded code is expressed using a dedicated intermediate language called FAÇADE [11], but source code can be written using several widespread languages. Translators are available in the software infrastructure that are deployed outside the card. For instance, C code can be compiled into FAÇADE using GCC.

The Camille OS provides the following three basic characteristics. *Portability* is inherited from the use of an intermediate code and by a limited set of hardware primitives. *Security* is ensured by a code-safety checking (which uses a PCC-like algorithm [19]) at loading time [25]. *Extensibility* is provided through a simple representation of the hardware that at the *root* of the system does not predefine any abstractions. Memory pages, numerical values, microprocessor, and native code can be easily manipulated by applications while ensuring strong security. Thus, applications have to build or import abstractions which match their requirements.

Embedded code is expressed using a dedicated intermediate language called FAÇADE [11], but source code can be written using several widespread languages. Portability is inherited from the use of such intermediate code. Security is ensured by a code-safety checking (which uses a type-inference algorithm) during loading. If a unit of code passes the loading step, its safety can be considered proven.

The usual downside of extensibility is performance. For some parts of the OS that require efficiency, Camille uses *Just-in-Time* techniques to compile intermediate code into native one. Smart cards by definition have limited computing power. Hence, the major optimization of intermediate code must be performed outside of the card, on the terminal. Only back-end optimizations can be performed by the on-board compiler. Increased performance also comes from the exokernel approach that does not introduce abstraction penalties in the core of the OS. Because smart cards have limited computing power, additional hardware independent optimizations are also performed out of the card, while the source code is translated to FAÇADE.

Experimental results validate the approach used in Camille. A prototype was designed on an AVR chipset from ATMEL (32 kb of ROM, 32 kb of EEPROM and 1 kb of RAM). Camille represents 17 kb of native code (3.5 kb for verification, 8.5 kb for native code generation, and the rest for hardware management). These numbers demonstrate that an on-board code generator suits smart cards constraints.

Table 1 shows that an average of three AVR instructions is generated for FAÇADE instruction, this means that the FAÇADE intermediate language is very close to hardware. It also indicates that half of the time is spent storing generated code in EEPROM pages, rather than producing the code. A significant gain in speed can thus be obtained by using a software cache for code writing.

On the one hand, on-board code generation becomes possible since the FAÇADE intermediate language is ready to compile (*i.e.* for example, the code generator can use FAÇADE code annotations to determine register mappings and data life cycle). On the other hand, code is less compact than the same code for a stack-based machine (this is why FAÇADE code is 1.8 larger than Java byte code for example). Furthermore, the overhead between Java Card (where byte code verification is performed off card) and FAÇADE in terms of compactness can be explained by the fact that a proof

| Measure | Average | Standard deviation |
|---|---|---|
| Number of cycles to process a FAÇADE instruction | 80727 | 26926 |
| Number of cycles ignoring writes in persistent memory | 37858 | 4596 |
| Number of AVR instructions per FAÇADE instruction | 3.05 | 1.04 |
| Size of the proof compared to FAÇADE code size | 30% | 3% |
| Size of the code compared to Java byte code | $\times$ 1.8 | $\times$ 0.6 |

Table 1: Measures Related to Code Generation

| Measure | Average | Standard deviation |
|---|---|---|
| Size of native code generated | 2934 bytes | - |
| Size of of typing information | 1539 bytes | - |
| Loading time | 4.194 s | - |
| Access time to a byte of a file of size less than 2kb | 25.9 $\mu$s | 6.4 $\mu$s |
| Access time to a byte of a file of size greater than 2kb | 52.3 $\mu$s | 30.4 $\mu$s |

Table 2: Measures related to ISO7816-4 file system

is added to the code to allow on card verification (the proof accounts for thirty percent of code size).

