USENIX Association

Proceedings of the 2nd Workshop on Industrial Experiences with Systems Software

Boston, Massachusetts, USA
December 8, 2002

Building an "impossible" verifier on a Java Card*

Damien Deville
deville@lifl.fr, http://www.lifl.fr/~deville
Gilles Grimaud
grimaud@lifl.fr, http://www.lifl.fr/~grimaud
Université des Sciences et Technologies de Lille,
Laboratoire Lifl, Bat M3,
cité scientifique, 59655 Villeneuve d'Ascq - France

Abstract

Java is a popular development platform for mobile code systems. It ensures application portability and mobility for a variety of systems, while providing strong security features. The intermediate code (byte code) allows the virtual machine to verify statically (during the loading phase) that the program is well-behaved. This is done by a software security module called the *byte code verifier*. Smart Cards that provide a Java Virtual Machine, called Java Card, are not supplied with such a verifier because of its complexity. Alternatives are being studied to provide the same functionality outside the card. In the present paper, we propose to integrate the whole verifier inside the smart card. This ensures that the smart card becomes entirely autonomous, which allows full realization of smart cards potential as pervasive computing devices. Our verifier uses a specialized encoding and a software cache with a variety of cache polices to adapt to the hardware constraints of smart card. Our experimental results confirm the feasibility of such a security system being implemented in a smart card.

Keywords: Smart card, Java Card, static verification, type inference, secure embedded system.

*This work was supported by a grant from Gemplus Research Labs, the CPER Nord-Pas-de-Calais (France) TACT LOMC C21, under the European IST Project MATISSE number IST-1999-11435, and our prototype was performed within the Gemplus Research Labs and considred by our partner as a proof of concept for further industrial development.

1 Introduction

This paper presents the motivations and techniques used to develop a stand-alone verifier inside a smart card. The verification process is one of the most important parts of the security in a Java Virtual Machine. In a JVM, this process is performed while a new application is loaded into the virtual machine. Due to smart card constraints, a verifier as it was defined by Sun is said to be impossible to embed in a Java Card Virtual Machine (JCVM).

"Bytecode verification as it is done for Web Applet is a complex and expensive process, requiring large amount of working memory, and therefore believed to be impossible to implement on a smart card" [12].

"Clearly, bytecode verification is not realizable on a small system in its current form. It is commonly assumed that downloading bytecode to a JAVACARD *requires that verification is performed off-card"* [18].

Nevertheless, we will present here some solutions to obtain a stand-alone verifier in a JCVM.

We first outline the standard verification process, in order to introduce the reader to standard verification principles. Next we present degraded solutions of this process that have been designed to cope with supposed smart card constraints. To overcome these difficulties, we show that these constraints are not actually a problem by detailing the properties of smart card hardware properties used for our stand-alone verifier. We then present our approach that consists of hadrware-

specific adaptations to efficiently match the standard algorithm with these detailed hardware characteristics. We do not change the standard verification algorithm, but instead propose some innovative techniques that allow an improved execution on top of the smart card limited hardware. In the last part we give some experimental results extracted from our prototype. Our prototype subsequently became a proof of concept of an embedded smart card verifier for our industrial partner.

2 Verification principles

First we present the standard verification process performed by the Java Virtual Machine in a conventional implementation. This verification process consists of performing an abstract interpretation of the byte code that composes each method of an application. The aim is to statically check that the control flow and data flow do not generate errors (underflow or overflow of the stack, variable used with invalid type, ...). This algorithm is called *type inference* and its concepts were first introduced in [10]. The paper [11] gives an overview of the whole Java verification process. Java Card Virtual Machine (JCVM [3]) is a stack-based machine that uses particular registers named *local variables*. The Java specification [14] imposes some constraints on the byte code generated by compilers so that it can be verified. Because of these constraints, the type for each variable used by the program can be inferred. We use *stack map* to mean a structure giving the type of each variable, for each particular point of our program. Java programs are not linear; one can jump to a particular instruction from various sources. Thus we can have different hypotheses for the type of the variables; we need to *merge* all these hypothesis into one, and we need to find the corresponding type that matches all, *i.e.* a *compatible* one. This operation is called *unification* and is notated $\cap$. For example, for classes, it corresponds to the inheritance relation. We give, in the next paragraph, the standard version of the verification algorithm [14]. We also use this later on for the description of our implementation techniques for smart cards.

