Special Military Academy of Saint-Cyr
Cadet Class General Simon (2003-2006)
2LT Daniel Reynaud-Plantey

J2ME Low Level Security

Implementation Versus Specification

Directed by Mr. John Healy and Lieutenant-Colonel Eric Filiol
September 12th 2005 - December 9th 2005

Galway - Mayo Institute of Technology, Galway, Ireland
Mathematics and Computing Department

Acknowledgements

My thanks go to Lieutenant-Colonel Eric Filiol, Head Scientist Officer at the French Army Signals Academy, Virology and Cryptology Laboratory, as well as to Mr. John Healy, Lecturer at the GMIT for their unconditional support and for their confidence in my capabilities.

I also have to thank the people on the KVM-Interest mailing list who took the time to read this thesis and who gave me their comments and pointed out some problems.

I would also like to thank Mr. Gabriel Hicks, Head of the Mathematics and Computing Department at the GMIT for accepting personally my application in the first place. And, last but not least, I have to thank everybody at USAC for hosting me and allowing me to work in good conditions and a great atmosphere.

Research Topic : J2ME Low-Level Security	**Project Nb :**

Author : Second Lieutenant **Daniel REYNAUD-PLANTEY**

Hosting Organisation : Galway Mayo Institute of Technology
Mathematics and Computing Department

Project Leader : Mr. John HEALY   **Tutor :** Lieutenant-Colonel Eric FILIOL

Jury : Professeur Ducassé
M. Healy
Lieutenant-Colonel Filiol
M. L'Haridon

Public Defense : January 13th 2005

Keywords : KVM, bytecode verification, J2ME, CLDC, class file format integrity

---

Study :

PRESENTATION :

    The Java language has been built from the beginning with security in mind and was designed for running mobile code over heterogeneous networks. Recently, Java has been ported to SmartPhones, and thus the new challenge is : how to run untrusted code safely on a phone or another mobile device with a limited amount of memory and processing power.
    My goal is to assess the security of the current implementation of Java on mobile phones, and more precisely to ensure that the Java low-level security strategy is enforced correctly.

CONSTRAINTS :

    I have to face several problems, one of the most important being that there is no standard way to edit directly the content of a binary class file. The other difficulty is to study an embedded system like the virtual machine on a cell phone, with limited output and error messages.

APPROACH :

First of all, I studied the different technologies involved in J2ME, which offer different capabilities and APIs to the Java programmer but also require a different development process, and I decided on a strategy for running tests.

Then, I created an Open Source toolkit to enable a low-level control over Java Archive and class files. For that purpose I produced new tools from scratch, but I also updated standard tools like Jasmin.

Finally, I used this toolkit to black-box test the KVM on a real phone and compared the behaviors of the KVM and an emulator.

RESULTS :

The tools produced have proved themselves very useful, and offer unique features. This toolkit should be reusable in other projects and can be viewed as a contribution to the Java community.

Several non-critical bugs have been found during the tests, as well as several differences in behavior between the phone used and the emulator. However, no security flaw has been discovered.

LIMITS :

The testing phase has also shown the limits of the toolkit for generating test cases : it still involves a lot of human interaction. The toolkit automates a lot of the process but it remains manual testing, and thus the quality of the test depends on the quality of the tester.

This approach, manual black box testing, has also clearly shown another serious limit : it can't prove the validity of the security model, and it won't exhibit non-trivial bugs.

CONCLUSION :

Although the approach taken is not extremely sophisticated, it shows that mature products like J2SE still contain "trivial" bugs. It also probably remains a good approach for testing file integrity checks, although the embedded nature of the KVM makes it more difficult.

It also provides a good ground for more sophisticated testing strategies, such as fully automated black box testing, which could prove that all the checks in a specification are actually enforced in a given implementation. This could be done in conjunction with the efforts made to formalize the safety of the bytecode verifier.

Sujet du mémoire : J2ME Low Level Security   **N° Projet :**

Auteur : Le Sous-Lieutenant **Daniel REYNAUD-PLANTEY**

Organisme d'accueil : Galway Mayo Institute of Technology
                          Mathematics and Computing Department

Directeur de Stage : M. John HEALY   **Tuteur :** Lieutenant-Colonel Eric FILIOL

Jury : Professeur Ducassé
           M. Healy
           Lieutenant-Colonel Filiol
           M. L'Haridon

Public Defense : 13 Janvier 2005

Mots clés : KVM, bytecode verification, J2ME, CLDC, class file format integrity

Étude :

PRÉSENTATION :

Le langage Java a, dès sa création, été conçu pour permettre d'exécuter du code mobile sur des réseaux hétérogènes. Récemment, la technologie Java a été adaptée sur les téléphones portables intelligents, ce qui implique un nouveau défi : comment exécuter en toute sécurité du code potentiellement hostile sur un équipement disposant de ressources limitées en termes de mémoire et de puissance de calcul ?

Mon but est d'évaluer la sécurité de l'implémentation actuelle de Java sur les téléphones portables, et plus précisément de m'assurer que la stratégie de sécurité bas niveau de Java est implémentée correctement.

CONTRAINTES :

Les difficultés auxquelles j'ai à faire face sont d'une part qu'il n'y a pas de moyen simple de modifier directement le contenu d'un fichier binaire Java, et d'autre part que l'on dispose de très peu d'informations sur le résultat de l'exécution d'un programme sur un système embarqué comme la KVM.

DÉMARCHE :

Dans un premier temps, j'ai étudié les différentes technologies en rapport avec J2ME, qui impliquent une interface différente et un processus de développement différent pour le programmeur. J'ai également décidé d'une stratégie pour tester la KVM.

Ensuite, j'ai créé un ensemble d'outils Open Source pour permettre de manipuler au plus bas niveau les fichiers Java Archive et les fichiers class. Pour cela, j'ai été amené à créer des outils totalement nouveaux, ainsi qu'à mettre à jour des outils existants comme Jasmin.

Enfin, j'ai utilisé ces outils pour générer un ensemble de fichiers invalides que j'ai essayé d'exécuter sur la KVM et observer son comportement.

RÉSULTATS :

Les outils se sont révélés être très utiles et offrent des fonctionnalités uniques. Ils devraient pouvoir être réutilisables dans d'autres projets, et peuvent donc être considérés comme une participation à la communauté Java.

Plusieurs bugs non-critiques ont été découverts pendant la phase de tests, ainsi que des différences de comportement entre le téléphone et l'émulateur. Cependant, aucune faille de sécurité n'a été découverte.

LIMITES :

La phase de tests a également mis en évidence les limites des outils pour créer des fichiers test : ils nécessitent encore beaucoup d'actions pour un humain. Ils permettent d'automatiser une bonne partie du processus, mais cela reste des tests manuels. La qualité des tests dépend donc de la qualité du testeur.

Cette approche, le test manuel sous forme de boîte noire, a aussi une autre limite très sérieuse : elle ne permet en aucun cas de prouver la sécurité d'un modèle, de même qu'elle ne permettra pas de mettre en évidence des bugs non triviaux.

CONCLUSION :

Bien que l'approche choisie ne soit pas extrêmement sophistiquée, elle montre que même des produits qui ont atteint leur maturité comme J2SE contiennent toujours des bugs triviaux. Cela reste également une bonne approche pour mener des tests de très bas niveau, notamment en ce qui concerne la vérification d'intégrité.

De plus, elle peut servir de base à une approche plus générale : les tests entièrement automatisés. Cette nouvelle approche pourrait permettre de prouver dans une certaine mesure que le comportement d'une implémentation donnée correspond à la spécification, si celle-ci a été suffisamment formalisée.

Table of Contents

Introduction

One of the strong selling arguments of Java is its security. Until recently, end-users knew Java only through web-based applets, demonstrating the idea of running untrusted code in a secure environment, the so-called sandbox. However, applets are run on desktops and benefit from the power of modern computers. Java has now been ported to new platforms such as SmartPhones and PDAs, and thus there is a new challenge : how to run untrusted code safely on a mobile device with a limited amount of memory and processing power ? To address this problem, Sun has designed a new security model. The goal of this study is to test the low-level security of Java 2 Micro Edition (J2ME).

In section 1, I am going to introduce J2ME by first presenting the general Java landscape and then the difference between J2ME and standard Java. Then I will present J2ME's security architecture at the end-to-end, application and low level. In section 3 I will analyse what was needed to test J2ME low-level security. My conclusion was that I needed new tools, including a java disassembler and an assembler. I have detailed the creation of these tools to make my testing strategy clear to the reader. In the last section I will explain how I have created the test case and will detail some of the problems it revealed.

Here are some of the guidelines I have decided to follow throughout this study :

- it must not remain theoretical. The final goal is to find real bugs in real phones, and among these bugs to find the ones which can be exploited. A side goal is to analyse the malware risk on J2ME.

- the tools had to be efficient enough to be useful for other projects. For me, writing tools just for a particular project would be a failure.

- the test case should be well documented and not specific to J2ME in order to be reusable by other projects.

Notations and Abbreviations

For this study, *The Java Virtual Machine Specification* by Tim Lindholm and Frank Yellin has been a central document. This document specifies the class file format and its constraints. Later on, I will refer to it using the term JVMS.

The items in the bibliography have been divided into different categories depending on their nature : books, specifications, white papers and electronic resources. Each category has been assigned a letter and each item has been assigned a number. For example [S1] refers to the first item in the specifications category (in this case, JVMS).

Some common Java terms and notions will not necessarily be defined later in this document, such as :

Virtual Machine (VM) : a program able to load and run Java class Files

applet : a Java program designed to run in a browser

MIDlet : a Java program designed to run on a MIDP mobile device (such as a mobile phone)

class file : it is a file representing a single Java class. They are generally (but not necessarily) produced by Java compilers and have the extension .class

malware : stands for malicious software. This is a very general term to refer to viruses, spyware, adware, trojans and other software with hidden or dangerous features.

application installer (or just **installer**) : the software on the phone responsible for the installation of MIDlets. This software is not part of the KVM.

I. Introduction to J2ME

The goal of this section is to provide a background on the technologies involved in Java 2 Micro Edition, for readers new to Java as well as readers used to Java 2 Standard Edition. Most of the information covered in this section can be found in [WP2] and [WP6].

1.1 General Java Landscape

1.1.1 "Write once, run everywhere"

Since its inception, the evolution of Java has been driven by the motto "Write once, run everywhere", meaning that Java is totally platform independent due to its interpreted nature. This is no longer exactly true nowadays. For performance reasons, the Java platform has evolved differently on different platforms. Java still achieves some sort of operating system independence, mainly because the Java programmer doesn't have access to pointer arithmetic. This means that the whole memory management process is realized by the virtual machine. But Java has evolved a lot since its early beginnings, at all levels : the bytecode specification has changed, a lot of additions have been made to the language itself, and there are many different optional Application Programming Interfaces (APIs). All of these facts tend to show that Java code can sometimes be hard to maintain from one version to the other, or from one platform to the other.

Another problem is that there are lots of different compilers and virtual machines (VMs) for each operating system. And sometimes these compilers and VMs come with their own set of classes and standard libraries. The situation has evolved in this direction because of the proprietary nature of Sun's products. They usually provide the source for their runtime environments (JRE) and standard libraries, but the license doesn't allow developers to modify and redistribute it. Of course, this clause makes it incompatible with the GNU standards, and as a consequence the GNU ClassPath project is born. The goal of this project is to provide a GNU replacement for Sun's standard libraries. From a developer's point of view, it means that you can't assume everybody is going to use Sun's products, some people might choose other products. This might break the compatibility of your software from one platform to the other, therefore the motto "Write once, run everywhere" is not absolutely true in practice.

1.1.2 Different platforms, different flavours

Java was also supposed to be a kind of universal language, able to run on smart cards as well as on mainframes. As obviously you don't expect the same features from such different devices, the Java technology has to adapt to what people expect from it and to the capabilities of the device it is running on. For example, a Java program must be able to run for days without rebooting on an enterprise server. But on a workstation, you have different expectations : you probably want to use short term applications with an efficient graphical user interface. That's why Java comes in different flavours :

- Java 2 Enterprise Edition (J2EE), for running scalable enterprise applications

- Java 2 Standard Edition (J2SE), aimed at workstations

- Java 2 Micro Edition (J2ME), for mobile devices

- Java Card, for smart cards

For the Java developer, it means two things : with a single language, you can write applications for every platform. But it also means that you will have to write code for a given platform, and it usually means that you may have to use different language constructs (for example you might not perform floating point operations on a smart card), and you will definitely have to use different APIs. This might or might not be a big deal, depending on what you want to do exactly. From a security perspective, the broad range of devices might sound appealing to malicious programmers but they will have to face another drawback of the Java technology : the lack of low-level control.