To prove the extensibility of Camille, a file system (conforming to smart card standard ISO7816-4) has been implemented. The results are summarized in Table 2. The first thing that should be noticed is the size of the file system abstraction. This proves that twelve abstractions with the same level of complexity can be built on top of the exo-kernel in a standard smart card. The loading time is considerable, but it is only performed once. Compared to a similar implementation in a Java Card, the access time to a byte is 20 times more efficient in Camille. If we face to an equivalent implementation in C language (without safety proof), the access time to a byte of data is similar and in some cases slightly faster (7%). The efficiency gap between Java Card and Camille implementation is due to the optimization performed by the on-card native code generator of Camille. On particular experiments (random bit generator), Camille is 70 times faster.

As a conclusion, Camille proves that recent work on extensible operating systems is indeed applicable to smart cards.

## 2.2 Bringing Smart Cards Closer to Distributed Computing

Smart cards are communicating devices that have to be plugged in a reader (also called *card acceptance device*) in order to be linked to remote applications. The communication interface relies on a request/response protocol derived from the OSI model. A smart card sequentially processes each incoming APDU[6] command sent by the reader. In the case of smart cards that host several applications, this processing can be seen as a remote procedure call or a method invocation (depending on the considered platform). One must notice that in this case, underlying runtime environments are not multi-task (application execution starts and stops with unitary APDU processing).

Smart cards have always been integrated into distributed applications using the same hypothesis: the smart card is a data/application server requested by a unique client using a synchronous client/server communication protocol. Nevertheless, certain models have raised new requirements for smart card execution models.

**Reactivity.** Smart cards are often used to ensure security. Some research work has demonstrated that security can be enhanced by giving smart cards a more active role: one example is the transaction coordinator proposed in [15]. Enabling this "reactivity" property means reversing the communication scheme by allowing smart card applications to be clients.

**Availability.** Since smart cards are off-line most of the time, you have to use them as efficiently as possible when on-line. In the case of open platforms, this implies that an embedded service must start when ready. That is what we refer to as "availability." Many applications use an asynchronous model that does not exist for today's smart cards. An example is the management of a fleet of thousands cards. When a service provider wishes to update a service, he deploys the change to each card, when he would rather send asynchronous commands that do not force him to wait

[6] Application Protocol Data Unit.

for each connection.

**Concurrency.** For the same reasons of efficiency, smart cards can no longer be considered as part of a one-to-one dialog. When using an open smart card, we can make no hypothesis about the number of clients for the embedded applications.

#### 2.2.1 Multi-Tasking

In order to fulfill the previously-identified requirements, a multi-tasking runtime environment has to be designed for two main reasons.

Concurrent accesses to the smart card by remote clients is the first problem to solve because it implies the ability to restore adequate internal states before each APDU is processed. Because applications involving smart cards are traditionally in three pieces (*i.e.* client-reader-card), the solution can rely on one of these three components. If we do not offer concurrency management on the client side, this amounts hiding smart card access behind a *mutual exclusion-like* API. This idea, proposed in [14], is not a viable solution because it hinders availability; a client may have to wait very long before gaining access. Placing the smart card terminal in this role, if it brings transparency to clients, is not a viable solution either because this implies that the terminal can restore the internal states of the smart card. In such a case, the smart card must provide at least a *get-set context* API which weakens security. For the previously mentioned reasons, the only viable solution is to have the card manage concurrency, this nevertheless implies that the card will be able to manage several application contexts at the same time (*i.e.* multi-tasking).

If we refuse to choose between reactivity and availability, there is a clear need for a multi-tasking environment. In a mono-tasking model, if an embedded application $A$ is frozen because it is waiting for the result of a remote procedure call, and if another application $B$ is requested by a client $C$, the dilemma is that if you want to serve $C$, the application $A$ will not be able to restart (the associated internal state will be lost), while keeping $A$ alive forbids us from starting $B$.

A multi-tasking runtime environment is thus mandatory. This however consists in managing several applications concurrently rather than managing several processes or threads simultaneously. In open platforms, virtual machines are often merged with operating systems. If we accept these requirements, our model is a virtual machine able to manage several application contexts.

