Enforcing Portability and Security Policies on Java Card Applications

Eric Vétillard and Renaud Marlet,
Trusted Logic
5 Rue du Bailliage – 78000 VERSAILLES

`Eric.Vetillard@trusted-logic.fr`, `Renaud.Marlet@trusted-logic.fr`

1   Introduction

The Java Card 2.1 specification was introduced almost five years ago. At that time, the expectations were very high in terms of impact for developers. There would immediately be "millions of Java developers" ready to develop Java Card applications, allowing hundreds or "killer smart card applications" to be developed and widely deployed in only a few years.

Things did not quite turn out that way. Until now, the millions of developers did not turn out, and most Java Card applications have been developed by about the same actors who previously developed native applications (card manufacturers, card issuers, and a few specialized software companies). There have been many reasons for this situation, but one of them undoubtedly is that the Java Card programming model is closer to the standard smart card programming model (counting the bytes used and watching for performance bottlenecks) then it is from the mainstream Java programming model. In particular, the memory management constraints are difficult to handle for most Java programmers [1].

Nevertheless, the situation is evolving, and there have been many new Java Card programmers in the last year, in particular in emerging markets such as North America and some parts of Asia. With these new developers, who have little experience about smart cards and about security software, the overall quality of the developed applications has dropped significantly. The direct consequence is that the risks related to the applications themselves have increased sharply, which has become an overall concern for the industry.

In previous years, the interoperability of smart card platforms has been the main concern for many issuers, in particular in the wireless market. There have been tremendous progress in this area, as the specifications were consolidated (moving from release 2.1 of the Java Card specification to releases 2.1.1 and 2.1.2, introducing GlobalPlatform release 2.1, …), and the implementations became more interoperable.

Today, the focus is on the applications, and there is a strong need for more methodology in the development of Java Card applications. The present paper addresses some aspects of the methodology of Java Card application development, and it focuses on portability and security issues. After presenting the issues, it introduces ways to address them by defining issuer policies, and it finally presents a validation technology that can be used to automatically enforce the good usage policies defined by the operator.

2   Issues to be addressed

The issues to be addressed are all related to the overall quality of applications. However, there are different categories of issues, which correspond to different needs. For instance, the wireless industry focuses on portability, which is extremely important for them, as they usually need to deploy an application on a wide array of cards. On the opposite, the banking and identity industry rather focuses

on security issues, since the applications that they need to manage on their smart cards are highly sensitive.

The aim of the present section is to describe some of the issues to be addressed when developing Java Card applications. These issues are here presented as examples, and the list does not mean to be exhaustive.

2.1   Portability

The Java Card specification [4] goes a long way into defining the interoperability of Java Card platforms, and these efforts have been quite successful. Nevertheless, there remain a few obstacles to the portability of applications:

- Versioning of Java Card virtual machine and APIs.
- Optional features.
- Partially specified features.
- Memory consumption patterns.

The first issue is that there have been several releases of the Java Card specifications. Cards have been deployed with different virtual machine versions, as well as different versions of API packages. For instance, there are today two versions of the Java Card virtual machine (corresponding to releases 2.1 and 2.2), and even three versions of the `javacard.security` API package. In addition, there are vertical APIs corresponding to each industry, which have also been evolving over time, which makes the issue even more complex.

The second issue is that some features of the Java Card specification are optional. The most publicized such feature is the support for 32-bit integers, which is optional on all Java Card specifications. The `javacardx.crypto` package also is optional, in order to enforce export restrictions when they are applicable. More importantly, Java Card does not mandate the presence on a card of any cryptographic algorithm implementation. Developers can therefore never assume that a specific algorithm will be supported on all cards. In addition, even if an algorithm is supported (*e.g.* DES), it does not mean that it is supported for all key lengths, or that all the padding algorithms that are associated to it are also supported.

The third issue concerns only a few detail points in the specification. For instance, when the execution of an application fails with an error (usually some kind of overflow), the specification describes a minimal behavior, but it also clearly indicates that additional measures can be taken by the card platform. For instance, a card platform may block the application in order to ensure that the error will not occur again, while another will simply adopt the minimal behavior. Similarly, when an object allocation is performed during a transaction and that transaction is then aborted, the specification indicates that the Java Card platform "may" reset the card, which is of course very annoying. There are a few other points in the specification which may affect the behavior of some applications on some cards.

The fourth issue is related to the way in which the Java Card platform consumes memory. There are here various factors that affect an implementation's memory consumption pattern. First, the implementation of the API objects may be different, leading to different object sizes. The difference may be quite important, in particular for cryptographic keys, which are typically stored in a way that includes redundant data, in order to thwart typical cryptography attacks. Then, the implementation may have different allocation strategies, and the allocation may be exact, or based on a block allocation, with various block sizes.