The different editions of Java and their associated platforms are summed up in the following figure[1] :
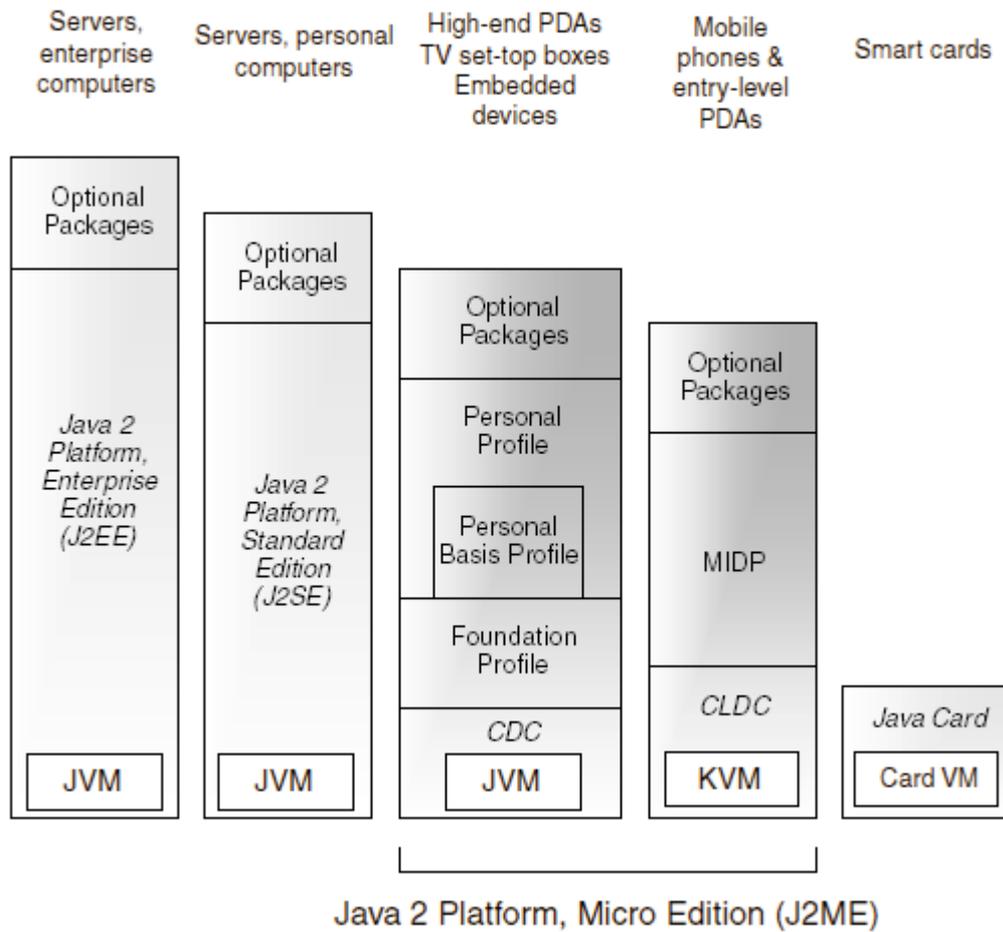

Illustration 1 : The Java Landscape

---

1   Taken from *Introduction to J2ME* in [WP6]

1.2 How does J2ME work ?

1.2.1 Configurations

So far, we have seen that J2ME was intended to bring Java to mobile devices. These devices include mobile phones and PDAs as well as washing machines, TV set-top boxes or car electronic equipment. What do they all have in common ? They are mobile, which means they have a limited amount of memory and computing power. All their other characteristics may vary : some of them might need batteries, some might have a good bandwidth or not, and some might not even have a screen. As a consequence, Java needs again to be adapted to different kinds of mobile devices, depending on their capabilities. This is the role of a configuration : provide the basic set of functionalities for a specific group of devices with similar characteristics, such as the total amount of memory and network connectivity. Therefore, **a configuration provides a virtual machine and a set of core classes**.

Two configurations have been defined :

- Connected, Limited Device Configuration (CLDC) : it is the configuration currently in use on SmartPhones and some PDAs. It is defined in the Java Specification Request (JSR) 30.

- Connected Device Configuration (CDC) : for devices with more memory and processing power. From the developer point of view it is closer to J2SE.

From now on, we will focus on CLDC 1.1 as specified in JSR 139 [S2]. This document is particularly important for this study, since J2ME security is addressed mainly in the CLDC specification.

CLDC 1.1 assumes that the target device has the following characteristics :
- at least 192 kB of total memory budget available for the Java platform (...),
- a 16-bit or 32-bit processor,
- low power consumption, often operating with battery power,

connectivity to some kind of network, often with a wireless, intermittent connection and with limited bandwidth.[2]

1.2.2 Profiles

We have seen that J2ME had to be split into configurations in order to fit on different devices. But even similar devices like mobile devices might offer different ranges of memory and processing power. They also might have more to offer in terms of underlying user interface, or advanced features like BlueTooth. This is why configurations are not enough and as a result profiles have been defined. **A profile is a set of APIs which sit on top of a given configuration** in order to take better advantage of the features of a device.

Six profiles have been defined, but only one is a CLDC profile : Mobile Information Device Profile (MIDP). It is made to take advantage of the capabilities of SmartPhones, and its APIs cover user interfaces, connectivity, data storage, messaging and gaming. MIDP 2.0 is specified through JSR 118 [S4]. It is commonly found on SmartPhones such as the one studied later, but the profiles are too high-level to fit in this study. They only address security at the application or end-to-end level, for example they can provide network or encryption APIs.

---

2 Taken from JSR-139 [S2]

1.3 The KVM

The KVM, also known as the Kilo Virtual Machine, is a virtual machine conforming to the CLDC specification. It is Sun's reference implementation of a Java virtual machine for mobile devices, designed to have a small memory footprint. It has been programmed in C, so that it can easily be ported to platforms for which a C compiler is available. The *KVM Porting Guide* [WP3] describes the different compilation requirements and options. It is meant to be used along with the KVM reference implementation, available for free with the complete source code. This fact is important because it can determine the way we are going to assess J2ME security.

The goal of the KVM is to provide an efficient Java environment for resource-constrained devices, with a support for the maximum of Java language features. Yet, some features of the language have not been implemented for performance or security reasons (as a consequence, the KVM is not fully compliant to the *Java Language Specification*) :

- no finalisation methods

- limited error and exception handling. For example, only 3 subclasses of `java.lang.Error` are defined : `NoClassDefFoundError`, `OutOfMemoryError` and `VirtualMachineError`. From the tester point of view, it means that it will be harder to understand the exact reason for a given failure.

Some features described in the *Java Virtual Machine Specification* [S1] have also been eliminated from the KVM as mentioned above :

- User-defined class loaders (JVMS §5.3.2)
- Thread groups and daemon threads (JVMS §2.19, §8.12)
- Finalization of class instances (JVMS §2.17.7)
- Asynchronous exceptions (JVMS §2.16.1)[3]

The class file verification is also different from what is described in JVMS, but this will be detailed later on. The most important eliminated feature from a security point of view is the possibility to have user-defined class loaders. In J2SE, it is possible to load classes either indirectly (by asking, or rather letting the virtual machine load them using its built-in class loader) or directly, by subclassing `java.lang.ClassLoader`. This can be used in obfuscation schemes, for example some classes can be encoded or encrypted and the only way to load them is to use a specific `ClassLoader`. This technique can also be used by sophisticated viruses to generate classes on the fly, as described in *New Threats of Java Viruses* [WP7]. Therefore, if Java viruses can be written for J2ME, they will have to rely on less sophisticated techniques.

The KVM is likely to be replaced by newer versions like the Monty VM in the next generation of mobile devices. The KVM is a simple bytecode interpreter loop, which has strong performance drawbacks. Monty VM uses a technology called Just In Time (JIT) compilation, which enables compilation of some parts of the bytecode to native code. This can increase performance by an order of magnitude, and this technology is now used widely in J2SE. As a tester, it implies a new constraint : the test case I am going to generate must not be VM dependent, so that the effort made to assess the security of a given VM will not be lost for the next major technology update.

---

3 Taken from *CLDC 1.1 Specification* [S2]

II. J2ME Security Architecture

The following section gives an overview of the general J2ME security architecture, and introduces key concepts such as the new verification process.

2.1 End-to-End Level

End-to-end security is provided by MIDP, through packages like javax.microedition.pki (standing for Public Key Infrastructure) and javax.microedition.io (which provides support for the Secure Socket Layer protocol). End-to-end security is beyond the scope of this study.

2.2 Application Level

2.2.1 Protection Domains

The application level security is also addressed by MIDP. In MIDP 1.0, the security model was very similar to the one used in J2SE with applets : a given application was untrusted by default and could run as trusted if it was digitally signed. The "untrusted" domain offered very limited access to sensitive API and OS functions such as network, disk or system access. On the other hand, the trusted domain granted all permissions to the application.

Since MIDP 2.0, the notion of protection domains has been introduced. **Protection domains are sets of permissions**, giving more control over what an application is allowed to do or not depending on its signer. Four protection domains are defined in MIDP 2.0's "Recommended Security Policy" :

- **Untrusted** : default, for unsigned applications. A warning message is displayed before installation, and access to every sensitive functionality requires user approval.

- **Third Party** : default for signed, commercial applications. No more warning message during installation, and permissions are slightly less restrictive.

- **Operator** : made for network operators, it doesn't require user approval for establishing communications.

- **Manufacturer** : equivalent to the old trusted domain with all permissions, it also gives access to device specific APIs which could be otherwise hidden.[4]

In order to access a different protection domain, the MIDlet has to be signed by a root certificate embedded on the target device. It is possible to have a MIDlet signed for the Third Party domain by undergoing the Java Verified program, which I am going to briefly study in the following section as an example of signing procedure.

2.2.2 Java Verified

The Java Verified program has been initiated by the industry and is a result of the Unified Testing Initiative (UTI). The signing process is the following :

- the developer registers with the Java Verified program

- he/she submits his/her application for testing

- the application undergoes an automatic pretesting phase, to ensure that the application is packaged correctly

- the developer then chooses a Testing Provider. According to *The Java Verified Program Guide* [E1] : "The Provider contacts the Developer, establishes a business relationship with the Developer and accesses the application."

---

4 As stated in *Getting Started with Security*, Forum Nokia, July 2005

- the main tests are then performed by the Testing Provider as specified in the *Unified Testing Criteria* [E2]

- if successful, a CA issues an identity certificate for the developer. This certificate is used to sign the MIDlet suite and is in turn signed by the UTI root certificate which is embedded in the target device

Once this process is completed, the developer can distribute his/her application along with the Java Powered logo and run in the Third Party domain. It is important to notice that this process requires payment, therefore this program is made for commercial applications.

Since this process grants access to a different security domain, it is reasonable to assume that the tests run by the Testing Provider include some security checks to ensure that no malware can go successfully through the testing process. The tests to be performed are detailed in the *Unified Testing Criteria* [E2] and are divided into the ten following categories :

- Application Characteristics
- Stability
- Application Launch
- User Interface Requirements
- Localization
- Functionality
- Connectivity
- Personal Information Management
- Security
- Retesting

Indeed, some of these tests are supposed to detect malware functionalities. For example, the test SE6 (in the Security category) :

SE6	Running environment	
Full Description The application must run in the sandbox environment and not exploit any malicious means of exiting the sandbox environment.		
Steps to conduct the test 1. Check the declaration statement on "Application Characteristics".	*Expected Result* It has been declared that the application runs in the sandbox environment and does not exploit any malicious ways of exiting the sandbox.	
Notes -	*Exceptions* -	
PASS ☐	FAIL ☐	PASSED WITH EXCEPTION ☐

We can see that the tester must check if the application does not try to break the security model by checking that you declared it doesn't. And it is true you have to fill out a form when you submit an application for testing :

F. Declarations:
1. Does your application override system or virtual machine generated security prompts or notifications or deceive the user by displaying misleading information just before a security prompt is shown to the user? Yes / No
2. Does your application simulate security warnings to mislead the user? Yes / No
3. Does your application simulate key-press events to mislead the user? Yes / No
4. Does your application run in the sandbox environment and not exploit any malicious means of exiting the sandbox environment? Yes / No

Of course these are not real tests and there is no way to tell if a given program is a malware just by executing it. And given the fact that Java Verified is made for commercial programs, you don't have to provide the tester with the source code of your application. Moreover, the application must be tested in its exact final package, because once it is signed it can't be modified any longer. As a consequence, it is possible to give the tester an obfuscated application. There is no theoretical limit to the level of obfuscation that Java applications can use, thereby making it impossible to tell whether a given program is a malware or not.[5]

The tester has to check that no undocumented event has taken place, such as tampering with the phone data or the user information. No outgoing connections should have taken place if they are not documented, and so on. But a trivial way to bypass this is to trigger the hidden features of a malware after a certain date, when the testing process is over and the application is signed.

To sum it up : i**t is possible for a malware to go through the Java Verified program successfully.** It requires some efforts for the malware author but it is possible. However, he/she faces legal problems since he/she intentionally made false declarations to the Testing Provider, and he/she had to give credentials to the CA. It might not be worth the effort though, since the Third Party domain does not automatically grant access to any sensitive API.

2.3 Low Level

We have seen that a Java program had limited access to the device's resources and it needed permissions to access some APIs. This model is called the sandbox, because of the idea that such a Java program runs in a closed, safe environment and can't do any harm to the device. But how is this model enforced ? What prevents a Java program from accessing other memory locations ?

2.3.1 Class File Loading

The whole low-level security process is part of the class file loading. What happens when the virtual machine is required to load a class C ?

- it looks for a file called C.class in a predefined list of directories and jar files.

- if C.class is contained in a jar file, it is first extracted from the archive.

