Incremental Dynamic Update For Java-based Smart Cards

Agnes C. Noubissi, Julien Iguchi-Cartigny, Jean-Louis Lanet
University of Limoges
Team SSD
France
{ *agnes.noubissi, julien.cartigny, jean-louis.lanet* }*@xlim.fr*

***Abstract*—One of the most appealing feature for multi-application smart cards is their ability to dynamically download or delete applications once the card has been issued. Applications can be updated by deleting old versions and loading the new ones. Nevertheless, for system components, the update is sligthly more complex because the systems never stop. Indeed, for smart cards based on Java called JavaCard, the virtual machine has a life cycle similar to the card because persistent objects are preserved after the communication sessions with the reader have expired. We present in this paper, our research in dynamic system components updating of JavaCard. Our technique requires a lot of off-card and on-card mechanisms. Our approach uses control flow graph to determine change between versions, a domain specific language to represent the change for minimization of the download overhead throughout the communication link with the card.**

***Keywords*-Smart Card; HotSwUp; Java Card; Dynamic update; e-passport;**

I. INTRODUCTION

Computer applications are getting more and more appealing with an increasing number of features available to users. In order to make them better, significant changes have been recordered to upgrade them as well as in regard to easily install and get rid of some features. But, there are some applications which should run continuously without interruption and should provide features to upgrade their functionnalities. This include in particular aeronautic applications like traffic control systems and financial transaction applications. Besides that, in the context of small embedded devices, we have the Java Card system in which Java Card virtual machine (JCVM) [4, 5] runs continuously and then never stops. In such devices, update must be applied at runtime. We propose in this paper, an approach to dynamically update non-stopping Java system components especially to fix bugs in some cryptographic algorithms. More precisely, this paper considers the update process in the form of on-line upgrades of existing classes associated with their runtime contexts which can be object instances in the heap, frames of methods to update in the stack, subclasses and other dependencies class in the system.

II. RELATED WORKS

A. Non Embedded Java Platforms

Several dynamic Java software update techniques have been presented in the literature [2, 7, 8, 9, 10]. Here, we consider updating techniques which are related to our work.

Al. Orso and An. Rao [10] have presented a technique based on class renaming and code rewriting, which then performs the upgrade by dynamically swapping the classes at runtime through a tool called DUSC. However, the class renaming is a time consumming process and there is a space overhead in the case of changes in the class hierarchy (renaming, inserting or deleting classes). Moreover, the approach can not deal with native methods and addition of non private methods or fields because the approach requires that the new classes have the same public interface.

Suriya Su., Michael H. and Kathryn S. have presented JVOLVE [7], a Java virtual machine which supports online softwares update. They use an Update Preparation Tool (UPT), the Jikes RVM dynamic class loader, a JIT compiler and class transformer functions to perform dynamic update. JVOLVE uses the UPT to determine list of changed, deleted and not modified classes in the application to update. And then, it loads changed classes, and compiles them in order to update existing classes. After that, the modified garbage collector converts existing object instances to refer to the new classes. Our approach follows this idea, but instead of using an UPT to get the list of direct or indirect changed classes, we use it to get the real changes in the content of these classes and we use a Domain Specific Language (DSL) specificaly designed to save efficiently these changes.

JDrums [8] and DVM (Dynamic Virtual Machine) [9] have modified their virtual machine to support dynamic update like JVOLVE. However, JDrums performed a lazy update from DVM and JVOLVE. Indeed, JDrums use to check objects pointer dereference to determine if a new objects class is available and this technique brings about an additional overhead.

B. Embedded Java Platforms like Java Card

Java Card is a fast growing technology based on embedded Java operating system. Java Card is shipped with a dedicated Java virtual machine called Java Card virtual

machine. The JCVM controls the access to smart card resources and thus serves as the smart cards operating system. Another good characteristic of Java Card is to provide the ability to run multiple applications and to securely load new applications to the card after it has been issued. Indeed, with Java Card, applications can be installed or uninstalled during post-issuance upon the user's request through the card manager.
Nowadays in Java Card, it is then possible to update applications by deleting and adding new applications in the card. But, for system components, the unique solution is to mask the new system in the ROM and then create a new card.