All of these issues can lead to non-portable applications, *i.e.* applications that do not exhibit the same behavior on two cards that faithfully follow the Java Card specification. Of course, each issue needs to be dealt with in a separate way:

- For versioning issues, it is important not to use features introduced in a later version without a good reason to do so (in order not to restrict the proportion of available targets). It is also important to convert the applications with the appropriate export files.

- For the optional features, developers need to have enough information about the target cards in order to know which algorithms are supported on all cards.
- For the partially supported features, the best solution is to avoid them altogether, since they usually correspond to unusual and avoidable situations.
- For the memory consumption pattern, the only solution is to get the actual information by running the application on all targeted card platforms, or to use another tool that will compute this information.

It is important to notice that, in that specific matter, developers cannot address the issue in a satisfactory manner unless they get the appropriate information from the card issuer, in particular regarding the targeted cards.

2.2 Safety

Safety issues are halfway between portability and security issues. The examples of safety issues that we will list below all correspond to features that are unambiguously specified by the Java Card specification, so there is no portability issues. These issues will not either directly affect the security of the application, since they should not corrupt the state of the application. However, these issues correspond to situations that the issuer does not want to see, usually because they indicate that the application may not have been thoroughly tested. Examples of safety features include:

- Proper management of transactions.
- Proper management of exceptions.
- Proper management of SIM Toolkit handlers.

About transactions, the Java Card specification clearly indicates that any transaction that is not closed by the application will be aborted by the JCRE the application returns. This is definitely a portable behavior, but it is not good practice, and it is much better to have the application close the transaction; even if this means aborting the transaction. At least, the issuer will be certain that the transaction abortion is voluntary. Other typical issues concern the use of sensitive or authentication objects during a transaction, since some actions may be undone without the developer knowing about it. More generally, good practice should lead to use transactions only in order to ensure the atomicity of complex persistent memory updates, and transaction code should be kept as small as possible.

About exceptions, the basic rule is to catch all exceptions except the **ISOException**'s that are used to indicate status words to the JCRE. That is particularly true in sensitive applications, where the "6F00" status word, indicating an uncaught runtime exception, could be interpreted by a terminal as a general failure of the card and/or application, and may lead to the blocking of the application. Additional safety issues regarding exceptions are to avoid **SecurityException**'s altogether, since they are used to indicate firewall violations. An application should not throw such an exception, and it should not attempt to catch it either. More generally, applications should abstain from throwing system exceptions, and they should include specific handling for all possible exceptions.

About SIM Toolkit handlers, the issue is related to the availability of the handlers. Because the handlers are overlaid, and because they have specific functions, they are not always available, and they cannot therefore be used freely. For instance, the **ProactiveResponseHandler** can obviously not be accessed before to send a proactive command. There also le ss obvious rules: during the processing of some SIM Toolkit events, it is not legal to send a proactive command; the **ProactiveHandler** is therefore not available during the handling of such events. Similar restrictions apply to all the other SIM Toolkit handlers. Reasonably tested applications should not have any access problem in normal situations, but applications also need to ensure that all the conditions are met in all cases, including when the handling of an event does not work well.

All of these issues are good practice issues, and they can only be addressed by the developers themselves, and good quality programs can be expected to address all the safety issues.

2.3 Security

The last set of issues regards the security of applications. Security issues are related to traditional security concerns such as integrity and confidentiality of data and communication, access control to sensitive resources, authentication and authorization, etc. The kind of issues that we can encounter on a Java Card application include for instance:

- Perform appropriate user authentication.

- Control access to an application's resources from other applications.

- Protect confidential data during sensitive operations.

- Use secure communication channels when required.

- Manage properly the application lifecycle.

The Java Card specification only defines very basic mechanisms for security. In particular, this specification is directly aimed at application developers, and it does not include any specification for card and application management mechanisms. Similarly, it does not define the roles of the specific actors on the card. However, these items are all covered by the GlobalPlatform card specification [2]. This specification precisely defines how a card and its applications should be managed, assigns precise roles to all actors, and then defines a set of tools for developers, made available to them through an API. This API allows application developers to perform security-related actions, such as using a global PIN to authenticate the end user, managing the lifecycle of their application, or establishing a secure channel between the card and a terminal using system-managed cryptographic keys.

The first issue mentioned above is user authentication. The Java Card API provides a class to manage PIN codes, and the GlobalPlatform API also provides a mechanism. There are here very few security constraints. Most importantly, PIN values should be large enough in order to guarantee a good level of security. In addition, it is better not to perform any authentication-related operation during a transaction, because the results (in particular if the transaction gets aborted) may not be those expected by the user.

The second issue is quite simple in principle. Java Card defines a firewall between applications, so the default behavior is that the applications cannot share any data. However, in some cases, applications need to share some objects. In such a case, there are many additional rules to apply. If the applications trust each other and they both trust the underlying platform, the minimal requirements are to follow the same good usage rules when writing the code, and to identify each other when initiating a sharing session. When the chain of trust is broken, it remain possible to use sharing, but there are many more precautions to take in order to avoid security breaches as well as `SecurityException`'s thrown by the system.