- then C.class is parsed and some checks are performed to ensure that its format is correct (later on, this process will be called *class file format integrity check*).

- if it has been parsed successfully, the code array of each method is verified by the bytecode verifier. This process is detailed in the following sub-section.

- class C is then ready to be used by other classes. Some security checks will only be performed at run time (such as the access to protected or private members).

The upper (Application and End-to-End) security levels can only be applied if there is no low-level compromising. Basically, it means that the class file loading and verification must be made correctly. It is possible to imagine several attacks at this level, jeopardizing the whole security model :

- if the class file lookup order can be modified, it is possible to load potentially dangerous classes instead of the requested ones.

- if the jar file is not handled safely, it might be possible to crash the virtual machine (for example by declaring that a file has a negative uncompressed length) or to exploit a buffer overflow. Many attacks rely on bad zip file handling in commercial programs.

---

5 see *Reverse Engineering and Java Viral Analysis* [WP8] for a study of obfuscation techniques usable by Java malware

- if the class file format integrity check is not correct, it might give an opportunity to an attacker to exploit the virtual machine in a dangerous way. For example, if class C declares that its super class is C itself, the virtual machine must not end up in an infinite loop but throw a ClassCircularityError.

- if the bytecode verification is not made correctly, it is possible to escape the sandbox and as a consequence to break the whole Java security model.

- if runtime checks are not performed correctly, it might, for example, be possible for an attacker to access private fields or methods in other classes and then to break the Java semantics.

2.3.2 The Bytecode Verifier

The bytecode verifier is a key component of Java low-level security. Its role is to guarantee that no stack overflows or underflows can occur, that all the local variables are accessed legally, and that the data types are consistent during the execution (the so-called type safety). It parses the code array of each method in a class file and checks that :

- there are no illegal instructions

- the execution can't proceed outside the code array

- there is no attempt to read from invalid memory locations

- the method is *type safe*

All of these checks are still structural checks, except for a very complex one : the type safety check. For security reasons, the in-memory representation of Java data are typed. For example, the type of a pointer to an object is `reference`. Therefore, if an instruction requires a `reference` as an argument (either on the stack or in a local variable), it is impossible to give it an argument of type `int` or `uninitialized`.s

The heart of the verifier is the type safety. But proving that a method is type safe is a very complex problem. The JVMS defines a type inference algorithm for the bytecode verifier. But it has several downsides :

- it has known several historical implementation issues which have been exploited by malware authors.

- it also suffers from a strong design problem : its complexity is $O(n^2)$. A denial of service attack is possible by creating a class using worst-case code arrays.[6]

- generally speaking, this algorithm is inefficient from the memory usage and required processing power point of view.

As a consequence a new algorithm has been defined in CLDC, more specifically in its Appendix 1 : *CLDC Byte Code Typechecker Specification* [S3]. It is suitable for mobile devices because it is more efficient, and the former DoS attack is no longer possible since its complexity is $O(n)$. We can also notice that a formalisation effort has been made, since most of the specification is made of Prolog rules indicating what type safety means for a class, a method, and eventually an instruction. However, this is not a formal proof that the type checker is correct, but scholars are working on this kind of formal proof ([WP4], [WP5]). The same kind of algorithm will become standard, beginning with Java 1.6.

---

6 *A Denial of Service Attack on the Java Bytecode Verifier*, [WP1]

2.3.3 Type Safety

In this section, I want to underline the fact that if type safety is broken, the whole Java security model collapses. The following figure shows the CLDC verification type hierarchy, as appearing in [S3] :
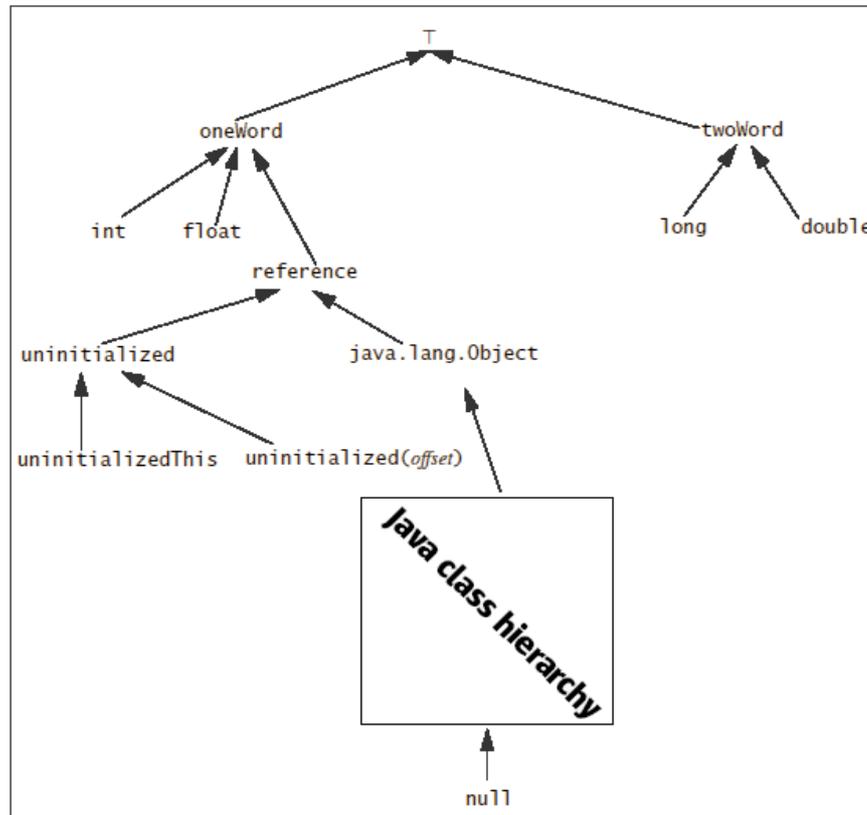
Illustration 2 : The Verification Type Hierarchy

The relation indicated by the arrow is "is assignable to". For example, a `reference` is assignable to a `oneWord`, but not to an `uninitialized`.. Let's assume that the typechecker encounters the instruction `iload x` in a method (which means load int from local variable at index x). [S3] specifies the following Prolog rules for checking the type safety of this instruction :

```
    instructionIsTypeSafe(iload(Index), Environment, _Offset, StackFrame,
NextStackFrame, ExceptionStackFrame) :-
        loadIsTypeSafe(Environment, Index, int, StackFrame, NextStackFrame),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

The predicates loadIsTypeSafe and exceptionStackFrame must be true. Here is the definition of loadIsTypeSafe :

```
    loadIsTypeSafe(Environment, Index, Type, StackFrame, NextStackFrame) :-
        StackFrame = frame(Locals, _OperandStack, _Flags),
        nth0(Index, Locals, ActualType),
        isAssignable(ActualType, Type),
        validTypeTransition(Environment, [], ActualType, StackFrame,
        NextStackFrame).
```

If the predicate `isAssignable(ActualType, int)` is not true, the `iload x` instruction will not be declared type safe. As Illustration 2 shows, no other types are assignable to `int`, except `int` itself (because "is assignable to" is reflexive). Therefore, if the local variable at index x is not an int, the instruction will not be declared type safe. The verification will fail, and the class will not be executed.

This simple example shows how the bytecode verifier checks the type safety for a single instruction. If these tests were not enforced correctly, it could be possible to access arbitrary memory locations. Consider the following example :

```
bipush x  ; pushes byte x on the stack
istore 1  ; stores x in local variable 1
aload 1   ; load reference from local variable 1
```

This code snippet tries to convert an `int` to a `reference`. This would cause arbitrary memory locations to be read, resulting in a total security failure. Hopefully, the bytecode verifier will detect that an `int` is not assignable to a `reference`, and will throw a `VerifyError`.

III. Testing J2ME Security

In this section, I analyse the different options for running tests : the testing strategy, the part of the security model to be tested and the tools used.

3.1 The Testing Strategy

There are two major strategies for testing software : white-box (or structural) and black-box (or behavioral) testing. The main difference between those strategies is that white-box testing assumes that the tester has access to the source code of the application. Brett Pettichord [E5] gives interesting ways to classify white-box and black-box testing, summed up in the following table :

	White-Box	**Black-Box**
People running the tests	Developers	Users
Coverage	Units, modules	The application as a whole
Risks assessed	Security	Usability
How the tests are run	Static analysis, formal inspection	Dynamic test execution
Evaluation (how to know when a bug is found)	Code instrumentation	Output or behavior of the program

In our case it is possible to choose white-box testing, because the source code of the KVM's reference implementation is available. This might not be a good idea however, because the actual implementation on the device might be different, even slightly, and these changes may introduce security flaws.

Here are the criteria to decide on a strategy in the case of J2ME, using Mr. Pettichord's classification :

- **People** : it is possible to play the role of a KVM developer by playing with the source code, but it means that the study would be focused on the reference implementation rather than the actual embedded implementation.

- **Coverage** : though the idea of testing each component of the KVM sounds appealing, it might not exhibit exploitable security bugs. For example, a bug in the class file format integrity checker might not go through the installer correctly, and is therefore not exploitable. The tests will later show that this scenario is very realistic. We can conclude that though black-box testing might not find all the bugs in an implementation, the bugs it will find are more likely to be exploitable.

- **Risks assessed** : here I disagree with Mr. Pettichord, white-box and black-box testing can both exhibit security flaws, but in different areas. For example : white-box testing can show that a security check is not enforced correctly ("I implemented this check, is it working as expected ?"), but on the other hand black-box testing can show that a security check is not enforced at all ("what did the programmer do ?"). Both methods are interesting from a security point of view.

- **How the tests are run** : it is possible to run tools which will look for common programming errors in the source code of an application. In our case it is not extremely interesting, because the KVM developer team probably uses such tools internally. Formal inspection is much more interesting but requires a lot of work, and it might be impossible to formalise a whole application such as the KVM. Most of the formalisation effort has been made on the verifier ([WP4] and [WP5]), but once again the bugs found might not be exploitable. It is simpler to generate executable test cases, even if the errors found will probably not be complete.

- **Evaluation** : in our case, code instrumentation is impossible due to the embedded nature of the KVM. But observing the behavior of the program might not be sufficient to find subtle bugs. And it turned out that this is a serious limiting constraint.

The analysis of these different criteria led me to choose black-box testing, because the test case is easier to generate and the bugs found are more likely to be exploitable. It is also particularly convenient to check that an implementation is consistent with the specification, and in our case the JVMS and the CLDC specification give a very good list of the checks a virtual machine should implement. Some of the drawbacks of black-box testing are that it will not prove that the implementation is correct if no bugs are found, and as we rely on the behavior of the KVM some bugs might not be straightforward to identify as such.

3.2 Coverage

Before creating the test case, I have to define precisely the limits of what I am going to test. Here are some of the different options :

- **the installer** : can be tested by playing with the jar file format, and it turned out that the class file format could also have an influence. It is worth noticing that the installer is not part of the KVM but is part of the life cycle of a MIDlet, so it can be worth testing it too. For example, it might be possible to exploit the installer to overwrite system files on the phone.

- **the jar handler of the KVM** : this part of the KVM is not documented at all, so we can expect to find bugs easily here, such as crashes and maybe buffer overflows.

- **the class file format integrity checker** : by playing with the class file format, we might find that the KVM does not parse class files correctly.

- **the bytecode verifier** : maybe the verifier does not check the type safety correctly. Bugs will probably be harder to find here because a lot of the development effort has been spent on the verifier, but if it contains bugs they are going to be critical bugs. For example, Adam Gowdiak found two of these critical bugs, which allowed him to break the sandbox completely [E4].

- **the runtime environment** : maybe some of the runtime checks are not performed correctly. However, there are not so many checks, and testing them is not as difficult as testing the verifier.

- **the security manager** : this is reaching the application level. The security manager is supposed to ensure that a MIDlet does only what is permitted for its protection domain.

- **the APIs** : some of the APIs on the phone might be buggy, leading to denial of service attacks. It might be interesting to look at the BlueTooth APIs for example.

I decided to focus on the lower levels : the jar handler and the class file format integrity checker. Creating a complete test case for the verifier is an extremely interesting but daunting task without specific tools, so I will only check for some known issues. The installer will also be indirectly tested, because the tests on the jar file format are also going to affect it. It would also be interesting to continue the tests for higher levels, knowing some of the bugs eventually found at the lower levels.

At this point, we can have an accurate outline of the strategy : I am going to **black-box test the jar file handler and the class file format integrity checker of the KVM**. As a side effect, the installer and the verifier will also be somewhat tested. These tests rely on input validation, so I need the ability to generate invalid jar files and class files. Unfortunately, it turned out that such tools were not available, or were available but not adapted to the mission. The conclusion is that I had to create my own set of tools to manipulate jar and class files.

3.3 Creating the Tools : the Tinapoc Project[7]

I couldn't find any low-level manipulation tools for jar (i.e. : zip) files. And one of the only standard tools for manipulating classes was Jasmin, a Java assembler. But its purpose was more to learn the virtual machine's instruction set than to create invalid files. And I was not satisfied with the existing Java disassemblers, most of them crashing or giving no output when they encounter an invalid class file. So I decided to write my own tools and ended up with a list of requirements for a suitable toolkit :

- it must give sufficient low-level control over the creation of binary files (zip and class).

- there must be tools to check that "the errors are correct". I mean that if I introduce an error in a binary file, a tool such as a disassembler must show me that the error is the one I wanted. This is necessary to ensure that the assembling tools are working correctly, so that the test case is accurate enough.

- the disassembling tools must work on broken, invalid, illegal or truncated files. This is a very strong constraint, and this is one of the top reasons to create new tools.

- the tools must automate a maximum of the process of creating invalid MIDlet suites.

- they must be useful to other projects, not only this one. This is a good way to obtain feedback from other people in order to improve the quality of the product.

3.3.1 Programming Language, APIs, Interface and License

I chose to program the toolkit in **Java**, because of its simplicity and portability. Moreover, the standard libraries provide interesting APIs. For example, the class `java.lang.DataInputStream` provides almost all the methods to parse a class file conveniently, such as `readUTF()`.

Then I had to choose which APIs I was going to use to manipulate binary files. The standard Java 1.5 library offers some zip manipulation utilities, but they have not been made to parse or create invalid files. And almost every API has the same problem : they implement things in a nice and clean way, but they are not made to be used with invalid files. Even the standard bytecode manipulation libraries, such as the ByteCode Engineering Library and ASM did not match my needs. Therefore I decided to use nothing but the standard Java libraries.

I chose to use a **command-line** interface. This is not the most user-friendly interface, but it can be used to combine multiple commands with scripts. This really speeded up the creation of the test case, since I was able to automate a lot of the process.

Finally, I decided to make the project available freely under the **GNU Public License** (GPL). This way, the toolkit is more likely to be used by other developers or advanced end-users. It is a good opportunity to obtain feedback from the community, and it led me to implement some features requested by experienced developers that I would not otherwise have thought of, or that I would have implemented in a bold way.[8] The project is now available at http://tinapoc.sourceforge.net/.

3.3.2 Zip Manipulation Tools

I designed two tools to manipulate zip files. The idea was to have a tool to convert a binary zip file to a text file, and another tool to convert this text file to a zip file. In the meantime, it should be possible to modify the text file easily, in order to produce a particular zip file. The problem was to find a convenient format for the text file. I didn't want to use a complex or sophisticated format because it meant that I would have to build a complex and sophisticated parser, and that I would have to define and learn yet another syntax.

I came up with a simple solution : I was going to use XML (eXtended Markup Language) as the format for the text files. It is simple, self-descriptive, doesn't require any learning, and is supported by many standard APIs. The tools have therefore been named zip2xml and xml2zip.

---

7   Tinapoc stands for "Tinapoc Is Not Another Pun On Coffee"
8   I have to thank the people from the GNU ClassPath and GCJ (GNU Compiler for Java)
    projects for providing me with useful insight during the development of the toolkit.

Example : here is the local file header data structure, as introduced in the *.ZIP file format specification* [S5] :