C. Short Conclusion

We have seen that, nowadays it is not possible to dynamically update system components on the card. But, we can use one of the techniques described for non embedded systems, however we can note that these existing techniques have been tested and benchmarked in platforms which are different to the smart card platform in the regard of resource constraints of the smart cards like memory, cpu, bandwidth of the communication link with the reader, etc. Our goal is then to provide a new technique that reduces bandwith consumption, optimizes runtime overhead and minimizes memory consumption. Minimizing overhead requires to: (1) optimize the size of input files that will be downloaded over the slow smart card communication link, (2) reduce memory and CPU time required during update, (3) reduce temporary resources needed to dynamic update. Moreover, security has always be a great concern for smart cards and it is even more relevant for multi-applicative and post-issuance cards. Indeed, while these new functionalities add important values, they bring new security issues that must be addressed. If a malicious update file is downloaded on the card, this could lead for example to a leak of cryptographic keys and PINS numbers. Then, our approach must provide a safe dynamic update and robust tool.

III. Case Study And General Issue

A. Case Study

It has been demonstrated that electronic passport could be hacked due to the weaknesses of the chip MIFARE [3]. This is a contactless smart chip technology based on ISO 14443. This chip is most widely installed on contactless smart card or proximity card like e-passport. Henryk Pltz and Karsten Nohl [11, 12] described a partial reverse-engineering method for the MIFARE classic chip. Similarly, G. de Koning Gans, J.-H. Hoepman, and F. D. Garcia [13] presented a technique that was able to manipulate the contents of a MIFARE classic card, and then determine some cryptographics keys, PIN code, modify some tests security, and so on. In the case of e-passport for example, if we need to fix a bug on a heacked or cracked cryptographic algorithm, the actual solution is to return all e-passports the issuer. This present solution is far too unwieldy in the case of a very large of e-passports has been recorded.
Currently, with post-issuance, it is possible to provide good updating by deleting old and adding new applications through operating system but it is not possible for systems components. So, it is therefore necessary to provide a mechanism for dynamically patching these system components, potentially in front of a terminal or a Near Field Communication (NFC) reader.

B. General Issues

The goal is to allow class upgrades, method redefinition as well as extending or modifying the class while preserving runtime type safety of the system and its coherence. Many questions need to be adressed: How should we represent changes? How are we going to find and replace references on the heap? How are we going to find and modify frames on the stack? How should we keep the journal for rollback update operation in the case of failure? How can we guarantee the integrity of the system during the upgrade process? In the case of dynamic update, we cannot wait until the system is idle before starting the upgrade. So, it is necessary to know the points of the systems called *safe update point* in which we can guarantee the coherence of the system. In general, the global process of upgrade can be divided into two steps.

1) Upgrade the byte code, object instances and associated frames: The aim of this step is to find and update byte code of the class in persistent memory. After that, we find and replace, in the heap of virtual machine, old objects instances of the updated classes to provide new ones with appropriate attributes and values. And finally, we find and update object instances references of the updated classes in stack frames. The process must ensure that new attributes of new objects rely effectively on the new classes attributes. However, those modifications of objects instances and frames must occur in a **safe update point** of the system to maintain its consistency and integrety. Indeed, detect safe update point is very important in the case of online update because of the likelihood of a system crash that will leave the system in a very unstable state. For example, if an update of object instance occurs while we continue accesses to the deleted attributes so non longer present in new object instance in the heap.

2) Upgrade of dependencies classes: The update process continues by propagating upgrade information to subclasses. It can be related to the lists of deleted methods, deleted interfaces, list of methods signatures or attributes changed. To maintain the integrity and coherence of the system, all these modifications must be done on subclasses and other dependents class of the update class if necessary.

IV. Our Approach

This figure illustrated our approach which is explained in the following paragraphs.