The third issue mentioned is about protecting data during sensitive operations. There are many things that may happen during a sensitive operation such as encryption and decryption, and these events need to be taken into consideration. Most problems will trigger an exception, which means that an appropriate way to protect sensitive operations is to catch all exceptions and to clear the sensitive data if anything has gone wrong before to continue the execution of the program.

The two last issues, related to secure communication channels and application lifecycle management, are not directly covered by the Java Card specification. However, there are APIs for these features in the GlobalPlatform card specification, which allow developers to handle these issues in a very simple and direct manner. In this case, the main requirements are to be systematic, in the sense that every time an application processes a command, it should check that the lifecycle is compatible with this processing, and that the proper transport security has been applied.

All of these issues need to be addressed by developers. However, it is quite obvious that not all applications need to follow the same level of security. While a payment application needs to be as secure as possible, a simple SIM Toolkit gaming application may not need to be upheld to the same standards. Since the sensitivity of an application also depends on the other applications that will reside on the same cards, the issuer also has a role to play in the definition of the security

requirements, since the developer needs to know the level of security required on a specific environment.

3   Addressing the Issues

In the previous section, we have listed some issues related to Java Card programming. Most of these issues are related to small details in the specification. If they are not addressed, an application will most likely run without a glitch in 99.99% of the cases. However, such a figure is not satisfactory for smart cards. If you consider that you have deployed the application on 1,000,000 cards and that cardholders uses the application on average once a day, this figure roughly means that 100 cards will fail every day. When thinking about smart card applications, developers and issuers need to never forget that smart card applications are mass-deployed.

Here, the situation is very different for SIM cards and banking or identity cards. Because SIM cards are network authentication tokens, they are very often connected, and their card issuer (the telecom operator) is able to send commands to the card and manage it over-the-air transparently from the user. In such a case, the direct costs cost of fixing a problem in an application mostly includes communication and management costs (indirect costs only occur as image degradation for the users who actually experienced a problem, since the update is performed transparently for the user). On the opposite, in the context of banking or identity cards, fixing a dysfunction in an application is extremely difficult. Since these cards are used episodically and they are almost never connected to a network, it is at least required to get the card connected to the issuer's network, for instance in a bank's branch office. Even if this is possible, the direct costs (sending mail or calling the user to ask him to come to the branch office), as well as the indirect costs (bad image, consumer trust), are staggering. In many cases, it may even be simpler and less expensive to replace the cards altogether.

Because of the probable high cost of an application bug, it is extremely important for card issuers to take all measures to reduce the risk of seeing such a bug. The rest of this section will focus on a specific measure that can be easily taken by issuers: defining card issuer policies, and then enforcing these policies.

3.1   Defining Issuer Policies

Traditional smart card issuers are used to define strong issuer policies. The number of consortia and standard bodies around smart cards shows that there has always been a strong need for interoperability of cards. Similarly, security has always been a clear focus for smart card issuers since, in most cases, a smart card is a security token. For instance, a typical smart card call for tenders mentions a list of standards that need to be supported, as well as a set of minimum security requirements (including for instance a description of the cryptographic attacks that the implementation of the cryptographic algorithms should b protected against).

When the target smart card is a Java Card, the same issuer policies should be stated. The main difference is here that two actors are involved: the Java Card platform provider, and the application developer. This means that the various responsibilities need to be split between these two actors, and the issuer needs to work on two different sets of policies:

- Card provider policies. The aim of this first set of policies is to define the "minimum service" that a card platform should provide. In particular, it should contain a set of interoperability guidelines, which define precisely the minimum requirements for the card, as well as a set of security guidelines, listing the precise security requirements for the smart card platform.
- Application developer policies. The aim of this second set of policies is to define as precisely as possible the duties of the application developer, in order to guarantee that all applications developed can run on the target card platforms, and that they satisfy basic security requirements.

An important thing here is that the policies may differ, depending on the target application. For instance, when developing games in a SIM Toolkit environment, developers may be extremely restricted in their access to resources (*e.g.* no cryptography, no access to GSM data, …), but they would also have to follow very few security restrictions. On the opposite, when developing a payment

application for a banking card, developers may have access to all resources available on the card, but they may also have to obey a large set of security-related restrictions.

These policies are complementary to specifications. Interoperability and security policies for card providers will often be refinements of the specifications, providing either additional restrictions or requirements (*e.g.* feature X must (not) be implemented), or providing additional requirements about the implementation of a feature (*e.g.* the implementation of algorithm X should protect against the known attacks Y and Z). On the other hand, portability and security guidelines for applications will usually need to be presented in a more detailed way, in order to map the requirements into good usage rules directly related to the API (hence ensuring that developers properly understand and apply the rules).