```
Local file header:
        local file header signature     4 bytes   (0x04034b50)
        version needed to extract       2 bytes
        general purpose bit flag        2 bytes
        compression method             2 bytes
        last mod file time             2 bytes
        last mod file date             2 bytes
        crc-32                         4 bytes
        compressed size                4 bytes
        uncompressed size              4 bytes
        file name length               2 bytes
        extra field length             2 bytes

        file name (variable size)
        extra field (variable size)
```

And here is an example of how it is represented in the XML files :

```
<local_file_header>
    <local_file_header_signature>0x4034b50</local_file_header_signature>
    <version_needed_to_extract>2.0</version_needed_to_extract>
    <compatibility_of_file_attribute_information_bis>0 (MS-DOS,
OS/2)</compatibility_of_file_attribute_information_bis>
    <general_purpose_bit_flag>0x0</general_purpose_bit_flag>
    <compression_method>8 (the file is Deflated)</compression_method>
    <last_mod_file_time>0x8779</last_mod_file_time>
    <last_mod_file_date>0x3361</last_mod_file_date>
    <crc_32>0x89e27c52</crc_32>
    <compressed_size>128</compressed_size>
    <uncompressed_size>184</uncompressed_size>
    <file_name_length>20</file_name_length>
    <extra_field_length>0</extra_field_length>
    <file_name>META-INF/MANIFEST.MF</file_name>
    <extra_field>(empty)</extra_field>
</local_file_header>
```

This example shows that there is a one-to-one mapping between the data structures in the specification and the elements in the XML tree. This gives the user a very fine-grained control over the data in the zip file.

We can also see in this example that it is possible to use numbers in decimal or hexadecimal notation (by pre-pending them with 0x). xml2zip also accepts comments as strings between brackets.

Finally, I would like to underline the fact that xml2zip is very permissive. Indeed, the XML file does not even need to be valid : if it is truncated (and hence does not represent a valid tree), the produced zip file will be truncated. The elements in the XML structure can be put in any order, or even be missing. This allows the user to do almost everything with the output zip file. zip2xml has been designed in a similar fashion : whatever appears in the zip file will appear in the XML file. Therefore, even an invalid zip file can be parsed successfully by zip2xml.

3.3.3 JasminXT

Then I needed a tool to create invalid class files. Creating a Java assembler is not straightforward, and the de-facto standard assembler, Jasmin, didn't allow enough low-level control for this study. Another drawback of using Jasmin is that it has not been maintained for years and it is really outdated. The solution was to **update Jasmin** itself : the new version should give more control over the creation of the class file, add some of the newer features of the Java language, and support the StackMap attribute (defined in the CLDC typechecker specification [S3]).

So I defined an extension of the Jasmin language, called JasminXT. Old Jasmin files can still be assembled with Jasmin. Here is a summary of the additions in JasminXT[9] :

- use of **offsets** for branch targets, local variable visibility and exception handlers. The offsets can either be absolute or relative :
```
goto  12  ; absolute offset : go to bytecode at offset 12
goto +5   ; relative offset : go 12 bytes forward
goto -8   ; relative offset : go 8 bytes backwards
```

- the following **access flags** are now supported : `ACC_ENUM`, `ACC_ANNOTATION`, `ACC_BRIDGE` and `ACC_VARARGS`

- the .bytecode directive has been added, to set the **bytecode version** in the class file. Example : `.bytecode 49.0`

- it is now possible to add a **SourceDebugExtension** attribute to the class with the following directive :
```
.debug "some string here"
```

- same thing for the **EnclosingMethod** attribute :
```
.enclosing method "some/package/Foo/someMethod(I)V"
```
or
```
.enclosing method "some/package/Foo"
```

- support for the **Signature** attribute (in the classes, methods and fields) :
```
.signature "<my::own>Signature()"
```
or
```
.field myField Ljava/lang/String; signature "<my::own>Signature()"
```

- support for the **StackMap** attribute, using the `.stack` directive in a method definition

- it is now possible to give the **offset of an instruction** before the instruction itself, like in the following code snippet :

```
0: aload_0
1: invokespecial java/lang/Object/<init>()V
4: aload_0
5: ldc2_w 3.14159
```

Most of the additions in JasminXT are meant to make Jasmin more permissive, in order to increase the control over the output class files. This is not particularly easy because Jasmin has not been designed for this. It has originally been designed to be a comfortable assembly language for the class file format. Some of the additions, like the support for new attributes, are just an extension to the set of Jasmin directives. But some of the other additions (like the use of offsets instead of labels) required deep modifications of the Jasmin parser and the Jasmin internal API for representing class files (JAS).

Jon Meyer, the author of Jasmin, has added me to the SourceForge developers of Jasmin.[10] It means that my updated version of Jasmin will not be a fork of the existing Jasmin, but will become the new official version.

3.3.4 Dejasmin

The last tool I needed was a reliable class file disassembler, which I programmed from scratch and called dejasmin. In this case, reliable means that the user must be able to see the actual content of the class file, as precisely as possible. To do this, I took a very simple and down to earth approach (very similar to the one used in zip2xml) :

```
while(there are more bytes) {
    read one byte
    print what it means
}
```

9 The complete syntax of JasminXT can be found in Appendix C.
10 http://jasmin.sourceforge.net

This basic principle ensures that there is no way to fool dejasmin, or to make it crash in any way, I just have to stick to the JVMS. This method is the main dejasmin disassembling engine, and can be triggered via the `--verbose` option. It is not the default output mode because it can sometimes produce a lot of garbage. For example, there is no need to print the content of the constant pool if the user wants to focus on disassembling the code of the class. This output mode becomes very valuable when the user wants to analyse invalid class files and to figure out exactly what is inside them.

The default output method is to print the content of the class file using the JasminXT syntax, which is more readable but less reliable. However, using a "pretty print" mode such as JasminXT is not straightforward and it requires some "high-level" processing. For example, Jasmin requires methods to be invoked like this :

```
invokevirtual java/io/PrintStream/println()V
```

But in the class file, the same instruction is represented as `invokevirtual x`, where x is an unsigned 16-bit number. This is interpreted as an offset in the constant pool, and the constant at this offset must be a CONSTANT_MethodRef. This constant points in turn to other constants, giving information on the name and descriptor of the method as well as the class defining it. Converting an offset to a high-level representation like `java/io/PrintStream/println()V` needs to keep the constant pool in memory and to process it correctly.

The conclusion is that the default disassembling engine, though reliable, was not adapted to such high-level processing. Moreover, there were already some disassemblers available supporting Jasmin output. So I integrated one of them, JasminVisitor, which is part of the ByteCode Engineering Library, and updated it so that it produces JasminXT output.

I finally implemented a last output mode for backwards compatibility, which can be triggered with the `--oldjasmin` option. The name of the option is self-descriptive enough : it is forwarded to JasminVisitor so that it does not use any of the features of JasminXT. As a result, the Jasmin file can then be assembled with older versions of Jasmin.

3.3.5 Requested Features and Future Development

After working with the toolkit to create the test case, I can now clearly underline one of its main limitations : Jasmin still does not allow a sufficient low-level control. I can not keep on updating Jasmin without totally breaking its initial design, because the problem is the Jasmin language itself. And if a new Jasmin language allowed a total low-level control, it would probably not be usable for other purposes, for which Jasmin has originally been designed. The solution would be to keep JasminXT and to implement a new disassembler/assembler pair : class2xml and xml2class.

Here are some of the requested features, which have not been implemented for the moment because they were not necessary for this study :

- add support for **external identifiers** to Jasmin, such as `java.io.PrintStream` instead of `Ljava/io/PrintStream;`

- add support for **constant pool indexes** in Jasmin, such as `invokevirtual #12`

- add support for **annotations** in Jasmin and dejasmin. This can be quite complex though.

- add support for **zip64** data structures to xml2zip and zip2xml

- compile the whole toolkit using standard **GNU libraries**. This would prevent people from having to use proprietary libraries if they don't want to.

IV. The Test Case

In the former section, we have seen that I decided that my strategy would be **black-box testing**. I also decided that I would focus on the **jar file handler** and the **class file format integrity checker**. For that purpose, I created jar and class file **low-level manipulation tools**.

In this section, I detail more precisely how I created the test case and how I ran the tests. I also present the test results.

4.1 Creation and Methodology

4.1.1 Limitations of Behavioral Testing

One of the difficulties of behavioral testing is to **know when a bug is found**, and if so, which error caused it. Observing the visible behavior of a program or its output does not always give a lot of interesting information about its actual internal state and this is a problem. Let's take the phone for instance. Here I only had 4 visible behaviors (and no output information, because there is no console on the phone) :

- "Authorisation failed" (AF) : error message indicating that the installation has failed

- nothing : installation completes successfully, but then the MIDlet won't run and fails with no error message

- almost nothing : the MIDlet starts but then exits immediately. This indicates a run-time error or a class loading error in a class used at run-time

- OK : the MIDlet works successfully

With such a restricted set of different reactions, it is sometimes hard to understand the exact reason for a given failure. For example, if I introduce a `breakpoint` instruction in the code array of a MIDlet, I don't expect it to work because `breakpoint` is an instruction reserved for debuggers and should not appear in class files. But knowing the error would be useful : is it a `ClassFormatError` (meaning that the class file couldn't be parsed successfully) or a `VerifyError` (meaning that the breakpoint instruction was parsed successfully and reached the verifier) ?

The other difficulty is to **know exactly what caused a bug** (assuming a bug has finally been identified as such). Here the problem is still the same : we can rely only on the behavior and output of the tested software. For example : if I run a class with a constant pool length of 0 (which is an invalid value) it will be rejected. But will it be rejected because the parser detects 0 as an invalid length, or because it assumed there was nothing in the constant pool and then failed to read the value of this class's name ?

4.1.2 Solutions

My solution to the first problem (knowing when a bug is found) was to run each invalid MIDlet on three different environments :

- the standard Java environment on a workstation (Sun's latest runtime environment, Java 1.5)

- a phone emulator, for which a console is available (Sun Java Wireless Toolkit 2.3 beta)

- a real phone (in this case, a Nokia 6680 with CLDC 1.1 and MIDP 2.0)

This solution is not perfect however, because we can only assume what is happening on the phone and it is still possible to miss some bugs.

My approach for the second problem (knowing what caused a bug) was to use a standard "hello world" MIDlet. Each time I had to study a particular error, I introduced this error in a new copy of the standard MIDlet. This way, I was more likely to observe the direct consequence of the error I introduced, since there are only a few different bytes between the working (standard) MIDlet and the invalid one. The standard MIDlet is made of :

- a main class, Test which extends `javax.microedition.MIDlet`. It has a `main()` method so that it can be run on J2SE. It also has some fields and a `computeSomething()` method with a try-catch-finally block.

- a class Test2, referenced in `Test.startApp()` and `Test.main()` so that it is loaded by both the standard runtime environment and the KVM

- an unused class Test3, to test the effect of class file format errors in unused classes

- an unused jpg file in the jar file, to test the effect of zip file format errors on unused files

- by default, the files are only stored in the archive (no compression used)

4.1.3 Creation of the Test Case

The idea behind this study is to create a test case as broad as possible. The goal is not to test for some random bugs but to cover the largest part of the specifications. Each time it was possible I tried to stick to a generic approach, expressed in the following guidelines :

- definition of **testing field X** : take a new copy of the standard MIDlet, and change field X. Field X will be considered tested after taking a few generic values (0, -1, 0xFFFF or 0xFFFFFFFF, expected value +/-1) and a few error-prone values (depending on the field).

  Example : the tested field is `attribute_length` in a `Code` attribute. The generic values are tested. An error-prone value tested is 0x7FFFFFFF (the maximum positive 32-bit number). Error-prone values are determined on a case-by-case basis. The success of the tests therefore depends on the knowledge of the tester and luck.

- **linearly test each field** defined in the specifications (both zip file format and class file format). For example, the local file header data structure defines the fields `local file header signature`, `version needed to extract`, `compatibility of file attribute info` and so on. All these fields have been tested one by one.

- **try to break the explicit rules** : if a specification states that a field must have a given value or must follow certain rules, test for values for which the rule is broken. This is a good way to ensure that a given check is correctly enforced if enforced at all.

  Example : the JVMS states that the "The value of the `code_length` item must be less than `65536`". So I checked for what happened if the code_length item was 65536. It is interesting to notice that some checks are not implemented intentionally.

- **spot switching points** : sometimes the tested software will have behavior B1 for value V1 of field X, and behavior B2 for value V2 of the same field X. A switching point is a value V such that "X = V" implies B1 and "X = V+1" implies B2.[11]

These guidelines ensure that the coverage is maximum for manual testing. However, we still infer that if something works for some chosen values, it works for all values. This is the big problem of manual testing, but at least the chosen values can be adapted to the problem.

Another approximation is that I usually infer that if a program has the same behavior for V1 and V2 and for "some" values between V1 and V2, it has the same behavior over the segment [V1, V2]. This approximation is particularly used for 32-bit values over the range [0x80000000, 0xFFFFFFFF] (negative numbers).

---

11 Sometimes it is impossible to spot switching points as there is a fuzzy limit between behavior B1 and B2. Some examples will be detailed in the test results

4.2 The Results

Most of the results I will present in this section are my own interpretation of what I have seen. I might be mistaken because many factors come into play and in some cases it took me a long time to figure out that my initial interpretation was wrong. The complete summary of the tests can be found in Appendix A (basically the tests raw data).

4.2.1 JAR File Tests

The jar file handling by virtual machines is almost not documented at all, that's why we can assume that each implementation has its own way to handle jar files. And the fact is that the zip file format is a very tricky one, so we can expect very different behaviors. The main problem in a zip file is that there is a lot of redundancy. Here is the structure of a zip file[12] :