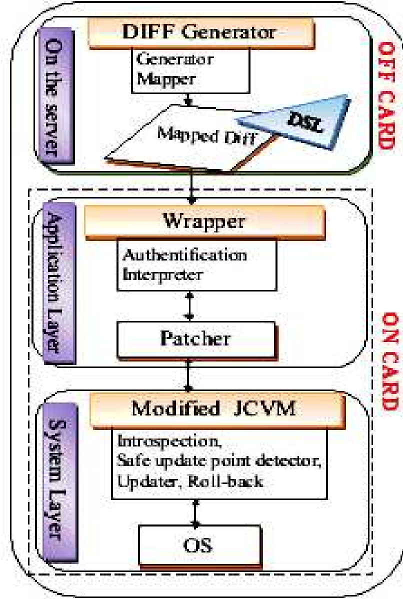

Figure 1. General work-flow of our approach of dynamic update

A. On Server

Off the card, the goal is to obtain some information about changes between versions (original and new) of the class of the system component module to update. First of all, we have written a tool that can provide information about global changes in the field of file. So, the tool can take as inputs two system components module (old and new) *(.jar)* and determine the list of added, deleted, modified or unchanged classes. For the modified classes, we provide another tool call **DIFF generator** which can take two modified classes and generate the real change between the two versions. To identify these changes in a proper manner, we have used control flow graph to represent the behavior of these classes and compare them. After that, we represent these changes in a *DSL* designed for the purpose. The final output file called DIFF is then passed to another tool called *mapper*. It takes the DIFF and the binary file *(.bin)* of the class to update, it searches all necessary relative address in *.bin* which are used to map symbolic information of identifiers, methods, attributes, fields present in the DIFF. After processing the **mapped Diff**, it is transferred to the smart card for patching. The goal of the DIFF is to reduce the overhead of the transfer over the slown smart card communication link and other side for SIM card for example, this technique can reduce down paid.

B. Application Layer

On the card, the **wrapper** authenticates sender, gets the *mapped Diff*, interprets it and then transfers update instructions to the patcher. To preserve coherence of the system, the patcher, before applying the update must ensure that the system have reached a safe update point. In our approach, *safe udate point* is restricted and defined by: **no frames relative to an update methods are in the stack**. Once the system reaches a *safe update point*, the update flag is positioned and the update is applied.

C. System Layer

In the system layer, we modified the virtual machine to offer also the following features: (1) search functions which can introspect data structures of the VM like references tables, threads table, class table, static object table, and also the heap and stack frame; (2) objects transformers functions; (3) and frame update functions. So, we developed *search method* which can find all objects instances of the class to update, all threads and frames relatives to objects references. For that, we have modified the source code of the garbage collector and allocated some specific space memory for update information. *Object searcher function* performs its work by distinguishing all active objects in the heap from any other objects reachable from those objects owned by the class to update and marks it with an update flag added for the purpose. We also developped *objects transformer functions* which can – from instructions provided by the **patcher** – modify object instances to refer to an instance of the new class and then save the new references in an appropriate data structure. For any object, say O, to be updated, **Updater** creates new object O with the size equal to the size of object instances of class to be updated, rewrites unmodified fields O.f with value of O.f and initializes other fields of O with provided values. Afterwards, it saves the new references which will be used to update references in the references table of VM and to update references in stack frames. After receiving final instructions, the **updater** can then update old references in the stack thread.

V. IMPLEMENTATIONS

In the off card part of the update system, the DIFF generator is written in Java and it is based on control flow graph. In the on card, the considered virtual machine is **SimpleRTJ** and the application layer called UAC (UpdateAppCard) which contains Wrapper and updater is developed in Java and C. *SimpleRTJ* is a simple Real-Time-Java virtual machine for small embedded and consumer device. It requires 18-24 KB of code memory to run and can virtually run on most 8, 16 or 32 bits embedded systems. Then, in this virtual machine, we add additional packages relative to **searcher**, **updater** and native functions for communications with the **patcher** since JNI is not available.

VI. MICRO BENCHMARKS

A. Experimental setup : Board ATMEL AT91/EB40

The AT91/EB40A [14] evaluation board has the following properties: (1) Flash memory device with 64 KB which

contains a boot software program, an Angel Debug Monitor, functional Test Software; (2) SRAM memory with 512 KB in which binary file can be downloaded through SRAM downloader; (3) A processor ARM7TDMI core at 32 Mhz; (4) Interfaces like JTAG inputs that allow code to be debugged on the board, JP5A/B and JP7A/B straps which establish the connection of an ammeter to measure the board power consumption respectively on VDDIO and VDDCORE.

