Hot Updates for Java-Based Smart Cards

Agnes C. Noubissi
Department of Computer Sciences
University of Limoges
Ph.D. Student
agnes.noubissi@xlim.fr

Julien Iguchi-Cartigny
Department of Computer Sciences
University of Limoges
Assistant Professor
julien.cartigny@xlim.fr

Jean-Louis Lanet
Department of Computer Sciences
University of Limoges
Professor
jean-louis.lanet@xlim.fr

***Abstract*—Systems need to be updated in order to correct vulnerabilities, fix bugs but also to enhance functionalities. Traditional software update mechanisms usually stop the software which need to be updated, apply the update then restart the system. However, this approach is not appropriate in critical systems such as banking, telecommunications, or air-traffic control.**

A solution is the dynamic software updating (DSU) approach: patch the software on-the-fly (*i.e.* during execution). Several mechanisms have been proposed to offer this functionality to native and virtual machine environments for desktop and server computers. But using DSU in embedded systems (and especially in smart cards) poses several challenges regarding the resource and security constraints.

In this paper, we present an on-going work, EmbedDSU, a framework for DSU on Java-Based Smart Cards. The first step is an off-card mechanism which compute the changes between two versions of classes in order to limit the update process only on the part of the component which is affected by the modification (DIFF). During the second step, the DIFF is interpreted on-card and the DSU mechanism updates the methods in the class definition and object instances. It also refresh others virtual machine data structures.

I. Introduction

To cope with new, modified or extended functionalities, software must be updated, with the goals to remove bugs or improve functionalities. A simple update approach is to stop the old version of software or the whole system, apply the update and restart the system or the software. However, this approach is not adequate for applications which must run continuously even during the update process. Dynamic Software Update (or DSU) (also called on-line, on-the-fly or hot update) is the process of updating an application or a software component without stopping neither the system on which it is executed, nor the application itself. There is already existing literatures about DSU on hardware or software approaches.

In this paper, we present EmbedDSU, a software-based DSU technique for Java-based smart cards which relies on the Java virtual machine. Our approach is based on the modification of an embedded virtual machine to offer dynamic update functionalities. EmbedDSU is divided in two parts : off-card and on-card. In off-card, in order to apply the update only on the parts of the application or component that are really affected by the modification or by the update, we have implemented a DIFF generator which determine and express what are really changed between versions of classes. Then, the DIFF file result is transfered to the card and used to perform the update. Under some specific assumptions, the approach permits generic updates of Java applications, Java components system and has some properties like:

1) No human intervention required: The process is fully automatic: given two versions of a class, the DIFF generator is able to compute and express the changes between the two versions for class interfaces, fields, method bodies and constant pools or class meta-data.

2) Support any granularity of change: This mechanism also supports various update granularity: fields, method signature, one or a set of instruction blocks, and related classes.

3) Atomicity of update: To ensure coherence of the system, update is performed atomically.

4) Semantic preservation: After update, the updated classes obtained by using the DIFF have the same behavior as if it were transfered entirely and linked into the card.

To validate our approach, we implemented and tested our technique with an existing embedded Java virtual machine called SimpleRTJ (Simple Real Time Java). We have modified SimpleRTJ and adapted it, in order to offer HotSwUp mechanisms. In this paper, we show our motivation, explain our update process architecture, present our implementation, describe clearly our future works and finally show some results.

II. Motivation and Background

A smart card is a piece of plastic, the size of a credit card, in which a single chip microcontroller is embedded. Usually, microcontrollers for cards contain a microprocessor and different memories: Ram (for run-time data and OS stacks), Rom (in which the operating system and the romized applications are stored), and Eeprom (in which the persistent data are stored). Due to strong size constraints on the chip, the amount of memory is small. Most smart cards sold today have at most 5 Kbytes of Ram, 256 KB of Rom, and 256 KB of Eeprom. This chip usually also contains some sensors (like light sensors, heat sensors, voltage sensors, etc.), which are used to disable the card when it is physically attacked. The communication uses a half duplex serial line with a very low level protocol: APDU. The card acts as a server and expects a request from a reader.