The process for writing these policies and turning them into practical guides for card and application providers can be split in three steps:

- Gather information.
  The first step is here to get all the information required for writing the guides. For interoperability and portability requirements, this information comes from the environment (capabilities of target cards, functions supported by target handsets). For security, the information comes in most cases from a risk analysis, in which the threats are described, as well as the required countermeasures.

- Split the responsibilities.
  The second step is to split the responsibilities between the providers. For interoperability, the idea is here to make sure that every issue is covered by an interoperability requirement (for card providers) and the corresponding portability requirement (for developers). For security, the idea is to make sure that every countermeasure is covered properly by either the card provider or the application developer.

- Consolidate the guidelines.
  The final step is to consolidate the guidelines into documents that are readily usable by card providers and application developers. The interoperability documents aimed at card platform providers are rather simple to define, since they mostly consist in a refinement of available specifications. The related security guidelines for card providers need more work, but the smart card industry already has many tools to deal with security, and is for instance well accustomed to practices such as Common Criteria security evaluations. On the opposite, the guidelines aimed at developers require a significant consolidation work, in order to ensure that the guidelines are well-understood and correctly applied.

In the rest of the present section, we will present two examples. The first one shows how the responsibilities can be split between card providers and application developers. The second one presents in detail how a basic security requirements can be merged with the security constraints, and then refined in order to get simple rules that can be applied by developers.

The first example regards the use of cryptographic keys, and how the constraints are defined for both the card platform provider and the application developer: The key types can be organized into four distinct categories:

- Forbidden key types.
  Such keys are in most cases forbidden for security reasons, usually because the keys are too weak. The situation in that case is simple. Card providers *must not* support this key type, and application developers *must not* use keys of this type.

- Recommended key types.
  Such keys correspond to the key types that are available on all cards, and whose use is recommended by the card issuer. The situation is again very simple. Card providers *must* support these key types, and application developers *should* use them.

- Recommended card-specific key types.
  Such keys correspond to keys that are recommended by the issuer, but only available on some kinds of cards, typically cards that support public key cryptography. Here, the situation is more complex. Card providers *must* support such key types on cards where they can be supported, and they *should not* support them on cards on which they cannot be implemented. Similarly, card

developers *should* use these algorithms on card types that support them, and they *must not* use them for cards that do not support them. In particular, a developer writing an application targeting all cards must not use these algorithms.

- Optional key types.
  Optional key types usually correspond to "exotic" key types that are not used by the card issuer, but are required for the implementation of some applications, usually targeted at specific markets (typically a vertical or a national market). Card providers *may* support these key types, in particular if they target a specific market on which these key types are required. Similarly, application developers *may* use these algorithms if they target the same specific markets, but they need to be aware that using these algorithms greatly restricts the range of target cards.

This example can then be instantiated with a few algorithms, as shown below:

Key type	On DES card	On PK card
TYPE_DES, LENGTH_DES	Forbidden	Forbidden
TYPE_DES, LENGTH_DES3_2KEY	Recommended	Recommended
TYPE_RSA, LENGTH_1024	Not available	Recommended
TYPE_DSA, LENGTH_1024	Optional	Optional

We here see the instantiation of the rules for a few typical algorithms. Here, 64-bit DES keys are forbidden on all cards because they are too weak; 128-bit DES keys are recommended on all cards. 1024-bit RSA keys are recommended on PK-enabled cards, but they are of course not available on cards that cannot perform RSA computations. Finally, DSA keys are optional on all cards, since they are only used in specific vertical applications.

These rules may sound trivial. However, there is a wide range of cryptographic key types and algorithms defined in the Java Card specification; if this range is not restricted by the issuer, application developers may decide to use algorithms that some card providers choose not to provide. Since very few cards implement all the algorithms defined in Java Card, this issue typically is quite important.

The second example is a very classical one, and it shows how a basic requirement can be refined stepwise into detailed rules that are applicable by developers. A very common requirement from issuers, especially for third-party applications, is that the applications should not use sharing to communicate with other applications. The intent is here to simplify the acceptance process, by ensuring that the applications are totally isolated from each other.

However, there is a problem with this requirement, for both wireless applications and GlobalPlatform applications. In both cases, the system APIs rely on the use of shareable interfaces:

- Applications need to share their own objects. In the SIM Toolkit API, the situation is most extreme, since all SAT-related invocations are performed through a shareable interface. In the GlobalPlatform API, the personalization of an application is performed through a shareable interface.
- Applications need to use shared objects. In the SIM Toolkit API, the GSM file system is accessible through a shareable interface. In the GlobalPlatform API, the secure channel functions are accessed through a shareable interface.

The issue is here that the basic requirement conflicts with the context constraints; in that specific case, it conflicts with the definition of the API. In the rest of the example, we will focus on the SIM Toolkit API, since it is more widely impacted than the GlobalPlatform API. The first step consists in merging the basic requirement with the context constraint. As a result, we get two basic rules:

- Applications can only share objects with the GSM application.
- Applications can only use objects shared by the GSM application.