B. Benchmark Cases

1) Payload cost: To represent the cost of sending the mapped DIFF compare to the cost of sending the entire new version of the class in the smart card through the NFC communication.

2) Times efficiency: The goal is to find the appropriate CPU time consumption. The factors, we use to determine dynamic update time can be stated as follows: (1) time to performs search of all object instances of a class to update in the heap and all frames in the thread stack; (2) time to find *safe update point*; (3) time to perform update of those object instances and frames determined by their numbers and type of update; (4) time to update source code in persistent memory depending of time of writing on FLASH of board ATMEL AT91.

3) Power consumption: In the case of smart card like SIM in which power consumption is very critical, we want to ensure that the approach can be applied also for Java Smart card virtual machine components for SIMs telephone.

4) Memory cost: Represents mainly the extra-memory used to create new objects to adapt and to obtain the object instances of the update class.

C. Results

Currently, we have modified the virtual machine and implemented search and update functions and the tests are underway. We started with the search test and considering that only data members of the class were modified. Microbenchmark presented here, has three simple classes with change and not change fields, which create object instance of the class to update and multiple frames in the stack frame.

Table I
SIZE OF FILE TO DOWNLOAD (BYTES)

	Size of new class	Size of DIFF	Percentage
class 1	1030	542	47,37%
class 2	1023	457	55,32%
class 3	993	413	58,40%
Average			**53,67%**

Table II
SEARCHING OBJECT INSTANCES TIME (MS)

	# total object inst.	# update class inst.	time
	70	1	0.04
	37	2	0.02
	44	6	0.03
Average	50.3	2.6	0.03

VII. CONCLUSION AND FUTURE WORKS

We present not only a new approach in the area of dynamic update of Java applications but also an approach in the smart card that can update system components such as classes of smart card virtual machine components. This approach is based on a mapped DIFF written in a defined DSL. That DIFF tells what has really changed between two versions of classes and express it with the appropriated relative address of methods, static variables and information of the original class in-card, to update. And we presented how the mapped DIFF is used in on-card to perform an update. Our results are actually in progress and tests are underway.

REFERENCES

[1] Agnes C. Noubissi, Jean-Louis Lanet and Julien Iguchi-Cartigny, *Carte puce, vers une dure de vie infinie*, MajecStic, Avignon, November 2009.
[2] Jonathan T. Moore, Michael Hicks, and Scott Nettles, *Dynamic software Updating*, Programming Language Design and Implementation (PLDI), ACM, 2001.
[3] Semiconductors Austria GmbH Styria, *http://www.mifare.net/*
[4] Z. Chen, *Java Card Technology for Smart Cards*, Addison, Wesley, 2000.
[5] *The Java Card 3.0 specification*: http://java.sun.com/javacard/, March 2008.
[6] Milan Fort, *Smart card application development using Java Card Technology*, SeWeS 2006.
[7] Kathryn S. McKinley, Michael Hicks, and Suriya Subramanian, *Dynamic Software Updates : A VM-Centric Approach*, PLDI, June 2009.
[8] Jesper Andersson and Tobias Ritzau, *Dynamic deployment of Java applications*, Java for Embedded Systems Workshop, London, May 2000.
[9] Earl Barr, J. Fritz Barnes, Jeff Gragg, Raju Pandey and Scott Malabarba, *Runtime support for type-safe dynamic java classes*, ECOOP, 2000.
[10] Alessandro Orso, Anup Rao, and Mary Jean Harrold, *A technique for dynamic updating of Java Software*, ICSM, 2002.
[11] Karsten Nohl, David Evans, Starbug and Henryk Pltz, *Reverse-Engineering a Cryptographic RFID Tag*, USENIX Security Symposium, San Jose, CA. July 2008.
[12] Karsten Nohl and Henryk Pltz, *MIFARE, Little Security, Despite Obscurity*. Presentation on the 24th Congress of the Chaos Computer Club in Berlin, December 2007.
[13] G. de Koning Gans, Jaap-Henk Hoepman, and Flavio D. Garcia, *A Pratical Attack on MIFARE Classic*, CARDIS, 2008.
[14] *ATMEL Corporation* : http://www.atmel.com/products/AT91/