The standard algorithm uses the hypothesis that each instruction has an associated stack map, a TOS (Top of stack), and one *changed*/*unhanged* bit.

Initialization:

- mark first instruction as *changed*;
- fill its typing information by using the signature of the method (variables with no type are given the type $\top$ that means unusable variable), initialize the top of stack (TOS) at 0;
- for all other instructions:
  - mark them as *unchanged*,
  - initialise their TOS at -1 (this means that this instruction has not been checked earlier).

Main Loop:

- while there remain instructions marked as *changed*;
- choose an instruction I marked as *changed*;
- mark I as *unchanged*;
- simulate the execution of I over corresponding typing information (if there is an error, method is rejected);
- for each successor S of I:
  - we use O to mean the stack map for I and R to mean the stack map for S,
  - if S's TOS is equal to -1, then copy the stack map from O into R (we also initialize the TOS of I with the one of S),
  - otherwise, perform unification between each cells of the stack map for R and the ones for O. Result is denoted by $R \cap O$ (if the TOS does not correspond, verification stops with an error),
  - if $R \cap O \neq R$, mark S as *changed*; $R \cap O$ is now the new stack map for S.

3 Java Cards and verifiers

The standard verification algorithm is usually presented as being impossible to embed in a smart card, due to hardware limitations. Instead, the verification process is simplified and adapted in order to fit on small devices. In the next parts we present existing solutions that guarantee a good security level in a standard Java Card.

In a regular Java architecture, the produced application is verified while being loaded in the virtual machine. The byte-code verifier is define by [14] JVM specification as the basic component of safety and security in a standard JVM. Currently, SUN Java Card is built upon a split VM scheme: one part is called the "off-card VM"; the other one is called the "on-card VM". As the on-card VM cannot load a CLASS file format because of its complexity, a converter is used to produce CAP files (Converted APplet) that is more convenient to smart card constraints (no runtime linking, easy to link on load, ready to use class descriptor [20], ready to run byte code). While converting the class file, verification is performed in order to ensure that the produced CAP file is coherent with the CAP file structure definition. Smart cards are considered secure devices, and because of the lack of on-card verification it is currently impossible to download a new application to a card after it has been issued to an end-user. Some solutions have been proposed in order to achieve a good security level of the Java Card without using a verifier with regard to smart card constraints.

- **Digital signature**. The application designer sends the CAP file to a trusted third party that digitally signs it; hence, the card can now easily check if the application has been modified. This model guarantees a high level of security; the only requirement is to dispose of cryptographic routines on board for checking the validity of the signature. As smart cards already have a cryptographic co-processor, the verification cost is low. However, there is one major problem: it is a centralised deployment scheme. This centralization decreases the flexibility of such an approach: all cards need to be declared to the trusted third party. Moreover it is not compatible with a smart card that operates off line, nor to a massive smart card distribution (wolrdwide, there are roughly 1000 times more smart cards than PCs with web browsers).

- **Defensive VM**. As SUN has defined the conditions and rules required for executing each byte code of the Java Card language, a defensive virtual machine [4] can be used. Before executing a byte code, the virtual machine can perform all the required tests. Thus, a very high level of security is achieved, but the efficiency of the virtual machine is decreased, and the amount of working memory needed for running the applet is also increased significantly.