```
[local file header 1]
[file data 1]
[data descriptor 1]
.
.
.
[local file header n]
[file data n]
[data descriptor n]
[archive decryption header] (EFS)
[archive extra data record] (EFS)
[central directory]
[zip64 end of central directory record]
[zip64 end of central directory locator]
[end of central directory record]
```

We can see that there is a succession of file headers and the corresponding data. After that, there is the central directory, structured like this :

```
[file header 1]
.
.
.
[file header n]
[digital signature]
```

So once again there are file headers. And there is almost no difference between the local file headers and the file headers found in the central directory. As a result, information like the crc32, file name, compressed and uncompressed size are present two times for each file in the archive. So how to parse a zip file ? It's up to the developer because there are many ways to parse them, either sequentially (like zip2xml), or by reading the central directory first and then accessing the data. And some programs read the compressed size in the local header, some in the central directory file header.

As a consequence, almost each implementation of a zip file reader is different from the others. And virtual machines are no exceptions. Here is what the tests revealed about the parsing (the description of the tests can be found in Appendix A) :

- the field `uncompressed_size` is ignored in the local header by all tested VMs (cf tests j001, j003, j004 and j005)

- the J2SE VM relies on the central directory offsets to parse the file (cf j012) and can therefore parse files considered incorrect by the others

- j101 to j103 show that the crc32 is ignored by all VMs in the local header. But the emulator doesn't ignore it in the file header (and therefore is the only one to fail if it is incorrect)

---

12 Taken from *The .ZIP File Format Specification* [S5]

- if there are duplicate entries (two files in the archive with the same name), the J2SE VM reads the second, whereas the emulator and the phone read the first. (cf j301 and j302)

- it is possible to crash the emulator very easily either by setting a large `uncompressed_size` or by using invalid values for the `file_name_length` field (cf j007, j200 and j201). The phone is more stable. The crashes can be very different : a crash report from zayit.exe, a MSVC++ assertion failed error...

Except for the crashes, we can't really say these are bugs. Actually the *.ZIP File Specification* [S5] is just that : a file specification. It shows how zip files are made, and doesn't give any instruction on how to parse them. This is why there are so many different behaviors. In comparison, the JVMS specifies both the class file format and the behavior of a VM.

These are just some examples of the difference between the different tested VMs. We can also expect that other tools will have different behaviors, such as disassemblers or debuggers. I think this can be used successfully by malware authors. For example, if a given mobile malware author wants to delay the analysis of his program, he just has to find errors tolerated by the phone but not by other VMs (and more particularly by disassemblers and debuggers or other analysis tools).

Example : we have seen that when there are duplicate entries, the phone and the emulator read from the first one. So if we set errors in the second one, they are only going to affect the J2SE VM (cf j301). Now, if we add an error in the first one tolerated by the phone but not by the emulator (such as setting the `compressed_size` to 0xFFFF), we obtain **a MIDlet which runs only on the phone** (cf j304). We can go further by adding an unused file with invalid attributes (`compressed_size = 0`, `uncompressed_size = 0`, `file_name_length = 0`) : this is going to puzzle zip utilities like WinZip, which is unable to open such an archive.

So these different behaviors are not directly related to security problems, but they can be used successfully by a malware author. The solution would be to document and implement a standard way to parse jar files. I want to emphasize that there is virtually no limit to the number of errors which can be added to an archive, as long as they are tolerated by the target VM. The problem for the analyst is that even analysis tools implement a particular parsing method, which make them vulnerable to certain errors. So if they don't parse zip files like the target VM it can be very hard to extract the data and finally analyse it.

To underline the fact that different VMs have really different behaviors, I have summed up some of the jar file test results in graphs 1 and 2 in the following pages. They show the behaviors of the phone, the emulator and the J2SE VM for whole ranges of values of fields `compressed_size` and `uncompressed_size` for Test.class in the central directory file header.
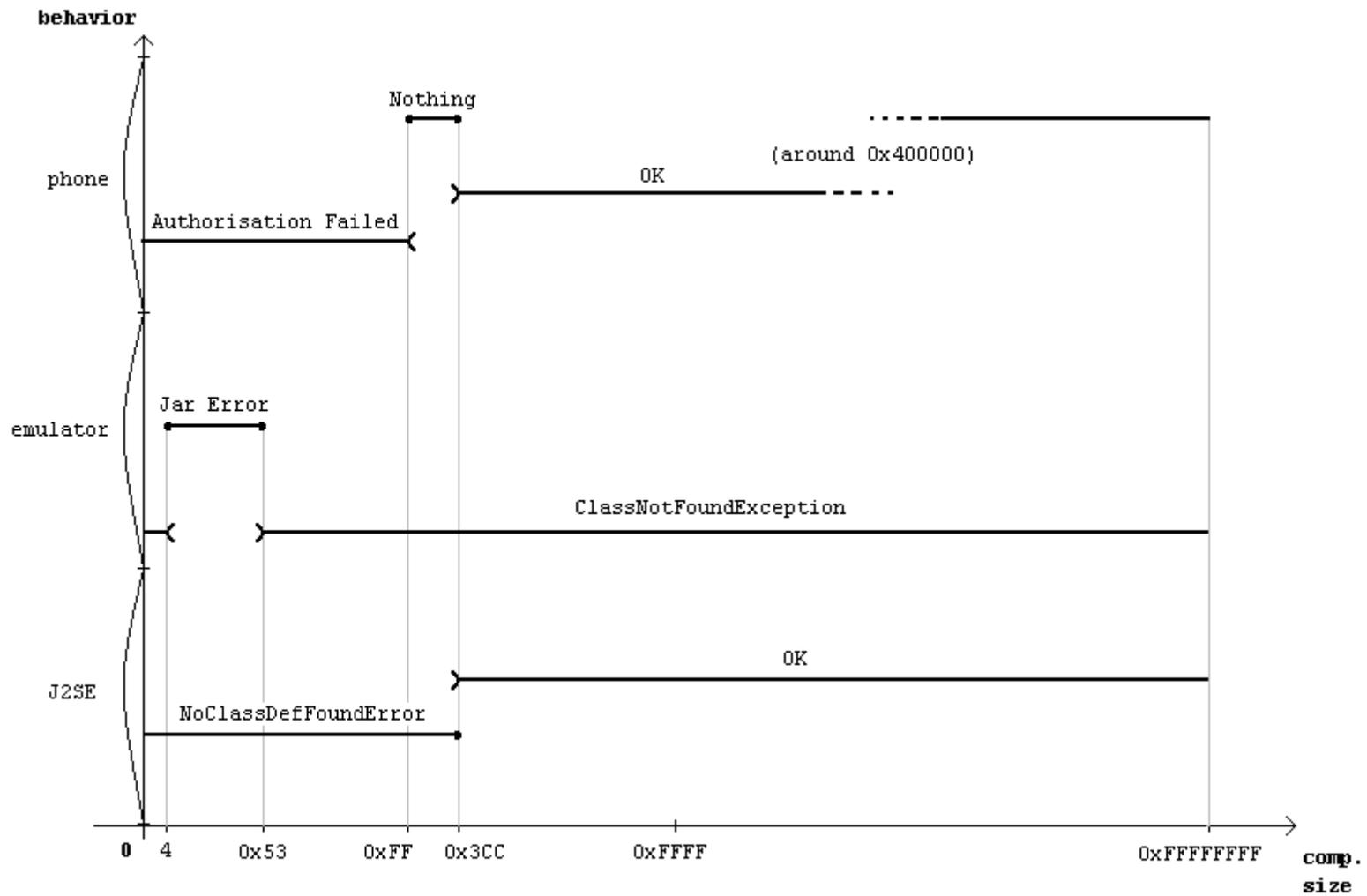
Graph 1 : compressed_size in file header for Test.class
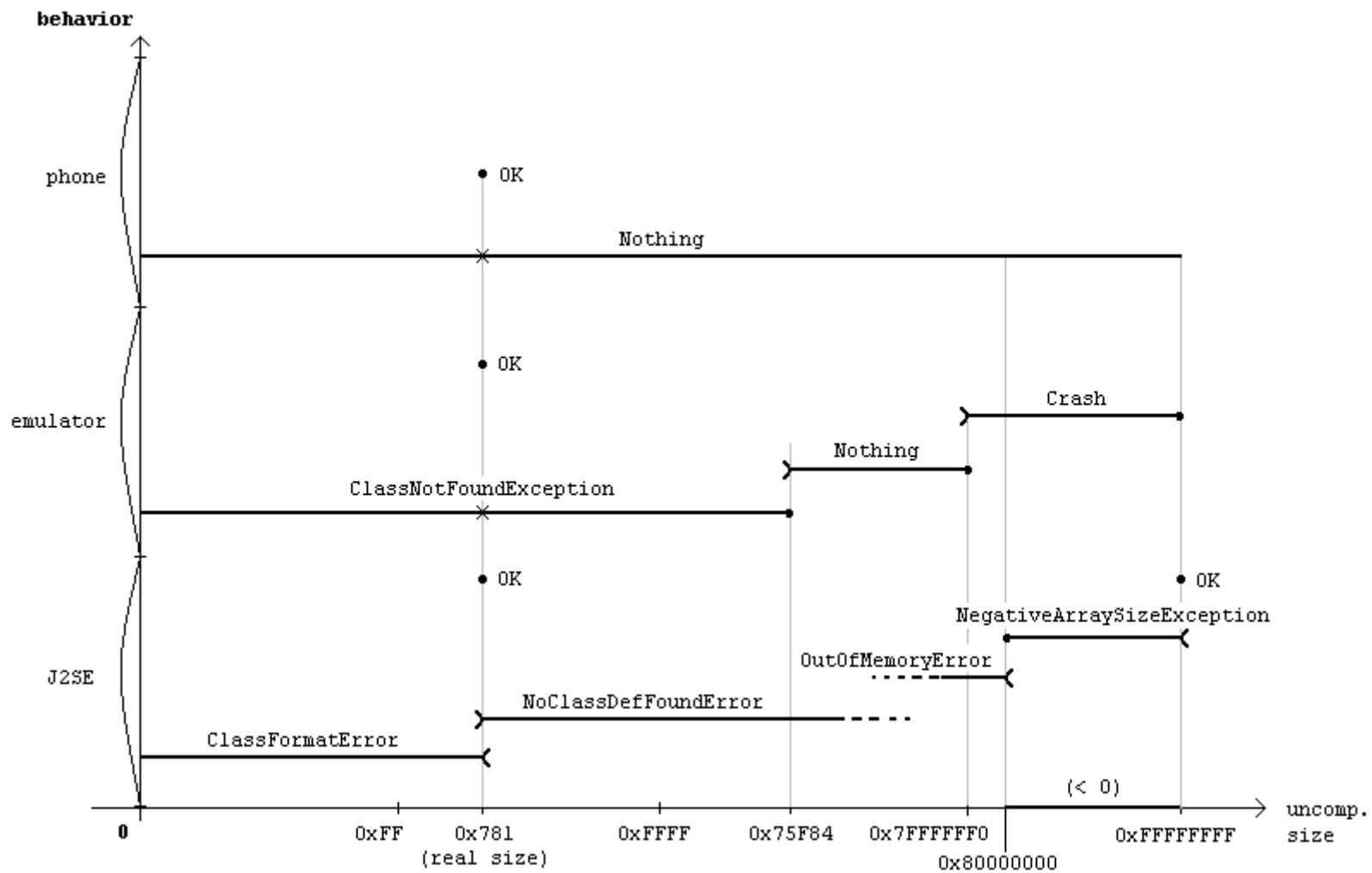
Graph 2 : uncompressed_size in file header for Test.class

4.2.2 Class File Tests

The class file format as defined in JVMS [S1] is simpler to parse, and the observed behaviors are more consistent because a class file has to be parsed sequentially. It can't be accessed randomly because the meaning of data at a certain offset depends on all the data that was before it, due to variable-length fields.

In many cases, JVMS also specifies the exception a VM has to return if some constraints are not respected. It doesn't mean that the behaviors of the different VMs will be exactly the same but that it will be easier to understand the errors and to generate a structured test case. Here is what the tests revealed (the full description of each test is in Appendix A) :

J2SE problems

• c121 and c190 show that bad identifiers are accepted in J2SE for class, field and method names. Examples of bad identifiers include empty strings, numbers or reserved keywords. However this seems to be intentional because an exception is thrown if the command-line option -Xfuture is used.

• test c193 throws a strange error ("invalid Code attribute name index 46848")[13] when using an empty string identifier and the bytecode major version is 45. I didn't expect the behavior of J2SE to depend on the bytecode version at all.

• a bug has been found in J2SE and reported to Sun.[14] The `attribute_length` item of an attribute is a 4-byte unsigned value. The problem is that the VM reads it as a signed value (cf c170). Another problem is related to this : when the VM reads an unknown attribute (and ignores it, as specified in JVMS) it reads the length and then skips the number of bytes read. This is probably done with a direct call to `fseek()`, therefore when using negative values the file pointer goes backward (cf c174). This is quite serious because it can be exploited to make the VM parse the class file non-sequentially. It can most notably be used by malware authors to "hide" methods in unexpected places, such as attributes.

Emulator problems

• the emulator handles the main class and the other classes differently. When an error is introduced in another class used by the main class, the emulator crashes instead of returning an exception (cf c031 and c070 to c094).

• it turned out that the emulator has a different behavior if it is launched from the command-line or using the KToolbar. In tests c121, c123 and c150 an exception is thrown if the emulator is run from the command-line whereas the MIDlet runs fine on the phone. It also runs fine on the emulator if it is launched from the KToolbar.

• miscellaneous zip file format errors crash the emulator (j007, j201)

Phone problems

• errors in unused classes provoke an Authorisation Failed error. It means that the installer parses every .class file in an archive (cf c030). This behavior was unexpected and I could find no documentation about it.

• the installer sometimes freezes. Choosing "Cancel" and then "Yes" produces nothing, and closing it from the task manager doesn't work. The only solution was to kill the process directly using a third-party utility called TaskSpy, or to reboot the phone.

• the reason for the former freeze is not clear, however it is possible to create MIDlets which will make the next install freeze (cf c035, c061 and c065). The cause of the problem seems to be related with invalid values of this_class, but I couldn't spot the problem exactly.

---

13 46848 corresponds to 0xB700. 0xB700 is often found in the code array of methods, because 0xB7 is the opcode of the `invokespecial` instruction. However it is not clear why the VM would suddenly fall in the middle of a code array and try to read an index there.

14 The bug report to Sun can be found in Appendix B.

- the two previous problems show that the behavior of a particular installation may depend on the previous installations. A reboot can also change the result of a test. Moreover, the same behavior can be produced by different errors and the same error in different files can produce different behaviors. As a consequence, it is almost impossible to report a bug.

- c046 and c047 show that the installation will succeed if `this_class` points to a `CONSTANT_Class`. Otherwise the behavior is erratic : either AF, the installer freezes, the next installation freezes, or nothing.

- c004 shows that the same error in the same file can produce different behaviors. In this case we have cyclic errors : the first time it is installed, the result is an AF. The second time, it is an installer freeze. After a reboot the installation succeeds, and then we obtain again an AF, and so on.

- c220 and c230 run correctly on the phone but result in an `OutOfMemoryError` on the emulator. This error is not even reported properly, since nothing seems to happen when the emulator is run from the command-line.

4.2.3 Test Conclusion

No security flaw has been found on the tested VMs, but a lot of undocumented or unexpected behaviors have been found. Some of them can be used by malware authors to delay the analysis of their programs, including errors in jar files and in class files. The errors in class files are both parsing errors and memory management errors.

One serious bug in the class file parsing of the J2SE VM has been found and reported to Sun. Some bugs have clearly been found in the application installer on the phone but it is almost impossible to make an accurate bug report. The emulator was very unstable and many different errors made it crash, but Sun's bug report interface doesn't include the Wireless Toolkit.

No bugs have been found in the KVM. First of all it is very stable and no crash has been experienced on the phone, but the lack of error messages and output makes it difficult to identify bugs. The intent is probably that developers should work on the emulator and then release their programs on the phone, but this study clearly shows that for subtle errors the phone and the emulator can have very different behaviors.

In retrospect the test case is poorly organised and is not conveniently reusable. It is not as generic as planned, because the standard MIDlet has evolved during the tests. The reason for this is that most of the problems have been found in what I expected to be "reference" implementations, and I had to adapt to that. Moreover, I still don't understand many of the results (such as the installer erratic behavior, or the fact that the emulator has a different behavior when it is launched from the command-line). When I originally designed the test case I also assumed that the phone and the emulator would have closer behaviors.