With these two rules, the ambiguity has been removed. However, this may not be enough to make developers comfortable, because these rules are not precise enough. For instance, the first rule can be refined as follows:

- Only applet classes may implement the `ToolkitInterface` interface.
- Only the null reference or instances of these classes may be returned by `getShareableInterfaceObject`.
- If an applet's `getShareableInterfaceObject` returns a non-null reference, it must only do so after checking that the AID of the requesting applet matches the GSM applet AID, and that the additional parameter value is 0.
- An applet should not define or use hidden communication channels with other applets, such as public static fields.

The guidelines defined above are very precise, which allows developers to follow them with a god level of confidence. Going into such details really is crucial for both portability and security guidelines, because developers need to be sure that they have applied the rules correctly and that they will be able to prove it. In fact, the precision in the rules is necessary for the enforcement of the rules, making it more difficult to dispute the application of rules.

3.2 Enforcing Issuer Policies

Assuming that detailed issuer policies have been defined, the next step is to enforce these policies. There are here two sets of issues:

- Enforcing card provider guidelines.
  Here, the requirement is to ensure that the card provider has followed the interoperability and security guidelines. A major difficulty in this assessment lies in the fact that the card provider's technology often is confidential, which means that only black-box testing is available to their customers.
- Enforcing application developer guidelines.
  Here, the requirement is to ensure that the applications developed for the card follow the portability and security guidelines defined by the card issuer. The main difficulty may here come from the sheer number of applications, which may be quite large for some issuers. In that case, traditional testing techniques such as source code inspection and black-box testing may be widely inefficient.

For card providers, there are two promising directions: standardized test suites and security evaluations. Test suites are used widely in all flavors of Java, including Java Card. In order to be able to use Java logos, Java implementers need to pass a Technology Compatibility Kit (TCK) on their implementation. Such a TCK exists for Java Card, and all Java Card implementations are required to pass it successfully. However, these tests are only available to licensees, which means that card issuers are not able to ensure that their needs are tested for, unless they get a Java Card license. Other test suites have been defined. For instance, for SIM Toolkit applications, the SIM Alliance has sponsored the definition of a test suite that verifies the interoperability of SIM Toolkit Java Card platforms.

For security, formal security evaluations, such as Common Criteria evaluations can also be very helpful. If the features required by a card issuer are included in a card's security target, then the card issuer may simply check the assurance requirements in that same security target in order to assess the kind of testing that has been performed by the independent testing lab that performed the security evaluation. Since the cards that undergo such evaluations usually are the most secure ones (for which specific security certificates are required), there is a good change for an issuer to find most of the features it requests defined in the card's security target. In fact, in most cases, these features will be listed in the Protection Profiles defined for Java Card-enabled cards or for GlobalPlatform-enabled cards.

For application developers, on the other hand, there are usually no standardized test suites, and formal security evaluations are seldom performed. Some test suites may exist, but they usually focus

on the purely functional behavior of applets, and they do not take into consideration the portability, safety and security aspects that are specific to Java Card applications.

Nevertheless, some tools exist that can check whether or not a specific application satisfies a specific set of rules. These tools allow the card issuers to automate the validation of applications, and makes their validation process more scaleable. These tools are discussed in the next section.

4 Automated Validation

4.1 *Rationale*

An automated validation tool takes as input a Java Card application and a set of rules, and it provides as output an analysis of the way in which the Java Card application follows the rules. The applications are usually provided in binary format, as CAP files[1], which means that the validation occurs on the exact code that will be downloaded in the card. The rules may consist of a standard set of rules, for instance the application portability rules defined by the SIM Alliance [3]. They may also include some rules customized for a specific card issuer, card program, or even for a specific application (for instance, a highly sensitive application may require specific rules that do not apply to other applications).

The result of the validation consists of a diagnosis and a report. The diagnosis may be accepted, which means that the application satisfies all the rules, rejected, which means that the application violates at least one of the rules, or warning, which means that the tool could not reach a decision for at least one rule (it has not been able to prove that the rule has not been violated, but it has not been able either to prove that the rule has been violated). In the case of rejected or warning diagnoses, the result is completed with a complete list of the violations and indeterminations, including as much information as available. The validation tool also provides a report, which summarized the most important features of the application, and lists its most notable requirements and properties. In some cases, the report may also include information such as size estimates for various Java Card platforms.

The information provided by such validation tools is crucial for card issuers who need to validate that applications developed for them follow their portability and security policies. These tools are based on static analysis techniques, and they aim at proving that the programs follow the rules rather than testing the programs' features. In fact, these tools are based on the same technology as class file and CAP file verifiers, which are universally used on all Java platforms. The main difference between the standard verifiers included in all Java platforms and these validation tools is in the sophistication of the algorithms. Since the validation tools need to infer much more information, their algorithms also need to be more complex.