- **Proof-Carrying Code**. PCC techniques were introduced by G. Necula and P. Lee [17]. E. and K. Rose have adapted it to Java [18]. G. Necula and P. Lee have proposed an architecture to use PCC verification on a Java platform [5, 16]. A PCC verifier was developped by G. Grimaud [7] for a specialized language with regard to smart card constraints. An off-card part produces a proof (or certificate) that is joined to the application. The on-card verifier has just to perform a linear verification phase in order to check if the code is malicious. This verification is simple enough to be performed by the card. This is the solution that is recommended by SUN when implementing a KVM [19]. The size of the downloaded application is increased because of its proof (from 10% to 30%). The major problem is that the application deployment is now more complex because of the need for the proof generator. Valid applications can also be rejected only because their proof is not provided. Such a Java Card verifier as been fully embedded in a smart card by Gemplus Research Lab. It was developed using formal method and is described in [2].

- **Code transformation**. X. Leroy [21] has proposed another solution that is described in [12, 13]. It consists of some off-card code transformations (also named "normalization"), in such a way that each variable manipulated by the virtual machine has one and only one type during all the different execution paths of the program. The embedded part just needs to ensure that the code respects this property. Valid applications can also be rejected only because they were not

transformed using Trusted Logic[21] normalizer. The major problem is that performing such code transformations would increase the number of variables and also decrease the re-usability of them. Thus the card would need more memory resources for executing the programs, and it might refuse fully optimized code.

- **An infeasible solution (?): stand-alone verification**. Each of the solutions described earlier has some disadvantages that a stand-alone verifier would not have. The major problem is the need of some external pre processing of applets that can make a non stand-alone verifier refuse some valid applets. But the main stand-alone verification problem is its time and memory complexity.

Table 1 summarizes advantages and drawbacks of each solution.

In the next part we give some information about the hardware of smart cards, in order to explain the difficulties of having a stand-alone verifier on board.

4 Smart card constraints

Smart card has some hardware limitations due to ISO [9] conbstraints that are mainly defined to enforce smart card tamper resistance. Now we are going to focus on each of the hardware features of the smart card, starting with the microprocessor, then the memory and ending with the I/O.

- **Microprocessors**. A wide class of microprocessors are used from old 8-bit CISC micro chip (4.44 Mhz) to powerful 32-bit RISC (100 to 200 Mhz). The type of CPU used for smart card is highly influenced by the ISO[9] constraints linked to the card. For example, as the card is a portable device that is stored in a wallet, it must meet standards related to torsion and bending capacity. Table 2 gives an overview of some processors commonly used in smart cards. Historically, smart card manufacturers used 8-bit processors because operating systems and applications code are more compact. But smart cards now need to be more efficient and new embedded applications require more and more computing power. So card designers now choose 32-bit RISC processors (or 8/16 bits CISC). For our experimental prototype presented in the last part of this article, we are using an AVR from Atmel [1]. It is an 8/16 bit processor with 32 (8 bits) registers. Some of them can be grouped to perform computation on 16 bits values. Its is a typical platform in the smart card community for operating system design [15].

  Computing performance of a smart card is not a significant problem for a stand-along verifier. What is really the limiting factor for smart card programs is the very small amount of memory, and some annoying hardware-specific problems.

- **Memories**. Different types of memory exist on the card. The first one is the RAM (Random Access Memory); there is also some ROM (Read Only Memory); and finally EEPROM (Electric Erasable Programmable Read Only Memory) or FLASHRAM that are writable persistent memories. Because smart card silicon is suposed to be limited to 20 mm^2, the physical space needed for storing 1 bit is an important factor. Each kind of memories used is of different size concerning this point; the smallest is the ROM. Table 3 gives an overview of the amount of memory present on board, and also present the "memory cell" which is the size for 1 byte of memory on the micro module.

  Persistent memory has a major drawback, linked to its electronic properties. Its writing delay is up to 10000 times slower than RAM one. Furthermore, writing in persistent memory may damage the memory cells (the stress problem occurs when using the *erase* operation: making a bit going from 0 to 1). These constraints have led others developing light-weight byte-code verifiers to consider persistent memory only as a memory to store data and not has a working memory.

  EEPROM provides 4 primitive operations:

  - *read*: reading a value,

Solution	*Pro*	*Cons*
Crypto	dedicated hardware	centralized solution
Defensive VM	easy to implement	important run time penality
PCC	no run time penality	code overload, non standard deployment scheme
Code transformation		
Stand alone	standard deployment scheme	impossible ?