#### 2.2.2 Exo-scheduling

The execution model chosen for AWARE is non-preemptive. This is motivated by smart card hardware constraints (writing in non-volatile memory is so costly that it would not make sense to interrupt an application when processing an APDU, Table 4 confirms this). The nature of the events inducing changes in application execution also suggested this model. These events (starting a new session[8], ending a session, switching sessions[9], performing a remote procedure call, receiving the answer of a pending remote procedure call), are implicitly linked to a non-preemptive scheduling policy because they occur either just before processing an APDU or when an embedded application stops.

If a non-preemptive policy is sufficient, one problem remains: without the terminal, the card cannot foresee when these events will occur. So the main idea of AWARE is that the terminal, which has a better view of what happens outside, will indicate to the card what to do. That is what we call *exo-scheduling*. In AWARE, the scheduler is split into an internal part that is truly in charge of saving and restoring application contexts and an external part that remotely controls the internal part by using a set (we call it VFD) of some scheduling-oriented APDU commands. It is worth noting that the solution chosen for AWARE does not solve any of the well-known security problems induced by the use of the terminal, but it does not raise any new problems either. As in Camille, externalizing crucial functions without weakening security is certainly helpful. A minimal infrastructure must be deployed on the terminal (it is also in charge of managing remote procedure calls) but this is relatively light. Besides it, an adequate middleware and associated dynamic configuration mechanisms have been designed in order to provide with publish/subscribe capabilities. Since this section focuses on operating system design issues, interested readers should refer to [12] for a more detailed presentation of the whole distributed software architecture.

[7] The *main* of every smart card operating system is a loop waiting for an APDU and branching to an associated function.

[8] by session we mean a dialog between a remote client and an embedded application.

[9] *i.e.* processing a command related to a session other than the current one.

| Description | Bytes In | Bytes Out | Duration |
|---|---|---|---|
| APDU router[7] "switch-case" processing | 0 | 0 | t0 = 54 ms |
| Empty Method Call | 1 | 0 | t1 = 76 ms |
| Remote Procedure Call Sending | 1 | 0 | t3 = 234 ms |
| Reactive Command Fetching | 0 | 4 | t4 = 60 ms |
| Suspended Session Resuming | 4 | 0 | t5 = 116 ms |
| Context Switching | 0 | 0 | t7 = 54 ms |

Table 3: Some Reference Measurements

#### 2.2.3 Measurements

In order to validate and evaluate AWARE's proposals, a prototype was developed using a CASCADE hardware emulator whose architecture was based on a 32-bit RISC microprocessor paced at 17 MHz with 512 bytes of RAM, 32 kb of ROM and 16 kb of Flash RAM. The implementation of AWARE's multi-context Java Card Virtual Machine has then been done by adapting the *GemXpresso0* prototype (*i.e.* the prototype of the first Gemplus Java Card, compliant with the 2.0 version of the standard.) based on the CASCADE chip. The prototype manages three simultaneous tasks, but the number of manageable tasks depends on the amount of Flash RAM that you want to devote to this task.

In order to obtain context-related duration measurements, we performed a set of tests. The results, presented in Tables 3 and 4, are average values obtained after 500 loops.

The context saving duration (T2) confirms the pertinence of a non-preemptive scheduling policy, most of the related execution time being dedicated to save the RAM-located execution stack into Flash RAM. The cost induced by context suspending is not prohibitive here, but it is entirely inadequate with a preemptive scheduling policy, where the overhead would be too great with regard to the typical method call execution duration.

Table 5 presents memory footprints for some pertinent elements. Context-related functions are not big RAM consumers, only 153 bytes are consumed. This consequently allows us to add other useful system-level mechanisms such as a cache, ... Flash RAM occupation is also relatively limited. In the prototype, the three contexts require a total of about 2 kb of persistent memory, this is reasonable with regards to the 32 kb commonly available now. Thus, more space remains for application code storage.