This study can therefore be seen as a preliminary study, and knowing these early results it is possible to build a better test case, leading to a better understanding of the results.

Conclusion

We have seen that Java was designed from the ground up with security in mind, and this is particularly true for J2ME. Built-in security features include the ability to run mobile code from untrusted sources with a fair level of confidence that it won't harm the device or perform any malicious operations. Other security features have been included at the application level and it would be interesting to investigate them. For example, some high-level security breaches could be found in the optional FileConnection and BlueTooth APIs. They might be used successfully by viruses or any other type of malware.

The approach taken here was to test as completely as possible the first layers of the Java security models, because any breach in a low security layer can have repercussions on all the layers above. The original constraints were that there were no tools suitable for this task, and that the project had to produce short-term results.

I have designed general tools to be able to manipulate jar and class files at a very low level. These tools have been very useful to generate the test case for J2ME low-level security, and can be used in many other projects. Even if several problems have been highlighted in J2SE, in the emulator and in the application installer on the phone, no security flaw has been found in the KVM.

Black-box testing turned out to be a good choice for revealing bugs very quickly. However, there are two main problems with this approach :

- some bugs might have been missed due to the lack of output on the phone
- it is hard to fully automate the process without a more general testing framework.

The solution to the first problem would be to switch to white-box testing by testing the reference implementation of the KVM on a workstation. But some problems related to the phone itself (such as the application installer) would have been missed.

The solution to the second problem would be to work with a formal specification and to produce a testing software which would test a given implementation against the specification. The software to be tested should be able to report its internal state accurately to the testing framework in order to minimize human intervention in the process.

A lot of research has been conducted by Sun and by independent researchers on Java security. Some people have developed a formal model of the bytecode verifier to prove that it was or was not type safe.[15] The goal of this kind of research is to develop a model and prove it is safe. I worked on the other end of the security model, by trying to check that a given implementation was conforming to the specification. It would be interesting to connect these works in order to produce a provably safe specification, and to ensure that an implementation is consistent with the specification.

---

15 Examples : *Formal Specification and Verification of a JVM and its Bytecode Verifier* [WP5] and *Toward a Provably Correct Implementation of the JVM Bytecode Verifier* [WP9] but a lot more has been published about this specific topic

Bibliography

Books

[B1] OAKS Scott,
Java Security, O'Reilly
618 pages, May 1998

[B2] BEIZER Boris,
Black-Box Testing, John Wiley and Sons
294 pages, 1995

[B3] ALLIN Jonathan,
Wireless Java for Symbian Devices, John Wiley and Sons
489 pages, 2001

[B4] Digia Inc,
Programming for the Series 60 Platform and Symbian OS, John Wiley and Sons
521 pages, 2003

[B5] MCLAUGHLIN Brett,
Java and XML Second Edition, O'Reilly
550 pages, August 2001

Specifications

[S1] LINDHOLM Tim and YELLIN Frank,
The Java Virtual Machine Specification (Java Series) Second Edition, Addison-Wesley
1999

[S2] Sun Microsystems Inc,
Connected, Limited Device Configuration 1.1 (JSR-139)
http://jcp.org/jsr/detail/139.jsp
60 pages, March 2003

[S3] BRACHA Gilad, LINDHOLM Tim, TAO Wei and YELLIN Frank (Sun Microsystems),
CLDC Byte Code Typechecker Specification
68 pages, January 2003

[S4] JSR 118 Expert Group,
Mobile Information Device Profile for J2ME, Version 2.0
560 pages, November 2002

[S5] PKWARE
.ZIP File Format Specification 6.2.1
1989, last revised April 2005

White Papers

[WP1] GAL Andreas, PROBST Christian, FRANZ Michael
A Denial of Service Attack on the Java Bytecode Verifier
13 pages, October 2003

[WP2] Sun Microsystems Inc,
J2ME Building Blocks For Mobile Devices, White Paper on KVM and the CLDC
36 pages, May 2000

[WP3] Sun MicroSystems Inc,
KVM Porting Guide
80 pages, May 2000

[WP4] KLEIN Gerwin,
Verified Java Bytecode Verification
190 pages, November 2002

[WP5] LIU Hanbing,
Formal Specification and Verification of a JVM and its Bytecode Verifier
36 pages, December 2004

[WP6] DE JODE Martin,
Introduction to J2ME (1st Chapter of *Programming Java 2 Micro Edition on Symbian OS*)
20 pages, June 2004

[WP7] REYNAUD-PLANTEY Daniel,
New threats of Java viruses, Journal in Computer Virology
12 pages, October 2005

[WP8] REYNAUD-PLANTEY Daniel,
Reverse Engineering and Java Viral Analysis, in Proceedings of the Virus Bulletin 2005
International Conference
6 pages, October 2005

[WP9] COGLIO Alessandro, GOLDBERG Allen, QIAN Zhenyu
Toward a Provably Correct Implementation of the JVM Bytecode Verifier
18 pages

Electronic Resources

[E1] *The Java(TM) Verified Program Guide, Version 1.0*
http://www.javaverified.com

[E2] *Unified Testing Criteria for Java(TM) Technology-Based Applications for Mobile Devices*,
Version 2.0
http://www.javaverified.com

[E3] BRNA Paul,
Prolog Programming, A First Course
197 pages, March 2001

[E4] GOWDIAK Adam,
J2ME Security Vulnerabilities, slides of the HITB04 presentation
October 2004

[E5] PETTICHORD Brett,
Five Ways to Think about Black-Box Testing
http://www.stickyminds.com
May 2001

Appendix A : The Complete Test Case

Naming

This appendix is the complete summary of the tests run. Each test MIDlet has been saved in a folder named after the category of the test. Here are the different categories :

- j : jar file format test
- c : class file format test
- v : verifier test (very few of them)

Abbreviations

Each test name is the letter of the category immediately followed by 3 digits. For example, c056 indicates a class file format test. The number has no special meaning, but usually the last digit indicates the tested field (for example c055 and c056 are different tests for the same field, whereas c060 is a test for a different field).

The behavior of the software is noted in the last 3 columns named "Java application", "Emulator" and "Phone". Namely, they correspond to the JDK 1.5 java command (on Windows XP), Sun Wireless Toolkit 2.3 beta emulator, and a Nokia 6680 phone. Here is a list of abbreviations used for the different observed behaviors :

- **CFE** : ClassFormatError
- **NCDFE** : NoClassDefFoundError
- **VE** : VerifyError
- **CNFE** : ClassNotFoundException (very common on the emulator)
- **-** : "nothing". On the phone, means that nothing happens when the MIDlet is launched (this is the normal behavior when an error is encountered). It sometimes happens on the emulator, but in this case we expect an error message instead.
- **~** : "almost nothing", happens on the phone when an error is encountered at run-time (the MIDlet runs and then suddenly exits)
- **AF** : corresponds to the "Authorisation Failed" error on the phone, during installation
- **OK** : the MIDlet runs correctly.

The following abbreviations are used in the descriptions :

- **comp** : stands for compression_method
- **LOC HDR** stands for local file header
- **FILE HDR** stands for file header (in the central directory)

Unless explicit mention, the modifications are made for Test.class. For example, j001's description reads "no compression, uncomp. size = 0 in LOC HDR", meaning that the uncompressed_size field is set to 0 in Test.class's local header.

If there are multiple descriptions for a single test name, the valid description for the actual test folder is the one with the '->' symbol.

Name	Description	Comment	Java application	Emulator	Phone
j001	no compression, uncomp. size = 0 in LOC HDR		OK	OK	OK
j003	comp = deflate, uncomp. size = 0 in LOC HDR		OK	OK	OK
j004	+--> same with uncomp. size = 0 in LOC and FILE HDR		CFE	CNFE	-
j005	+--> same with uncomp. size = 0xFFFF in LOC HDR		OK	OK	OK
j006	+--> same with uncomp. size = 0xFFFF in FILE HDR		NCDFE	CNFE	-
j007	+--> same with uncomp. size = 0xFFFFFFFF = -1 in FILE HDR	works in J2SE	OK	zayit.exe crashes	-
j008	+--> same with uncomp. size = 0x0FFFFFFF in FILE HDR		OutOfMemoryError	-	-

Name	Description	Comment	Java application	Emulator	Phone
j009	comp = deflate, comp. size = 0 in FILE HDR		NCDFE	CNFE	AF
j010	+--> same with comp. size = -1 in FILE HDR	works in J2SE	OK	CNFE	-
j011	+--> same with comp. size = -1 in LOC HDR		OK	OK	OK
j012	+--> same with comp. size = -1 in LOC and FILE HDR	J2SE uses central dir offsets	OK	CNFE	-
j013	+--> same with comp. size = 0xFFFF in FILE HDR	phone and emulator differ	OK	CNFE	OK
j014	+--> same with comp. size = 0x0FFFFFFF in FILE HDR		OK	CNFE	-
j015	+--> same with comp. size = 0xFF in FILE HDR		NCDFE	CNFE	AF
j017	+--> same with comp. size = 0xF in FILE HDR		NCDFE	Jar Error	AF
(...j033)	j017 to j033 are different values for comp. size, summed up in Graph 1 (cf 4.2.1)				

Name	Description	Comment	Java application	Emulator	Phone
j100	crc_32 = 0 in LOC HDR		OK	OK	OK
j101	crc_32 = 0 in LOC and FILE HDR	phone and emulator differ	OK	CNFE	OK
j102	crc_32 = 0 in FILE HDR		OK	CNFE	OK
j103	crc_32 = -1 in FILE HDR		OK	CNFE	OK

(different values have been tested for version_needed_to_extract, but it had no effect whatsoever)

Name	Description	Comment	Java application	Emulator	Phone
j200	comp = deflate, file_name_length = 0 in LOC HDR		NCDFE	zayit.exe crashes	-
j201	+--> same with file_name_length = 1 in LOC HDR		NCDFE	MSVC++ assertion failed	-
-	many of the other values tested crashed the emulator				

j300	duplicate Test.class entries, along with compressed data		OK	OK	OK
j301	+--> same with uncomp. size = 0 in 2nd FILE HDR		CFE	OK	OK
j302	+--> same with uncomp. size = 0 in 1st FILE HDR		OK	CNF	-
j303	+--> same with uncomp. size = 0 in both FILE HDR		CFE	CNF	-
j304	+--> j301 + comp_size = 0xFFFF in 1st FILE HDR	runs only on the phone	CFE	CNF	OK

j310	comp = deflate, uncomp. size = 0 in labo.jpg FILE HDR		OK	OK	OK
j311	+--> same with comp. size = 0 and file_name_length = 0 in labo.jpg FILE HDR	winzip fails to open it	OK	OK	OK

Name	Description	Comment	Java application	Emulator	Phone
c000	Test is an interface rather than a class		CFE (field not public)	CFE	-
c001	+--> same thing with public fields		CFE	CFE	-
c002	+--> c001 + abstract methods		CFE	CFE	-
c003	this_class points to a CONSTANT_String		CFE	CNFE	-
c004	this_class = 0	variable behavior on the phone	CFE	CNFE	AF then install freezes then nothing
c005	this_class = 0xFFFF		CFE	CNFE	AF or freeze
c010	super_class = 0, 0xFFFF or this_class		CFE	CNFE	-
c020	implement a class		IncompatibleClassChangeError	IncompatibleClassChangeError	AF
c021	implement index 0		CFE	CNFE	AF
c022	implement the same interface twice		CFE	CNFE	AF
c023	implement Test		ClassCircularityError	CNFE	AF
c030	constant_pool_count = 0 in Test2 (unused)		OK	OK	AF
c031	constant_pool_count = 0 in Test2 (referenced in startApp())	emulator crash	OK	silent crash	AF
c032	constant_pool_count = 0 in Test2 (ref. in main())		CFE	OK	AF
c033	constant_pool_count = 0xFFFF in Test2 (ref. in main())		CFE	OK	AF
c034	constant_pool_count = 0xFFFF in Test2 (ref. in startApp())		OK	silent crash	AF