Table 1: Summarize of each solution

Model	Architecture	Data BUS size	Registers	Frequency
68H05	CISC	8 bits	2 (8 bits)	4.77 Mhz
AVR	RISC/CISC	8 bits	32 (8/16 bits)	4.77 Mhz
ARM7TI	RISC	32 bits	16 (32 bits)	4.77 to 28.16 Mhz

Table 2: Characteristics of common smart card microprocessors.

– *erase*: changing some bits from 0 to 1,

– *write*: changing some bits from 1 to 0,

– *update*: an *erase* followed by a *write*.

Erase is a stressing operation, it can provoke a lapse of the memory cell. It is also 38% slower than a *write*. Table 4 illustrates this characteristic. This characteristic is also true for FLASHRAM.

Operation	time for writing a 64 bytes page
erase	$\simeq 2.77$ ms
write	$\simeq 2$ ms

Table 4: Differences between the two writing operations in a typical EEPROM usage.

Using this memory well is the key technological challenge for smart card developers.

- **I/O**. We have one half duplex serial line (from 9600 to 15700 baud in conventional smart cards). This rate means that 1 KB of data is transferred in less than 1 second. Compared to the number of CPU cycles available in 1 second, the I/O capacity reduces the viability of technical solutions that involve an additional data transmission.

5 A fully embedded Java Card byte code verifier?

The byte code verifier is said to be impossible to be implemented inside a smart card; the impossibility is due to its high algorithmic complexity and also to its large memory consumption. In this part, we focus on all these constraints, and give solutions for allowing its usage on board, overcoming the hardware limits described previously.

5.1 Time and space complexity

The elementary operation we are going to use is the interpretation for a byte code working on a simple variable. Complexity is limited by $D * S * J * V$. D is the depth of the type lattice. S is the number of instructions of the verified program. J is the maximum number of jump (branching instructions), and V is the number of variable used at most. In the worst case, all instructions are jumps ($S = J$). By analyzing all the different byte codes of Java Card we can find the one that manipulates the highest number of variables. Let c represent this number, thus $V = c * S$ in the worst case. We can also state that we need one instruction for creating a new type in our lattice, thus $D = S$. Finally time complexity is $O(S^4)$. Hence, type inference is a polynomial form of the number of analyzed instructions. Let us now evaluate the memory that is needed for performing a typing inference. We need the type information for each label (destination of a branch) of

Type	memory point	capacity	write time	page size
ROM	reference	32-128 KB	read only	1 byte
FlashRAM	x 2-3	16-64 KB	2.5ms	64 bytes
EEPROM	x 4	4-64 KB	4ms	2-64 bytes
RAM	x 20	128-4096 B	$\leq 0.2\mu s$	1 byte

Table 3: Card memory characteristics

our program. The size of each of them is the number of local variables plus the maximal stack size. We also need one more frame for the current one. Thus the memory we need is $T*(S+L)*(J+1)$ where J is the number of branching instructions and/or exception handlers, T the size needed to encode one type, S the stack size, and L the number of local variables. Practically, we can easily require more than 3 KB of RAM. Thus we need to store the proof in persistent memory. Doing this we need to be aware of the stressing problem and also of the slowness of writing of persistent memory.

5.2 Our solutions

We propose to use persistent memory (*i.e.* EEPROM or FLASHRAM) for storing stack maps of the current method. Nevertheless, the amount of memory needed is smaller than the one for a PCC verifier [2] that needs to store the proof for every methods in persistent memory. Our strategy consists of using RAM to hold the entire stack map whenever possible, and when not possible to use it as a cache for persistent memory. The first challenge is the bad property of persistent memory related to cells stressing. We propose the usage of a non stressing type encoding. The second challenge is to find the best cache policy according to our problem.