A smart card can be viewed as a secure data container, since it securely stores data and it is used securely during

short transactions. Its safety relies on the underlying hardware and the correct design of the operating system. Java Card is a kind of smart card that implements the standard Java Card 3.0 [5] in one of the two editions Classic Edition or Connected Edition. Such a smart card embeds a virtual machine which interprets codes already romized with the operating system or downloaded after issuance. Due to security reasons, the ability to download code into the card is controlled by a protocol defined by Global Platform [16]. This protocol ensures that the owner of the code has the necessary credentials to perform such an action. Java Card is an open platform for smart cards, i.e. able of loading and executing new applications after issuance. Thus, different applications from different providers run in the same smart card. Thanks to type verification, the byte codes delivered by the Java compiler and the converter (in charge of delivering compact representation of class files) are safe, i.e. the application loaded is not hostile to other applications in the Java Card. Furthermore, the Java Card firewall checks permissions between applications in the card, enforcing isolation between applications. The API is romized but with possibilities to patch the operating system or the API in the Eeprom area before the issuance. After issuance no mechanism is provided to modify these components.

Java card is a subset of Java, with no multithreading, simple type (no float) only one-dimensional array, no garbage collection and so on. Applications are relatively simple to develop and are based on the Java Card applet model. The main method Process is invoked while receiving incoming bytes and according to the content of the header executes the request and sends back data. Such a programming model is simple.

In the case of E-passport, if weakness in a cryptographic algorithm is discovered, the actual solution is to return all of them to the issuer. This solution is too complex when there is an important number of E-passports in circulation. This is not an hypothetical threat. Indeed, recent exploits have been shown on the MIFARE chip [3] (a contact-less smart chip technology based on ISO 14443). For instance Pltz and Karsten [11, 12] described a partial reverse-engineering method for the MIFARE classic chip. Similarly, Gans and *al.* [13] presented a technique to manipulate the contents of a MIFARE classic card, and then compute some cryptographic keys, PIN code, or modify some tests security. Currently, it is possible to provide update of applications on Java-based smart cards by replacing old version by new one, an operation known as post-issuance. But for this, the application needs to be stopped before the update, which is impossible for Java system components. So, it is necessary to provide a mechanism to dynamically patch the system components when the chip is inside a terminal or near a Near Field Communication (NFC) reader. Our research focuses especially on a Java-based technology smart card called Java Card. A Java Card [5, 6] is shipped with a dedicated Java virtual machine called Java Card virtual machine (JCVM), which interprets Java card applications and manages access to smart card resources, thus serving as the smart card operating system. The JCVM has the ability to run multiple applications, sometime uploaded in the card after it has been issued. But the JCVM is a 'not-stop' VM: the life cycle is the card life because persistent objects are preserved even after expiration of the communication sessions with the reader. Thus the JCVM runs continuously and then update must be applied at runtime.

III. The proposed architecture

The architecture (Fig. 1) of our system is divided into two parts: on-card and off-card.

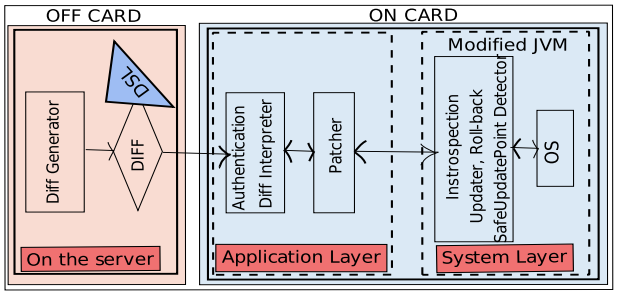

Fig. 1. Workflow of the dynamic update using EmbedDSU

A. Off-card

The first step is to compute the changes between two versions (the original and new one) of the system component module to update, with the goal to found the parts of the component that are modified. EmbedDSU uses a DIFF Generator to generate a DIFF file. This file contains the changes between two versions of classes, expressed using a Domain Specific Language (DSL). Then the DIFF file is parsed and converted into a compressed binary format to minimize the size and to reduce the interpretation cost by the card. Finally, the binary format file is transferred to the smart card for patching.

