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

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

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

ABSTRACT

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

1. INTRODUCTION

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

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

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

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

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

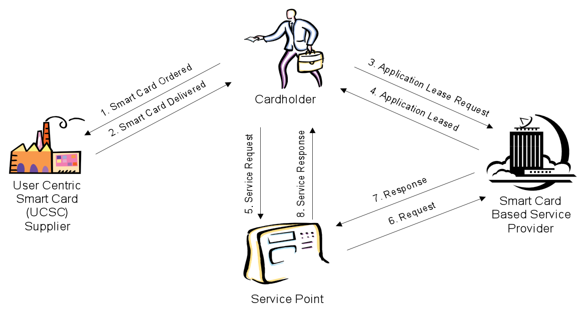
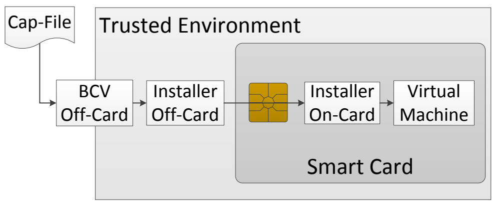


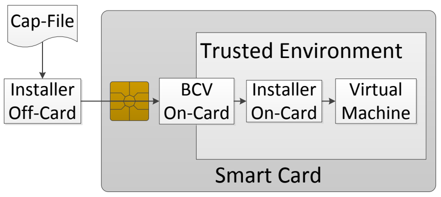
Figure 1: Relationships in the User Centric Smart Card Ownership Model between card supplier, card user, application issuer and the service point(from [1])

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

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



(a) off-card BCV



(b) on-card BCV

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

2. RELATED WORK

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

2.1 Java Card Security Model

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

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

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

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

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

2.2 Bytecode Verification

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

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

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

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

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

2.3 On-card Verifier

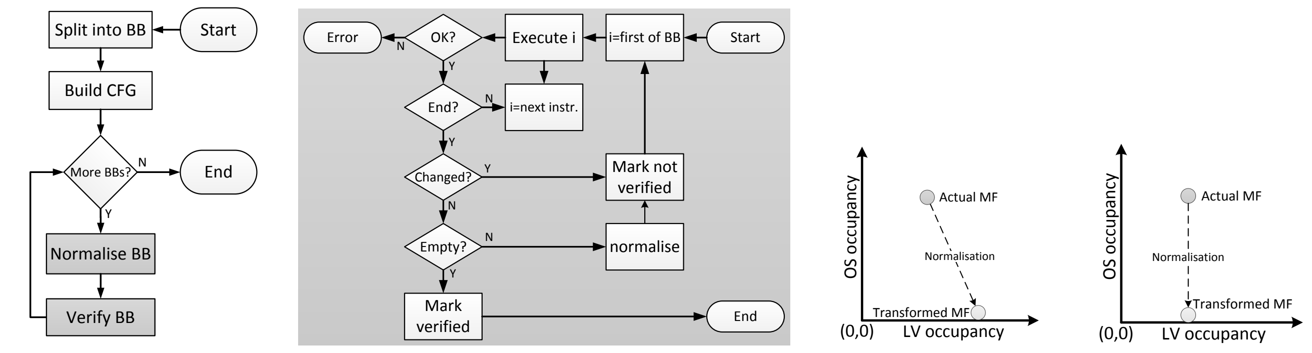
Soon after the initial presentation of Java Card the scientific community began with the research for a possible solution of an on-card BCV. The main problem with the existing algorithm is the memory consumption. For a smart card with only hundreds of byte of RAM or tenth of kilobytes of EEPROM/Flash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

2.3.1 *Reducing the dictionary*

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

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

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



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

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

2.3.2 Adding off-card components

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

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

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

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

3. CONCEPT OF THE NEW VERIFICATION METHOD

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

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

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

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

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

3.1 Normalization

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

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

3.2 Verification

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

4. IMPLEMENTATION

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

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

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

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

5. RESULTS

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

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

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

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

6. CONCLUSION AND FUTURE WORK

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

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

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

Acknowledgments

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

7. REFERENCES

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

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

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

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

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

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

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

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

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

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