5.3 A non stressing type encoding

In order to solve the problem of persistent memory stress, we propose to find a good encoding of the type lattice used during type inference. The goal is to use a non-stressing write when storing a type in persistent memory. The paper [8] presents examples of type lattice encoding that are used in compilation or in databases. It proposes techniques to perform a bijection between lattice operations and Boolean operations. Described encodings allow finding the Least Upper Bound (LUB) of two elements using a composition of logical functions. Type unification consists of finding the LUB of two types. Typing information needs, sometimes, to be store in persistent memory due to its size, and persistent memory has a stressing problem. We observe that when unifying two types, the result type is higher in the type lattice, so we would like to be able to move upward in the lattice while only changing bits from 1 to 0. Such an encoding causes no stress on persistent memory, and unification is reduced to a logical AND that has the property of only clearing bits (1 to 0 transitions). As non stressing writes takes less time than a stressing one, unification goes faster.

We could find a more compact encoding (*i.e.* using less memory) by using a more complex Boolean function, but this would take less into account persistent memory properties.

In our prototype, dynamic downloaded classes do not take benefits of the encoding, as it would require computing the whole type encoding. However our experiments show that complex unification (*i.e.* one between two classes, producing a class that is not `java.lang.Object`) happens very rarely. In fact, smart card software engineering implies using specific rules that reduce the number of different class used at the same time. Accordingly, we do not try to optimize these cases, we just accept the minor performance degradation. This choice has no effect on security. Experimentally, we found that less than 1% of unification are performed between two classes. All these techniques reduce $\cap$ computation time and ensure a non stressing usage of persistent memory but they do not deal with the latency of write operation.

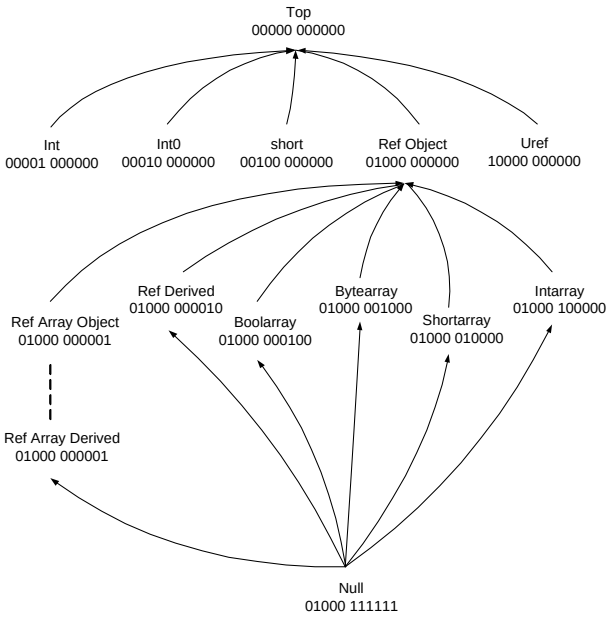

Figure 1: Our non stressing type encoding

5.4 Efficient fixed points using software cache policy

As described earlier, type inference is an iterative process that stops when a fixed point is reached. By using our type encoding we decrease the persistent memory stress, but we can also obtain better results by trying to maximize the RAM usage. The aim is to maximize the size of the data we can store and work on in a RAM. For example, some specific methods extracted from our application test pool need more than 4 KB of memory for the verification process. As RAM memory is faster than persistent memory, but is less present on a smart card, we use a *software cache* for working on typing information. Such a technique can speed up the verification of huge (13KB) applets as it highly decreases the number of write in persistent memory.