c035	constant_pool_count += 1 in Test2 (ref. in startApp())	makes the next install freeze	OK	silent crash	AF
c036	super_class = this_class in Test2 (ref. in Test())		OK	-	-

Name	Description	Comment	Java application	Emulator	Phone
c037	Test2 self-implementing interface (ref. in startApp())		OK	silent crash	AF
c038	interface_count = 0xFFFF in Test2 (ref. in startApp())		OK	silent crash	AF
c039	c036 with super_class = 0xFFFF		OK	-	~
c040	+--> same thing with super_class = 0		OK	-	~
c041	c036, but Test2 is unused		OK	OK	OK
c042	constant_pool_count += 1 in Test2 (unused)		OK	OK	AF

Name	Description	Comment	Java application	Emulator	Phone
c043	this_class = 0xFFFF in unused Test2		OK	OK	AF
c044	this_class = super_class in unused Test2		OK	OK	AF
c045	this_class = 0 in unused Test2		OK	OK	AF
c046	this_class = 1 in unused Test2 (CONSTANT_MethodRef)		OK	OK	AF / freeze
	this_class = 4 in unused Test2 (CONSTANT_Utf8)		OK	OK	AF / freeze
	this_class = 5 in unused Test2 (CONSTANT_Utf8)		OK	OK	AF / freeze
	this_class = 6 in unused Test2 (CONSTANT_Utf8)		OK	OK	AF / freeze
	this_class = 7 in unused Test2 (CONSTANT_NameAndType)		OK	OK	AF / freeze
	this_class = 10 in unused Test2 (cp_count)		OK	OK	AF / freeze
->	this_class = 11 in unused Test2 (cp_count + 1)		OK	OK	AF / freeze

Name	Description	Comment	Java application	Emulator	Phone
c047 ->	this_class = 0 in Test	same as c004	CFE	CNFE	- / AF / freeze
	this_class = 1 in Test (CONSTANT_MethodRef)		CFE	CNFE	AF / freeze
	this_class = 2 in Test (CONSTANT_Class)		CFE	CNFE	-
	this_class = 3 in Test (CONSTANT_MethodRef)		CFE	CNFE	-
	this_class = 4 in Test (CONSTANT_Class)		CFE	CNFE	-
	this_class = 5 in Test (CONSTANT_String)		CFE	CNFE	-
	this_class = 105 in Test (cp_count)		CFE	CNFE	AF

Name	Description	Comment	Java application	Emulator	Phone
c050	Test2 extends Test3 and vice versa		ClassCircularityError	silent crash	~

Name	Description	Comment	Java application	Emulator	Phone
c060	interface_count = 0xFFFF in Test3		OK	OK	OK
c061	this_class = 0 in Test3	makes the next install freeze	OK	OK	AF
c062	interface_count += 1 in Test3		OK	OK	OK
c063	interface_count += 1 in Test2		CFE	silent crash	-
c064	interface_count += 1 in Test		CFE	CNFE	-
c065	this_class = 0 in Test2	makes the next install freeze	CFE	silent crash	AF

Name	Description	Comment	Java application	Emulator	Phone
c070	fields_count += 1 in Test	(main class)	CFE	CNFE	-
c071	fields_count += 1 in Test2	(used)	CFE	silent crash	~
c072	fields_count += 1 in Test3	(unused)	OK	OK	OK

Name	Description	Comment	Java application	Emulator	Phone
c080	0x00 char in String in Test	(main class)	CFE	CNFE	-
c081	0x00 char in String in Test	(used)	CFE	silent crash	~
c082	0x00 char in String in Test	(unused)	OK	OK	OK

Name	Description	Comment	Java application	Emulator	Phone
c090	MethodRef(0,0) in Test	(main class)	CFE	CNFE	-
c091	MethodRef(0,0) in Test2	(used)	CFE	silent crash	~
c092	MethodRef(0,0) in Test3	(unused)	OK	OK	OK
c093	MethodRef(0xFFFF, 0xFFFF) in Test		CFE	CNFE	-
c094	MethodRef(0xFFFF, 0xFFFF) in Test2		CFE	silent crash	~

Name	Description	Comment	Java application	Emulator	Phone
c100	super_class = 0 in Test3		OK	OK	OK
	super_class = 1 in Test3		OK	OK	OK
->	super_class = this_class in Test3		OK	OK	OK
	super_class = 0xFFFF in Test3		OK	OK	OK

Name	Description	Comment	Java application	Emulator	Phone
c110	this_class = "Test!" in Test2		NCDFE (wrong name)	silent crash	~
c111	Test2 in a file called TestA.class		NCDFE	silent crash	~
c112	this_class = x+1, x being the index of a CONSTANT_Long in Test		CFE	CNFE	-

Name	Description	Comment	Java application	Emulator	Phone
c120	field[0].name_index = 0 in Test		CFE	CNFE	-
c121 ->	field[0].name_index = 32 in Test ("I")	phone and emulator differ	OK	CNFE from command OK from KToolbar	OK
	field[0].name_index = 36 in Test ("<init>")		OK	CNFE	-
	field[0].name_index = 55 in Test ("All your base")		OK	CNFE	-
	field[0].name_index = 76 in Test ("Exit")		OK	CNFE	OK
	field[0].name_index = 0xFFFF in Test		CFE	CNFE	-
c122	field[0].name_index = 3 in Test (middle of a long)		CFE	CNFE	-

Name	Description	Comment	Java application	Emulator	Phone
c123	field[0].name_index = 0 in Test2		CFE	silent crash	~
	field[0].name_index = 3 in Test2 (middle of a double)		CFE	silent crash	~
	field[0].name_index = 4 in Test2 (CONSTANT_FieldRef)		CFE	silent crash	~
->	**field[0].name_index = 12 in Test2 ("ConstantValue")**	not the same from command line and from GUI	OK	silent crash from command OK from KToolbar	OK
	field[0].name_index = 34 in Test2 (cp_count)		CFE	silent crash	~

Name	Description	Comment	Java application	Emulator	Phone
c130	field[0].descriptor_index = 0 in Test		CFE	CNFE	-
	field[0].descriptor_index = 1 in Test (CONSTANT_MethodRef)		CFE	CNFE	-
	field[0].descriptor_index = 33 in Test ("ConstantValue")		CFE	CNFE	-
	field[0].descriptor_index = 38 in Test ("Ljavax/...")		CFE	CNFE	-
	field[0].descriptor_index = 35 in Test ("I")		CFE	CNFE	-
->	field[0].descriptor_index = 0xFFFF in Test		CFE	CNFE	-

Name	Description	Comment	Java application	Emulator	Phone
c140	field[0].attribute_count = 0 in Test		CFE	CNFE	-
	field[0].attribute_count += 1 in Test		CFE	CNFE	-
->	field[0].attribute_count = 0xFFFF in Test		CFE	CNFE	-

Name	Description	Comment	Java application	Emulator	Phone
c150	field[0].attribute[0].name_index = 0 in Test		CFE	CNFE	-
->	field[0].attribute[0].name_index = 42 in Test ("()V")	not the same from command line and from GUI	OK	CNFE from command OK from KToolbar	OK
	field[0].attribute[0].name_index = 43 in Test ("Code")		OK	CNFE from command OK from KToolbar	OK
	field[0].attribute[0].name_index = 0xFFFF in Test		CFE	CNFE	-

Name	Description	Comment	Java application	Emulator	Phone
c160	field[0].attribute[0].attribute_length = 0 in Test (ConstantValue)		CFE	CNFE	-
	field[0].attribute[0].attribute_length = 3 in Test		CFE	CNFE	-
	field[0].attribute[0].attribute_length = 0xFF in Test		CFE	CNFE	-
	field[0].attribute[0].attribute_length = 0xFFFF in Test		CFE	CNFE	-
->	field[0].attribute[0].attribute_length = 0xFFFFFFFF = -1 in Test		CFE (invalid length -1)	CNFE	-

Name	Description	Comment	Java application	Emulator	Phone
c170	field[0].attribute[0].attribute_length = 0 in Test ("()V")		CFE	CNFE	-
	field[0].attribute[0].attribute_length = 0 in Test (really 0 bytes long)		ZipException	Invalid Jar	AF
	field[0].attribute[0].attribute_length = 3 in Test		CFE	CNFE	-
	field[0].attribute[0].attribute_length = 0xFF in Test		CFE	CNFE	-
	field[0].attribute[0].attribute_length = 0xFFFF in Test		CFE (truncated class file)	CNFE	-
->	field[0].attribute[0].attribute_length = -1 in Test	**bug in J2SE**	CFE (but not truncated)	CNFE	-
	field[0].attribute[0].attribute_length = -2 in Test		CFE	CNFE	-
c171	field[0].attribute[0].attribute_length = -14 in Test		CFE (repetitive field name/signature)	CNFE	-

Name	Description	Comment	Java application	Emulator	Phone
c180	public private startApp() in Test		CFE	CFE	-
	public protected startApp() in Test		CFE	CFE	-
	private protected startApp() in Test		CFE	CFE	-
	static init() in Test		CFE	CFE	-
	abstract init() in Test		CFE	CFE	-
	native init() in Test		CFE	CFE	-
->	final init() in Test		CFE	CFE	-

Name	Description	Comment	Java application	Emulator	Phone
c190	name_index = 0 for computeSomething()		CFE	CNFE	-
	name_index = 107 for computeSomething() ("println")		NoSuchMethodError	CNFE	~
	name_index = 53 for computeSomething() (" omputeSomething")		OK	CNFE	-
	name_index = 53 for computeSomething() (".omputeSomething")		OK	CNFE	-
	name_index = 53 for computeSomething() ("6omputeSomething")		OK	CNFE	-
	name_index = 53 for computeSomething() ("(omputeSomething")		OK	CNFE	-
	name_index = 53 for computeSomething() ("c(mputeSomething")		OK	CNFE	-
	name_index = 1 for computeSomething() (MethodRef)		CFE	CNFE	-
->	name_index = 73 for computeSomething() (cpcount)		CFE	CNFE	-
c191	name_index = 53 for computeSomething() ("")		OK	CFE	-

Name	Description	Comment	Java application	Emulator	Phone
c192	Test2 with empty name ("")		OK	CFE	Unable to install : invalid file
c193	field HELLO_STRING with empty name	different results depending on bytecode version	CFE (invalid code attribute name) if version = 45 OK otherwise	CFE	-
c194	field iamverylong with empty name		OK	CFE	-
c195	"computeSomething" replaced with "", bytecode_version = 45.0		CFE	CFE	-
c196	"computeSomething" replaced with "", bytecode_version >= 46.0		OK	CFE	-

Name	Description	Comment	Java application	Emulator	Phone
c200	descriptor_index = 0 for computeSomething()		CFE	CNFE	-
	descriptor_index = 1 (CONSTANT_MethodRef)		CFE	CNFE	-
	descriptor_index = 24 (CONSTANT_String)		CFE	CNFE	-
->	descriptor_index = 0xFFFF		CFE	CNFE	-

Name	Description	Comment	Java application	Emulator	Phone
c210	computeSomething() without a Code attribute		CFE : no code	CNFE	-
c211	computeSomething() with 2 Code attributes		CFE : multiple code	CFE : duplicate code	-

Name	Description	Comment	Java application	Emulator	Phone
c220	max_stack = 0 in computeSomething()		VE : stack too large	VE	-
	max_stack = 0xFFFF in computeSomething()		OK	-	-
	max_stack = 0xFF in computeSomething()		OK	OK	OK
	max_stack = 256 in computeSomething()		OK	OK	OK
->	**max_stack = 512 in computeSomething()**	phone and emulator differ	OK	- from command OutOfMemoryError from KToolbar	OK

Name	Description	Comment	Java application	Emulator	Phone
c230	max_locals = 0 in computeSomething()		VE	CFE : bad stack map	-
	max_locals = 256 in computeSomething()		OK	OK	OK
	max_locals = 512 in computeSomething()	phone and emulator differ	OK	-	OK
->	max_locals = 0xFFFF in computeSomething()		StackOverflowError	-	-

Name	Description	Comment	Java application	Emulator	Phone
c240	code_length = 0xFFFFFFFF in computeSomething()		CFE	-	-
->	code_length = 0 in computeSomething()		CFE	CFE	-
	code_length = 0 in Test.recurse() (unused), physically 0 too		OK	OK	OK
c241	code_length = 0 in <clinit>() in Test2		CFE	VE	~

Name	Description	Comment	Java application	Emulator	Phone
c250	start_pc = end_pc in computeSomething()		CFE	CFE	-
	start_pc > end_pc in computeSomething()		CFE	CFE	-
	end_pc > code_length in computeSomething()		VE	VE	-
->	start_pc in the middle of an instruction in computeSomething()		CFE	VE	-

Name	Description	Comment	Java application	Emulator	Phone
c260	handler_pc inside [start_pc, end_pc) in computeSomething()		VE	VE	-
->	handler_pc = 0xFFFF		CFE	CNF	-
c270	catch_type = 0 in computeSomething()		VE	CNF	-
	catch_type = 23 in computeSomething() (MIDlet)		VE	CNF	-
->	catch_type = 0xFFFF in computeSomething()		CFE	CNF	-
v000	Test2 is an interface extending Test		CFE	CFE (bad super class)	~
v010	Test3 is an interface implemented by Test2 Test tries to fetch a field in Test2 defined in Test3		OK	OK	OK
v020	invokespecial computeSomething() in startApp()		OK	IncompatibleClassChangeError	~
v021	invokespecial in startApp() to invoke in Test2 a method defined in Test3 (Test2 extends Test3)		VE	VE	-
v030	Test3 defines `protected void doSomething()` Test2 extends Test3 and defines `public void doSomething()` Test calls Test2.doSomething()		OK	OK	OK
v031	same thing but Test3.doSomething() now returns "Test3" and Test2.doSomething() now returns "Test2"		"Test2"	"Test2"	OK
v032	same thing with Test2 and Test3 in different packages		"Test3"	"Test3"	AF
v040	goto 666 with StackMap for offset 666	Gowdiak's bug 1	VE (bad target)	VE (bad stack map)	-
v050	goto_w 0x10000	Gowdiak's bug 2	VE (bad target)	VE (bad stack map)	-

Appendix B : J2SE Bug Report

(as appearing on Sun's bug database : http://bugs.sun.com)

Bug ID: 6352834
Votes   0
Synopsis   Java Runtime reads attribute_length as a signed value
Category   java:runtime
Reported Against
Release Fixed
State   In progress, bug
Related Bugs
Submit Date   18-NOV-2005
Description