B. On-card

1) Application Layer: The binary DIFF file is uploaded into the card. After signature check with the wrapper, the binary DIFF is interpreted and the resulting instructions are transfered to the patcher in order to perform the update. The Patcher has the role to initialize update data structures. These data structures are read by the updater module to know what to update and how to update, by the safeUpdatePoint detector module to know when apply the update and by the roll-backer to know how to return in the previous version. All these issues pass through introspection and modification of data structures of the virtual machine, this requiring Virtual Machine modification.

2) System Layer: The last element of EmbedDSU is the modified virtual machine to support the followings features: (1) Introspection module which provides search functions to go through VM data structures like references tables, threads table, class table, static object table, the heap and stack frames for retrieving necessary informations to other modules; (2) updater module which can modify and updates object

instances, method bodies, class metadata, references, affected registry in the stack thread and affected VM data structures; (3) SafeUpdatePoint detector module which permits to detect safe point in which we can apply the update by preserving coherence of the system; (4) Rollback module which permits to obtain previous versions of codes, instances, stack thread and VM data structures. All these modules interact to provide HotSwUp functionalities. In the next section, we will see how they are implemented.

IV. Implementations

A. Off-card: Realization of Diff Generator

There are a lot of existing tools able to compute textual differences between files, however theses tools are limited to provide only textual differences and therefore do not consider changes in program behavior which can be caused indirectly by textual modifications. Then, we defined a differencing algorithm called DIFF generator that uses the control flow graph (CFG) in order to identify changes between two versions of program and then compute syntatic patches.

Indeed, Diff Generator can handle object-oriented features and can capture the behaviors of object-oriented components, and then identifies changes between versions. It takes as input an original class file C and modified version C' of class file to update and generate a Diff file formatted with the DSL.

It starts by generate the corresponding arborescence structure of the two classes (*.class*) using BCEL [15] and then performs its comparison first at the class interface and field levels. After that, at the method level, it matches method of the class file C with methods having the same name in C' to obtain a set of method pair. For each pair, it compares signature, interfaces, local variables, exception tables, and finally, bytecode instructions. And, for bytecode instruction, it constructs the corresponding control flow graphs (CFG) in order to model behaviors.

Each CFG represents the control flow behavior of the corresponding method, and it is consist of an nodes called basic blocks and egdes which represent possible flows from one basic block to another. Basic block, in this case, is a set of bytecode related to a statement (or instruction) which can be entered only at the beginning and leaved at the end. For comparing the two CFGs G and G' for m and m', the goal of our algorithm is to find, for each statement in m, a matching or corresponding statement in m', based on the method structure. Thus, it requires to find for a basic block of G, the corresponding one in G' and then compare their contents so compare relative statements to identify differences and similarities between them. DIFF generator can then provides the list of added, deleted or modified instructions with appropriate information about their location and attributes (for example their parameters which can be an *index* in the constant pool). In general, at the methods and fields level, the rule is: **methods or fields in C that do not appear in C' are deleted, whereas those in C' that do not appear in C are added**. Furthermore, Diff Generator also compares entries of constant pools of two classes to obtain information in order to modify class metadata of the old version in on-card for corresponding to the new version.
In conclusion, Diff generator can detect added, deleted or modified constant pool entries, class interfaces, fields, method signatures or interfaces, local variables, bytecodes or method bodies and exceptions. Thus, produces Diff file to transfer in on-card for the update.

B. On-card

In on-card, the main element is the modified virtual machine which contains HostSwUp modules. This new virtual machine can have three modes of execution.

- The mode before detecting a new version of an application, called the *standard mode*,
- The mode after receiving the DIFF file and before detecting a safe point or a quiescent state, called the *QuiescentStateSearch mode*,
- The mode after detecting the safe point, called the *update mode*.