Our experiments have consequently validated our proposals and demonstrate the viability and the pertinence of proposed mechanisms. They particularly allow us to affirm that the coupling of a non-preemptive multi-tasked runtime environment and an exo-scheduling mechanism is a viable solution both in terms of performance and memory footprints.

# 3 Challenges for Next Generation Operating Systems

In the previous subsection, we have explained how a non-preemptive multi-tasking runtime environment could comply with current generation smart card constraints. There is no doubt that emerging technologies will soon allow us to embed a preemptive concurrency model. So, when this occurs, new problems will have to be solved. First, all applications have different needs in terms of resource management. Some applications will need bandwidth, others will need processing time, ... Their particular requirements have to be taken into account. Furthermore, if open models have led to multi-purpose smart cards, an application should not necessarily suffer from the presence of others (*i.e.* the way it runs should be the same).

The two major challenges we have to face are resource control and real-time. This task becomes harder if you try to adopt these properties without giving up extensibility. The following section introduces the issues of the early work in this context.

## 3.1 Resource Control

From the emergence of smart cards [23] to the present, integrating a high level of safety with so few resources has always been the main challenge of smart card manufacturers. Hence, one has to keep in mind that optimizing physical and logic resources usage is of prime importance in such

[10] N represents the number of managed contexts.

| Instruction or instructions sequence | Deduced duration |
|---|---|
| VM starts, empty method invocation and VM ends | `T0` = t1 - t0 = 22 ms |
| Remote Procedure Call Marshaling | `T1` = t2 - t1 = 69 ms |
| Context Saving | `T2` = t3 - t2 = 89 ms |
| Context Restoration | `T3` = t5 - t1 = 40 ms |
| Context Switching | `T4` = t7 - t0 = $\varepsilon$ |

Table 4: Deduced Durations

| Element | Storage | Size |
|---|---|---|
| Context-Related Functions Code | ROM | 2608 bytes |
| Full Embedded System Code | ROM | 18592 bytes |
| Persistent Context Data | Flash RAM | (651 * N + 3) bytes[10] |
| Volatile Context Data | RAM | 153 bytes |

Table 5: Memory Footprints

a constrained system. Today, modern smart card platforms offer the opportunity to download code into the card while it is in the user's possession[11]. This new functionality raises new problems as far as the security of mobile code for smart cards is concerned, since hostile applets can be developed and downloaded into the card. In Java Card [2], various solutions have been studied to integrate a Java bytecode verifier into a smart card in order to make sure that programs are well-typed [16, 1, 7]. After type-safe verification, resource control is the logical next step to ensure reliability [6]. Indeed application providers would like guarantees that their applets will have all the required resources for safe execution throughout their lifespan.

#### 3.1.1 Description of the problem

When mobile code is uploaded to a new host, there is no guarantee that there will be enough resources to complete its execution. The most commonly adopted solution is to use a contract-based approach of resource management [26]. In this approach, each applet must declare its resource requirements in a contract. Once the smart card accepts a contract, it must meet its requirements during the applet's lifespan. Since the uploaded applet is not considered as trustworthy, safeguards must be established. Runtime techniques are generally used to control allocations, which implies costly monitoring. Moreover, when the contract is cancelled, it is often too late to recover applet execution even if a call-back mechanism can be used [5].

[11] *post issuance*

To reduce runtime extra-costs, it may be preferable to check once and for all whether an applet respects its own contract. This generally implies bounding its resource consumptions by means of static control-flow analysis or type systems. These techniques are complex and usually take into account a subset of the targeted language. For the same reason, it is also difficult to incorporate them directly into a smart card. In this case, security policies require that resource bounds computed off card be checked on card before being used [18].