Nevertheless, validation tools keep the main advantages that have been essential in the success of the bytecode verification technology: the actual executable code is analyzed, and the user does not need to provide the source code. This allows validation tools to be used as filtering technology in the Java Card application provisioning process.

Static analysis is a very mature technology, which has been around for over 20 years. Java bytecode verification is one of its most well-known applications, and the technology has encountered numerous problems, linked to its expressiveness (what are the possible outcomes of the analysis?), performance (can I reasonably use the technology?), and flexibility (isn't the analysis enforcing rules too tightly, making it impossible to use?).

In the specific context of Java Card, these problems can be handled in an efficient way, mostly because of the specific features of the Java Card framework. In particular, the facts that Java Card represents only a small subset of Java and that it is based on a very restrictive programming model make it much simpler to use static analysis technology. In particular, it allows us to use very detailed analysis domains, which in turn allows us to define a wide range of properties to be checked. The rest

---

[1] A CAP file is the binary distribution file format for Java Card applications.

of this section describes the way in which Trusted Logic's validation tool deals with the most typical static analysis issues.

4.2   Expressiveness

The analysis algorithms developed by Trusted Logic for the Java Card validation tool are quite complete. In particular:

- Specific abstract domains have been defined to analyze the values of the arguments passed to a method.

- The analysis includes a specific abstract domain to analyze the content of the heap. This allows algorithms to check properties on objects, and also to gain in precision.

- The analysis is global, and it analyzes all possible execution paths of the Java Card application. This allows algorithms to put conditions on sequences of operations.

- The analysis is able to track the constraints imposed on some variables on each execution path, making it possible to put conditions on these constraints.

All these analyses have been studied in the past, in particular through research prototypes, but the simplicity of the Java Card programming model makes it possible to use them altogether in the same tool.

With the data gathered by these algorithms about the program, it is possible to verify a wide range of properties. The simplest ones do not in fact require any specific analysis, as the information is available in the CAP files. For instance, it is possible to check the dependencies of a CAP file, or to verify the classes and methods that are used.

Some other properties require a more complete analysis. For instance, it is possible to check some conditions on the actual arguments passed to a method, or to check the result of the execution of a method (making sure that the method never throws an exception, or that the return value verifies a given condition). Similarly, it is possible to check that a system variable (for instance the event variable used in SIM Toolkit event notification) is or not constrained to take a given value. It is also possible to perform some checks on objects stored in the heap, for instance to check that the firewall rules are not broken. It is also possible to verify that some operations occur in a given sequence, for instance that every opened transaction is properly closed.

It is also possible to perform more complex checks, but these usually are combinations of the simpler checks discussed above. For instance, the verification of the rule about sharing with the GSM applet is in fact a combination of simpler rules; the most complex ones being the verification that proper checks are performed before to return a non-null value.

4.3   Performance

Performance has always been an issue with static analysis programs, because most algorithms used in these programs are not linear, and their behavior on a particular program is highly unpredictable. Although Java Card defines a very simple programming model, this performance issue remains present, and needs to be addressed. There are at least two ways to address this issue:

- Systematically limit the precision of the algorithm.
  By making the algorithms less precise, the analysis can be faster, but it will infer less information, and there may a larger number of "undetermined" rules, for which the validation tool is not able to conclude. In that case, the tool can be used in a non-interactive environment, but the developers may need to adopt specific programming techniques to avoid the indeterminations.

- Use the tool interactively.
  In that case, the default behavior of the validation tool is to attempt the analysis with the best precision possible. This should allow the tool to get good results on most programs, but its execution time will be too long on some complex programs. For those programs, it is then necessary to manually adjust some settings and to analyze them with a limited precision, in order to get a result from the tool. In that case, the results obtained from the tool are optimal, but the use of the tool requires some interaction with the user.

The default choice for the validation tool is to favor an interactive use. In fact, there are two such uses. First, in a development or acceptance environment, the tool is expected to be used by a developer or a tester in an interactive way (verifying an application, and then checking the results). Then, in a production environment, for instance in a provisioning chain, the use should be less interactive, but the user (for instance the developer who registers an application) is given the opportunity to specify the level of precision required for the analysis of its program.

The idea behind this choice is to favor the expressiveness of the framework, and to minimize the constraints imposed on the developers' programming style.

4.4 Flexibility

Another common issue with static analysis tools is that they do not work on all programs. Developers need to adapt to the tools, usually by changing some programming habits. If the tools are not flexible enough, these changes can become really annoying for developers, finally leading them not to use the tools.

With the Java Card validation tool, the core restrictions are very limited. The programs should not include recursive calls, because allowing recursion makes the algorithms far less efficient; however, using recursion is strongly discouraged in Java Card, where the execution stack is always small. In addition, the program should not use non-standard libraries that include native methods, because the tool only processes Java bytecode and would have no way to know the effect of these native methods.