1) Standard mode: During this mode, the virtual machine works normally without hotSwUp process overhead. However, we need to know how the notification about the new update can be provided to the process managing updates (update manager). In our technique, the knowledge of an available new update is obtained when the smart card is powered up by contact or contacless reader. So, if an update is available, the reader can send information to the update manager which can load the corresponding DIFF file from a centralized repository and stored it in persistent memory in order to be used in safe moment to perform update. To search that moment, the virtual machine switches directly to QuiescentStateSearch mode. We will discuss about centralized repository in the **section future works**.

2) QuiescentStateSearch mode: To obtain a safe point, we need to introspect the corresponding component state. In our approach, it can be identified and taken at any point in time during execution. It consists of:

- Classes. It is set of classes related to class to be updated (their method bodies and class metadata).
- Frames. It means all frames in the execution stack related to a method of the class to be updated. Knowing that, for each call of a method, a frame is created on the execution stack and it contains references to local variables, method parameters and a program counter which reflect the lines of method's bytecode.
- Objects. These are the active instances of the class to be updated present in the heap of the virtual machine. Active objects mean objects that are referred from method frames on execution stack or referred by a field of any other active instances.

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. And then, we defined the safe point state, like a state in which we do not have frames related to a modified method in the stack thread.

So, when switching to the QuiescentStateSearch, all calls to a class method to be modified are blocked, in order not to have others frames related to a method to be modified in the stack thread, nevertheless others frames can continue to be created on the stack. To determine a safe point, only frames of the component state are used. At starting point, we count all frames related to a modified method present in the stack thread. If the value is not equals to zero, then the update is delayed, the virtual machine can continue to execute others applications. However, the value is decremented each time, these methods finished their execution. When the value equals zero, then the safe update point is obtained and the virtual machine can switch to the update mode.

Moreover, it is necessary to provide the case where the safe point is never reached. And for that, in parallel of searching safe point, we initialized an int value which represent the time counter. This time counter permits to wait for a fixed waiting time. If the fixed time is reached, then we stop the search of safe point and switch to the standard mode.

3) Update mode: In this state, update must be done in an atomic way, so updates are either completely or not applied at all. The goal is to modify the component state of the old version of the class to obtain the new ones corresponding to that of the new version. Then, the update of classes (method bodies and class metadata), of frames, of instances and of others affected VM data structures.

Update process starts by updating method bodies and class metadata of the class to be updated and related classes. Indeed, with the Diff file of the class to be updated, we can know entries of constant pool to modify and for each modified method, parameters, local variables and bytecode to modify. Then, for updating the code of the class, updater module copies while modifying the old version of class metadata like constant pool, field table, method table, constant table and for each method that does not deleted, it copies while modifying method header, and bytecode instructions. So, the old version is transfered to a new space while modifying to obtain a the corresponding new version of the class.

After updating the class, update process continue with the update of all other constant pools of related classes to modify references to old methods or fields in order to point to the new field entry table, or to point to the new method.

When updating a class C to C', we must ensure that all instances of class C are updated to be instances of class C'. For updating instances, garbage collection approach is used. It consists of a heap traversal from persistence root - which include object references in reference table, statics and stacks local variables- and for each encountered object O which belongs to C or which contains a reference to class C, need to be updated. The modified garbage collector contained by the Updater module, creates a new object O' with the appropriate size, rewrites unmodified fields O'.f with value of O.f and initializes added fields of O' with initial values computed off card. Afterwards, the Updater saves the new object references in the reference table of the VM. The old version of an instance is deprecated immediately after the new version has been created and filled with values, but the old object references is also saved in order to be used for roll-back if necessary.

Update process continue by updating references in frames to point to the new object instances and return addresses to point to the method bodies in the new space of the class. After realising the update, garbage collector is called to free the memory space.

4) Roll-Back: When there are errors at any point during update, in order not to leave any incoherence in the application, the update process must stop and the application must continue with previous component execution state. Indeed, roll-back functions allow to revert back to:

- The old version of instances by using old object references saved during instance updates and replace new references by old.
- The old version of code by changing all references in class metadata of related class to point to the old methods and fields in the old space of the class.
- The old version of object references and return addresses in the stack frames. And of course others VM data structures.