Once a smart card commits itself to respecting specific resource requirements, the simplest solution to ensure availability is to reserve and lock all the required resources at start-up. This is the solution used in Java Card [2] for heap memory allocation. Smart card developers gather all necessary memory initializations needed at start-up to avoid running out of memory later. This implies that additional memory allocations cannot be ensured.

The drawback of this solution is the waste of resources when multiple applets are used. Indeed applets will seldom use all their reserved quotas simultaneously. On the contrary it is more likely that peaks of resource usage will occur at different times. Moreover, if this is not the case, we might consider delaying one or more tasks to avoid this situation. In short, resource usage is currently far from being optimized.

Our objective is to guarantee resource availability to every applet, while minimizing the global needs of the system. Most approaches described

above do not solve these two problems simultaneously. Thus we are looking for a framework that is more economical than one that blocks all resources at start-up, while offering the same level of dependability.

#### 3.1.2 Our approach

Reserving all the required resources at start-up is a simple and efficient way to guarantee resource availability. The drawback of this solution is the waste of resources as it is scaled up. It would be interesting to balance the resource requirements according to the execution time, because we could reserve resources "just in time". This solution becomes advantageous in a multi-application context when the resources reserved but not used at a given time by one application can be used by another.

In most systems, when a program requests more resources than the amount available, an error is reported and the task terminated. For our purpose, however, a thriftier solution would be to suspend the requesting task temporarily, in the hope that some resource might be released later. Our approach is to schedule applets according to the resource requirements in order to minimize the global needs of the system. Resource requirements are not set but inferred because they must be trustworthy. The scheduler, then, uses these resource consumptions to schedule applets. The scheduling policy must:

- guarantee that each applet will always meet resource needs,
- minimize the global needs of the system (*i.e.* the amount of available resource).

We propose an architecture that can both guarantee the availability of one resource and optimize its usage with a deadlock-avoidance algorithm [8]. Its specificity lies in its ability to improve the task scheduling in order to spare resources in a multi-process operating system. Our architecture comprises two parts. The first statically computes the resource needs. The second guarantees at runtime that there will always be enough resources for every application to terminate, thanks to an efficient deadlock-avoidance algorithm.

Due to the limited computing power of smart cards (memory and CPU), not all operations can be done on card. Therefore, we use a split architecture (off card/on card) to delegate complex computations off the card. The off-card part is in charge of resource prediction on Java applications while the on-card part prevents resource deadlock through a resource server.

### 3.2 Real-Time

First of all, what is the motivation for introducing real-time in smart card operating systems? In current smart cards, some software are designed to support real-time hardware constraints. As defined in ISO7816 standard, a smart card operating system must answer terminal queries within five seconds, otherwise the card will be considered mute and thus unusable (this is an implicit deadline). A more complete example can be found in Java SIM card used by GSM phones, in which two kinds of applications co-exist. The main one is the application that generates cryptographic session keys between cell phone and BTS[12]. Sessions keys need to be delivered with regard to a firm deadline (one per communication unit used). SIM application is so close to the operating system and to the virtual machine that an application developer would not be able to write it using the standard application development scheme and tools. For example, SIM provides context commutation to ensure the respect of the deadline linked to cryptographic key generation. Smart card operating systems clearly have RT deadlines.

The standard exo-kernel architecture, as it has been defined by the MIT, does not offer any dedicated real-time primitives neither to applications nor to extensions. The exo-kernel is designed to support extensibility while ensuring maximum security. It just offers "equity" for the access to the processor to each "system" [13]. For each CPU time slice the exo-kernel uses a "round-robin" policy to elect an application (a *yield* primitive is given to extensions which want offering the remaining time of a slot to others). The main motivation in Camille RT is to offer to user applications and to system extensions the capability to implement real-time primitives, and to make standard applications coexist with real-time applications.

The key points to support real-time in an extensible operating system bases on exo-kernel architecture are:

**(*i*)** quantify the execution time for each of the exo-kernel primitives (*e.g.*, delay related to a virtual TLB access),

**(*ii*)** find a schedule that succeeds all running applications deadlines,

[12] Base Transmission Station.