In order to perform type inference, we need to find the target of branching instructions. Having the branching instructions and also the branched ones, we can build (at low cost) the control graph flow of the verified program. Then, when verifying a particular code section we know which are the ones we can jump to by using the control graph flow. If we look at the example given in Figure 2, when verifying the last code sec-

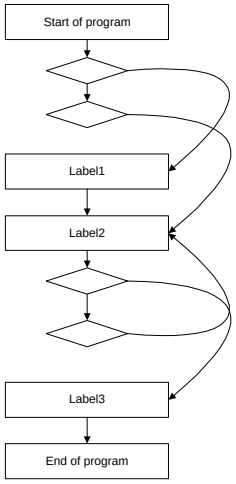

Figure 2: An example of a cache policy using control graph flow

tion (the one beginning with Label 3), we can see that none of the other code section are accessible. The control graph flow is used for piloting the eviction of data in the cache. Moreover, we can sometimes extract from the graph the property that a label is not reachable from any further point of the program. In this case, eviction performs no write in persistent memory as the type information will not be used anymore. Pratically, this happens frequently in conventional Java Card code. The eviction significantly increase the speed of stand-alone byte code verification.

We note that the techniques we propose are compatible with future types of persistent memory like FeRAM (Ferro-Electric Random Access Memory) or MRAM (Magnetic Random Access Memory), and will take benefits of using these memories. The techniques we have presented here have been delivered to the industry and are protected under the patent [6].

6 Experiments

We have implemented a prototype that includes each of the techniques we have mentioned. This prototype shows that the stand-alone byte code verifier is feasible in a smart card. Moreover, we have used common benchmarks for evaluating its performance against a system using a PCC verifier.

6.1 Our prototype

We have implemented the concepts and techniques described in the earlier part of this article on an AVR 3232SC smart card microprocessor from Atmel. This chip includes 32 KB of persistent memory, 32 KB of ROM, and a maximum of 768 bytes of RAM. The amount of available RAM highly depends on the state of the verifier and also of the virtual machine. Practically, in most of cases, using less than 512 bytes of RAM does not alter stand-alone byte code verification's efficiency. The type inference algorithm has an important advantage: we can compute the amount of memory we will need to verify a particular method of an application. We propose different algorithms and heuristics: we have implemented different cache policies that suit a particular amount of RAM (an "all in RAM policy", a LRU policy, a more evolved LRU, and finally the one described earlier that uses control graph flow of the verified program). We also use some heuristics for selecting the current label to verify. These heuristics have different properties in term of performance. We choose a heuristic dynamically in order to fit the amount of working memory. We give in Table 5 the size of the important part of our verifier.

Module name	code size in bytes
Byte code transition rules	13552
Unification	2242
Cache policies	3700

Table 5: Memory footprint in ROM

We have a total of 30 KB of code for our stand-alone type verifier. The remaining part which is not described in table 5 consists of routines that deal with the cap file format, I/O, and RAM and EEPROM allocators. We did not tune this implementation for our proof of concept, and we would expect a production version to be both smaller and faster. For example, a standard JCVM represents 20KB of code; its API represents 60 to 100KB; and a huge user application is about 13KB (these measures are extracted from products from our industrial partner).

6.2 Practical metrics

We give in Table 6 the duration of the loading phase in milliseconds, and also of verifications, for three applets which size are from 3 KB to 4.5 KB. The PCC verifier is the one described in [2].

The loading phase for the PCC is higher than the one for the stand-alone (between 20% to 30%). It is due to the fact that it needs to load and write the entire proof in persistent memory. The overhead for the proof in a PCC verifier is said to be between 20 to 30 % in related work. The time taken to write the proof in persistent memory before verification by the PCC could be avoid by performing the verification on the fly. Thus the only difference between a PCC and a stand-alone verifier would be the extra loading

	Applet	PCC	Stand-alone	Stand-alone Vs PCC
Load	grl-com.utils.wallet	3335	2744	0.82
	grl-com.games.tictactoe	9594	8933	0.93
	com.gemplus.pacap.utils	9414	7331	0.78
Verif	grl-com.utils.wallet	411	318	0.77
	grl-com.games.tictactoe	1232	1102	0.89
	com.gemplus.pacap.utils	1312	1463	1.12
Total	grl-com.utils.wallet	3746	3062	0.82
	grl-com.games.tictactoe	10826	10035	0.92
	com.gemplus.pacap.utils	10726	8794	0.82

Table 6: Time for loading and verifying (in ms)

time of the proof. Of course, the techniques described earlier could be used on the PCC verifier to reduce the time needed to use the proof (on the fly verification, non stressing encoding to speed up operations on types, ...). We need also to remind that the off-card generation of the proof for a PCC verifier as a cost in terms of deployment tools and also in terms of time. Thus PCC as a model is more expensive than a stand-alone verifier.