V. Micro Benchmarks

To determine EmbedDSU instance updates time, we compare cost of the updated garbage collector to the normal garbage collector. So, we determine the perfomance during instance updates when we added, deleted or re-ordered fields in the class to be updated and we compare to the normal garbage collector perfomance. The table 1 describes the different instance versions or configurations used for the micro benchmarks.

Re-ordered fields	Added fields	Deleted fields
class C' { int var2; short var3; int var1; Object var4; }	class C' { *short var5;* int var1; *int var6;* int var2; short var3; Object var4; }	class C' { int var1; short var3; }

TABLE I
THE THREE INSTANCE VERSIONS USED

The microbenchmark have three classes with re-ordered fields, added fields or deleted fields. The original class contains four fields (two integers, one short and one reference to an object initialize to null). We measure the coast of perfoming update while varying the number of objects for each configuration. We create 400 objects of the class to be updated and for each fraction of objects between 0% and 100%. So, after each set of 40 objects, we execute each configuration 25 times and report the median time obtained. And we realise the same for normal garbage collector but without dead objects, so just

run through the heap of the vm from persistance roots. Figure 2 shows the results.

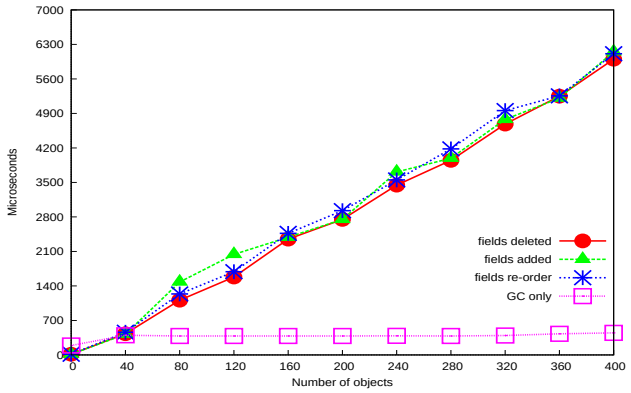

Fig. 2. MicroBenchmarks results for changing the fields of a class compared to a garbage collection run

We need to precise that normal garbage collector is applied without dead objects in the heap and then, without overhead brings by space memory free. In this case, the normal GC is highly optimized, whereas update instances, have overhead due to others functions like transformer functions which copie one field at a time and initialize new fields with values provided in off card and specified in the Diff. This extra functions make the process more expensive compared to this normal GC.

In other hands, we can see that adding fields is the most expensive operation under all others instances modifications because, it need to search a space memory with the new appropriate size and then copies the old mapped fields and initializes the new ones by the initial values provided. Re-ordering instance fields is also costly because we need to create a new object, make a copy of the fields one-by-one for all instances, but in this case, we do not need to read initial values provided by the Diff and this can reduce cost compared to adding fields operation. However, deleting instances field remains the least expensive operation, the objects are copied to their new space skipping the deleted fields and we do not need to initialize some fields with the Diff.

VI. Related works

Several dynamic Java software update techniques have been presented in the literature [2, 7, 8, 9, 10].

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).

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.

Our approach follows this idea, but instead of use UPT to get the list of direct or indirect modified classes, we use a tool to express the real changes in the content of classes, so obtain the real modifications between two versions of classes for example and represent them using a DSL designed specifically for capturing differences between classes in order 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.

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 in platforms, which are different of the smart card platform in the regard of resource constraints.

VII. Future works

A. Full State transfer

Currently in EmbedDSU, for updating instances, the strategy for initializing fields consists of searching fields that match by their name and type. And for matching fields, we copy the values from the old to the new instance. All others fields are initialized with a constant value obtained in off-card and then provided by the Diff. But, sometimes, initial values can be a result of a method execution or a result of an expression to evaluate during execution. Then, it is necessary to provide dynamic initialization of new instances fields during update process in order to apply an instance full state transfer by providing state transfer functions.