However, there are other factors that limit the programming style for developers:

- The most important is a consequence of the goal of the tool. Since the tool is intended to check the quality of a program, many of the rules it verifies impose a specific programming style from the programmer. For instance, if a rule mandates that all exceptions need to be caught, then the programmer must conform to this rule, but the requirement does not come from the tool itself.

- Another factor comes from the nature of the analysis. Since the analysis is global, it cannot take any input value into consideration, and the content of the objects on the application's heap needs to be approximated. For some rules, these values can therefore not be used directly.

We can consider a small example, in which the installation parameters are used to specify the type of a key to be allocated. A first implementations is shown below:

```
public static void install(byte[] buf, short ofs, short len)
{
  …
  myKey = KeyBuilder.buildKey(buf[ofs++], buf[ofs++]) ;
  …
}
```

In this example, the input parameters (*i.e.* the elements of the buffer passed as input to the installation method) are used directly as parameters to the key factory method. The consequence is that the analysis will systematically signal a potential error, since some values of the input parameters could lead to the allocation of forbidden key types. The following style is more acceptable:

```
public static void install(byte[] buf, short ofs, short len)
{
  …
  switch(buf[ofs++]) {
    case USE_3DES:
      myKey = KeyBuilder.buildKey(TYPE_DES, LENGTH_DES3_2KEY) ;
      break ;
    case USE_RSA_1024:
      myKey = KeyBuilder.buildKey(TYPE_RSA, LENGTH_RSA_1024) ;
      break ;
    default:
      ISOException.throwIt(ISO7816.SW_UNKNOWN_ERROR) ;
  }
  …
}
```

In this second example, the input parameter is used as a key to a switch statement, and specific key types are allocated depending on its value. With this method, the validation tool can determine which key types may be used; if both key types used in this program are allowed by the rules, no violation will be reported for the excerpt shown here.

In addition, one can note that, like the constraints coming from the rules, the constraints coming from the algorithms have a tendency to favor well-written programs, as shown in the examples above. A positive side effect, especially in the context of Java Card applications, could be that some dangerous practices, common in Java Card for obscure optimization reasons, may be discouraged.

The Java Card validation tool may also have a positive impact for developers, because it can provide them some information that is difficult to get in other ways:

- The validation can locate common mistakes that are linked to Java Card's specific execution model. Since these mistakes only lead to problems when the programs run on actual cards, they can be very hard to identify, and automatically flagging them can be very beneficial for junior developers.

- The validation tool can identify all kinds of abnormal behaviors, such as uncaught exceptions, memory allocation issues, … Senior developers can take advantage of such a tool by getting a quick diagnosis on an application at all phases of its development. Once again, the productivity gain can be significant, since most errors flagged by the validation tool are difficult to identify using standard testing.

To conclude about flexibility, the use of a Java Card validation tool in the context of the present paper, *i.e.* to enforce the policies provided by a card issuer, should not lead to major constraints for the developer. On the contrary, by forcing the card issuer to be very explicit about the policies, and by providing an efficient way to provide to the issuer a proof that its policies have been followed, automated validation tools can be a great help for developers.

5   Putting it Into Practice

Putting the elements described in the present paper is quite easy. Once the issuer has defined its portability and security policies, the development cycle can be defined as shown below.
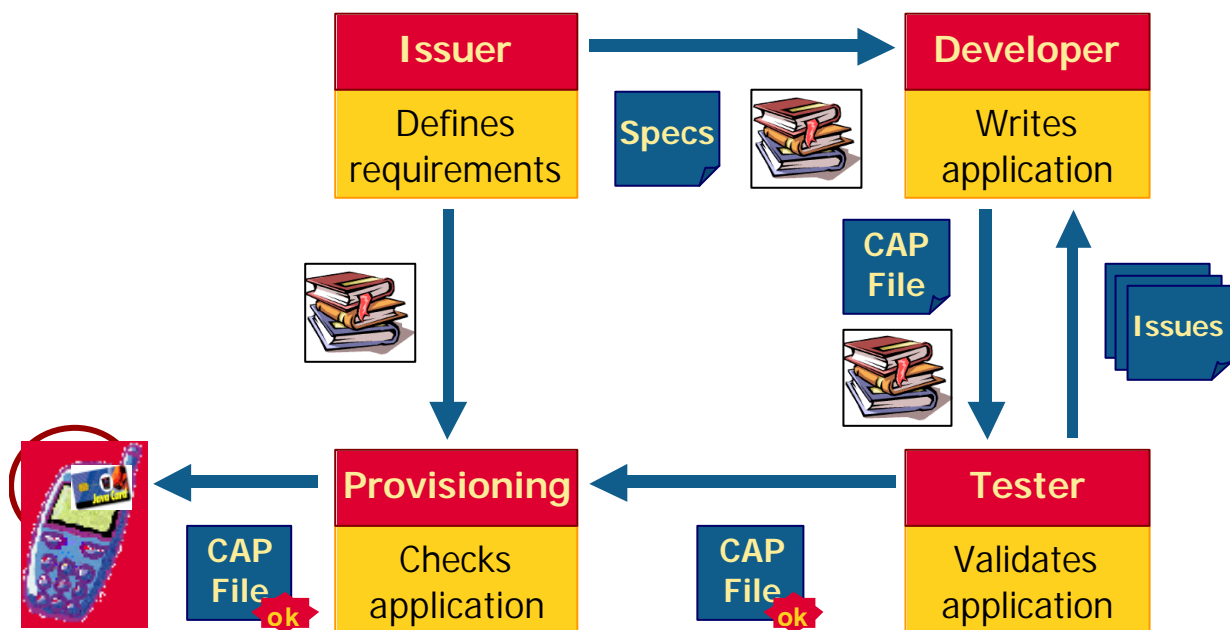
The issuer defines the requirements, and gives the specifications and the policies (in the form of a developer's handbook) to the developer. The developer then writes the application, compiles it, and gives the resulting binary file (CAP file), together with the developer's handbook, to a tester. The