[13] extension in the exo-kernel architecture.

**(*iii*)** define a way to allow applications to notify their real-time requirements (deadline, rate, start time, ...) to the exo-kernel,

**(*iv*)** quantify the execution time for each of the RT code expressed in FAÇADE intermediate language.

The first three points concern the real-time problematic in a standard exo-kernel, the last one is specific to Camille as it uses an intermediate language to increase the portability.

#### 3.2.1 Real-Time and Downloaded Code

Dealing with real-time constraints requires the knowledge of execution time for every RT code. As Camille is an open extensible operating system, applications are converted to or written in an intermediate language named FAÇADE. In its current form, it offers extensibility at both user and OS level, but does not support WCET computation. Algorithms for WCET [3] and languages allowing easy WCET computation are well known [4, 27] but have the drawbacks of dramatically constraining application programming. We then have to propose a FAÇADE sub-model, more restrictive, used for developing real-time tasks and extensions and use the existing one for generic applications. Thus standard and real-time applications can coexist in Camille. WCET computation need to be performed in the card for different reasons. The main one is that FAÇADE is compiled "on the fly" by the card, thus only the card exactly knows the time needed for executing each FAÇADE instruction because it depends on the physical platform. Computing WCET outside the card would be possible by exporting a "chip profile" containing the exact code generated by the on fly compiler and the cycle number for each CPU's instructions. Smart card manufacturers do not like these informations to be public for industrial and economical purpose such as keeping secret the detail of technologies of smart card chip, and also to prevent timing attack or DSA [17] of cryptographic softwares/hardwares.

#### 3.2.2 Real-Time and Kernel

To extend Camille into a real-time operating system, we need to quantify the execution time for every primitives of the exo-kernel. Exo-kernels only offers simple primitives to access efficiently and safely the hardware. Thus, due to their simplicity, exo-kernel's primitives are easy to bound in time. Similar analysis have been performed on other real-time kernel code like RTEMS [3]. Another "tricky" way is to compile the code of the Camille kernel using our GCC and to analyze the resulting FAÇADE code using our WCET algorithms and tools.

The major technological challenge on a smart card is linked to persistent memory (EEPROM) which presents annoying electronic properties. Writing one page is slow (it takes about 5ms) and deny any access to the current physical page within the whole writing delay. EEPROM writing operations implies hardware locks. This challenge may create difficulties to predict primitives duration to compute their WCET.

#### 3.2.3 Scheduling and Exo-Kernel

Provided with the knowledge of the execution time required for each of the exo-kernel primitives and also by each of the downloaded applications, the problem is now to find a schedule for a set of standard and real-time tasks. Exo-kernels only support "equity" in term of access to the micro-processor using a "round-robin" policy (simple and impartial). The time slices are granted to operating system extension which elect a runnable application according to their own scheduling policy. Having a *round robin* ensure that the real-time extension will have access to the processor for one quantum of time each $N$ quantum ($N$ stands for the extension number in the exo-kernel). Each real-time extension can schedule its own tasks and match deadlines regarding this property. Nevertheless, when there are many real-time extensions, real-time scheduling may not be optimal. A problem may occur when loading a new task. Real time tasks that have validated their real-time requirements with a rate of $\frac{1}{N}$ will tend to have problem meeting their deadline with a new rate of $\frac{1}{N+1}$. The simplest solution is a *vote* to accept or reject the new extension. The user could solve problems by deleting incompatible system extensions. This two levels CPU sharing process is close to hierarchical schedulers [24]. It is this sufficient to build a RT scheduler over an exo-kernel, but it introduces sub-optimal solutions.

The key points we have presented prove that the exo-kernel, real-time operation and an extensible open operating system are not mutually antagonist. Nevertheless, some points are mandatory to support real-time operation on top of exo-kernel. Quantifying execution time and optimizing the

scheduling are exo-kernel problems, computing application execution time and giving a way for the application to notify its requirements to the kernel concern more Camille design and architecture.