For implementing that feature, we expect to couple state transfer file with the Diff file in on-card, with state transfer functions provided by the developers. We believe that developers themselves have the best understanding of evolutionary changes in their applications. However, to not permit human intervention in the process, the automatic way will be better.

B. Late loading of DIFF file

In this section, we discuss how the notification of available new updates can be provided to on-card update manager. We propose to use a centralized repository, where all updates (DIFF files) are stored. The on-card update manager can then, for an application or a system component, check whether any new versions are available. However, this solution requires that in the repository, we have a service registry that is able to give for a precise smart card, all installed applications and system components. Currently, the question is how to implement the update manager in order to check and load updates or DIFF files in order to apply dynamic update? For simplicity, we assume that, all card readers can have some primitives update informations about DIFF file which are really stored on the

centralized repository. Then, when the reader powered up the card, it can send update message to on-card update manager to inform it that new updates are available. After that, on-card update manager can provide a push request for obtaining a set of DIFF files related to classes to be updated.

C. Dynamic wrapper methods or proxies methods

Currently, EmbedDSU permits update in cascade of a class and related classes. But, this approach can made the card unavailable during a certain time for the users if we have an important numbers of class to update. Therefore, we think to update only related classes which contains calls to deleted methods but for other classes which contains only modified methods including the method signature, we expect to provide dynamic wrapper or proxies methods created during update. Those wrappers forwarded old method calls for corresponding to the new calls with real number and type of parameters. However, we need to precise that all those signature modifications can not always be wrapped. We can have many type of changes, for instance a method RT m(T1 p1, T2 p2) can be modified to perform change in:

- Orders of parameter : → RT m(T2 p2, T1 p1), wrapper method can be such as :
  RT m(T1 p1, T2 p2) { **return** m(p2, p1); }
- Less of parameters : → RT m(T2 p2), in such case, wrapper can look as :
  RT m(T1 p1, T2 p2) { **return** m(p2); }
- Additional parameters : → RT m(T1 p1, T2 p2, T3 p3).

That later case is the very difficult, because we can not generate always a initial value for the added parameters. Indeed, for parameter like password, secret code, or pin code, if that type of parameter is added, we can not predict the value to affect. But for some cases, in which initialization is possible, we can then have a wrapper similar to :

RT m(T1 p1, T2 p2) {
T3 p3 = null ; // or other initial value depending on type of parameter;
return m(p1, p2, p3); }

For static or final methods where behavior cannot be modified, it is little more complicated. Our assumption is to require that new versions always follows the inheritance hierarchy of the old version.

VIII. Conclusion

In this paper, we present our on-going work EmbedDSU, a technique for dynamic update code in the context of Java-based smart card. We show that EmbedDSU is divided in off-card and on-card mechanism and allow arbitrary modifications to java classes in on-card through the Diff file obtained in off-card. We need to precise that our benchmarks are done on an AMD Athlon Dual core machine, running at 1 GHZ with 2GB of RAM on which Ubuntu kernel version 2.6.28. But, we apply some constraints like using 2MB for heap of the VM. However, currently, we transfer the modified virtual machine to an evaluating board AT91 EB40A [14] to have the realistic benchmarks related to time of transfering the Diff, interpret it, searching the safe point, applying instance updates to compare to the time to transfer a new version of a class, link and continue execution. And determine also memory used during those operations.

References

[1] Agnes C. Noubissi, Jean-Louis Lanet and Julien Iguchi-Cartigny, *Incremental Dynamic Update For Java Smart Cards Applications*, ICONS'10, France, April 2010.

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

[5] *The Java Card 3.0 specification*: http://java.sun.com/javacard/

[6] Milan Fort, *Smart card application development using Java Card Technology*, SeWeS 2006.

[7] Suriya Subramanian, Michael Hicks, Kathryn S. McKinley, *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/

[15] *Apache* : http://jakarta.apache.org/bcel/

[16] *Global Platform* : http://www.globalplatform.org/