tester, which may or not be in the same company as the developer, then validates the application and returns a list of issues to the developer. After a few exchanges, the tester accepts the application and forwards it to the provisioning agent.

At this point, there are two possible options. In the first case, the provisioning agent trusted the tester, and its role may then simply consist in integrating the new application into its provisioning chain, allowing it to be downloaded into a SIM card on the field. In the second case, the provisioning agent also gets the developer's handbook from the issuer, and it first runs the automated validation tool on the CAP file with the rules provided by the issuer. Since the CAP file has previously been accepted by the tester, this validation is expected to succeed.

In all cases, the issuer is likely to suggest the use of a Java Card validation tool. In that case, the developer's handbook should be provided as a paper document (for reference purpose), and also as an electronic document, to be used with the validation tool. Since the Java Card application tool is able to use several sets of rules in a single run, it then becomes possible for developers to use the tool to verify the requirements of several card issuers in a single run.

6 Conclusion

The deployment of mobile code on pervasive devices such as smart cards and mobile phones is just starting. However, we have seen that some issues emerge rapidly. Among these issues, the portability of applications and the security of applications are the most critical.

In the present paper, we have presented a general methodology for the development of platform provider guidelines and developer handbooks, and the automated validation of the latter. The approach has been experimented on Java Card, which presents several advantages:

- It is a mature technology.
  Java Card has been around since 1996, and its specification has not undergone any major change since 1998. The current specifications are therefore very stable, and many issues linked to the specification itself have already been identified and addressed.

- It is widely deployed today.
  Over 300 million Java Card cards have been deployed as of today, most of them in the wireless market. Smart card providers and telecom operators have worked together (in particular through the SIM Alliance) in order to enhance the level of interoperability of the cards. As a result, most card interoperability issues have been addressed, and the focus of developers is shifting toward application portability.

- It offers a very restricted programming model.
  Since Java Card is aimed at smart cards, its programming model has been adapted to the meager resources available on such platforms, and this model is extremely restrictive. This greatly simplifies the work of defining and enforcing development rules to Java Card programs.

This approach has been very successful on Java Card. Important organization have issued application development guidelines, and most of the major issuers are defining their own guidelines for applications downloaded on their phones. In addition, a commercial validation product is available from Trusted Logic, and it has been used for instance all the members of the SIM Alliance.

The most important mobile Java platform today is MIDP 2.0, the new release of the profile aimed at wireless devices. This platform is far less stable than Java Card, as the first compliant products are just arriving on the market, and telecom operators can still expect significant interoperability issues for some amount of time.

Nevertheless, it is extremely important to apply the methodology described here on MIDP 2.0 deployments. The lack of interoperability simply makes the need for application portability greater, and the novelty of MIDP 2.0's security model make application security and safety checks extremely important. The MIDP 2.0 programming model is far less restrictive than the Java Card model, which means that fewer rules can be automatically verified. Automated validation still remains interesting for telecom operators, because it allows them to perform an automated screening of the applications based on their own metrics.

A version of the Trusted Logic validation tool for MIDP 2.0 programs is available today as a prototype. However, the most important task is now to define the portability and security guidelines, and to then develop developer handbooks. This step should take quite some time, because the first MIDP 2.0 devices are just being released, but when satisfactory guidelines will be available, we can expect that automated validation techniques will gather a lot of interest from wireless operators and other users of the technology.

7   References

[1] Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley, 1999.

[2] GlobalPlatform Consortium. GlobalPlatform Card Specification v2.1.1. March 2003.
Available from http://www.globalplatform.org .

[3] SIM Alliance. Interoperability Stepping Stones, release 2001.
Available from http://www.simalliance.org .

[4] Sun Microsystems. The Java Card Specification (Runtime Environment, Virtual Machine, and API). June 2002.
Available from http://java.sun.com/products/javacard .