If we look at the global time taken for verification, we can point out that stand-alone verification is not slower than PCC one which has a linear complexity. In some particular case, stand-alone verification can be faster than PCC one. Smart card application are often simple in terms of generated byte code. Thus stand-alone verification is nearly linear. With this table we show that the extra time needed for stand-alone verification (with our technical approach) is often less than those needed for downloading and storing the PCC proof.

Some experimental results of a verifier using code transformation were presented in [13] but with only few details. Nevertheless, we are in the same order of magnitude for verification execution time (1 sec for 1 kb of code). We did not have access to smart card using others verifications strategies so we could not compare them with our prototype. But cryptographic signature supposes an extra time to check the basic cap file signature after download (for example, a smart card usually performs a RSA in something like 10ms per 64 bytes). Concerning the solution using digital signature, it is the worst solution in term of infrastructure as it assumes a trusted third party to sign applications.

Conclusion

We have shown that careful attention to the smart card hardware allows us to integrate a stand-alone verifier on a smart card. Stand-alone verification is no longer a mismatch to the smart card constraints. The usage of hardware-specific techniques allows the stand-alone verifier to be as efficient as a PCC one.

References

[1] Atmel Corporation. Atmel AVR. `http://www.atmel.com`.

[2] L. Casset, L. Burdy, and A. Requet. Formal development of an embedded verifier for java card byte code. In *DSN-2002. The International Conference on Dependable Systems and Networks*, 2002.

[3] Z. Chen. *Java Card Technology for Smart Cards*. Addison Wesley, 2000.

[4] R. M. Cohen. Guide to the djvm model version 0.5 alpha ** draft **, 1997.

[5] C. Colby, G. C. Necula, and P. Lee. A Proof-Carrying Code Architecture for Java. In *Computer Aided Verification*, 2000.

[6] D. Deville, G. Grimaud, and A. Requet. Efficient representation of code verifier structures, 2001. International pending patent.

[7] G. Grimaud, J. L. Lanet, and J. J. Vandewalle. Façade: A typed intermediate language dedicated to smart card. In *Software*

Engineering - ESEC/FSE'99, pages 476–493, 1999.

[8] H. Ait-Kaci and R. Boyer and P. Lincoln and R. Nasr. Efficient Implementation of Lattice Operations. *ACM Transactions on Programming Languages and Systems, TOPLAS*, 11(1):115–146, 1989.

[9] International Standard Organisation: ISO. Integrated circuit(s) cards with contacts, parts 1 to 9, 1987-1998.

[10] Gary A. Kildall. A unified approach to global program optimization. In *ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[11] X. Leroy. Java bytecode verification : an overview. In *Computer Aided Verification*, 2001.

[12] X. Leroy. On-card bytecode verification for java card. In *Esmart*, 2001.

[13] X. Leroy. Bytecode verification for Java smart card. *Software Practice & Experience*, 32:319–340, 2002.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.

[15] Matthias Bruestle. SOSSE - Simple Operating System for Smartcard Education, 2002. `http://www.franken.de/users/mbsks/sosse/html/`.

[16] G. C. Necula. A scalable architecture for proof-carrying-code. In *Fifth International Symposium of Functionnal and Logic Programming*, 2001.

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

[18] E. Rose and K. H. Rose. Lightweight bytecode verification. In *Workshop of the Formal Underpinnings of the Java Paradigm, OOPSLA'98*, 1998.

[19] Sun Microsystem. Connected Limited Device Configuration and K Virtual Machine. `http://java.sun.com/products/cldc/`.

[20] Sun Microsystem. The javacard™ 2.1 specification. `http://java.sun.com/products/javacard/`.

[21] Trusted Logic. Formal methods, smart card, security. `http://www.trustedlogic.com/`.