# 4 Conclusion

In the last two decades, the smart card industry has moved from pure hardware design to embedded software providing, also enabling third-party developers to easily build applications on top of "wide spectrum" execution environments. Despite the necessity for smart card software developers to preclude the use of conventional solutions, smart card software has evolved according to conventional software engineering practices. The emergence of operating systems was initially motivated by simplifying application design while removing hardware management code from functional software. Today's operating systems provide useful abstractions of the hardware functionalities. However, a large panel of abstractions can be provided according to different *kinds of* or *classes of* applications, each application class promoting a *preferred* abstraction (*i.e.* a preferred operating system).

Nowadays, smart cards industry provide application designers with few standard execution platforms (and, for each, a set of APIs) that allow to load, remove and execute applications written no longer in 80's assembly language but in well-known higher level languages such as Java. Nevertheless, today's smart cards use reaches the limit of "extensibility-by-application-only". New perspectives must be opened, such as efficient memory management, multiple data transport layers, ... Multi-tasking is one of such underlying system abstractions. However, building a multi-tasking runtime environment for smart cards is not trivial with regards to devices constraints. Nevertheless, this can simply be brought by safely balancing efforts between the card and the terminal in which it is plugged. That is what AWARE does, also opening the card to an ever distributed computing world. An immediate conclusion in smart card operating system design is always that advances induce benefits only if an adequate middleware is coming with. It makes sense to try to increase smart card runtime environment usage with multi-tasking. But this sounds better when providing smart cards applications with efficient means to access remote resource and services and when providing remote applications to be able to interact with smart card applications from anywhere at anytime (considering that smart cards are mainly disconnected).

However, applications need much more system-level abstractions that are not offered by current platforms, and since these platforms are supposed to be generic they cannot offer all possible abstractions. To allow the largest panel of applications running on the same card, its operating system must be able to define new hardware abstractions. Today, Camille is the only prototype of such a safely extensible smart card operating system. This exo-kernel allows defining new hardware abstractions and applications, from a basic hardware interface. The safety guarantee enforces security between the smart card and the *outside world* but also between different sets of embedded code. Safety guarantees, however, only concern confidentiality and integrity. And yet, an application's activity can today be stopped by a denial of service attack. In this context, resource control becomes an important part of the safety guarantees when untrustworthy software can be loaded over this trusted computing basis. Smart card operating system research now needs to face the new challenges of resource control and real-time. Overriding the limits of operating system extensibility according to *resource safe* systems now seems to be a key problem in smart card operating system research for the next decade. As usual, the way residing in smartly coupling external complex computation and internal lightweight verification seems to be the green mile to follow.

# References


[1] Ludovic Casset, Lilian Burdy, and Antoine Requet. Formal Development of an embedded verifier for Java Card Byte Code. In *the IEEE International Conference on Dependable Systems & Networks*, Washington, D.C., USA, June 2002.

[2] Zhiqun Chen. *Java Card™ Technology for Smart Cards : Architecture and Programmer's Guide*. The Java™ Series. Addison Wesley, 2000.

[3] A. Colin and I. Puaut. Worst-case execution time analysis of the RTEMS real-time operating system. In *13th Euromicro Conference on Real-Time Systems*, pages 191–198, Delft, The Netherlands, june 2001.

[4] Karl Crary and Stephanie Weirich. Resource Bound Certification. In *the 27th ACM Symposium on Principles of Programming Languages*, 2000.

[5] Grzegorz Czajkowski and Thorsten von Eicken. JRes: A Resource Accounting Interface for Java.

In *Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 21–35, Vancouver, British Columbia, Canada, October 1998.