```
FULL PRODUCT VERSION :
java version "1.5.0_01"
Java (TM) 2 Runtime Environment, Standard Edition (build 1.5.0_01-b08)
Java HotSpot(TM) Client VM (build 1.5.0_01-b08, mixed mode)

ADDITIONAL OS VERSION INFORMATION :
Microsoft Windows XP [version 5.1.2600]

A DESCRIPTION OF THE PROBLEM :
The runtime interprets large attribute_length values incorrectly.
Values in the range 0x80000000 and 0xFFFFFFFF are considered negative.

STEPS TO FOLLOW TO REPRODUCE THE PROBLEM :
1/ Compile a Test class containing a final field.
2/ Set the attribute_length to 0xFFFFFFFF in the ConstantValue
attribute
3/ run Test

or
1/ Compile a Test2 class containing a final field
2/ replace "ConstantValue" with something else (any name not
recognized by the VM)
3/ set the attribute_length to 0xFFFFFFF2 (-14)
4/ run Test2

EXPECTED VERSUS ACTUAL BEHAVIOR :
EXPECTED -
Exception in thread "main" java.lang.ClassFormatError: Truncated Class
File for both Test and Test2
ACTUAL -
Exception in thread "main" java.lang.ClassFormatError: Invalid
ConstantValue field attribute_length -1 in class file Test

and

Exception in thread "main" java.lang.ClassFormatError: Repetitive
field name/signature in class file Test2

The latter shows another problem : the file pointer goes backwards
when a negative attribute_length is read. In this case we go 14 bytes
backwards, so that the same field is read again.

REPRODUCIBILITY :
This bug can be reproduced always.

---------- BEGIN SOURCE ----------
No source code, manual edition required.
---------- END SOURCE ----------
```

Work Around   N/A
Evaluation   N/A

Appendix C : JasminXT Syntax

This document is an excerpt from the JasminXT documentation, available at
http://jasmin.sourceforge.net/xt.html

JasminXT File Format

This new version is an extension of the existing Jasmin language, therefore old Jasmin files
should still compile correctly with newer versions of Jasmin. JasminXT is supported by Jasmin
2.0 or higher.

In the rest of this document, words between '[' and ']' are optional. The syntax of a JasminXT file
is the following :

```
<jas_file> {
    <jasmin_header>
    [<fields>]
    [<methods>]
}
```

JasminXT Header

```
<jasmin_header> {
    [.bytecode <x.y>]
    [.source <sourcefile>]
    <class_spec>
    <super_spec>
    <implements>
    [.signature "<signature>"]
    [.debug "<debug_source_extension>"]
    [.enclosing method <method_name>]
}

example :
.bytecode 49.0
.source hello.j
.class hello
.super java/lang/Object
.signature "<my::own>Signature()"
.debug "this string will be included in the SourceDebugExtension
attribute"
.enclosing method foo/bar/Whatever/someMethod()
```

The .bytecode directive sets the version of the bytecode in the class file.

The .signature directive, when used in the header of the Jasmin file, sets the Signature attribute
for the class (the argument is a string between double quotes)

The .debug directive sets the SourceDebugExtension attribute for the class (the argument is
also a string between double quotes)

The .enclosing directive sets the EnclosingMethod attribute for the class. The argument is a
supposed to be a method name, but it can be any string between double quotes.

JasminXT Class, Super Class and Interfaces Definition

```
<class_spec> {
    .class <access_spec> <class_name>
}
```

where <access_spec> is any number of words taken from this list : public, private, protected, static, final, synchronized, native, final, super, interface, abstract, annotation, enum, bridge/volatile, transient/varargs

and <class_name> is the fully qualified internal form of the class, such as my/package/MyClass

```
<super_spec> {
    .super <class_name>
}
```

```
<implements> {
    .implements <class_name>
    (...)
}
```

The .super and .implements directives have not been modified in JasminXT
The .implements directive can be repeated in order to implement multiple interfaces

JasminXT Field Definition

```
<fields> {
    .field <access_spec> <field_name> <descriptor> [signature
<signature>]
            [ = <value> ]
    (...)
}
```

The only addition is the optional signature attribute. If present, the Signature attribute will be set in the class file for this field with the given quoted string as an argument.

example :

```
.field enum myField Ljava/lang/String; signature
"<my::own>Signature()" = "val"
```

JasminXT Method Definition

The general format of a method definition has not changed in JasminXT.

```
<methods> {
    <method>
    (...)
}
```

```
<method> {
    .method <access_spec> <method_name> <descriptor>
        <statement>
        (...)
    .end method
}
```

JasminXT Method Statements

```
<statement> {
    .limit stack <integer>
    | .limit locals <integer>
    | .line <integer>
    | .var <var_number> is <var_name> <descriptor> from <label1> to
<label2>
    | .var <var_number> is <var_name> <descriptor> from <offset1> to
<offset2>
    | .throws <classname>
    | .catch <classname> from <label1> to <label2> using <label3>
    | .catch <classname> from <offset1> to <offset2> using <offset3>
    | .signature "<signature>"
    | .stack
        offset <pc>
        [locals <verification_type> [<verification_arg>]]
        (...)
        [stack  <verification_type> [<verification_arg>]]
        (...)
    .end stack
    | <instruction> [<instruction_args>]
    | <Label>:
}
```

In Jasmin XT you can now use offsets instead of labels for the local variable definitions and for the exception handlers definitions.

The .signature sets the Signature attribute for this method with the given quoted string.

You can now also define StackMap attributes using the .stack directive. <pc> is an offset in the local bytecode array. <verification_type> is one of the following keywords : Top, Integer, Float, Long, Double, Null, UninitializedThis, Object or Uninitialized. Object takes a <classname> as a parameter. Uninitialized takes an integer as a parameter.

example :

```
.stack
    offset 16
    locals Null
    locals Top
    locals Object allo
    stack Uninitialized 12
.end stack
```

This statement defines a single stack map frame. All the stack map frames defined in a method are then aggregated and form the StackMap attribute for the method.

JasminXT Instructions

```
<instruction> {
    [<pc>:] <opcode> [<instruction_args>]
}
```

The main change in JasminXT is that it is now possible to put the offset of the instruction before the opcode (the <pc>: statement). The pc is processed as a label, therefore you can virtually put any number as the pc but it won't change the actual pc of the bytecode.

Another update is that it is now possible to use offsets (both relative and absolute) as branch targets instead of labels. The offset is considered to be relative if it begins with a plus or a minus sign.

example :

```
goto  n  ; absolute offset : go to the bytecode labelled n
goto +n  ; relative offset : go n bytes forward (from the offset of
this goto)
goto -n  ; relative offset : go n bytes backwards
```

If something hasn't been documented here, it means that it hasn't changed, so you can still refer to the Jasmin user guide