[6] Damien Deville, Antoine Galland, Gilles Grimaud, and Sébastien Jean. Smart Card Operating Systems: Past, Present and Future. In *the 5th USENIX/NordU Conference*, Västerås, Sweden, February 2003.

[7] Damien Deville and Gilles Grimaud. Building an "impossible" verifier on a Java Card. In *2nd USENIX Workshop on Industrial Experiences with Systems Software*, Boston, USA, 2002.

[8] Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[9] Dawson R. Engler and M. Frans Kaashoek. Exterminate All Operating System Abstractions. In *the 5th IEEE Workshop on Hot Topics in Operating Systems*, Orcas Island, USA, 1995.

[10] Gilles Grimaud. *CAMILLE : un système d'exploitation ouvert pour carte à microprocesseur*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille (LIFL), 2000.

[11] Gilles Grimaud, Jean-Louis Lanet, and Jean-Jacques Vandewalle. FAÇADE: A Typed Intermediate Language Dedicated to Smart Cards. In *Software Engineering–ESEC/FSE*, volume 1687 of *LNCS*, 1999.

[12] Sébastien Jean. *Models and Software Architectures for Internal and External Cooperation in Open Smart Cards*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille (LIFL), 2001.

[13] Sébastien Jean, Didier Donsez, and Sylvain Lecomte. Using some Database Principles to Improve Cooperation in Multi-Application Smart Cards. In *the 21st IEEE International Conference of the Chilean Computer Science Society*, Punta Arenas, Chile, 2001.

[14] Roger Kehr, Michael Rohs, and Harald Vogt. Mobile Code as an Enabling Technology for Service-oriented Smart Card Middleware. In *the 2nd IEEE International Symposium on Distributed Objects and Applications*, Antwerp, Belgium, 2000.

[15] Sylvain Lecomte, Gilles Grimaud, and Didier Donsez. Implementation of Transactional Mechanisms for Open Smartcards. In *GEMPLUS Developer Conference*, 1999.

[16] Xavier Leroy. Bytecode Verification for Java smart card. *Software Practice & Experience*, 32:319–340, 2002.

[17] T. S. Messerges, E. A. Dabbish, and R. H. Sloan. Investigations of power analysis attacks on smartcards. In *USENIX Workshop on Smartcard Technology*, pages 151–162, 1999.

[18] Mobile Resource Guarantees (MRG). European Project IST-2001-33149. `http://www.dcs.ed.ac.uk/home/mrg/`.

[19] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997.

[20] Pierre Paradinas. Smart Card Operating Systems: Overview and Trends. In *8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Schloss Elmau, Germany, May 2001. Invited Speaker.

[21] Pierre Paradinas and Jean-Jacques Vandewalle. A Personal and Portable Database Server: the CQL Card. In *Application of Databases*, volume 819 of *LNCS*, Vadstena, Sweden, 1994.

[22] Pierre Paradinas and Jean-Jacques Vandewalle. New Directions for Integrated Circuit Cards Operating Systems. *Operating Systems Review*, 29(1):56–61, January 1995.

[23] Jean-Jacques Quisquater. The adolescence of smart cards. *Future Generation Computer Systems*, 13:3–7, 1997.

[24] J. Regehr. *Using Hierarchical Scheduling to Support Soft Real-Time Applications on General-Purpose Operating Systems*. PhD thesis, University of Virginia, 2001.

[25] A. Requet, L. Casset, and G. Grimaud. Application of the B formal method to the proof of a type verification algorithm. *HASE*, 2000.

[26] Nicolas Le Sommer and Frédéric Guidec. A Contract-Based Approach of Resource-Constrained Software Deployment. In *the 1st IFIP/ACM Working Conference on Component Deployment*, volume 2370 of *LNCS*, pages 15–30, Berlin, Germany, June 2002.

[27] Scott Thibault, Jerome Marant, and Gilles Muller. Adapting Distributed Applications Using Extensible Networks. In *the 19th IEEE International Conference on Distributed Computing Systems*, 1999.